SLIDING INTO BDOS

PART I

SLIDING INTO BDOS

THE SMOOTH AND EASY WAY

by: Michael J. Karas
2468 Hansen Court
Simi Valley, CA 93065

What is this thing everybody is talking about called BDOS? This series will attempt to answer this question in some detail but first we need a little basis to understand WHY in the first place. Digital Research CP/M is an operating system for smaller type micro processor computer systems that is designed to remove much of the normal computer operation drudgery experienced by the computer operator. The operating system software embodies a "system philosophy" that structures and generalizes upon the operating environment of a piece of electronics hardware. The environment presented actually allows that piece of quiet, transistorized machinery to be used at a much higher level. The full impact of what this operating system provides to a computer is most probably felt by the typical micro computer hacker that worked the hard way to get a computer system up and running. While building, debugging, and integrating the pieces, the computer was just a whole bunch of parts interfaced together in an organized manner. However, when the thing is finally a "computer" how does it get used. The low level process of poking data into memory from a front panel or even filling, dumping, or block moving memory data with an EPROM based "monitor program" hardly makes this computer "useful". The process of putting on disks and bringing up CP/M lights the torch for computer usability. In this case the hacker experiences an elated feeling now "NOW I CAN DO SOMETHING!"

Buried inside of the total operating system presentation is the concept of generalization brought up in the previous paragraph. One of the major requirements in order to make a computer useful is that there has to be applications software that performs the jobs intended for the computer. Jobs like accounting, word processing, spread sheet data analysis, or inventory control. Unfortunately the process of producing applications software is very, very expensive. A good package may

take anywhere from one to ten man years of development effort to make. If the process of making an applications package had to be custom taylored to a specific hardware environment, then there would not be affordable software available for use upon a given XYZ computer. Generalization in the operation of a computer environment solves this problem however. With the understanding that at a certain level "all microprocessor computer systems are alike" it is possible, with minimum constraints, to define a set of logical type operations that make a computer useful.

This logical set of operations, for the Digital Research CP/M operating system, is defined within the BDOS portion of the operating system. Here in about 3 1/2 K bytes of tightly written assembly language is the "generalization converter" that takes I/O requests for hardware independant applications programs and turns them into a lower level set of simplistic hardware oriented functions that are then processed through the BIOS. This conversion process is beneficial in the light that CP/M Ver 2.2 can be setup to run on a typical brand XYZ computer for about one half of the effort needed to convert even one of the simplest application packages had that application been written in a hardware dependant manner. Conclusion; software developers can make better, more sophisticated applications available for lower cost and computer users find a competitive software market place where there are many times multiple packages available that perform similar functions.

The thrust of this presentation is to show the prospective applications programmer how to use most of the generalized set of "BDOS System Calls" within Digital Researches CP/M Ver 2.2. The presentation scheme will be to describe all of the functions and use simple examples. The reader is assumed to be modistly familiar with 8080 Assembly Language Programming as all of the examples will be given in machine language. Likewise, in this environment it is assumed by default that the prospective programmer is planning to code in assembly language. If a CP/M compatible high level language is used for programming, such as Digital Research PL/I-80 or Microsoft BASIC-80, then of course the program interface at the "System Call" level becomes transparent to the programmer. Run time subroutines make the high level coded application get converted through yet another step. (One major reason applications code in a high level language runs slower than the equivalent function written in assembly language).

SUMMARY OF CP/M SYSTEM CALLS

The set of system or "BDOS" I/O entry points available to the CP/M programmer is complete yet simple. The primary beauty of the CP/M system is this small world of completeness. Many programmers familair with other operating systems complain that

the CP/M system is weak, unflexible, and incomplete. However, in a microprocessor type computer world, the generalization level defined for the CP/M system allows 85% of all microprocessor type appliciation jobs to be programmed with relative ease. Also, in my opinion, 8-bit microprocessor hardware is easily capable of performing about 90 percent of the typical tasks targeted for microcomputers. So what is this set of functions? The chart of Figure 1 summarizes, in function number order, all of the system operations specific to CP/M Version 2.2 that will be covered in this presentation. In the subsequent sections that follow the functions will be grouped into categories so that related operations may become familiar with reference to one another.

FIGURE 1. DETAILED SUMMARY OF CP/M 2.2 SYSTEM CALLS

Function Number			Entry Value to BDOS Passed in	Return Value from BDOS Passed in
DEC	HEX	Function	(DE) or (E) regs	(HL) or (A) register
0	00	System Reset	****	****
1	01	Console Input	****	(A)=character
2	02	Console Output	(E)=character	****
3	03	Reader Input	****	(A)=character
4	04	Punch Output	(E)=character	****
5	05	Printer Output	(E)=character	****
6	06	Direct Console I/O	(E)=0FFH is input	(A)=character
			(E)=chr is output	****
7	07	Get IOBYTE	****	(A)=IOBYTE
8	08	Set IOBYTE	(E)=IOBYTE	****
9	09	Display Console String	(DE)=string addr	****
10	0A	Input Console String	(DE)=string addr	(A)=# chr input
11	0B	Get Console Status	****	(A)=000H idle
				(A)=0FFH ready
12	0C	Get CP/M Version Number	****	(HL)=Version #
13	0D	Reset Disk Subsystem	****	****
14	0E	Select Disk Drive	(E)=disk number	****
15	0F	Open a File	(DE)=FCB address	(A)=dir code
16	10	Close a File	(DE)=FCB address	(A)=dir code
17	11	Search for File	(DE)=FCB address	(A)=dir code
18	12	Search for Next	****	(A)=dir code
19	13	Delete File	(DE)=FCB address	(A)=dir code
20	14	Read next Record	(DE)=FCB address	(A)=error code
21	15	Write next Record	(DE)=FCB address	(A)=error code
22	16	Create New File	(DE)=FCB address	(A)=dir code
23	17	Rename File	(DE)=FCB address	(A)=dir code
24	18	Get Login Vector	****	(HL)=login vector
25	19	Get Logged Disk Number	****	(A)=logged disk
26	1A	Set R/W Data Buff Addr	(DE)=buffer addr	****
27	1B	Get Allocation Vector	****	(HL)=alloc vector
				address

```
28    1C | Write Protect Disk     | (E)=disk number  |      ****         |
29    1D | Get Read Only Vector   |      ****        | (HL)=R/O vector   |
30    1E | Set File Attributes    | (DE)=FCB address | (A)=dir code      |
31    1F | Get Addr of Disk Parms |      ****        | (HL)=parm addr    |
32    20 | Get/Set User Select    | (E)=0FFH get     | (A)=current user  |
33    21 | Read Random Record     | (DE)=long FCB adr| (A)=error code    |
34    22 | Write Random Record    | (DE)=long FCB adr| (A)=error code    |
35    23 | Get Size of File       | (DE)=long FCB adr| (r0-2=rec cnt)    |
36    24 | Set Random Record Num  | (DE)=long FCB adr| (r0-2=rec numb)   |
37    25 | Reset Drive            | (DE)=drive vector|      ****         |
38    26 | Not used               |                  |                   |
39    27 | Not used               |                  |                   |
40    28 | Write Random with      | (DE)=long FCB adr| (A)=error code    |
-------------------------------------------------------------------------
```

The technical means required to "use" or interface to the CP/M system for each function contains a certain common structure that will be discussed here. The base memory page of a CP/M system memory map includes, at a specific memory address, a JUMP instruction to the CP/M BDOS entry point. For most CP/M systems this is address 00005H. To accomplish BDOS I/O the number of the function is placed into the (C) register. If the parameter requires input parameters, then they are passed in the (DE) register pair or the individual (E) register depending upon whether the parameter is a word or byte value. Result information returned by some functions is sent back to the users program in either the (A) register or the (HL) register pair depending upon if the value is a byte or word. The following simple program segment demonstrates the scheme used to output the 26 characters A-Z to the console screen through the use of function number 2.

```
BDOS      EQU       0005H          ;SYSTEM ENTRY
CONOUT    EQU       2              ;OUTPUT FUNCTION

          ORG       0100H          ;TPA BASE
          MVI       B,26           ;PRINT 26 COUNTER
          MVI       C,'A'          ;START WITH 'A'
;
LOOP:
          PUSH      B              ;SAVE COUNTER & LETTER
          MOV       E,C            ;LETTER TO (E) FOR OUTPUT
          MVI       C,CONOUT       ;BDOS FUNC TO (C)
          CALL      BDOS           ;GO GO OUTPUT
          POP       B
          INR       C              ;SEQUENCE TO NEXT CHAR
          DCR       B              ;DECREASE CHR COUNTER
          JNZ       LOOP           ;MORE TO DO IF NOT TO ZERO
          RET                      ;IMMEDIATE CCP RETURN
```

SYSTEM CALLS FOR OPERATOR CONSOLE INPUT AND OUTPUT

Intrinsic to the operation of any computer system, especially of the CP/M gender, is the operator console. The device provides the human interface to the machine and as such the BDOS includes a generalized set of operator communication functions to perform I/O with the console device. The various options available will each be presented with a brief example.

INPUT FROM CONSOLE KEYBOARD: Function 1.

This function waits for and reads in a character from the console device keyboard. The operator typed character is echoed automatically back to the console display if the character is an ASCII printable character (020H to 07EH) or it is a carriage return, line feed, back space, or tab. Note that the BDOS automatically expands tabs to columns of eight characters. Upon outputting the character for the echo, a check is made for console start/stop, CTL-S, and if so the console input routine does not return to the users program until another arbitrary key is depressed.

```
;CONSOLE INPUT EXAMPLE
;
CONIN     EQU       001H          ;FUNC # 1
BDOS      EQU       0005H         ;SYSTEM ENTRY

          ORG       0100H         ;START
          MVI       C,CONIN       ;FUNCTION
          CALL      BDOS          ;GO GET CHARACTER
          STA       INCHAR        ;SAVE FOR WHATEVER REASON
          RET                     ;IMMEDIATE CCP RETURN
;
INCHAR:
          DS        1             ;PLACE TO STORE INPUT CHAR
;
          END
```

OUTPUT TO CONSOLE DISPLAY: Function 2.

The ASCII character in the (E) register is sent to the console display device. The output may be any byte value but many times the hardware driver BIOS routines automatically strip off the upper bit of the byte. Upon output the printer echo flag within BDOS is checked (CTL-P) and if set the character is also sent to the printer peripheral device. Note that the BDOS automatically expands output tabs to columns of eight characters. Upon outputting the character a check is made for input of console start/stop, CTL-S, and if so the console output routine does not return to the users program until another arbitrary key is depressed.

```
;CONSOLE OUTPUT EXAMPLE
;
CONOUT    EQU       002H          ;FUNC # 2
BDOS      EQU       0005H         ;SYSTEM ENTRY

          ORG       0100H         ;START
          LDA       OUTCHAR       ;GET CHARACTER TO OUTPUT
          MOV       E,A
          MVI       C,CONOUT      ;FUNCTION
          CALL      BDOS          ;GO SEND CHARACTER
          RET                     ;IMMEDIATE CCP RETURN
;
OUTCHAR:
          DB        'X'           ;PLACE TO GET OUTPUT CHAR
;
          END
```

DIRECT USER INTERFACE TO CONSOLE: Function 6.

Some programming applications require that the BDOS not monitor the input/output character stream as is done with functions 1 & 2. To allow for these functions the direct I/O function is supported. The following example shows how it is used to input values and echo them until an input control-Z character is typed.

```
;DIRECT CONSOLE I/O EXAMPLE
;
DIRCIO    EQU       006H          ;FUNCTION NUMBER
BDOS      EQU       0005H         ;SYSTEM ENTRY POINT
CTLZ      EQU       'Z'-040H      ;ASCII CTL-Z CHARACTER
INPUT     EQU       0FFH          ;DIRECT INPUT FLAG

          ORG       0100H         ;CONSOLE INPUT
;
LOOP:
          MVI       E,INPUT       ;SET FOR INPUT
          MVI       C,DIRCIO      ;FUNCTION
          CALL      BDOS          ;GET INPUT OR STATUS
          ORA       A             ;IF (A)=0 NO CHAR WAS READY
          JZ        LOOP          ;CONTINUE TO WAIT FOR INPUT
          CPI       CTLZ          ;IF INPUT WAS CTL Z THEN END
          RZ                      ;CCP RETURN ON END
          MOV       E,A           ;CHARACTER TO (E) FOR OUTPUT
          MVI       C,DIRCIO      ;SAME FUNCTION NUMBER AGAIN
          CALL      BDOS          ;GO OUTPUT IT
          JMP       LOOP          ;NEXT CHARACTER INPUT LOOP
;
          END
```

PRINTING STRINGS OF CHARACTERS TO THE CONSOLE: Function 9.

Message string sequences of characters to be sent to the console are quite common in applications programming. Typical uses may be for user prompt messages, program sign-on messages etc. The BDOS provides a convenient mechanism to allow the programmer to output a whole string of characters rather than having to loop with single character outputs. The string is intended to be stored in consecutive memory locations and end with the ASCII '$' character. The (DE) registers are used to point to the start of the string. The '$' signals the end of the string to display and is not sent to the console. The output bytes may be any 8-bit value but many times the hardware driver BIOS routines automatically strip off the upper bit of the byte. Upon output of each character the printer echo flag within BDOS is checked (CTL-P) and if set the character is also sent to the printer peripheral device. Note that the BDOS automatically expands output tabs to columns of eight characters. Upon outputting each character a check is made for input of console start/stop, CTL-S, and if so the console string output routine does not return to the users program until another arbitrary key is depressed.

```
;CONSOLE STRING PRINT EXAMPLE
;
CONSTR    EQU       009H          ;FUNC # 9
BDOS      EQU       0005H         ;SYSTEM ENTRY
CR        EQU       0DH           ;ASCII CARRIAGE RETURN
LF        EQU       0AH           ;ASCII LINE FEED

          ORG       0100H         ;START
          LXI       D,MESSAGE     ;POINT AT STRING TO SEND
          MVI       C,CONSTR      ;FUNCTION
          CALL      BDOS          ;GO SEND STRING
          RET                     ;IMMEDIATE CCP RETURN
;
MESSAGE:
          DB        CR,LF,'Hello Operator',CR,LF,'$'
;
          END
```

READING A STRING OF CHARACTERS IN FROM KEYBOARD: Function 10.

The CP/M console command processor (CCP) assumed to be vary familiar to most CP/M system operators allows buffered command input with editing features. It turns out that this operation is a much needed function for getting in strings of text from the operator console. Use of this function allows standardization of the command input functions so that the operator can easily learn the editing key functions. It also removes the pain of writing the same function over and over again by the applications programmer. The read string command inputs the edited text to a

```
buffer pointerd to by the (DE) register pair. The caller
specifies the maximum length desired and the BDOS returns the
actual length of string entered if carriage return is entered
prior to exceeding the maximum input length. The input length is
returned in both the (A) register and as part of the buffer.
Bytes in the string buffer past the end of the entered text are
uninitialized. The example shown below gives an assembly language
view point of the buffer structure and how to program an input
function.

     The editing functions supported are the following control
and/or special characters:

          rub/del        removes and echos the last entered char
           ctl-C         initiates system reboot if first char
           ctl-E         echos a CR & LF to console without
                           putting them into buffer
           ctl-H         (or back space key) back spaces one char
                           removing last entered character
           ctl-J         (or line feed key) terminates line input
           ctl-M         (or carriage return) terminates input
           ctl-R         retypes currently entered characters
                           under current line
           ctl-U         deletes all of currently entered data
                           and restarts buffer input on new line
           ctl-X         deletes all of currently entered data
                           and restarts buffer input on same line

;CONSOLE INPUT BUFFER EXAMPLE
;
CONBUF    EQU       00AH           ;STRING INPUT FUNCTION
BDOS      EQU       0005H          ;SYSTEM ENTRY POINT
LENGTH    EQU       32             ;DESIRED MAXIMUM CHARACTERS

          ORG       0100H          ;START POINT
          LXI       D,STRING       ;POINT AT BUFFER AREA
          MVI       C,CONBUF       ;FUNCTION NUMBER
          CALL      BDOS           ;GO GET STRING
          RET                      ;RETURN TO CCP WITHOUT
                                   ;...DOING ANYTHING WITH DATA
;
;
;CONSOLE INPUT BUFFER LAYOUT
;
STRING:
          DB        LENGTH         ;MAXIMUM DESIRED INPUT LENGTH
AMOUNT:
          DS        1              ;BYTE WHERE BDOS RETURNS
                                   ;..ACTUAL BYTE COUNT
STRBF:
          DS        LENGTH         ;RESERVED STORAGE FOR UP TO
```

```
                                    ;"LENGTH" NUMBER OF CHARACTERS
;
          END
```

DETERMINING IF THERE IS PENDING KEYBOARD INPUT: Function 11.

Some computer programs are designed to spend large amounts of time processing inside of the computer or manipulating data within disk files without stopping to ask the user if he/she desires to stop the processing sequence. Also it is many times desirable to have a "terminate" capability for application programs without waiting for the operator to answer a character input request. If the normal console input function is used the user computer is not resumed until a character is already input. The console input status check function may be used to poll the user keyboard to determine if a character input is pending. If no input is ready then the user program is immediately resumed with an indication of if there was a pending input. If a character is pending a 0FFH is returned in the (A) register. Otherwise a 000H value is returned. The following example illustrates the use of console status to terminate a normally endless loop that prints the same string over and over.

```
;CONSOLE STATUS USAGE EXAMPLE
;
CONSTAT   EQU       00BH          ;FUNC # 11
CONSTR    EQU       009H          ;PRINT STRING FUNCTION
BDOS      EQU       0005H         ;SYSTEM ENTRY
CR        EQU       0DH           ;ASCII CARRIAGE RETURN
LF        EQU       0AH           ;ASCII LINE FEED

          ORG       0100H         ;START
LOOP:
          LXI       D,MESSAGE     ;POINT AT STRING TO SEND
          MVI       C,CONSTR      ;FUNCTION
          CALL      BDOS          ;GO SEND STRING
          MVI       C,CONSTAT     ;GET ABORT STATUS
          CALL      BDOS
          ORA       A             ;CHECK STATUS
          JZ        LOOP          ;NO KEY SO CONTINUE LOOP
          RET                     ;IMMEDIATE CCP RETURN IF ABORT
;
MESSAGE:
          DB        CR,LF,'Depress any Key to STOP','$'
;
          END
```

```
AUXILLIARY PERIPHERAL CHARACTER INPUT AND OUTPUT FUNCTIONS

     The  generalized CP/M BDOS provides the capability for three
character by character logical I/O devices to be atteched to  the
computer  system.  This requirement stems from the fact that most
computers  are  designed to interface to the real world  in  more
ways than just a console device. The three devices are classified
as:

     a)  A lister type device that is generally expected to be  a
printer  of  some sort.  This classification is  an  output  only
device.

     b)  An input device supporting character input from a source
other than the console.  The device is specifcally an input  type
unit.  CP/M  jargon refers to this device as the "READER" for  no
particular reason.

     c)  A  generalized character output only device  used  as  a
specific data destination other than the console or standard list
device.  Some  computer  systems  use this  device,  often  times
referred to as the "PUNCH" device as a second printer output.

     The  three  following examples  illustrate  the  programming
techniques used to talk to each of these three devices.

;LIST DEVICE OUTPUT EXAMPLE
;
LIST      EQU       005H           ;FUNC # 5
BDOS      EQU       0005H          ;SYSTEM ENTRY

          ORG       0100H          ;START
          LDA       LSTCHAR        ;GET CHARACTER TO OUTPUT
          MOV       E,A
          MVI       C,LIST         ;FUNCTION
          CALL      BDOS           ;GO SEND CHARACTER
          RET                      ;IMMEDIATE CCP RETURN
;
LSTCHAR:
          DB        'L'            ;PLACE TO GET OUTPUT CHAR
;
          END



;READER DEVICE INPUT EXAMPLE
;
READER    EQU       003H           ;FUNC # 3
BDOS      EQU       0005H          ;SYSTEM ENTRY

          ORG       0100H          ;START
```

```
          MVI       C,READER        ;FUNCTION
          CALL      BDOS            ;GO GET CHARACTER
          STA       RDRCHR          ;SAVE FOR WHATEVER REASON
          RET                       ;IMMEDIATE CCP RETURN
;
RDRCHR:
          DS        1               ;PLACE TO STORE INPUT CHAR
;
          END



;PUNCH DEVICE OUTPUT EXAMPLE
;
PUNCH     EQU       004H            ;FUNC # 4
BDOS      EQU       0005H           ;SYSTEM ENTRY

          ORG       0100H           ;START
          LDA       PNCHCHR         ;GET CHARACTER TO OUTPUT
          MOV       E,A
          MVI       C,PUNCH         ;FUNCTION
          CALL      BDOS            ;GO SEND CHARACTER
          RET                       ;IMMEDIATE CCP RETURN
;
PNCHCHR:
          DB        'P'             ;PLACE TO GET OUTPUT CHAR
;
          END
```

SYSTEM CONTROL BDOS FUNCTIONS

This family of system calls supported by the CP/M BDOS are designed to allow the programmer a degree of flexibility in manipulating the operation of general CP/M environment. Each function here will generally be discussed individually due to the unique nature of each operation.

SYSTEM RESET: Function 0.

The system reset function is designed to allow restart of the CP/M system command processor after a user application completes execution or is aborted. The system reset function is equivalent to a JMP to address 0000H or a CTL-C which forces a system WARM Reboot. The reboot operation de-activates all active drives except drive A: which is re-logged. Operation is extremely simple as:

```
RESET     EQU       000H            ;SYSTEM RESET FUNC
```

```
BDOS      EQU       0005H          ;SYSTEM ENTRY POINT

          ORG       0100H
          MVI       C,RESET
          JMP       BDOS           ;CALL ALSO PERMISSABLE
                                   ;EXCEPT THAT FUNCTION
                                   ;DOES NOT RETURN TO USER
                                   ;PROGRAM


GET AND SET IOBYTE: Functions 7 & 8.

     The    generalized   CP/M   operating   system   environment
communicates via I/O to "logical" type devices.  This means  that
the console,  lister, "reader", and "punch" are just treated as a
generic  device classsifications.  The CP/M system allows for and
supports, to a degree, the capability for the hardware to contain
multiple  physical devices (peripherals and/or real I/O  devices)
within  each of the generic logical device  classifications.  The
means to support the assignment of multiple physical devices to a
given classification is done through the IOBYTE,  normally stored
at address 00003H of the base page of the CP/M memory.  The  BIOS
hardware  I/O software may thusly be written to easily know which
one  of two printers to talk to when the BDOS requires output  to
one of two printers.  A "default standard" IOBYTE format has been
adopted  based  upon an 8-bit  microprocessor  system  convention
developed by Intel Corp as follows:

                        (lister) (punch)  (reader) (console)
      Logical Devices =>   LST:     PUN:     RDR:     CON:
          IOBYTE bits =>   7 6      5 4      3 2      1 0
      ------------------------------------------------------
        Bit pattern
        dec  binary
         0     00          TTY:     TTY:     TTY:     TTY:
         1     01          CRT:     PTP:     PTR:     CRT:
         2     10          LPT:     UP1:     UR1:     BAT:
         3     11          UL1:     UP2:     UR2:     UC1:

     The  designators in the table specify the "standard types of
physical devices and are defined as follows:

     TTY: A teletype console with keyboard, hard copy display and
          possibly an integral tape reader/punch
     CRT: An interactive cathode ray type terminal with keyboard
          input and display screen
     BAT: A batch processor workstation with a card reader type
          input device and a hard copy display/output device
     UC1: A user defined alternate "console" unit
     LPT: Line printer
     UL1: A user defined list device
```

```
     PTR: Paper Tape Reader
     UR1: User defined "reader" character input device
     UR2: User defined "reader" character input device
     PTP: Paper Tape Punch
     UP1: User defined "punch" character output device
     UP2: User defined "punch" character output device
```

The BDOS support for the I/O device assignment is a standard mechanism to access the IOBYTE's current value and switch it to some other value. Suppose a CP/M computer had two printers connected as LST: and UL1:. If the applications program needs to switch printing output to another printer, the process could be handeled as follows:

```
;GET AND SET IOBYTE EXAMPLE
;
SETIOB    EQU       008H           ;SET IOBYTE FUNCTION
GETIOB    EQU       007H           ;GET IOBYTE FUNCTION
BDOS      EQU       00005H         ;SYSTEM ENTRY POINT
LSTMASK   EQU       11$00$00$00B   ;IOBYTE MASK FOR LIST
                                   ;..DEVICE
LPT       EQU       10$00$00$00B   ;BIT VALUE FOR LPT #1
UL1       EQU       11$00$00$00B   ;BIT VALUE FOR LPT #2

          ORG       0100H          ;PROGRAM START
          MVI       C,GETIOB       ;GO GET CURRENT IOBYTE VAL
          CALL      BDOS
          ANI       (NOT LSTMASK) AND 0FFH  ;KEEP ALL OTHER BITS
          ORI       UL1 AND LSTMASK ;SET IOBYTE FOR PRINTER #2
          MOV       E,A
          MVI       C,SETIOB       ;FUNCTION TO RESET THE IOBYTE
          CALL      BDOS
          RET                      ;IMMEDIATE CCP RETURN
;
          END
```

GET CP/M VERSION NUMBER: Function 12.

Sometimes it is necessary for an applications program to "know" what version of CP/M the program is running under. Version 2.0 and above support a feature to tell the application program what the version number is. One reason is to permit version dependant functions such as random record file I/O to be used if it is supported by the version of CP/M being used. The system call to get the version number returns a two byte value split into two parts as follows:

```
          if (H)=0 then this is a CP/M System
             (H)=1 then this is an MP/M System
             (L)=version number in hex
```

```
          if (L)=00 then older than CP/M 2.0
             (L)=20 then version CP/M 2.0
             (L)=21 then version CP/M 2.1
             (L)=22 then version CP/M 2.2
```

A program to read the CP/M version number is as follows:

```
;VERSION NUMBER EXAMPLE
;
GETVERS   EQU       00CH          ;FUNCTION 12
BDOS      EQU       00005H        ;SYSTEM ENTRY POINT

          ORG       0100H         ;PROGRAM START
          MVI       C,GETVERS     ;FETCH VERSION NUMBER
          CALL      BDOS
          MOV       A,L           ;SAVE CP/M VERSION NUMBER
          STA       CURVERS
          RET                     ;BACK TO CCP
;
CURVERS:
          DS        1             ;STORE THE VERSION NUM HERE
          END
```

RESETTING THE CP/M DISK SYSTEM: Function 13.

The CP/M operating system contains features to control access to files upon the disk drives. A directory checksum scheme, beyond the scope of this presentation, permits the operating system to determine when a disk has been changed in a drive thus preventing the a wrong disk from being written upon. This is neat except that in many cases an appliciations program may require disk changes as functions are changed or new files are required. This system control function permits the application to force read/write status to be set for all drives, drive A: to be logged, and reset of the default disk record buffer address to its default value of 080H within the CP/M base page. The following program sequence shows how to reset the disk system.

```
;RESET DISK SYSTEM EXAMPLE
;
RESET     EQU       0DH           ;FUNCTION 13
BDOS      EQU       0005H         ;SYSTEM ENTRY POINT

          ORG       0100H         ;PROGRAM START
          MVI       C,RESET       ;SET UP FUNCTION
          CALL      BDOS          ;GO RESET THE DRIVES
          RET                     ;BACK TO THE CCP
;
          END
```

GET AND SET OF CURRENT USER CODE: Function 32.

CP/M Version 2.2 permits the file system on a given drive to be partitioned into up to 15 individual directory areas so that usage areas can be setup. For instance, the system operator could put all assembly language development programs in one user area while having disk utility programs in another. The BDOS allows the application programmer to determine the currently logged user number and to modify it if necessary. The following example sets the current user number up by one. If the highest user number is currently logged then the user 0 area is selected.

```
;GET/SET USER EXAMPLE
;
GSUSR     EQU       020H          ;FUNCTION 20
GET       EQU       0FFH          ;GET FLAG
BDOS      EQU       0005H         ;SYSTEM ENTRY POINT

          ORG       0100H         ;START UP POINT
          MVI       E,SET         ;MAKE THIS A FETCH NUM RQST
          MVI       C,GSUSR
          CALL      BDOS          ;GET THE CURRENT USER #
          INR       A             ;BUMP RETURNED USER UP 1
          ANI       00FH          ;MASK TO MOD(15)
          MOV       E,A           ;MOVE FOR SET TO NEW USER
          MVI       C,GSUSR
          CALL      BDOS
          RET                     ;CCP GETS US BACK
;
          END
```

SYSTEM FUNCTIONS THAT CONTROL THE DISKS

The data storage files for applications programs are stored upon the disk drives attached to the CP/M computer. The BDOS supports a number of functions that allow the state and selection status of the drives to be controlled.

SELECT DISK: Function 14.

The simplest control function is to select the current disk with which to refer to as the logged or default disk. The function is equivalent to the console CCP command:

```
          A>B:<cr>
          B>
```

Which changed the currently logged disk to drive B:. A BDOS program to affect the same thing is given in the example program of the next section below. Drive numbers correspond to the

```
console displayed drive designators as follows:

          A: = Drive # 0
          B: = Drive # 1

              ***

          P: = Drive # 15

Once  a drive has been selected it has its directory  "activated"
and is maintained in a logged in status until the next warm boot,
cold boot, or disk reset BDOS function.


DETERMINE LOGGED DISK: Function 25.

     An  applications program can determine which disk  drive  is
the  currently  logged  or  default drive  through  use  of  this
function.  The BDOS will return in the (A) register the number of
the currently selected drive according to the table given above.

     The program segment below shows a sequence of BDOS interface
code  that first determines if drive B:  is selected,  and if not
then does a BDOS call to change it.

;SELECT AND POLL LOGGED DISK DRIVE EXAMPLE
;
SELECT    EQU       0EH            ;FUNCTION 14
ASKDRV    EQU       19H            ;FUNCTION 25
BDOS      EQU       0005H          ;SYSTEM ENTRY POINT

          ORG       0100H          ;PROG START
          MVI       C,ASKDRV       ;FIND OUT IF B: IS SELECTED
          CALL      BDOS
          CPI       'B'-'A'
          RZ                       ;DONT SELECT IF ALREADY
                                   ;..LOGGED
          MVI       E,'B'-'A'      ;SET TO LOG AND SELECT B:
          MVI       C,SELECT
          CALL      BDOS
          RET                       ;FINISHED WITH ANOTHER PROG
;
          END


DRIVE STATUS SET AND RESET: Functions 28 & 37.

     Drive  status  may  be  individually  controlled  by   these
functions.  Operation 28 allows a the currently selected drive to
be write protected (set to read/only). The process is simply:
```

```
WPDSK     EQU       01CH
BDOS      EQU       0005H
          MVI       C,WPDSK        ;WRITE PROTECT DISK
          CALL      BDOS
```

The write protect status of a specific disk may be removed by function 37 which deactivates the directories of each drive specified at call time. Each drive by default then becomes read/write again but requires reactivation through reselection. The reset drive vector is a 16-bit value passed to the BDOS with a "1" bit in each bit position for a drive that equires resetting. The most significant bit of the 16 bit quanity corresponds to drive P: and the LSB to drive A:. The code sequence to reset drive B: would be:

```
RESDSK    EQU       025H
BDOS      EQU       0005H
          MVI       C,RESDSK       ;FUNCTION CODE
          LXI       D,0000$0000$0000$0010B  ;DRIVE B: BIT SET
          CALL      BDOS
```

GET DRIVE LOGIN AND READ?ONLY VECTORS: Function 24 & 29.

The BDOS keeps track of all drives that have been selected since the last boot or disk reset functions. These drives are considered in a online status in that the system knows immediately what the space allocation map of the drive is and whether the drive is in read/only status or not. Function 24 allows the application program to determine what subset of the current drive complement are in this online logged status. The vector returned in the (HL) register pair is a bit map like above where a "1" bit means the drive is active. The most significant bit of the 16-bit number corresponds to drive P:. The code below fetches the vector and saves it in a local data area.

```
;LOGIN VECTOR EXAMPLE
;
LOGIN     EQU       018H           ;FUNCTION 24
BDOS      EQU       0005H          ;SYSTEM ENTRY POINT

          ORG       0100H
          MVI       C,LOGIN        ;FUNCTION
          CALL      BDOS
          SHLD      LOCLOG         ;SAVE VECTOR HERE
          RET                      ;TO CCP
;
LOCLOG:
          DS        2
          END
```

In a similar manner the BDOS allows determination of which

```
drives are in the write protected read/only status.  A "1" bit in
the  returned  vector indicates read/only status for  a  specific
drive. The code here shows how to fetch it.

;READ/ONLY VECTOR EXAMPLE
;
ROVEC     EQU       01DH           ;FUNCTION 29
BDOS      EQU       0005H          ;SYSTEM ENTRY POINT

          ORG       0100H
          MVI       C,ROVEC        ;FUNCTION
          CALL      BDOS
          SHLD      LOCROV         ;SAVE VECTOR HERE
          RET                      ;TO CCP
;
LOCROV:
          DS        2
          END

GET ALLOCATION VECTOR AND DISK PARM POINTER: Function 27 & 31.

     Two  more  miscellaneous disk drive interface functions  are
provided  that  permit several special types of functions  to  be
performed.  The first, function 27 returns an address in the (HL)
registers that points to a bit string in memory that  corresponds
to the data block allocation map of the currently selected drive.
The  map  contains  one  bits  in each  position  where  a  block
allocated, starting with the MSB of the forst byte in the string.
The  length of the bit string depends upon the total capacity  of
the  drive  in  allocatable  blocks.   Function  31  permits   an
application  to  determine the characteristics of  the  currently
selected drive. The BDOS returns an address in the (HL) registers
that  points  to  a table of 33 bytes that describe  the  current
drive.  Data  in  the  table  includes such  data  as  number  of
possible  directory entries on the disk,  number  of  allocatable
blocks on the disk, and, indirectly, the size of each disk block.
The  program  below  is  a comprehensive  example  of  how  these
functions  can be used to determine the remaining space left on a
the selected drive. The program stores the available space of the
drive specified in the first byte of the default FCB into  memory
location "KPDISK" and then exits to the CCP. The reader can adapt
the code as desired.

;
;CP/M BDOS INTERFACE EQUATES
;
BASE    EQU     0000H           ;BASE OF CP/M SYSTEM
LOGDRIV EQU    0004H+BASE    ;LOCATION OF CURRENTLY LOGGED DRIVE
BDOS    EQU    0005H+BASE    ;THE BDOS I/O VECTOR
SLCTDSK EQU    14        ;SELECT DISK DRIVE
GALVEC  EQU    27        ;GET ADDRESS ALLOCATION VECTOR
```

```
GDSKP   EQU    31        ;GET ADDRESS OF DISK PARAMETER TABLE
;
;
    ORG    0100H
;
;
;PROGRAM TO FETCH REMAINING DISK SPACE IN KBYTES
;
SPCGET:
    LDA    LOGDRIV        ;GET CURRENTLY LOGGED DRIVE AND SAVE
    ANI    0FH        ;STRIP OUT USER NUMBER
    STA    SAVDRIV        ;SAVE CODE
;
    LDA    FCB        ;CHECK IF SAME AS SELECT
    DCR    A        ;ADJUST FCB DRIVE TO MATCH SELECT DRIVE
    MOV    E,A        ;..SELECT IN BDOS
    MVI    C,SLCTDSK    ;SELECT DISK FUNCTION
    CALL    BDOS
;
    MVI    C,GDSKP        ;FIND ADDRESS OF DISK PARAMETER HEADER
    CALL    BDOS
    LXI    B,0002H        ;INDEX TO BLOCK SHIFT FACTOR
    DAD    B
    MOV    B,M        ;(B) = BYTE BLOCK SHIFT FACTOR
    INX    H
    INX    H
    INX    H
    MOV    E,M        ;(DE) = WORD DISK BLOCK COUNT
    INX    H
    MOV    D,M
    INX    D
;
    MOV    A,B        ;ADJUST SHIFT FOR KBYTE SIZE
    SUI    03H
    LXI    H,0001H        ;CALCULATE BLOCK SIZE
SPCCAL:
    ORA    A        ;KNOW KBYTES PER BLOCK?
    JZ    SPCKNW
    DAD    H        ;DOUBLE # SECTORS PER TRACK
    DCR    A        ;DECREMENT BLOCK SHIFT
    JMP    SPCCAL
;
SPCKNW:
    MOV    C,L        ;(BC)=KBYTES PER BLOCK
    MOV    B,H
    LXI    H,0        ;INITIALIZE KPDISK
    SHLD    KPDISK
    PUSH    B        ;SAVE KBYTES/BLOCK
    PUSH    D        ;SAVE NUMBER OF BLOCKS
    MVI    C,GALVEC    ;NOW POINT TO THE ALLOCATION VECTOR
    CALL    BDOS        ;(HL)=ALLOCATION VECTOR ADDRESS
```

```
    POP    D
    POP    B
;
    SHLD    ALLSAVE        ;SAVE ALLOCATION POINTER
    MVI    H,1        ;SET MINIMUM START BIT COUNT
;
UALLOC:
    DCR    H        ;DEC BIT COUNT
    JNZ    STACT        ;STILL ACTIVE BYTE
;
    LHLD    ALLSAVE        ;GET POINTER
    MOV    A,M
    INX    H
    SHLD    ALLSAVE        ;SAVE NEW POINTER
    MVI    H,08H        ;SET BIT COUNTER TO MAX
;
STACT:
    RLC            ;GET ALLOCATION BIT TO CARRY
    JC    ALLOC        ;DONT COUNT ALLOCATED BLOCKS
    PUSH    H
    LHLD    KPDISK        ;GET KBYTES LEFT COUNT
    DAD    B        ;ADD IN ONE MORE BLOCK COUNT
    SHLD    KPDISK
    POP    H
;
ALLOC:
    DCX    D        ;DEC TOTAL BLOCK COUNT
    MOV    L,A
    MOV    A,D
    ORA    E        ;ALL BLOCKS SCANNED YET
    MOV    A,L        ;RESTORE ALLOC BIT PATTERN
    JNZ    UALLOC        ;MORE TO COUNT
;
    LDA    SAVDRIV        ;RETURN DISK SELECT TO PREVIOUS
    MOV    E,A        ;..SELECT IN BDOS
    MVI    C,SLCTDSK    ;SELECT DISK FUNCTION
    CALL    BDOS
    RET                    ;BACK TO THE CCP
;
;
;PROGRAM DATA STORAGE ALLOCATIONS
;
BLKSIZ:
    DS    2        ;STORAGE FOR ALLOCATION BLOCK SIZE
ALLSAVE:
    DS    2        ;STORAGE FOR ALLOCATION PNT SAVE
SAVDRIV:
    DS    1        ;SAVE CURRENT DISK SELECT DURING RELOG
KPDISK:
    DS    2        ;STORAGE FOR KBYTES PER DRIVE LEFT
;
```

```
    END
```

The next part in this series will present the the CP/M file system as viewed from the BDOS interface aspect. The FILE CONTROL BLOCK (FCB) will be presented. In addition the procedures to prepare files for I/O and then the actual I/O procedures will be presented. The series will round out to a conclusion with a comprehensive programming example that presents a sequential file I/O set of subroutines that permit character by character I/O with a file to be done.

PART II

SLIDING INTO BDOS (Part II)

WITH FILES MADE EASY

by:

Michael J. Karas
2468 Hansen Court
Simi Valley, CA 93065
(805) 527-7922

Since I know that all devoted Life Lines readers have anxiously been waiting for this "second in a series" tutorial on using files with the CP/M BDOS, I will not go on a long time telling you why this thing about CP/M BDOS file interface is so important. Nor will I try to justify why the turorial should be valuable. You wouldn't be reading here at this time if you had any inclination to find my work disinteresting. If you are new on the scene and have some questions about what this is all about I would like to direct your attention to the November 1982 issue of Life Lines where the first part of this tutorial series was presented. There the purpose of the BDOS and the general interface concepts were presented. The article went on to include a description of the physical device system calls and other miscellaneous system control type functions.

THIS TIME IT'S FILES

This month the tutorial continues with a description of the sequential file I/O system supported within the BDOS. The concepts of CP/M file storage are to be described along with appropiate CP/M directory structure definition as it relates to

the access of the files stored upon a CP/M disk. The FILE CONTROL BLOCK (FCB) will be described in terms of its functions as related the a file to be accessed upon a disk. I have also included a comprehensive programming example that allows a sequential file to be accessed character by character.

HOW FILES ARE STORED UPON THE DISK

The CP/M operating system manages the available space on a disk by dividing the total available space up into a number of relatively small data block storage areas called "GROUPS". A group size is usually described as the minimum allocatable space that a file can occupy. What this means is that the operating system, in its disk space management scheme, lumps sets of the normal 128 byte logical records of a file together into these things called groups. The number of groups that may be contained on a disk depends upon the total file storage space of the disk in logical 128 byte records divided by the number of 128 byte logical records lumped together into a group. (A note to the less casual reader is that the number of groups on a disk is limited by design to 65K groups. Secondly a group is always an integral power of two number of 128 byte logical records with a minimum size of 8 records (1K byte). Group size is necessarily limited to 16K bytes due to the extent system described below).

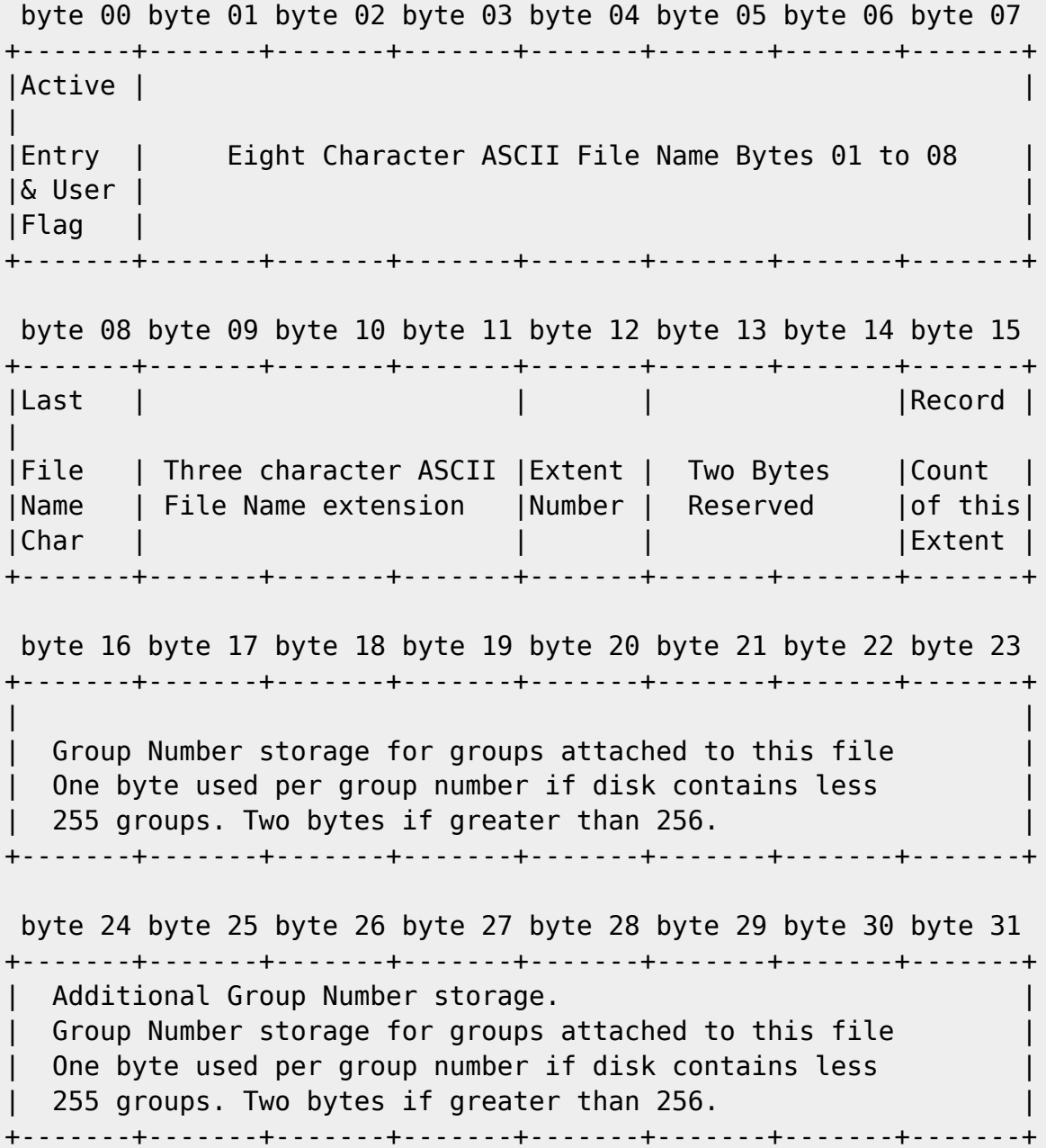
As a file is stored upon a CP/M disk it consumes disk space in 128 byte logical records. Each time a group becomes filled with records the operating system allocates another group to the file. Hence the term "minimum allocatable size". If, as the file grows in size, the last allocated group assigned to a file is not completely filled the remaining space in the group is "burned" in that it is not usable by other files. The CP/M system keeps track of the group assignments made to the various files on a disk, the files names, and the total number of 128 byte logical records in each file through a stored directory. The first portion of the disk is reserved for the file directory. A fixed number of directory entries, determined by the system's BIOS design, are available, usually a number like 64, 128, or 256, depending upon the size of the disk.

Each file has a unique directory entry "set" that describes the file location upon the disk. A "set" of directory entries is specified because each entry is designed to "point to" or store the group allocation numbers for that file. Each directory entry has a number slots where group numbers can be stored. The system design allows each directory entry to specify the storage for 16K bytes of storage space. For files larger than 16K bytes a seperate directory entry is used for each 16k bytes (or remainder portion thereof). Each such piece of a file is referred to as an "EXTENT" of the file. The directory entry "set" for a file

contains a byte in each extent directory entry that stores the extent number of the file. Extent numbers start with 0 and may increase to a theoretical limit of 255 or the size of the disk in 16K byte pieces, whichever is smaller.

The chart below describes the functions of all bytes in a typical directory entry. Each entry is 32 bytes long and they are packed four to a logical sector with the number of logical sectors filled up with directory entries limited to the predetermined number of directory entries divided by four.

Figure 1. DISK DIRECTORY ENTRY DEFINITION



The bytes of the disk directory entry are each described in the following paragraphs. The first byte stored in an entry is set to indicate if this slot in the predetermined directory area

is empty or if it describes an active file extent. A value of 0E5H indicates an empty slot. This value was chosen presumably due to that a freshly formatted diskette contains all 0E5H bytes in the empty sectors, thus making such disk appear to have no files contained thereon. If the byte value is non 0E5H, then the slot contains a valid file extent descriptor. The CP/M user number area to which an active file is associated is stored in the first directory entry byte. User number values range from 0 to 15.

The next eight bytes contain the primary name of the file in ASCII characters. If the name is shorter than 8 characters then the name is padded to the right with spaces. Following the name field is a three byte file name extension field in ASCII characters. The extension field, if shorter than 3 characters is padded to the right with spaces. For CP/M version 2.2, the upper bits (bit 7) of the extent name bytes are used to describe certain attributes about the file. If the upper bit of the first extent name character is set, then the file is described as a read-only file. The upper bit of the second extent name character, if set, indicates that the file name should not be displayed in directory listings.

Each directory entry, as a file descriptor extent, has the next byte set to a number that specifies which 16K byte chunk of the file that this entry describes. Two bytes after the extent byte are not used within the directory and are normally set to zero by default. The number of records stored in the extent, described by this directory entry, is recorded in the byte 15 position. The maximum value for the record count is 128 (080H) which if equal to (128 * 128) or 16K bytes, the maximum size of an extent.

Byte positions 16 to 31 contain the group numbers upon the disk that contain the data belonging to the file named in the directory entry. The number of bytes within the total 16 available that are used for group number storage is dependant upon the amount of file data described by this extent and by the group size of the disk. The group numbers are single byte numbers, up to 16 total, if the number of groups upon the disk is less than or equal to 255. If the number of groups upon the disk is more than 255 then byte positions 16 to 31 contain two byte group numbers, stored in low byte/high byte order. The group numbers contained within a directory entry do not have to be in increasing sequential order nor do they have to be consecutive.

The figure below shows two logical records of the directory from a single sided double density disk with 2K byte groups. The total number of groups available is 243 so the group numbers are single byte numbers. Note that only one half of the 16 byte space for group numbers is used due to the fact that 8 entries for 2K

byte groups is all that is needed to describe the storage for one full 16K byte extent.

Figure 2. EXAMPLE HEX/ASCII DIRECTORY RECORD DISPLAY

```
00  00414449 52202020  20434F4D 0000000B   .ADIR    COM....
10  07000000 00000000  00000000 00000000   ................
20  004D4552 47505249  4E4F5652 0000003C   .MERGPRINOVR...<
30  16171819 00000000  00000000 00000000   ................
40  00434F50 59202020  20434F4D 0000000E   .COPY    COM....
50  0C000000 00000000  00000000 00000000   ................
60  00435243 4B202020  20434F4D 0000000A   .CRCK    COM....
70  0D000000 00000000  00000000 00000000   ................

00  E5555345 52202020  204C4F47 00000030   eUSER    LOG...0
10  04050600 00000000  00000000 00000000   ................
20  00444454 20202020  20434F4D 00000026   .DDT     COM...&
30  0F101100 00000000  00000000 00000000   ................
40  0044552D 56373520  20434F4D 0000002E   .DU-V75  COM....
50  12131400 00000000  00000000 00000000   ................
60  00464F52 4D415420  20434F4D 0000000C   .FORMAT  COM....
70  15000000 00000000  00000000 00000000   ................
```

The above examlpes all show files that are less than 16K bytes each. Note also the display showing the erased "USER.LOG" file.

HOW FILES ARE ACCESSED

The files upon a disk are accessed through a user description block called a File Control Block (FCB for short). The file control block, used by virtually all file access BDOS system calls, has the structure as shown in Figure 3. This chart is taken from a Digital Research CP/M manual and is included here for quick educational reference.

Note that the structure of a file control block is much the same as that of a directory entry with a few minor changes. The changes and/or differences are as follows, otherwise the byte descriptions are the same as for the disk directory entry.

The first byte of an FCB allows the programmer to specify which drive should be used for the file access. Drive A: to P: are specified as 1 to 16 respectively while a value of zero indicates that the currently logged default drive should be used for the access.

An FCB contains four additional bytes that are used as pointers for file access position. The "cr", current record

```
number,  indicates  the  sequential record number of this  extent
that  will  be  accessed upon the next file read  or  file  write
system  call.  The  user normally sets the "cr" byte to  zero  to
begin file access at the first logical record of the  file.  Each
time  a  read or write is performed the current record number  is
incremented.  When the "cr" byte attains a value of 080H during a
sequential  file operation the BDOS automatically  realizes  that
the  current  extent  of  the file has been  fully  accessed  and
performs  the necessary disk directory accesses to setup the  FCB
to allow file access to the next extent.  For reading this simply
means  that the next extent descriptor directory entry  from  the
disk,  for  this  file,  is  read  into  memory  (ie.  the  group
allocation  numbers from the disk are copied into the d0-dn bytes
of  the FCB,  the extent number becomes one greater,  the  record
count  from the disk for the new extent is copied into  the  "rc"
byte and the cr byte is zeroed).  During a writing operation  the
"cr"  byte  attaining a value of 080H indicates that the  current
extent  of the file is full and so the BDOS  automatically  finds
the  appropiate directory entry spot on the disk to write in  the
newly  assigned group allocation bytes,  record count  value  and
extent number.  The BDOS will then create another directory entry
on the disk for the new extent of the file.  In this case the d0-
dn  bytes of the FCB are zeroed to indicate that storage has  not
yet been allocated for this extent.
            Figure 3. FILE CONTROL BLOCK DESCRIPTION

   ---------------------------------------------------------------
   |dr|f1|f2|/ /|f8|t1|t2|t3|ex|s1|s2|rc|d0|/ /|dn|cr|r0|r1|r2|
   ---------------------------------------------------------------
    00 01 02 ... 08 09 10 11 12 13 14 15 16 ... 31 32 33 34 35

   where:

     dr        drive code (0 - 16)
               0 => use default drive for file access
               1 => select drive A: for file access
               2 => select drive B: for file access
               ...
               16=> select drive P: for file access

     f1...f8   contain the files name in ASCII upper case
               with high bits equal to zero.

     t1,t2,t3  contain the file type in ASCII upper case
               with high bits normally equal zero. tn' denotes
               the high bit of these bit positions.
               t1' = 1 => Read/Only file
               t2' = 1 => SYS file, no DIR list

     ex        contains the current extent number,
               normally set to 00 by the user, but is
```

```
               in the range 0 - 31 during file I/O.

     s1        reserved for internal system use

     s2        reserved for internal system use, set to
               zero on call to OPEN, MAKE, SEARCH system
               calls.

     rc        record count for extent "ex," takes on values
               0 to 128.

     d0...dn   filled-in by BDOS to indicate file group numbers
               for this extent.

     cr        current record to read or write in a sequential
               file operation. Normally set to zero by the user
               upon initial access to a file.

     r0,r1,r2  optional random record number in the range of 0 to
               65535, with overflow to r2. r0/r1 are a 16 bit value
               in low/high byte order.
     The  last three bytes of the FCB,  r0,r1,  & r2 are used for
random record file I/O and will be covered in the third and final
part of this turorial. For simpler sequential I/O the FCB in fact
does  not even need to be setup for the 36 bytes of  storage.  33
bytes suffice for all sequential file I/O FCB operations.


FILE ACCESS SETUP SYSTEM FUNCTIONS

     The  procedure for the programmer to use in accessing a file
generally  starts in one of two ways.  The first  senario  starts
with,  "Lets  see if our file exists on the disk?" There are  two
BDOS  system calls related to the functions of searching the disk
directory for a file name match against the FCB specified by  the
user.  These operations allow for the programmer to find out if a
specific file name already exists upon the disk.  In addition  it
provides  a  mechanism to scan a directory to determine all  file
names  that exist in the directory.  The second  situation  comes
into being if the programmer is already aware of the file  status
with respect to "presence" on the disk or as the logical sequence
of events following the first senario. These latter functions are
used to work with specific files for opening,  closing, creating,
renaming and deleting.


SEARCH FIRST AND SEARCH NEXT: Functions 17 and 18.

     The search functions scan the directory for match of a  file
name  that compares with the user specified FCB pointed to by the
(DE) register pair.  The match is made on the basis of  comparing
```

the f1-f8, t1-t3, and ex bytes of the FCB to the corresponding bytes of the disk directory entries. Any FCB position that contains an ASCII question mark "?" (03FH) is specified as a "match any character" from the disk directory. The function calls return a value of 0FFH in the (A) register if no more matched directory entries can be found. The search functions cause the currently valid disk buffer address and the following 128 bytes to be filled with a copy of the directory record containing the matched entry, if one is found. The (A) register is returned with a 0 to 3 value to indicate which one of the four possible 32 byte chuncks of the directory record contain the matched entry.

Search first means to find the first occurrance of a matched entry to the FCB. The search next function scans the directory from the current search position instead of from the beginning. Note that it is not normally valid to perform the search next functon without first performing the search first function. Also it is not valid to perform other directory or file operations between the search first and search next functions.

The program example below shows a technique for reading all directory entries from the disk drive specified by the first FCB byte into a memory resident list. The list starts at the LIST label with the total matched file count stored in the FILECNT variable. The LISTPOS label stores the next available list load point during the directory scan operation. The search FCB uses the CP/M default FCB location at address 05CH and specifies a total wild card (*.*) match. The "ex" byte is zeroed before the search first call so that only the zero extents of the files are returned. The file names are stored in the list in character strings of 16 bytes each with a preceeding drive designator byte and padded to the right with 4 zero bytes. Please note that this program is a segment only and will not directly assemble and run as a CP/M .COM file without a little added lead in and error exit coding.

Listing 1. A DIRECTORY SCANNING PROGRAM

```
BUFR    EQU    80H+BASE    ;DEFAULT CP/M BUFFER
BDOS    EQU     0005H           ;ENTRY POINT FOR BDOS OPERATIONS
;
SRCHF   EQU    17        ;SEARCH DIR FOR FIRST OCCUR.
SRCHN   EQU    18        ;SEARCH DIR FOR NEXT OCCUR.
STDMA   EQU    26        ;SET DMA ADDRESS
;
FCB EQU    5CH+BASE    ;DEFAULT FILE CONTROL BLOCK
FCBEXT  EQU    FCB+12          ;EXTENT BYTE IN FCB
FCBRNO  EQU    FCB+32        ;RECORD NUMBER IN FCB
;
;
;SETUP SIZE OF ELEMENTS IN THE FILE NAME LIST
```

```
;
ITEMSZ  EQU    16        ;EACH LIST ITEM IS 16 BYTES
;
;
;SETUP WILD CARD FILE IMAGE LIKE *.*
;
    LXI    H,FCB+1        ;PLACE TO PUT WILD CARD IMAGE
    MVI    B,11        ;SIZE TO SET
ALFN:
    MVI    M,'?'        ;PUT IN A JOKER CHAR
    INX    H        ;BUMP FILL POINTER
    DCR    B        ;DCR BYTE COUNTER
    JNZ    ALFN
;
;
;ZERO INITIAL TOTAL FILE COUNT
;
    LXI    H,0000H
    SHLD    FILECNT
;
;
;HERE IF NAME PROPERLY POSITIONED IN THE DEFAULT FCB AREA FOR LIST BUILD
;
NAMEPRES:
    MVI    C,STDMA        ;INITIALIZE DMA ADDRESS TO DEFAULT BUFFER
    LXI    D,BUFR
    CALL    BDOS
;
    XRA    A        ;CLEAR APPROPIATE FIELDS OF SEARCH FCB
    STA    FCBEXT        ;EXTENT BYTE
    STA    FCBRNO        ;AND RECORD NUMBER
;
    LXI    D,FCB        ;USE DEFAULT FCB FOR SEARCH
    MVI    C,SRCHF        ;SEARCH FOR FIRST OCCURRANCE
    CALL    BDOS
    CPI    0FFH        ;SEE IF FOUND
    JNZ    LOADLIST    ;IF SOME FOUND THEN GO BUILD LIST
;
;
;PUT INSTRUCTIONS HERE TO HANDLE A SITUATION WHERE NO FILES
;MATCHING THE FCB WILD CARD IMAGE ARE FOUND.
;
        JMP     ERROR$EXIT      ;TO USER SUPPLIED ROUTINE
;
;
;BUILD UP LIST WITH ALL FOUND ENTRIES
;
LOADLIST:
    LXI    H,LIST        ;INITIALIZE LIST POINTER PARAMETERS
    SHLD    LISTPOS        ;START = CURRENT POS OF LIST
;
```

```
;
;PUT CURRENTLY FOUND NAME TO LIST
;(A) = OFFSET IN DEFAULT BUFFER OF NAME
;
;
NM2LST:
    ANI    3        ;ZERO BASED TWO BIT INDEX
    ADD    A        ;TIMES 32 TO MAKE POSITION INDEX
    ADD    A
    ADD    A
    ADD    A
    ADD    A
    MOV    C,A        ;PUT IN BC
    XRA    B        ;CLEAR HIGH ORDER
    LXI    H,BUFR        ;TO NAME POSITION IN DEFAULT BUFFER
    DAD    B           ;(HL) = CURRENT FOUND NAME POINTER
    LDA    FCB        ;PUT DISK DRIVE NUMBER INTO NAME PLACE
    MOV    M,A        ;INTO BUFFER
    XCHG
    LHLD    LISTPOS        ;POINTER TO CURRENT LOAD POINT IN LIST
    XCHG
    MVI    B,12        ;MOVE DRIVE DESIGNATOR AND NAME TO LIST
MOVLP:
    MOV    A,M        ;GET NAME BYTE FROM DEFAULT BUFFER
    STAX    D        ;PLACE INTO LIST
    INX    H        ;BUMP POINTERS
    INX    D
    DCR    B        ;CHECK MOVE BYTE COUNT
    JNZ    MOVLP
    XCHG           ;(DE) WAS LEFT WITH LEXT LOAD POINT ADDRESS
;
    MVI    B,ITEMSZ-12    ;REMAINING LIST ITEM SPACES TO ZERO OUT
FILZRO:
    MVI    M,00H        ;PUT IN A ZERO BYTE
    INX    H
    DCR    B        ;ALL REST FILLED YET
    JNZ    FILZRO
;
    SHLD    LISTPOS        ;KEEP NEXT LOAD POINT IN SAFE PLACE
    LHLD    FILECNT        ;INCREASE FILE COUNT FOR EACH FILE
    INX    H
    SHLD    FILECNT
;
;
;SEARCH FOR NEXT OCCURANCE OF SPECIFIED FILE NAME
;
    MVI    C,SRCHN        ;SEARCH NEXT FUNCTION CODE
    LXI    D,FCB        ;FILE NAME SPECIFICATION FIELD
    CALL    BDOS
    CPI    0FFH        ;SEE IF ALL THROUGH DIRECTORY YET
    JNZ    NM2LST        ;IF NOT GO PUT NAME INTO LIST
```

```
;
;
;PROGRAM EXECUTION TO HERE IF THE LIST CONTAINS SOME FILE NAMES
;FROM THE DISKETTE
;
;USER DOES HIS OWN THING FROM HERE
;
;
;DIRECTORY NAME LIST FOR STORAGE OF INPUT NAMES
;
FILECNT:
    DS    2        ;COUNTER FOR NUMBER OF FILES

LISTPOS:
    DS     2         ;STORAGE FOR CURRENT LIST
                ;LOAD POINTER
;
LIST:
    DS    1        ;START POINT FOR FILE NAME LIST
;
;+++...END OF LISTING 1.
```

OPEN FILE: Function 15.

An existing file on a disk may not be read until the user FCB contains the information about where the file is stored upon the diskette. Function 15 provides a means where the user fills in the file name and then calls the operating system to get the d1-dn bytes of the FCB filled in. Once the file is OPEN then it may be read because subsequent calls to the BDOS to READ will "know where" the file is located. The OPEN function returns a value of 0FFH if the file cannot be found, otherwise the (A) register contains a value of 0 to 3 to indicate that the file was successfully opened. To open a file the programming procedure is simply:

```
;
;OPEN FILE EXAMPLE
;
OPEN      EQU       15        ;OPEN FUNCTION CODE
BDOS      EQU       0005H     ;SYSTEM ENTRY

          ORG       0100H     ;START
          LXI       D,FCB     ;POINT AT FILE CONTROL BLOCK
          MVI       C,OPEN    ;FUNCTION
          CALL      BDOS
          CPI       0FFH      ;CHECK IF NOT FOUND
          JZ        ERROR
          RET                 ;IF OPEN GO TO CCP
;
ERROR:
```

```
          MVI       C,9       ;PRINT ERROR MESSAGE
          LXI       D,ERRMS
          CALL      BDOS
          RET
;
ERRMS:
          DB        'FILE NOT FOUND','$'
;
;
;FILE ACCESS FILE CONTROL BLOCK
;
FCB:
          DB        00H       ;SET TO USE DEFAULT DRIVE
          DB        'TEST    DAT',0,0,0,0
          DS        16        ;STORAGE FOR D1 TO DN BYTES
          DB        0         ;CURRENT RECORD BYTE
;
          END
```

CLOSE FILE: Function 16.

Whenever a file is accessed for writing new space is allocated for that file on the disk. This implies that the user FCB contains disk group numbers that are not stored upon the diskette in the directory entry for the file. Function 16 provides a means where the user completes the file writing operation and then calls the operating system to set the directory entry group allocation bytes, the rc byte and the extent byte from the corresponding bytes of the FCB. A file that has been opened for reading only need not be closed because there is no change in the stored disk directory information. The CLOSE function returns a value of 0FFH if the file cannot be found, otherwise the (A) register contains a value of 0 to 3 to indicate that the file was successfully closed. To close a file the programming procedure is simply:

```
;
;CLOSE FILE EXAMPLE
;
CLOSE     EQU       16        ;CLOSE FUNCTION CODE
BDOS      EQU       0005H     ;SYSTEM ENTRY

          ORG       0100H     ;START
          LXI       D,FCB     ;POINT AT FILE CONTROL BLOCK
          MVI       C,CLOSE   ;FUNCTION
          CALL      BDOS
          CPI       0FFH      ;CHECK IF NOT FOUND
          JZ        ERROR
          RET                 ;IF CLOSED GO TO CCP
;
ERROR:
```

```
          MVI       C,9       ;PRINT ERROR MESSAGE
          LXI       D,ERRMS
          CALL      BDOS
          RET
;
ERRMS:
          DB        'FILE NOT FOUND','$'
;
;
;FILE ACCESS FILE CONTROL BLOCK
;
FCB:
          DB        00H       ;SET TO USE DEFAULT DRIVE
          DB        'TEST    DAT',0,0,0,0
          DS        16        ;STORAGE FOR D1 TO DN BYTES
          DB        0         ;CURRENT RECORD BYTE
;
          END
```

DELETE FILE: Function 19.

Often time the programmer will create and write files which will subsequently not be needed. The file or files may be deleted through use of function 19. The user sets an FCB to the appropiate file name in the f1-f8, and t1-t3 bytes. The BDOS function then removes the specified file from the directory of the appropiate disk. The user specified file name in the FCB may contain ASCII question marks in which case the delete function may delete multiple files if the file name matches more than one file on the disk with the name. The "?" matches any character at the position of its occurrance in the name. The DELETE function returns a value of 0FFH if the file(s) cannot be found, otherwise the (A) register contains a value of 0 to 3 to indicate that the file was successfully deleted. To delete a file the programming procedure is simply:

```
;
;DELETE FILE EXAMPLE
;
DELETE    EQU       19        ;CLOSE FUNCTION CODE
BDOS      EQU       0005H     ;SYSTEM ENTRY

          ORG       0100H     ;START
          LXI       D,FCB     ;POINT AT FILE CONTROL BLOCK
          MVI       C,DELETE  ;FUNCTION
          CALL      BDOS
          CPI       0FFH      ;CHECK IF NOT FOUND
          JZ        ERROR
          RET                 ;IF CLOSED GO TO CCP
;
```

```
ERROR:
          MVI       C,9       ;PRINT ERROR MESSAGE
          LXI       D,ERRMS
          CALL      BDOS
          RET
;
ERRMS:
          DB        'FILE NOT FOUND','$'
;
;
;FILE ACCESS FILE CONTROL BLOCK
;
FCB:
          DB        00H       ;SET TO USE DEFAULT DRIVE
          DB        'TEST    DAT',0,0,0,0
          DS        16        ;STORAGE FOR D1 TO DN BYTES
          DB        0         ;CURRENT RECORD BYTE
;
          END
```

CREATE FILE: Function 22.

Whenever a new file is desired it must first be created so that there is a spot in the directory to later save the file allocation information (see close function above). The BDOS assumes that the programmer has specified a file name that does not exist upon the disk. If there is a chance that a new file is desired that may duplicate the name of one already upon the disk the peviously described delete function should be used to erase the old file before creating the new file. Otherwise the directory may contain two files by the same name. The CREATE function returns a value of 0FFH if there is no room in the directory to store the freshly created directory entry, otherwise the (A) register contains a value of 0 to 3 to indicate that the file was successfully created. A newly created file may be immediately written since the BDOS prepares the user FCB to look like an empty file. To create a file the programming procedure is simply:

```
;
;CREATE FILE EXAMPLE
;
CREATE    EQU       22        ;CREATE FUNCTION CODE
BDOS      EQU       0005H     ;SYSTEM ENTRY

          ORG       0100H     ;START
          LXI       D,FCB     ;POINT AT FILE CONTROL BLOCK
          MVI       C,CREATE  ;FUNCTION
          CALL      BDOS
```

```
          CPI       0FFH      ;CHECK IF DIRECTORY FULL
          JZ        ERROR
          RET                 ;IF CLOSED GO TO CCP
;
ERROR:
          MVI       C,9       ;PRINT ERROR MESSAGE
          LXI       D,ERRMS
          CALL      BDOS
          RET
;
ERRMS:
          DB        'DIRECTORY FULL','$'
;
;
;FILE ACCESS FILE CONTROL BLOCK
;
FCB:
          DB        00H       ;SET TO USE DEFAULT DRIVE
          DB        'TEST    DAT',0,0,0,0
          DS        16        ;STORAGE FOR D1 TO DN BYTES
          DB        0         ;CURRENT RECORD BYTE
;
          END
```

RENAME FILE: Function 23.

Sometimes it is necessary to change the name of a disk file from that already existing in the disk directory. With function 23 the user specifies the name of an existing file on the disk with a standard FCB format except that on calling the BDOS the d1-dn byte area of the FCB are set to the new name desired for the file. All occurrances of the existing file name (ie. all extents) are changed to match the new name. The drive select byte specifies the drive upon which the rename operation should be done. The first byte of the second 16 bytes of the FCB (d0) is expected to be zero. The RENAME function returns a value of 0FFH if the old name file could not be found, otherwise the (A) register contains a value of 0 to 3 to indicate that the file was successfully renamed. To rename a file the programming procedure is simply:

```
;
;RENAME FILE EXAMPLE
;
RENAME    EQU       23        ;RENAME FUNCTION CODE
BDOS      EQU       0005H     ;SYSTEM ENTRY

          ORG       0100H     ;START
          LXI       D,FCB     ;POINT AT FILE CONTROL BLOCK
          MVI       C,RENAME  ;FUNCTION
```

```
          CALL      BDOS
          CPI       0FFH      ;CHECK IF DIRECTORY FULL
          JZ        ERROR
          RET                 ;IF CLOSED GO TO CCP
;
ERROR:
          MVI       C,9       ;PRINT ERROR MESSAGE
          LXI       D,ERRMS
          CALL      BDOS
          RET
;
ERRMS:
          DB        'FILE NOT FOUND','$'
;
;
;FILE ACCESS FILE CONTROL BLOCK
;
FCB:
          DB        00H       ;SET TO USE DEFAULT DRIVE
          DB        'TEST    DAT',0,0,0,0     ;OLD NAME
          DB        00H       ;BYTE ASSUMED TO BE ZERO
          DB        'NEWNAME DAT',0,0,0,0     ;NEW NAME
          DB        0         ;CURRENT RECORD BYTE
;
          END
```

ACCESSING FILE DATA

The previous section showed the reader how to find and setup files for subsequent I/O. Other file/directory handling functions were also presented. This has all led up to the big moment when the users program is finally ready to read or write data from/to a disk file. So here it is at last...

CP/M disk file data is moved between the disk and memory in blocks of 128 bytes called logical records or "sectors" in older fashioned CP/M lingo. Two functions to be presented here are included in the CP/M BDOS function code to allow sequential access to blocks of data in a file. The READ function starts at the beginning of a file and reads data blocks till the end of the file. The opposing WRITE operation moves data blocks to a new disk file and writes till the end of the users data when the file is closed (or the disk is full if the programmer has too much data). The BDOS includes one other function that allows the user to specify the area in his program where the 128 byte disk record buffer is to be located. These three functions will each be individually described below.

SET DISK BUFFER ADDRESS: Function 26.

The 128 byte data buffer that is to be used by the BDOS for file I/O is based at an address commonly referred to as the "DMA ADDRESS". This address or "buffer pointer" is passed to the BDOS in the (DE) registers when performing function 26. The program below simply sets the buffer address to "DATBF", a storage area after the end of the short program.

```
;
;SET BUFFER ADDRESS EXAMPLE
;
STDMA     EQU       26        ;SET BUFFER ADDRESS FUNCTION CODE
BDOS      EQU       0005H     ;SYSTEM ENTRY

          ORG       0100H     ;START
          LXI       D,DATBF   ;POINT AT DATA BUFFER
          MVI       C,STDMA   ;FUNCTION
          CALL      BDOS
          RET                 ;BACK TO CCP
;
DATBF:
          DS        128       ;SETUP 128 BYTE BUFFER
;
          END
```

READ AND WRITE DISK RECORDS: Functions 20 and 21.

The disk read and write functions are very similar in operation in that both move 128 bytes of data to/from the users program. The READ assumes entry with (DE) pointing to an active FCB setup by the open file function. The read sequential function reads the 128 byte record specified by the "cr" field of the FCB into the buffer pointer to by the current disk buffer address. After each READ operation the "cr" field is incremented to the next record number. If the "cr" field overflows past the end of the extent without encountering the end of the file then the BDOS automatically opens the next extent in preparation for the next read operation. The READ function returns a 00H code in the (A) register if the READ was performed successfully. If the end of file is encountered a non zero value is returned in (A).

The WRITE function assumes, on entry to the BDOS, that the (DE) registers point at a validly opened of created FCB. The WRITE will move 128 bytes of data from the buffer specified by the current disk buffer address to the disk. The written record is placed at the "cr" record position of the extent. As each record is written the "cr" field is incremented in preparation for the next write operation. Similar to the READ, if the "cr" field overflows past the end of the current extent, the BDOS automatically closes the current extent and creates a new extent in preparation for the next write operation. The WRITE command

```
may  be  performed on an existing file.  If  the  file  currently
contains  data at the "cr" record then the WRITE will overlay the
current  data with the new 128 byte record.  The  WRITE  function
returns  a  00H  value in the (A) register if  the  operation  is
successful.  A  non-zero value is returned if the write  function
was unsuccessful due to a full disk or directory.

     The small program below is designed to read the first record
of  a  file  'TEST.DAT',   and  write  it  into  the  small  file
'ONEREC.DAT'. The program should be reasonably self documenting.



;
;READ AND WRITE FUNCTION EXAMPLES
;
READ      EQU       20        ;READ FUNCTION CODE
WRITE     EQU       21        ;WRITE FUNCTION CODE
OPEN      EQU       15        ;OPEN FUNCTION CODE
CLOSE     EQU       16        ;CLOSE FUNCTION CODE
DELETE    EQU       19        ;DELETE FUNCTION CODE
CREATE    EQU       22        ;CREATE NEW FILE
STDMA     EQU       26        ;SET DISK BUFFER ADDRESS
BDOS      EQU       0005H     ;SYSTEM ENTRY

          ORG       0100H     ;START
          LXI       D,DATBF   ;POINT AT DATA BUFFER
          MVI       C,STDMA   ;FUNCTION
          CALL      BDOS
;
          LXI       D,FCBIN   ;POINT AT AND OPEN INPUT FILE
          MVI       C,OPEN
          CALL      BDOS
          CPI       0FFH      ;CHECK FOR OPEN ERROR
          JZ        ERROR
;
          LXI       D,FCBOUT  ;DEFAULT DELETE OF NEW FILE
          MVI       C,DELETE  ;..IN CASE IT EXISTS ALREADY
          CALL      BDOS
          LXI       D,FCBOUT  ;POINT AT FILE CONTROL BLOCK
          MVI       C,CREATE  ;FUNCTION TO MAKE NEW FILE
          CALL      BDOS
          CPI       0FFH      ;CHECK IF DIRECTORY FULL
          JZ        ERROR
          XRA       A         ;CLEAR THE INPUT CR FIELD TO READ
          STA       INCR      ;..FIRST RECORD
          LXI       D,FCBIN   ;READ FIRST FILE
          MVI       C,READ
          CALL      BDOS
          ORA       A         ;CHECK IF READ WAS O.K.
          JNZ       ERROR
          LXI       D,FCBOUT  ;WRITE TO OUTPUT FILE
```

```
          MVI       C,WRITE
          CALL      BDOS
          ORA       A         ;CHECK THAT DISK WASNT FULL
          JNZ       ERROR
;
          LXI       D,FCBOUT  ;CLOSE THE OUTPUT FILE
          MVI       C,CLOSE
          CALL      BDOS
          CPI       0FFH      ;CHECK CLOSE STATUS
          RNZ                 ;BACK TO CCP IF NO ERROR
;
ERROR:
          MVI       C,9       ;PRINT ERROR MESSAGE
          LXI       D,ERRMS
          CALL      BDOS
          RET
;
ERRMS:
          DB        'PROGRAM FILE ERROR','$'
;
;
;FILE ACCESS FILE CONTROL BLOCKS
;
FCBIN:
          DB        00H       ;SET TO USE DEFAULT DRIVE
          DB        'TEST    DAT',0,0,0,0
          DS        16        ;STORAGE FOR D1 TO DN BYTES
INCR:
          DB        0         ;CURRENT RECORD BYTE
;
FCBOUT:
          DB        00H       ;SET TO USE DEFAULT DRIVE
          DB        'ONEREC  DAT',0,0,0,0
          DS        16        ;STORAGE FOR D1 TO DN BYTES
          DB        0         ;CURRENT RECORD BYTE
;
DATBF:
          DS        128       ;SETUP 128 BYTE BUFFER
;
          END
```

SEQUENTIAL FILE I/O PROGRAMMING EXAMPLE

The assembly language code of Listing 2 presents a comprehensive set of I/O routines that allow either an input or output sequential file to be processed on a byte by byte basis. The routines perform all necessary sector buffering. The reader is encouraged to fully study the code and gain an understanding of how it all works. The program uses most of the BDOS functions

presented in this turorial.

Listing 2. CHARACTER BY CHARACTER DISK I/O ROUTINES

```
;*****************************************************************
;
;   DEMONSTRATION SEQUENTIAL CP/M FILE CHARACTER BY
;   CHARACTER I/O ROUTINES. NOTE THAT THE MAIN BODY
;   OF THIS PROGRAM IS NOT DESIGNED TO RUN AS IS IN
;   ANY NORMAL MANNER.
;
;   MANY THANKS ARE DUE TO WARD CHRISTENSEN WHO PREPARED THE
;   ORIGINAL SET OF SIMILAR I/O ROUTINES BURIED INSIDE OF
;   THE CP/M USERS GROUP MODEM PROGRAM THAT HAS BECOME SO
;   VERY POPULAR. THANKS AGAIN WARD.
;
;*****************************************************************
;
;
;CP/M BDOS EQUATES
;
RDCON   EQU    1
WRCON   EQU    2
PRINT   EQU    9
OPEN    EQU    15        ;OPEN FILE
CLOSE   EQU    16        ;CLOSE FILE
SRCHF   EQU    17        ;SEARCH FOR FIRST
ERASE   EQU    19        ;DELETE FILE
READ    EQU    20        ;READ FILE RECORD
WRITE   EQU    21        ;WRITE FILE RECORD
MAKE    EQU    22        ;CREATE NEW FILE
STDMA   EQU    26        ;SET DATA BUFFER POINTER
BDOS    EQU    0005H        ;SYSTEM I/O ENTRY POINT
FCB EQU    5CH        ;SYSTEM FCB
FCBEXT  EQU    FCB+12        ;FILE EXTENT
FCBSNO  EQU    FCB+32        ;SECTOR #
FCB2    EQU    6CH        ;SECOND FCB
DSKBUF  EQU    080H        ;DEFAULT DISK BUFFER ADDRESS
SECSIZ  EQU    080H        ;CP/M SECTOR SIZE
;
WBOOT   EQU    00        ;CP/M WARM BOOT ENTRY ADDRESS
;
;
;DEFINE ASCII CHARACTERS USED
;
LF  EQU    10        ;LINEFEED
CR  EQU    13        ;CARRIAGE RETURN
EOFCHR  EQU    01AH        ;CP/M END OF FILE CHAR
```

```
;
;
;START OF EXECUTABLE CODE
;
    ORG    100H
    LXI    SP,STACK    ;SETUP A STACK TO USE
;
;
;SEQUENTIAL I/O WRITE OF CP/M FILE ENABLED BY USING THIS SEQUENCE
;OF SUBROUTINE CALLS. THE FILE CONTROL BLOCK IS ASSUMED TO BE
;STORED AT THE DEFAULT LOCATION AT 05CH IN THE BASE PAGE OF
;CP/M MEMORY MAP.
;
SIOWR:
    CALL    ERASFIL        ;ERASE RECIEVED FILE
    CALL    MAKEFIL        ;ESTABLISH NEW FILE
    CALL    INITWR        ;INITIALIZE FILE WRITE PARAMETERS
;
;
;MAKE FOLLOWING CALL TO PLACE A CHARACTER FROM THE (A) REGISTER
;INTO THE CP/M FILE. LOOP DOING THIS TILL YOU HAVE ALL IN FILE THAT
;IS NEEDED.
;
    CALL    WRCHAR        ;PUT CHAR IN FILE
;
    CALL    WREOF        ;FLUSH LAST SECTOR TO CP/M FILE
    CALL    CLOSFIL        ;CLOSE IT UP
;
;
;SEQUENCE OF COMMAND CALLS TO OPEN AND USE A SEQUENTIAL CHARACTER
;FILE FOR READING. THE FILE CONTROL BLOCK IS ASSUMED TO BE LOCATED
;AT THE DEFAUT LOCATION OF 05CH IN THE BASE CP/M PAGE.
;ONCE THE FILE IS INITIALIZED THE CHARACTERS CAN BE READ ONE BY
;ONE UNTIL THE RDCHAR SUBROUTINE RETURNS A SET CARRY FLAG
;INDICATING A END OF PHYSICAL FILE CONDITION. EOF IS SENSED AS
;PHYSICAL END OR 01AH CHARACTER WHICHEVER COMES FIRST
;
SIORD:
    CALL    OPENFIL        ;OPEN THE CP/M FILE
    CALL    INITRD        ;GO INIT FOR FILE READ
    CALL    RDCHAR        ;GET CHAR FROM CP/M FILE
    JC    EOF        ;CHECK FOR EOF
;
EOF:
;   PLACE CODE HERE FOR END OF FILE HANDLING
;
;I/O HANDLING SUBROUTINES
;
;
;
;>--> ERASFIL: ERASE THE INCOMING FILE.
```

```
;
;IF IT EXISTS, ASK IF IT MAY BE ERASED.
;
ERASFIL:
    LXI    D,FCB        ;POINT TO CTL BLOCK
    MVI    C,SRCHF     ;SEE IF IT..
    CALL    BDOS        ;..EXISTS
    INR    A        ;FOUND?
    RZ            ;..NO, RETURN
    CALL    ILPRT        ;PRINT:
    DB    '++CP/M FILE EXISTS, TYPE Y TO ERASE: ',0
    CALL    KEYIN        ;GET A CHARACTER FROM CONSOLE
    ANI    5FH        ;MAKE UPPER CASE
    CPI    'Y'        ;WANT ERASED?
    JNZ    EXIT        ;QUIT IF NOT ERASE
    CALL    CRLF        ;BACK TO START OF LINE
;
;
;ERASE OLD FILE
;
    LXI    D,FCB        ;POINT TO FCB
    MVI    C,ERASE        ;GET BDOS FNC
    CALL    BDOS        ;DO THE ERASE
    RET            ;FROM "ERASFIL"
;
;
;>--> MAKEFIL: MAKES THE FILE TO BE RECEIVED
;
MAKEFIL:
    LXI    D,FCB        ;POINT TO FCB
    MVI    C,MAKE        ;GET BDOS FNC
    CALL    BDOS        ;TO THE MAKE
    INR    A        ;FF=BAD?
    RNZ            ;OPEN OK
;
;
;DIRECTORY FULL - CAN'T MAKE FILE
;
    CALL    ERXIT
    DB    '++ERROR - CANNOT MAKE FILE',CR,LF
    DB    '++DIRECTORY MUST BE FULL',CR,LF,'$'
;
;
;>--> OPENFIL: OPENS THE FILE TO BE SENT
;
OPENFIL:
    LXI    D,FCB        ;POINT TO FILE
    MVI    C,OPEN        ;GET FUNCTION
    CALL    BDOS        ;OPEN IT
    INR    A        ;OPEN OK?
    RNZ            ;FILE OPENED OK
```

```
    CALL    ERXIT        ;..NO, ABORT
    DB    '++CANNOT OPEN CP/M FILE','$'
;
;
;>--> CLOSFIL: CLOSES THE RECEIVED FILE
;
CLOSFIL:
    LXI    D,FCB        ;POINT TO FILE
    MVI    C,CLOSE        ;GET FUNCTION
    CALL    BDOS        ;CLOSE IT
    INR    A        ;CLOSE OK?
    RNZ            ;..YES, RETURN
    CALL    ERXIT        ;..NO, ABORT
    DB    '++CANNOT CLOSE CP/M FILE','$'
;
;
;>-->  INITRD: INITIALIZES FILE READ PARAMETERS
;
INITRD:
    MVI    A,00H        ;SET THE BUF CNT TO EMPTY
    STA    CHRINBF
    LXI    D,DSKBUF    ;SET THE DMA BUFFER POINTER
    PUSH    D
    MVI    C,STDMA
    CALL    BDOS
    POP    D
    XCHG            ;SET SECTOR POINTER
    SHLD    SECPTR
    RET
;
;
;>--> RDCHAR: READS A CHARACTER FROM FILE
;
;RETURN IS WITH DESIRED CHARACTER IN
;THE A REGISTER. IF EOF, THEN
;RETURN IS WITH THE CARRY FLAG SET.
;
RDCHAR:
    LDA    CHRINBF        ;GET NUMBER OF CHAR IN BUF
    ORA    A        ;CHECK IF BUFFER EMPTY
    JZ    RDBLOCK        ;GO GET A SECTOR IF EMPTY
    DCR    A        ;DECREMENT
    STA    CHRINBF
    LHLD    SECPTR        ;GET BUFFER POINTER
    MOV    A,M        ;GET CHARACTER FOR CALLER
    INX    H        ;INCREMENT POINTER
    SHLD    SECPTR
    CPI    EOFCHR        ;CHECK FOR LOGICAL CP/M EOF
    STC
    RZ            ;RETURN EXIT FOR LOGICAL EOF
    CMC            ;CLEAR CARRY SO EOF NOT INDICATED
```

```
                ;ON NORMAL RETURN
    RET             ;FROM "RDCHAR"
;
;
;BUFFER IS EMPTY - READ IN ANOTHER SECTOR
;
RDBLOCK:
    LXI    D,FCB
    MVI    C,READ
    CALL    BDOS
    ORA    A        ;READ OK?
    JZ    RDBFULL        ;YES
    DCR    A        ;EOF?
    JZ    REOF        ;GOT EOF
;
;
;READ ERROR
;
    CALL    ERXIT
    DB    '++CP/M FILE READ ERROR','$'
;
REOF:
    STC             ;SET CARRY FLAG FOR EOF EXIT
    RET
;
;
;BUFFER IS FULL
;
RDBFULL:
    MVI    A,SECSIZ    ;INIT BUF CHAR COUNT
    STA    CHRINBF
    LXI    H,DSKBUF    ;INIT BUFFER..
    SHLD    SECPTR        ;..POINTER
    JMP    RDCHAR        ;PASS CHAR TO CALLER
;
;
;>-->  INITWR: INITIALIZES FILE WRITE PARAMETERS
;
INITWR:
    MVI    A,00H        ;SET THE BUF CNT TO EMPTY
    STA    CHRINBF
    LXI    D,DSKBUF    ;SET THE DMA BUFFER POINTER
    PUSH    D
    MVI    C,STDMA
    CALL    BDOS
    POP    D
    XCHG            ;SET SECTOR POINTER
    SHLD    SECPTR
    RET
;
;
```

```
;>--> WRCHAR: WRITE A CHARACTER TO FILE
;
;ENTRY IS WITH CHARACTER IN A
;ENTRY AT WREOF FILLS REMAINING BYTES
;OF SECTOR WITH 01AH PER CP/M CONVENTION.
;
WRCHAR:
    LHLD    SECPTR        ;PUT CHAR IN BUFFER
    MOV    M,A
    INX    H        ;BUMP POINTER
    SHLD    SECPTR
    LDA    CHRINBF        ;INCR CHAR COUNT
    INR    A
    STA    CHRINBF
    CPI    SECSIZ        ;CHECK IF SECTOR FULL
    RNZ            ;GO BACK IF OK
;
WRBLOCK:
    LXI    D,FCB        ;IF FULL THEN WRITE
    MVI    C,WRITE        ;..THE..
    CALL    BDOS        ;..BLOCK
    ORA    A
    JNZ    WRERR        ;OOPS, ERROR
    MVI    A,00H        ;RESET THE CHAR CNT
    STA    CHRINBF
    LXI    H,DSKBUF    ;RESET BUFFER..
    SHLD    SECPTR        ;..POINTER
    RET
;
WRERR:
    CALL    ERXIT        ;EXIT W/MSG:
    DB    '++ERROR WRITING CP/M FILE',CR,LF,'$'
;
WREOF:
    LDA    CHRINBF        ;FILL REST OF SECTOR WITH 01AH
    LHLD    SECPTR
    MVI    B,EOFCHR
WREND:
    MOV    M,B        ;PUT IN THE CP/M EOF CODE
    INX    H
    INR    A        ;INC THE CHAR CNT
    CPI    SECSIZ        ;BUFFER FULL YET
    JNZ    WREND
    JMP    WRBLOCK        ;GO PUT FILLED BLOCK ON DISK
;
;
;>-->  KEYIN: GETS A KEY CODE IN FROM CONSOLE
;
KEYIN:
    PUSH    B        ;SAVE..
    PUSH    D        ;..ALL..
```

```
    PUSH    H        ;..REGS
    MVI    C,RDCON        ;GET CON CHAR FUNCTION CODE
    CALL    BDOS        ;GET CHARACTER
    MOV    A,E
    POP    H        ;RESTORE..
    POP    D        ;..ALL..
    POP    B        ;..REGS
    RET
;
;
;>--> CTYPE: TYPES VIA CP/M SO TABS ARE EXPANDED
;
CTYPE:
    PUSH    B        ;SAVE..
    PUSH    D        ;..ALL..
    PUSH    H        ;..REGS
    MOV    E,A        ;CHAR TO E
    MVI    C,WRCON        ;GET BDOS FNC
    CALL    BDOS        ;PRIN THE CHR
    POP    H        ;RESTORE..
    POP    D        ;..ALL..
    POP    B        ;..REGS
    RET            ;FROM "CTYPE"
;
;
;>--> CRLF: TYPE A CARRAGE RETURN LINE FEED PAIR AT CONSOLE
;
CRLF:
    MVI    A,CR
    CALL    CTYPE
    MVI    A,LF
    CALL    CTYPE
    RET
;
;
;>--> ILPRT: INLINE PRINT OF MSG
;
;THE CALL TO ILPRT IS FOLLOWED BY A MESSAGE,
;BINARY 0 AS THE END.  BINARY 1 MAY BE USED TO
;PAUSE (MESSAGE 'PRESS RETURN TO CONTINUE')
;
ILPRT:
    XTHL            ;SAVE HL, GET HL=MSG
ILPLP:
    MOV    A,M        ;GET CHAR
    ORA    A        ;END OF MSG?
    JZ    ILPRET        ;..YES, RETURN
    CPI    1        ;PAUSE?
    JZ    ILPAUSE        ;..YES
    CALL    CTYPE        ;TYPE THE CHARACTER OF MESSAGE
ILPNEXT:
```

```
    INX    H        ;TO NEXT CHAR
    JMP    ILPLP        ;LOOP
;
;
;PAUSE WHILE TYPING HELP SO INFO DOESN'T
;   SCROLL OFF OF VIDEO SCREENS
;
ILPAUSE:
    CALL    ILPRT        ;PRINT:
    DB    CR,LF,'PRESS RETURN TO CONTINUE OR ^C TO EXIT'
    DB    CR,LF,0
    CALL    KEYIN        ;GET ANY CHAR
    CPI    'C'-40H        ;REBOOT?
    JZ    EXIT        ;YES.
    JMP    ILPNEXT        ;LOOP
;
ILPRET:
    XTHL            ;RESTORE HL
    RET            ; & RETURN ADDR PAST MESSAGE
;
;
;>--> PRTMSG: PRINTS MSG POINTED TO BY (DE)
;
;A '$' IS THE ENDING DELIMITER FOR THE PRINT.
;NO REGISTERS SAVED.
;
PRTMSG:
    MVI    C,PRINT        ;GET BDOS FNC
    JMP    BDOS        ;PRINT MESSAGE, RETURN
;
;
;>--> ERXIT: EXIT PRINTING MSG FOLLOWING CALL
;
ERXIT:
    POP    D        ;GET MESSAGE
    CALL    PRTMSG        ;PRINT IT
;
EXIT:
    LXI    D,080H        ;RESET DEFAULT DMA ADDRESS FOR EXIT
    MVI    C,STDMA
    CALL    BDOS
    LHLD    STACK        ;GET ORIGINAL STACK
    SPHL            ;RESTORE IT
    JMP    WBOOT        ;GO DO A WARM BOOT OF CP/M TO BRING
                ;BACK IN CCP
;
;
;FOLLOWING 2 USED BY THE CP/M DISK BUFFERING ROUTINES
;
SECPTR  DW    DSKBUF        ;POINTER TO DISK BUFFER POS
CHRINBF DB    0        ;# OF CHARACTERS IN BUFFER
```

```
;
;
;SETUP A STACK AREA
;
    DS    38        ;STACK AREA
STACK   DS    2        ;STACK POINTER
;
;  --------------
;
    END
;
;+++...END OF LISTING 2
```

The reader is invited to be with us again next month when the tutorial continues into its third and final part. The functions of random record file I/O will be presented with complete programming examples to show how random I/O works. Several special file I/O tricks will be shown that permit unique problems to be solved under the CP/M operating system. One of these will be a program that does "update" on an exisiting file without the use of the random record I/O capabilities. So long till January and I hope that all Life Lines readers have a joyous holiday season.

PART III

SLIDING INTO BDOS (Part III)

UNDERSTANDING RANDOM FILES

by:

Michael J. Karas
2468 Hansen Court
Simi Valley, CA 93065
(805) 527-7922

The time has arrived to complete the third and final part of

this series on the operation of the CP/M BDOS as viewed from the assembly language programmers perspective. Presently we will build upon the extensive treatment of sequential files presented in Part II of the series to provide a basis for understanding the CP/M 2.2 random file I/O capability. Please note that functions of the BDOS presented here are specific to CP/M Versions 2.2 and 3.0. Older CP/M systems using Version 1.4 do not directly support random access file I/O and as such are not compatible with the programming examples presented below.

WHY RANDOM FILE I/O ANYWAY

In the beginning of the CP/M era, sometime around the release of Version 1.3 by Digital Research, small inexpensive single-user micro processor systems were typically used for simple-minded data processing applications. Most computing operations were linear with respect to the data handling by the CPU. Data entered from paper tape, cassette, card readers, or human entry from a keyboard tended to be limited to a sequential processing from start to finish. The usage of such data by the computer in data analysis, program compilation, or logging applications was also largely sequential. Finally the data output operations based upon the needs of hard copy, backup, and transmission from micro to micro were relegated to sequential processing applications.

Anticipated applications of micro type computer hardware by operating system designers, at that time, seemed to dictate that the disk file structures of the operating systems should be sequential in nature. This was true for the earliest releases of CP/M and Intel's ISIS II operating system. Other simple floppy disk operating systems like PERTEC's FDOS and MITS' Disk Extended Basic operating systems were also strictly sequential in the treatment of the disk file allocation and storage. However, these two systems permitted random record I/O within the bounds of an already existng file provided the space to store the records was previously pre-allocated as contiguous disk space in the file structure. The process of random I/O was then easy as a relative offset between the beginning record number for the file and the offset desired within the file.

As the micro processor applications market opened up in the late 1970's it seemed that new uses for computers were being found weekly. It has gotton to the point that micro processor computer users have a large array of very sophisticated software packages to choose from and utilize in their business and hobby activities. The main thing that can be pointed out about many of these packages is that the processes they perform are hardly linear with respect to the handling of data. Interactive programs like word processors, data base managers, spelling checkers, and

spread sheet analysis programs may very well need to be able to store or access data to/from a disk file in a manner that cannot be handled in the old sequential manner. The sequential philosophy generally limited file update to appending to the end of the file and read access to a particular record had to read N-1 records from the beginning of the file prior to being able to read record N.

Random access file I/O within an operating system anticipates the requirements of non-sequential I/O by permitting access to various records directly. Any record that was previously written may be read upon demand. Likewise the user/programmer may write any record desired. The Digital Research CP/M operating system supports this type of I/O in a powerful yet elegantly simple manner through a set of four BDOS system functions. These calls allow random access disk files to be implemented within the standard CP/M compatible file structure.

RANDOM FILE STRUCTURE UNDER CP/M 2.2

The structure of random files under the CP/M operating system is much the same as that for sequential files. Part II of this series (Lifelines, January 1982) described and illustrated the sequential structure in detail. The reader will recall that CP/M treats disk data in fixed records of 128 bytes. These records are collected together into "groups" that are stored on the disk as an allocated group. The disk space reserved for a given file, in its directory entry, is always marked, identified, and allocated in the even multiples of the "allocation group size".

I previously mentioned two older operating systems that supported random file I/O within the confines of a pre-allocated file. This system requires that all of the space for an "N" record file be reserved as contiguous disk space even if the file only contains two records (#0 and #N). Making a random access file bigger than the pre allocated size was virtually impossible. The CP/M Ver 2.2 random file access system has overcome the problems described above. A random file under CP/M contains only the number of allocated groups required to hold the stored records. The holes between the defined records do not consume unused disk space.

If a file under CP/M is created with only random record 0 of the file written then that file contains 128 bytes of real data and consumes one allocation group of disk space. The allocation group consumed also may contain other adjacent random records to fill out the size of the group. For instance, on single density 8" disks with a 1024 byte allocation group size, a one record (#0) file would be able to be written with additional record

numbers 1 to 7 within the same allocation group. Likewise if a single record file was created with only record number 9 written, that file would consume only one allocation group of disk space. Additional record numbers 8, and 10 to 15 could then be written without requiring additional disk space.

RANDOM FILE I/O SYSTEM CALLS

Let us next investigate the five BDOS system calls that CP/M supports for random I/O within files. The chart of Figure 1 on the following page details the look of a random access file control block. Note that the file control block contains three bytes at the end that are used to store the random record number that will currently be accessed. The random access system calls all utilize this field to determine the portion of the file to access at read/write time.

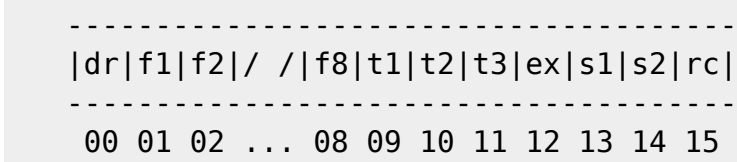
A CP/M random file may contain up to 64K records of 128 bytes numbered from 0 to 65535. Two bytes of the file control block hold this record number, r0 as the low byte and r1 as the high byte. This provides accessability to records up to a maximum file size of 8 megabytes. The r2 byte of the file control block is not used except as the overflow or carry out of the r1 byte. If byte r2 ever contains a value that is non-zero the record number is beyond the end of the 8 megabyte limit for the file.

To access a random file, it must first be opened in the normal manner with the "open" BDOS function call. In the case of creating a new random file the make file BDOS call is sufficient in that the the results of the make operation are equivalent to the open function on a zero length file.

READ RANDOM RECORD: Function 33.

This system call is made with the (DE) register pair pointing to a 36 byte file control block. Bytes r0-r2 are set up with the random record to read. The BDOS then fetches the addressed record from the file and places it in the callers record buffer pointed to by the last set buffer address function

Figure 1. FILE CONTROL BLOCK DESCRIPTION



where:

```
     dr        drive code (0 - 16)
               0 => use default drive for file access
               1 => select drive A: for file access
               2 => select drive B: for file access
               ...
               16=> select drive P: for file access

     f1...f8   contain the files name in ASCII upper case
               with high bits equal to zero.

     t1,t2,t3  contain the file type in ASCII upper case
               with high bits normally equal zero. tn' denotes
               the high bit of these bit positions.
               t1' = 1 => Read/Only file
               t2' = 1 => SYS file, no DIR list

     ex        contains the current extent number,
               normally set to 00 by the user, but is
               in the range 0 - 31 during file I/O.

     s1        reserved for internal system use

     s2        reserved for internal system use, set to
               zero on call to OPEN, MAKE, SEARCH system
               calls.

     rc        record count for extent "ex," takes on values
               0 to 128.

     d0...dn   filled in by BDOS to indicate file group numbers
               for this extent.

     cr        current record to read or write in a sequential
               file operation. Normally set to zero by the user
               upon initial access to a file.

     r0,r1,r2  optional random record number in the range of 0 to
               65535, with overflow to r2. r0/r1 are a 16 bit value
               in low/high byte order.
call.  The r0-r2 fields of the file control block are not changed
as  a  result of the random read function such that a  subsequent
random read operation would read the same record. The random read
function may return a number of error codes as described below:

     Error Code  00  - The random read  function  worked  without
          error and the user buffer contains the desired data.

     Error Code 01 - The random read operation addresses a record
          that  is  contained  in  a disk  allocation  group  not
          allocated to the file.  This means that the group field
          number slot of the appropriate extent of the file  that
```

```
          should contain the record is equal to 0.

     Error Code  03  - The random read operation  just  requested
          required  that a different extent descriptor  directory
          entry  had  to  be open for  the  impending  operation,
          however  prior  to opening the new extent  the  current
          extent could not be closed due to disk read/only status
          or a disk change.

     Error  Code  04 - The random read operation  just  requested
          required  access to an extent of the file that does not
          exist on the disk.

     Error Code  06  - The random read operation  just  requested
          required access to a record number beyond the bounds of
          the  disk  drive,  ie  the disk drive is  less  than  8
          megabytes  and  the  record  requested  is  within  an
          allocation group beyond the end of the disk.



WRITE RANDOM RECORD: Function 34.

     This  system  call  is  made with  the  (DE)  register  pair
pointing to a 36 byte file control block.  Bytes r0-r2 are set up
with the random record to write.  The BDOS then moves the data in
the  callers  record  buffer pointed to by the  last  set  buffer
address  function call to the addressed record in the  file.  The
r0-r2  fields  of  the file control block are not  changed  as  a
result of the random write function such that a subsequent random
write  operation  would write the same record.  The random  write
function may return a number of error codes as described below:


     Error Code  00  - The random write function  worked  without
          error and the user buffer contains the desired data.

     Error Code  03 - The random write operation  just  requested
          required  that a different extent descriptor  directory
          entry  had  to  be open for  the  impending  operation,
          however  prior  to opening the new extent  the  current
          extent could not be closed due to disk read/only status
          or a disk change.

     Error Code  05 - The random write operation  just  requested
          required  access to an extent of the file that does not
          exist on the disk.  In the process of creating the  new
          extent the disk directory was found to be full.

     Error Code  06  - The random write operation just  requested
          required access to a record number beyond the bounds of
```

the disk drive, ie the disk drive is less than 8 megabytes and the record requested is within an allocation group beyond the end of the disk.

WRITE RANDOM RECORD WITH ZERO FILL: Function 40.

This system call is made with the (DE) register pair pointing to a 36 byte file control block. Bytes r0-r2 are set up with the random record to write. The BDOS then moves the data in the callers record buffer, pointed to by the last set buffer address function call, to the addressed record in the file. The r0-r2 fields of the file control block are not changed as a result of the random write function such that a subsequent random file operation would access the same record. If the random write operation caused a new allocation group to be allocated to the file the other records of the same block are filled with zeros. The random write with zero fill function may return a number of error codes identical to those described for function number 34 above.

COMPUTE FILE SIZE: Function 35.

This system call determines the number of 128 byte records in a file and sets the number of records into the r0 and r1 bytes of the 36 byte file control block addressed by the (DE) register pair. The returned size is a virtual size in that if the file was created by random write operations and the file contains "holes" the file size function does not take the holes into account. Another way of looking at this is to think of this function as returning a record number that is one greater than the maximum record number currently in the file. If the file had no "holes" or it had been written in the conventional sequential fashion, then the file size reported by this function is the real file size. This function provides a convenient function of positioning a file at its end so that subsequent sequential or random update could be performed.

SET RANDOM RECORD: Function 36:

The (DE) register pair is set to point to a 36 byte file control block that has previously been used to reference a file in the sequential mode. Upon reference with this system call the r0 to r2 fields are filled in with the random record number that corresponds to the current file position, ie the BDOS simply computes the real current record number as follows:

```
     The current extent number is multiplied by 128,  the  number
     of  records  per extent,  and to this product is  added  the
     numerical  value  of the CR field,  current record  in  this
     extent.  The final result is placed into the r0-r2 fields of
     the FCB.



LOOKING AT SOME EXAMPLES


     The  following simple assembly language program is  designed
to  write record numbers 0 and 143 into a file on the  disk.  The
write  random function is used to write the first record with all
A's and the second record, # 143, with all B's.


;
;
;RANDOM RECORD I/O DEMONSTRATION FOR CP/M 2.2
;
;   THIS FIRST LEVEL DEMONSTRATION IS DESIGNED TO
;   SHOW HOW TO INITIALLY SET UP A FILE TO BE A RANDOM FILE
;   AND TO WRITE TWO RECORDS INTO THE FILE SUCH THAT THE
;   FIRST RECORD (RECORD NUMBER 0) AND THE SEVENTEENTH
;   RECORD OF THE SECOND EXTENT (RECORD NUMBER 143) BOTH
;   CONTAIN DATA. THE PURPOSE IS TO DEMONSTRATE THE
;   RESULTING DISK DIRECTORY ENTRIES THAT RESULT FROM
;   AN INCOMPLETE FILE. THIS DEMO PROGRAM DOES NO RANDOM
;   WRITE ERROR CHECKING.
;
;
;SYSTEM LEVEL INTERFACE EQUATES
;
BDOS    EQU    0005H        ;SYSTEM INTERFACE VECTOR
MAKE    EQU    22          ;MAKE NEW FILE FUNCTION
SBADDR  EQU    26          ;SET DISK BUFFER ADDR
OPEN    EQU    15          ;OPEN FILE FUNCTION
CLOSE   EQU    16          ;FILE CLOSE FUNCTION
DELETE  EQU    19          ;DELETE FILE FUNCTION
RRAND   EQU    33          ;READ RANDOM FUNCTION
WRAND   EQU    34          ;WRITE RANDOM FUNCTION
WRANDF  EQU    40          ;WRITE RANDOM WITH 00 FILL
;
;
    ORG    0100H         ;START OF A PROGRAM
;
    XRA    A           ;ZERO BYTES OF THE FCB
    STA    EXT           ;EXTENT FIELD
    STA    CR           ;CURRENT RECORD COUNT
    STA    RR+2          ;AND THE R2 FIELD
```

```
    LXI    H,0000H        ;ALSO ZERO RANDOM RECORD FIELED
    SHLD    RR
;
    LXI    D,BUFFER    ;SET DISK BUFFER ADDRESS
    MVI    C,SBADDR
    CALL    BDOS
;
    LXI    D,RANDFCB    ;POINT AT OUR FCB
    MVI    C,DELETE    ;ERASE TEST FILE IF IT ALREADY EXISTS
    CALL    BDOS
;
    LXI    D,RANDFCB    ;MAKE A NEW FILE FOR TEST
    MVI    C,MAKE
    CALL    BDOS
;
    MVI    A,'A'        ;FILL FIRST RECORD WITH A'S
    CALL    FILL        ;GO FILL
    LXI    H,0000H        ;SET RECORD NUMBER TO WRITE A'S INTO
    SHLD    RR
    LXI    D,RANDFCB    ;WRITE RECORD OF A'S
    MVI    C,WRAND        ;NORMAL WRITE RANDOM FUNCTION
    CALL    BDOS
;
    MVI    A,'B'        ;FILL NEXT RECORD WITH B'S
    CALL    FILL        ;GO FILL
    LXI    H,143        ;SET RECORD NUMBER TO WRITE B'S INTO
    SHLD    RR
    LXI    D,RANDFCB    ;WRITE RECORD OF B'S
    MVI    C,WRAND        ;NORMAL WRITE RANDOM FUNCTION
    CALL    BDOS
;
    LXI    D,RANDFCB    ;CLOSE JUST WRITTEN FILE
    MVI    C,CLOSE
    CALL    BDOS
;
;
    RET            ;BACK TO CCP BY IMMEDIATE RETURN
;
;
;SUBROUTINE TO FILL BUFFER WITH A PATTERN
;
;   ENTRY WITH (A) CONTAINING BYTE TO FILL BUFFER WITH
;
FILL:
    LXI    H,BUFFER    ;POINT AT BUFFER FOR FILL
    MVI    B,128        ;FILL BYTE COUNTER
FILLP:
    MOV    M,A        ;PUT A BYTE INTO BUFFER
    INX    H        ;BUMP POINTER
    DCR    B        ;DECREMRNT BYTE COUNT
    JNZ    FILLP        ;CONTINUE TILL BUFFER FULL
```

```
    RET
;
;
;RANDOM FILE TEST DATA AREA
;
RANDFCB:
    DB    00        ;USE CURRENT LOGGED DRIVE FOR TEST
    DB    'RANDFILE'    ;NAME OF FILE TO PLAY WITH
    DB    'TST'        ;..AND THE EXTENSION NAME
EXT:
    DB    00,00,00,00    ;EXTENT, S1, S2, AND FCBSZ BYTES
    DS    16        ;STORAGE FOR THE ALLOCATION NUMBERS
CR:
    DS    1        ;CURRENT RECORD BYTE
RR:
    DS    2        ;RANDOM RECORD NUMBER (R0,R1)
    DS    1        ;RANDOM RECORD OVERFLOW BYTE (R2)
;
;
;RANDOM DISK I/O DATA BUFFER
;
BUFFER:
    DS    128        ;ONE RECORD BUFFER
;
    END
```

The above program was assembled and caused to run on an empty single density disk in the default disk drive. The following display shows how the directory upon the disk looked after running the program. Notice that the file only consumes two allocated groups. Due to the fact that this was a single density disk with 1024 byte allocation groups of 8 records each, then if record number 8 was subsequently written the directory entries would change to include an allocation block number in the second group number slot of the first extent of the file.

```
G=00:00, T=2, S=1, PS=1

00  0052414E 4446494C  45545354 00000001  *.RANDFILETST....*
10  02000000 00000000  00000000 00000000  *................*
20  0052414E 4446494C  45545354 01000010  *.RANDFILETST....*
30  00030000 00000000  00000000 00000000  *................*
40  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
50  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
60  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
70  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
```

The following two sector displays off the single density

```
disk show the A's and B's written by the program above. All other
sectors  in the group numbers 02 and 03 were empty,  ie contained
whatever data that used to be there.  This brings up the  subject
of  the write random with zero fill function.  A small segment of

G=02:00, T=2, S=17, PS=20

00  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*
10  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*
20  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*
30  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*
40  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*
50  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*
60  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*
70  41414141 41414141  41414141 41414141  *AAAAAAAAAAAAAAAA*

G=03:07, T=3, S=6, PS=5

00  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*
10  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*
20  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*
30  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*
40  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*
50  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*
60  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*
70  42424242 42424242  42424242 42424242  *BBBBBBBBBBBBBBBB*



the  first demonstration program was changed to cause the  second
write operation to be done with zero fill. The changed portion of
the program is shown below:


    LXI    D,RANDFCB    ;WRITE RECORD OF A'S
    MVI    C,WRAND        ;NORMAL WRITE RANDOM FUNCTION
    CALL    BDOS
;
    MVI    A,'B'        ;FILL NEXT RECORD WITH B'S
    CALL    FILL        ;GO FILL
    LXI    H,143        ;SET RECORD NUMBER TO WRITE B'S INTO
    SHLD    RR
    LXI    D,RANDFCB    ;WRITE RECORD OF B'S
    MVI    C,WRANDF    ;WRITE RANDOM ZERO FILL FUNCTION
    CALL    BDOS
;
    LXI    D,RANDFCB    ;CLOSE JUST WRITTEN FILE


     Note  from  the  directory display below that  there  is  no
change  in the appearance of the entries from the first  example.
```

This time the only thing that changed was the data in allocation group 3. Due to the second write this allocation group contains a sector of B's at GROUP=03:07 with the other seven sectors of the group now containing zeroes from the zero fill operation. The function of zero fill is to leave a clean slate on records numbers subsequently read from the same allocation block. The BDOS is capable of reporting unwritten record information for records that correspond to group number slots in the directory entries that contain a '00' byte indicating unallocated. However once a group is allocated for one record the BDOS cannot determine if other sectors of that group have been written or not. Thus ero function may be issued when creating a random access file for the first time. The programmer may then use a record of 128 zeroes to indicate that the record is not used as opposed to accidentally mistaking the garbage data from un-initialized sectors written without zero fill as real data.

G=00:00, T=2, S=1, PS=1

```
00  0052414E 4446494C  45545354 00000001  *.RANDFILETST....*
10  02000000 00000000  00000000 00000000  *................*
20  0052414E 4446494C  45545354 01000010  *.RANDFILETST....*
30  00030000 00000000  00000000 00000000  *................*
40  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
50  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
60  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
70  E5E5E5E5 E5E5E5E5  E5E5E5E5 E5E5E5E5  *eeeeeeeeeeeeeeee*
```

The next example program is included here to show a clever means of implementing arbitrary record selection I/O within a file without resorting to random file I/O. The intent is not to indicate that the following scheme is the preferred method. The program below was developed with the CP/M Ver 1.4 operating system in mind. However the algorithm works fine with CP/M 2.2 as well. The technique used to play with random records by using sequential read and write operations is to manipulate the "cr" field of a standard 33 byte file control block. The "cr" byte is the only meand that the BDOS uses to indicate the next record to access. The programmer may change this byte value to force the BDOS to go to any record within the current extent.

If the first extent of a file is opened, the group allocation values for that extent lie in the file control block. If the technique of performing "your own" random I/O is done, the code must access record numbers not to excede 07fh without first closing the current extent and opening the next. This can be done with either the conventional open and close operations or the programmer, when done working with the current extent may open next automatically by performing a dummy read of record 080H of

the current extent. The programming example below uses the "roll your own" technique but does not anticipate a file size greater than 16K (one extent size).

The program below is a skeleton structure of a .COM file serialization procedure. The idea is to insert a six byte serial number string into the target file PROG.COM on drive B:. The serial number is inserted into the file at the places specified by the labels in the table at the start of the listing. These values are stripped out of the symbol table that is generated at the assembly of the PROG.ASM file. If the assembler does not generate a symbol table then the label values may be pulled off the .PRN listing output. The insert points are places within the "to be serialized" program where the programmer has determined that he would like to place the serial number string. Within the file itself, the labels point to the place where the string is to be inserted with respect to run time load address. The real file offset is 0100H bytes less. In addition, the scheme does not insert all six bytes of the program serial number at each location. The byte at each label address minus one contains a value between 1 and 6 of thenumber of serial number bytes that should actually be inserted at seralization time.

The list of label values in the program below is used to build, at assembly time, a table of record numbers where the specific serial number strings are to be inserted. This table is then used to fill in the "cr" byte of the file control block as each serial number is to be inserted. The table also contains the byte offset within the record where the insert point is to start. As each serial number is to be inserted the appropriate record is read, the number is inserted (with length specified by the value from the file record just accessed), and the record is written back to the disk. Sequentail read and write operations are used for both operations. Logic within the code listing below also provides for the occurrance that the serial number string may cross the end of the first record and flow into the next record. In this case the first is rewritten followed by reading of the next with the remainder of the insert proceeding from the beginning of the second record.

Please note that the program example is given as a skeleton only and the serial number entry process, increment process, and the disk I/O error exit points are left for the reader/programmer to fill in with code of his own choosing.

```
;
;
;PROGRAM SERIAL NUMBER INSERTION EQUATES
;   EACH ADDRESS IS A VALUE INSIDE OF THE "PROG.COM"
;   FILE THAT IS THE PLACE TO PUT THE SERIAL NUMBER.
```

```
;
SERA    EQU    0132H
SERB    EQU    01E9H
SERC    EQU    0278H
SERD    EQU    039AH
SERE    EQU    06FFH
SERF    EQU    0732H
SERG    EQU    0BBCH
SERH    EQU    0C08H
;
;
;CP/M BDOS SYSTEM CALLS FUNCTION NUMBERS
;
BOOT    EQU    0000H        ;REBOOT LOCATION ENTRY POINT
BDOS    EQU    0005H        ;BDOS FUNCTION ENTRY POINT
RESET   EQU    13         ;RESET DISK SYSTEM
OPEN    EQU    15         ;OPEN FILE FUNCTION
CLOSE   EQU    16         ;CLOSE FILE FUNCTION
DMAADR  EQU    26         ;SET DATA BUFFER ADDRESS
READ    EQU    20         ;READ SEQUENTIAL
WRITE   EQU    21         ;WRITE SEQUENTIAL
;
;
;DEFINE BASE EXECUTION AREA FOR THIS PROGRAM
;
START   EQU    0100H
;
;
    ORG    START        ;BASE OF EXECUTION AREA
;
;
;START UP HERE WITH PROGRAM INITIALIZATION AND
;DEFINE PROCEDURE TO FETCH IN SERIAL NUMBER TO INSERT INTO
;THE FILE
;
SERASK:

;
;ENTER APPROPIATE CODE HERE TO PUT A SIX BYTE SERIAL NUMBER
;INTO VARIABLE "SERSTR"
;

;
;
;SERIAL NUMBER INSERT POINT PROCESSING
;
;
SERCOPY:
    MVI    C,RESET        ;RESET DISK SYSTEM UPON INSERT
    CALL    BDOS
    LXI    D,PROGFCB    ;SET TO OPEN THE PROG.COM FILE
```

```
    MVI    C,OPEN
    CALL    BDOS
    INR    A        ;CHECK IF OPEN ERROR
    JNZ    SERCP1        ;OPEN SO GO START WRITE



;
;PRINT ERROR MESSAGE HERE AS TO INDICATE THAT THE FILE
;"PROG.COM" IS NOT PRESENT ON DRIVE B:.
;
    JMP    SERASK        ;IF ERROR BACK TO GET A NEW SERIAL
                ;..NUMBER OR TO EXIT

SERCP1:
    MVI    B,00H        ;INDEX COUNTER FOR TABLE VALUES
SERIST:
    MOV    L,B
    MVI    H,00H
    DAD    H        ;DOUBLE TO WORDS
    LXI    D,INSTAB    ;INTO TABLE
    DAD    D
    MOV    A,M        ;GET RECORD NUMBER FOR PLACE
    STA    PROGFCB+32    ;SET TO READ THIS RECORD
    INX    H
    MOV    C,M        ;GET BYTE LOCATION OF COUNTER
    PUSH    B
    LXI    D,PROGFCB    ;USE PROG FCB TO READ
    MVI    C,READ
    CALL    BDOS        ;GO READ SECTOR
    POP    B        ;INDEX TO LENGTH
    MOV    L,C
    MVI    H,0
    LXI    D,080H        ;BASE OF DEFAULT BUFFER
    DAD    D
    MOV    C,M        ;GET LENGTH
    INX    H        ;POINT TO NEXT BUFFER BYTE
    LXI    D,SERSTR    ;POINT (DE) TO SERIAL LOCATION
;
MOVLP:
    MOV    A,H        ;SEE IF PAST THE END OF BUFFER
    CPI    01H
    JNZ    SAMSEC        ;STILL IN THE SAME SECTOR
;
    MVI    H,0        ;RESET TO NEXT SECTOR BASE
    PUSH    B
    PUSH    H
    PUSH    D
    LXI    H,PROGFCB+32    ;DECREASE RECORD FOR WRITE
    DCR    M
    LXI    D,PROGFCB
    MVI    C,WRITE        ;WRITE LAST SECTOR
```

```
    CALL    BDOS
    LXI    D,PROGFCB
    MVI    C,READ        ;READ NEXT SECTOR
    CALL    BDOS
    POP    D
    POP    H
    POP    B
;
SAMSEC:
    PUSH    B
    LDAX    D        ;GET A SERIAL NUMBER BYTE
    MOV    M,A        ;AND SLAM INTO BUFFER
    POP    B
    INX    H
    INX    D
    DCR    C        ;DONE ALL BYTES HERE YET
    JNZ    MOVLP
;
    PUSH    B
    LXI    H,PROGFCB+32    ;SET BACK CURRENT RECORD FOR WRITE
    DCR    M
    LXI    D,PROGFCB
    MVI    C,WRITE        ;REWRITE THIS SECTOR
    CALL    BDOS
    POP    B
    INR    B        ;BUMP TABLE SCAN INDEX
    LDA    TABLEN        ;CHECK FOR DONE
    CMP    B
    JNC    SERIST        ;GO FOR NEXT TABLE ENTRY

;
;PUT IN LOGIC HERE TO SPECIFY THE NEXT OF SEQUENTIAL SERIAL NUMBERS
;OR TO GO BACK TO THE TOP OF THE PROGRAM TO GET A NEW SERIAL NUMBER.
;


;
;
;PARAMETER DATA AREA FOR SERAL NUMBER PROGRAM
;
;
;"PROG.COM" FILE ACCESS CONTROL BLOCK
;
PROGFCB:
    DB    'B'-040H    ;DISK DRIVE B: ALL THE TIME
    DB    'PROG    COM',0,0,0,0
    DS    17        ;ALLOCATION SPACE
;
;
;
;SERIAL NUMBER INSERTION POINT REFERENCE TABLE
;
```

```
INSTAB:
    DB    ((SERA-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERA-0100H-1) AND 07FH) ;BYTE OFFSET
    DB    ((SERB-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERB-0100H-1) AND 07FH) ;BYTE OFFSET
    DB    ((SERC-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERC-0100H-1) AND 07FH) ;BYTE OFFSET
    DB    ((SERD-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERD-0100H-1) AND 07FH) ;BYTE OFFSET
    DB    ((SERE-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERE-0100H-1) AND 07FH) ;BYTE OFFSET
    DB    ((SERF-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERF-0100H-1) AND 07FH) ;BYTE OFFSET
    DB    ((SERG-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERG-0100H-1) AND 07FH) ;BYTE OFFSET
    DB    ((SERH-0100H-1)/128)       ;RECORD NUMBER
    DB    ((SERH-0100H-1) AND 07FH) ;BYTE OFFSET
;
TABLEN:
    DB    (($-INSTAB)/2)-1    ;NUMBER OF TABLE ENTRIES
;                   ;..MINUS 1 FOR LOOP EASE
SERSTR:
    DS    10H        ;PLACE TO KEEP BINARY SERIAL NUMBER
;
;
    END
;
;
;...END OF SERIAL NUMBER INSERT PROGRAM
```

The next and final example is a fully functional program that uses random record I/O under CP/M 2.2 to perform a "useful" function. The program mixes up the records of a file in an ordered yet bizarre way in order that the file contents may be encoded to prevent its use until such time that it is unscrambled. The unmixing process is also performed by the program below. The records or "sectors" of the file are mixed and unmixed in place on the disk in that the disk file is not copied. Random access file I/O is used to swap records directly. The comment block at the beginning of the program listing contains an explanation of the program "intent" and the record mixing algorithm chosen. Operation of the program, should the reader wish to utilize the encoding and decoding functions provided, is also described in the listing.

This example program is presented as a working example of random file I/O in use. Detailed description of the internal workings of the program are beyond the scope of this tutorial but may be inferred by studying the listing and reading the rather

prolific comment statements. For readers that would like to avoid the aggravation of typing in the source code for the program below or for the other programs presented in this BDOS tutorial series, Part I in Lifelines, November 1982 and Part II in Lifelines, January 1983, a machine readable copy of the source code files on an eight inch single density diskette may be obtained from Michael J. Karas, 2468 Hansen Court, Simi Valley, California 93065. Please send diskettes preformatted, labeled and in a returnable mailer of some sort. Also include either stamps or money for return postage (no postage meter tapes, those are accepted on date of printing only) for your return package.

LISTING FOR SECRET.ASM A RANDOM I/O PROGRAM EXAMPLE

```
;
;
;RANDOM RECORD I/O DEMONSTRATION FOR CP/M 2.2
;
;   THIS THIRD LEVEL DEMONSTRATION PROGRAM IS DESIGNED TO
;   DEMONSTRATE RANDOM FILES BY DEVELOPING A 'NOT NECESSARILY
;   PRACTICAL' ALGORITHM FOR ENCODING A PROGRAM FILE ON A DISK.
;   THE INTENT IS TO MAKE THE TRANSMISSION OF AN OBJECT FILE
;   ARBITRARILY SCRAMBLED ON A 128 BYTE BY 128 BYTE RECORD BASIS
;   SUCH THAT IF THE TRANSMITTED FILE, EITHER ON FLOPPY DISKETTE
;   OR ON THE PHONE LINE WERE INTERCEPTED BY AN ILLICIT THIRD
;   PARTY, THEN THE THIRD PARTY WOULD RECEIVE GARBAGE UNLESS
;   HE HAD POSSESSION OF THE DECODING ALGORITHM.
;
;   THIS PROGRAM WILL IMPLEMENT SUCH AN ALGORITHM IN BOTH AN
;   ENCODING AND DECODING FORMAT. HERE IS THE ALGORITHM USED.
;   (OBVIOUSLY DUE TO THE FACT THAT THIS APPEARS IN THE
;   PUBLIC IMAGE AS A MAGAZINE ARTICLE WILL PREVENT THE FOLLOWING
;   ALGORITHM TO BE OF 'SECRET' USE).
;
;   THE OPERATOR ENTERS THE COMMAND TO RUN THE PROGRAM AS:
;
;       A>SECRET filename.typ E<cr>
;
;               where filename.typ is the
;               file to encode. And "E"
;               indicates to encode the file
;
;   or:
;
;       A>SECRET filename.typ D<cr>
;
;               where filename.typ is the
;               file to decode. And "D"
;               indicates to decode the file
;
;   THE ENCODING PROCESS WRITES THE ENCODED FILE RIGHT IN PLACE
```

```
;   WITHIN THE USER SPECIFIED FILE. NO MEANS IS USED TO SPECIFY
;   IN THE ENCODED FILE THAT IT IS ENCODED.
;
;   THE DECODE PROCESS READS AND DECODES THE FILE RIGHT IN PLACE
;   WITHIN THE USER SPECIFIED FILE NAME.
;
;   THE ALGORITHM LEAVES THE FIRST RECORD OF THE FILE INTACT AND
;   DOES NOT ENCODE THE PART OF A FILE BEYOND 128 RECORDS IN SIZE.
;   FOR FILES LARGER THAN 128 RECORDS THE FINAL RECORDS BEYOND THE
;   128'TH ARE LEFT UNTOUCHED. THE BDOS IS CALLED TO DETERMINE THE
;   SIZE OF THE FILE SO THE NUMBER OF RECORDS IN THE FILE ARE
;   KNOWN. THIS NUMBER OF RECORDS WILL BE REFERRED TO HERE AS "NR".
;   IF "NR" IS GREATER THAN 128 THEN "NR" IS SET TO 128. THEN THE
;   FIRST "NR-1" BYTES OF THE FIRST RECORD ARE READ SEQUENTIALLY
;   TO MAKE A LIST OF ONE BYTE BINARY NUMBERS WITH A NUMBER OF
;   ENTRIES EQUAL TO THE NUMBER OF RECORDS IN THE FILE MINUS ONE,
;   UP TO A MAXIMUM OF 127 NUMBERS.
;
;   THIS LIST IS THEN PROCESSED TO CONVERT ALL OF THE NUMBERS IN THE
;   LIST TO BE WITHIN THE RANGE OF 1 TO "NR-1". THIS CONVERSION IS
;   DONE BY FIRST "ANDING" EACH OF THE BYTES IN THE LIST WITH A MASK.
;   THE MASK HAS A NUMERICAL VALUE EQUAL TO "NR-1" ROUNDED UP TO
;   THE NEXT BIGGEST [(2 ^ N) - 1] VALUE, IE IF THE FILE HAS 5
;   RECORDS THE MASK IS 07H. IF THE FILE HAS 59 RECORDS THE MASK
;   HAS A VALUE OF 3FH. THE LIST IS THEN SCANNED FOR VALUES THAT
;   ARE GREATER THAN "NR-2". EACH VALUE THAT IS GREATER THAN
;   "NR-2" IS DIVIDED BY TWO IGNORING THE REMAINDER. FINALLY EACH
;   LIST VALUE IS INCREMENTED BY ONE TO MAKE A REAL FILE READABLE
;   RECORD NUMBER.
;
;   THE LIST IS THEN USED AS A RECORD SCRAMBLE/UNSCRAMBLE LIST.
;   FOR SCRAMBLING IT IS SCANNED FROM THE BEGINNING WHILE
;   UNSCRAMBLING SCANS THE LIST FROM THE END. SCRAMBLING PROCEDES
;   AS FOLLOWS (THE UNSCRAMBLE PROCESS IS THE REVERSE):
;
;       THE SECOND FILE RECORD IS NOW INTERCHANGED IN
;       POSITION WITH THE RECORD POINTED BY THE FIRST
;       NUMBER IN THE LIST. THE THIRD FILE RECORD IS
;       INTERCHANGED WITH THE RECORD POINTED TO BY THE
;       SECOND LIST VALUE. THIS PROCESS CONTINUES UNTIL
;       THE END OF THE LIST. DURING THE PROCESS OF
;       INTERCHANGING THE FILE SECTORS IN THIS RATHER
;       BIZARRE MANNER, EACH TIME A LIST VALUE IS FOUND
;       TO HAVE A LEAST SIGNIFICANT BIT THAT IS EQUAL
;       TO "1" THEN THAT RECORD HAS EACH BYTE XOR'ED
;       WITH THE RECORD NUMBER.
;
;                               WRITTEN BY:
;                                    MICHAEL J. KARAS
;                                    2468 HANSEN COURT
;                                    SIMI VALLEY, CA 93065
```

```
;                                   (805) 527-7922
;
;
;
;SYSTEM LEVEL INTERFACE EQUATES
;
BDOS    EQU    0005H        ;SYSTEM INTERFACE VECTOR
MAKE    EQU    22          ;MAKE NEW FILE FUNCTION
SBADDR  EQU    26          ;SET DISK BUFFER ADDR
OPEN    EQU    15          ;OPEN FILE FUNCTION
CLOSE   EQU    16          ;FILE CLOSE FUNCTION
DELETE  EQU    19          ;DELETE FILE FUNCTION
RRAND   EQU    33          ;READ RANDOM FUNCTION
WRAND   EQU    34          ;WRITE RANDOM FUNCTION
WRANDF  EQU    40          ;WRITE RANDOM WITH 00 FILL
PRINT   EQU    9          ;PRINT STRING TILL $
FSIZE   EQU    35          ;COMPUTE FILE SIZE FUNCTION
DEFCB   EQU    05CH          ;DEFAULT FILE CONTROL BLOCK
DEFBUF  EQU    080H          ;DEFAULT BUFFER LOCATION
;
EXEC    EQU    08000H         ;EXECUTE SPOT FOR SMALL PROGRAM
BOOT    EQU    00000H         ;SYSTEM REBOOT ENTRY POINT
;
;
;ASCII CHARACTER DEFINITIONS
;
CR  EQU    0DH         ;CARRIAGE RETURN
LF  EQU    0AH         ;LINE FEED
;
;
    ORG    0100H         ;START OF A PROGRAM
    LXI    SP,STACK     ;SETUP A STACK FOR EXECUTION
    LXI    D,SNGMSG     ;PRINT SIGNON MESSAGE
    MVI    C,PRINT
    CALL    BDOS
;
;
;CHECK IF THERE WAS A COMMAND LINE FILE NAME
;
    LDA    DEFCB+1         ;IF FIRST BYTE 20 THEN NO NAME
    CPI    ' '
    JZ    CMDERR         ;IF NO FILE NAME PRINT ERROR
    LDA    DEFCB+17     ;GET OPTION CHARACTER
    CPI    'E'         ;CHECK FOR ENCODE
    JZ    PROCESS         ;GO TO PROCESS IF ENCODE
    CPI    'D'         ;CHECK IF DECODE
    JZ    PROCESS         ;GO PROCESS OF DECODE
;
CMDERR:
    LXI    D,ERRM1         ;PRINT ERROR MESSAGE
    MVI    C,PRINT
```

```
    CALL    BDOS
    JMP    BOOT        ;EXIT IF NO FILE NAME OR OPTION
;
;
;HERE IF AN ENTRY FILE NAME AND A VALID OPTION
;
PROCESS:
    STA    OPTION        ;SAVE OPTION CHAR FOR LATER
                ;...REFERENCE
    XRA    A        ;SETUP FCB FOR OPEN
    STA    DEFCB+12    ;ZERO EXTENT BYTE
    STA    DEFCB+32    ;ZERO CURRENT RECORD BYTE
    STA    DEFCB+35    ;ZERO R2 BYTE
    LXI    H,0000H
    SHLD    DEFCB+33    ;ZERO RANDOM RECORD NUMBER
;
    MVI    C,OPEN        ;OPEN FILE USER SPECIFIED
    LXI    D,DEFCB        ;USE DEFAULT FCB BUILT BY CCP
    CALL    BDOS        ;GO ATTEMPT OPEN
    INR    A        ;CHECK IF FOUND
    JNZ    FOUND
;
    MVI    C,PRINT        ;PRINT NOT FOUND ERROR
    LXI    D,ERRM2
    CALL    BDOS
    JMP    BOOT        ;EXIT
;
;
;FOUND FILE SO LETS NEXT COMPUTE ITS FILE SIZE
;
FOUND:
    LXI    D,DEFCB        ;THAT SAME FCB AGAIN
    MVI    C,FSIZE
    CALL    BDOS        ;GET THE FILES SIZE IN RECORDS
    LHLD    DEFCB+33    ;GET SIZE OF THE FILE
    MOV    A,H        ;CHECK IF GREATER THAN 128 RECORDS
    ORA    A
    JNZ    TOBIG
    MOV    A,L
    ORA    A        ;CHECJ IF FILE EMPTY OR ONLY ONE RECORD
    JZ    TOSMALL
    CPI    1
    JZ    TOSMALL
    CPI    129
    JC    SIZINA        ;WE HAVE SIZE IN (A)
TOBIG:
    MVI    A,128        ;SET SIZE TO 128 DEFAULT
SIZINA:
    STA    NR        ;SAVE NUMBER OF RECORDS
    JMP    READFST
;
```

```
TOSMALL:
    MVI    C,PRINT        ;PRINT FILE SIZE ERROR MESSAGE
    LXI    D,ERRM3
    CALL    BDOS
    JMP    BOOT
;
;
;READ FIRST RECORD INTO LIST BUFFER
;
READFST:
    LXI    D,LIST        ;SET DMA ADDRESS TO LIST BUFFER
    MVI    C,SBADDR
    CALL    BDOS
    LXI    H,0000H        ;SET FIRST RECORD
    SHLD    DEFCB+33
    XRA    A
    STA    DEFCB+35    ;CLEAR R2 BYTE
    MVI    C,RRAND        ;READ RANDOM FIRST RECORD
    LXI    D,DEFCB
    CALL    BDOS        ;NO NEED TO CHECK READ ERROR BECAUSE
                ;..WE KNOW THAT THESE RECORDS EXIST
;
;
;HERE TO PROCESS LIST INTO A SET OF NUMBERS THAT FIT OUT FILE
;RECORD COUNT RANGE.
;
    LDA    NR        ;FETCH NUMBER OF RECORDS
    DCR    A        ;SET NR-1
;
    MVI    B,0FFH        ;INITIAL MASK VALUE
    MVI    C,07H        ;NUMBER OF TIMES TO ROTATE FOR MASK
;
MKLP:
    RAL            ;CHECK FOR ZERO BIT IN NR-1
    JC    HMSK        ;EXIT WE HAVE OUR MASK ONE BIT FROM (A)
    PUSH    PSW
    MOV    A,B        ;PUT A ZERO BIT INTO MASK
    ORA    A        ;CLEAR CARRY
    RAR            ;PUT ZERO IN
    MOV    B,A
    POP    PSW
    DCR    C        ;DEBUMP SHIFT COUNT
    JNZ    MKLP
;
HMSK:                ;HERE IF (B) HAS LIST MASK VALUE
    LDA    NR        ;GET NUMBER OF VALUES IN LIST
    DCR    A
    MOV    C,A        ;PUT LOOP COUNTER INTO (C)
    MOV    D,A        ;SAVE NR-1 IN (D)
    LXI    H,LIST        ;POINT AT LIST
LSTPROC:
```

```
    MOV    A,M        ;GET A LIST BYTE
    ANA    B         ;MASK IT
    CMP    D         ;IS RESULT GREATER THAN NR-2
    JC    VALOK        ;VALUE IS OK
    ORA    A         ;DIVIDE BY TWO IF TOO BIG
    RAR
VALOK:
    INR    A         ;SET VALUES TP FOR REAL RECORD NUMBERS
    MOV    M,A        ;PUT CONVERTED NUMBER INTO LIST AGAIN
    INX    H         ;BUMP LIST POINTER
    DCR    C         ;DEC LOOP COUNTER
    JNZ    LSTPROC        ;DO ALL BYTES OF LIST
;
;
;ENCODE/DECODE THE FILE HERE
;
ENCODE:
    LXI    H,LIST        ;KEEP A POINTER TO THE LIST
    LDA    OPTION        ;IF OPTION IS 'E' WE GO FORWARD
    CPI    'E'
    MVI    A,1        ;DEFAULT FORWARD CURRENT RECORD
    JZ    FORWA        ;GO FORWARD
    LDA    NR        ;INDEX TO END OF LIST FOR DECODE
    DCR    A         ;SET START RECORD FOR DECODE
    MOV    E,A
    DCR    E         ;ZERO BASE INDEX
    MVI    D,0
    DAD    D
;
FORWA:
    SHLD    LISTP        ;SAVE LIST POINTER
    STA    CURR        ;SET CURRENT RECORD NUMBER TO START
    LDA    NR
    DCR    A
    STA    CNTR        ;SET NUMBER OF SWAPS
;
ENCLP:
    LXI    D,BUF1        ;SET BUFFER ONE AS DMA ADDRESS
    MVI    C,SBADDR
    CALL    BDOS
    LDA    CURR        ;READ CURRENT RECORD
    MOV    L,A
    MVI    H,00
    SHLD    DEFCB+33    ;SET RECORD NUMBER
    LXI    D,DEFCB
    MVI    C,RRAND        ;READ THAT RECORD
    CALL    BDOS
    ORA    A         ;CHECK ERROR
    JNZ    DSKERR
;
    LXI    D,BUF2        ;SET BUFFER 2 AS DMA ADDRESS
```

```
    MVI    C,SBADDR
    CALL    BDOS
    LHLD    LISTP        ;GET SWAP POSITION
    MOV    L,M
    MVI    H,00
    SHLD    DEFCB+33    ;SET SWAP RECORD NUMBER
    LXI    D,DEFCB
    MVI    C,RRAND        ;READ SWAP RECORD
    CALL    BDOS
    ORA    A        ;CHECK ERROR
    JNZ    DSKERR
;
    LHLD    LISTP        ;IS SWAP RECORD AN ODD NUMB
    MOV    B,M        ;SABE XOR PATTERN IN (B)
    MOV    A,M
    RAR
    JNC    SWRT        ;GO DO SWAP WRITE DIRECTLY IF EVEN
    LDA    OPTION        ;WHICH BUFFER TO XOR
    LXI    H,BUF2        ;DEFAULT FOR 'E'
    CPI    'E'
    JZ    INB2        ;USE BUFFER 2
    LXI    H,BUF1        ;IF DECODE USE BUFFER 1
INB2:
    MVI    C,128        ;BUTE COUNT OF XOR
XORLP:
    MOV    A,M        ;GET A BYTE TO XOR
    XRA    B
    MOV    M,A        ;PUT BYTE BACK
    INX    H        ;BUMP BUFFER POINTER FOR XORING
    DCR    C        ;DEC BYTE COUNT
    JNZ    XORLP
;
SWRT:
    LXI    D,BUF1        ;SET BUFFER ONE AS DMA ADDRESS
    MVI    C,SBADDR
    CALL    BDOS
    LHLD    LISTP        ;GET SWAP POSITION
    MOV    L,M
    MVI    H,00
    SHLD    DEFCB+33    ;SET SWAP RECORD NUMBER
    LXI    D,DEFCB
    MVI    C,WRAND        ;WRITE SWAP RECORD
    CALL    BDOS
    ORA    A        ;CHECK ERROR
    JNZ    DSKERR
;
    LXI    D,BUF2        ;SET BUFFER 2 AS DMA ADDRESS
    MVI    C,SBADDR
    CALL    BDOS
    LDA    CURR        ;WRITE CURRENT RECORD
    MOV    L,A
```

```
    MVI    H,00
    SHLD    DEFCB+33    ;SET RECORD NUMBER
    LXI    D,DEFCB
    MVI    C,WRAND        ;WRITE THAT RECORD
    CALL    BDOS
    ORA    A        ;CHECK ERROR
    JNZ    DSKERR
;
    LDA    CURR        ;FETCH LOOP PARMS
    MOV    B,A
    LHLD    LISTP
;
    LDA    OPTION        ;CHECK OPTION
    CPI    'E'
    JZ    INCF        ;IF ENCODE INCR FORWARD
;
DECB:
    DCX    H        ;DECREMENT DOWN THROUGH LOOP
    DCR    B
    JMP    PSVE        ;SAVE PARMS
INCF:
    INX    H
    INR    B
PSVE:
    SHLD    LISTP        ;SAVE NEW LIST POSITION
    MOV    A,B
    STA    CURR
;
    LDA    CNTR        ;FETCH LOOP COUNTER
    DCR    A
    STA    CNTR
    JNZ    ENCLP        ;GO TO LOOP TO PROCESS MORE IF
             ;NOT DONE YET
;
;
;HERE WE ARE DONE WRITING SO LETS CLOSE UP AND GO HOME
;
    LXI    D,DEFCB
    MVI    C,CLOSE
    CALL    BDOS
    INR    A        ;CHECK ERROR CODE
    JZ    DSKERR
;
    MVI    C,PRINT        ;PRINT DONE MESSAGE
    LXI    D,DONMSG
    CALL    BDOS
    JMP    BOOT        ;EXIT
;
;
;EXIT POINT WITH ERROR MESSAGE IF THE DISK WRITE OPERATION
;RESULTED IN AN ERROR
```

```
;
DSKERR:
    LXI    D,ERRM4        ;PRINT GARBAGE FILE ERROR
    MVI    C,PRINT
    CALL    BDOS
    JMP    BOOT        ;EXIT FOR THE POOR GUY
;
;
;PROGRAM OPERATIONAL MESSAGES
;
SNGMSG:
    DB    CR,LF,'MICRO RESOURCES Disk File Scramble and'
    DB    CR,LF,'Unscramble Utility Designed to Demonstrate'
    DB    CR,LF,'CP/M Ver 2.2 Random Record I/O. (1/24/82)','$'
;
DONMSG:
    DB    CR,LF,'File Processing Complete','$'
;
ERRM1:
    DB    CR,LF,'No File Name Specified or Improper Option','$'
;
ERRM2:
    DB    CR,LF,'Specified File Not Found','$'
;
ERRM3:
    DB    CR,LF,'Cannot Process Files with 0 or 1 Record(s)','$'
;
ERRM4:
    DB    CR,LF,'File I/O Error, This Error Should NOT Normally'
    DB    CR,LF,'Happen, But the File is now Garbaged...','$'
;
;
;PROGRAM DATA STORAGE SECTION
;
OPTION:
    DS    1        ;PLACE TO STORE COMMAND LINE OPTION CHAR
;
NR:
    DS    1        ;NUMBER OF RECORDS TO SWAP
;
CNTR:
    DS    1        ;ENCODE/DECODE LOOP COUNTER
;
CURR:
    DS    1        ;CURRENT SWAP SECTOR
;
LISTP:
    DS    2        ;LIST SCAN POINTER
;
LIST:
    DS    128        ;LIST BUFFER
```

```
;
BUF1:
    DS    128         ;DATA BUFFER 1
;
BUF2:
    DS    128         ;DATA BUFFER 2
;
    DS    36
STACK   EQU    $    ;USER STACK AREA
;
;
    END
;
;
;+++...END OF FILE
```

From:
https://hc-ddr.hucki.net/wiki/ - **Homecomputer DDR**

Permanent link:
https://hc-ddr.hucki.net/wiki/doku.php/cpm/sliding_into_bdos?rev=1516801609

Last update: **2018/01/24 13:46**