FLEX Programmer's Manual

Technical Systems Consultants, Inc.

CONTENTS

Preface

The purpose of the Advanced Programmer's Manual is to provide the
assembler language programmer with the information required to make
effective use of the available system routines and functions. This
manual applies to all 6800 versions of FLEX. The programmer should keep
this manual close at hand while learning the system. It is organized to
make it convenient as a quick reference guide as well as a thorough
reference manual. The manual is not written for the novice programmer and
assumes the user to have a thorough understanding of assembler language
programming techniques.

Introduction

The FLEX Operating System consists of three main parts: the Disk
Operating System (DOS) which processes commands, the File Management
System (FMS) which manages files on a diskette, and the Utility Command
Set, which are the user-callable commands. The Utility Command Set is
described in the FLEX User's Guide. Details of the Disk Operating System
and File Management System portions of FLEX are described in this manual,
which is intended for the programmer who wishes to write his own commands
or process disk files from his own program.

When debugging programs which use disk files and the File Management
System, the user should take the following precautions:

1.   Write-protect the system diskette by exposing or covering the
write-protect cutout on the diskette. See the FLEX User's Guide for
further details on this operation. This will prevent destruction of the
system disk in case the program starts running wild.

2.   Use an empty scratch diskette as the working diskette to which your
program will write any data files. If something goes wrong and the
diskette is destroyed, no valuable data will have been lost.

3.   Test your program repeatedly, especially with "special cases" of
data input which may not be what the program is expecting. Well-written
programs abort gracefully when detecting errors, not dramatically.

A careful programmer, using the information in this manual, should be
able to make the fullest use of his floppy disk system.

                       THE DISK OPERATING SYSTEM

The Disk Operating System (DOS) forms the communication link between the
user (via a computer terminal) and the File Management System. All
commands are accepted through DOS. Functions such as file specification
parsing, command argument parsing, terminal I/O, and error reporting are
all handled by DOS. The following sections describe the DOS global
variable storage locations (Memory Map), the DOS user callable
subroutines, and give examples of some possible uses.

                             DOS MEMORY MAP

The following is a description of those memory locations within the DOS
portion of FLEX which contain information of interest to the programmer.
The user is cautioned against utilizing for his own purposes any
locations documented as being either "reserved" or "system scratch", as
this action may cause destruction of data.

$A080-$A0FF - Line Buffer
     The line buffer is a 128 byte area into which characters typed at
     the keyboard are placed by the routine INBUF. All characters
     entered from the keyboard are placed in this buffer with the
     exception of control characters. Characters which have been deleted
     by entering the backspace character do not appear in the buffer,
     nor does the backspace character itself appear. The carriage return
     signalling the end of the keyboard input is, however, put in the
     buffer. This buffer is also used to hold the STARTUP file during a
     cold start (boot) operation.

$AC00 - TTYSET Backspace Character
     This is the character which the routine INBUF will interpret as the
     Backspace character. It is user definable through the TTYSET DOS
     Utility. Default = $08, a Control-H (ASCII BS).

$AC01 - TTYSET Delete Character
     This is the character which the routine INBUF will interpret as the
     line cancel or Delete character. It is user definable through the
     TTYSET DOS Utility. Default = $18, a control-X (ASCII CAN).

$AC02 - TTYSET End of Line Character
     This is the character DOS recognizes as the multiple command per
     line separator. It is user definable through the TTYSET Utility.
     Default = $3A, a colon (:).

$AC03 - TTYSET Depth Count
     This byte determines how many lines DOS will print on a page before
     Pausing or issuing Ejects. It may be set by the user with the
     TTYSET command. Default = 0.

$AC04 - TTYSET Width Count
     This byte tells DOS how many characters to output on each line. If
     zero, there is no limit to the number output. This count may be set
     by the user using TTYSET. Default = 0.

$AC05 - TTYSET Null Count
     This byte informs DOS of the number of null or pad characters to be
     output after each carriage return, line feed pair. This count may
     be set using TTYSET. Default = 4.

$AC06 - TTYSET Tab Character
     This byte defines a tab character which may be used by other
     programs, such as the Editor. DOS itself does not make use of the
     Tab character. Default = 0, no tab character defined.

$AC07 - TTYSET Backspace Echo Character
     This is the character the routine INBUF will echo upon the receipt
     of a backspace character. If the backspace echo character is set to
     a $08, and the backspace character is also a $08, FLEX will output
     a space ($20) prior to the outputting of the backspace echo
     character. Default = 0.

$AC08 - TTYSET Eject Count
     The Eject Count instructs DOS as to the number of blank lines to be
     output after each page. (A page is a set of lines equal in number
     to the Depth Count). If this byte is zero, no Eject lines are
     output. Default = 0.

$AC09 - TTYSET Pause Control
     The Pause byte instructs DOS what action to take after each page is
     output. A zero value indicates that the pause feature is enabled; a
     non-zero value, pause is disabled. Default = $FF, pause disabled.

$AC0A - TTYSET Escape Character
     The Escape character causes DOS to pause after an output line.
     Default = $1B, ASCII ESC.

$AC0B - System Drive Number
     This is the number of the disk drive from which commands are
     loaded. If this byte is $FF, all ready drives will be searched.
     Default = drive #0.

$AC0C - Working Drive Number
     This is the number of the default disk drive referenced for non-
     command files. If this byte is $FF, all ready drives will be
     searched. Default = drive #0.

$AC0D - System Scratch

$AC0E-$AC10 - System Date Registers
     These three bytes are used to store the system date. It is stored
     in binary form with the month in the first byte, followed by the
     day, then the year. The year byte contains only the tens and ones
     digits.

$AC11 - Last Terminator
     This location contains the most recent non-alphanumeric character
     encountered in processing the line buffer. See commentary on the
     routines NXTCH and CLASS in the section "User-Callable System
     Routines".

$AC12-$AC13 - User Command Table Address
     The programmer may store into these locations the address of a
     command table of his own construction. See the section called
     "User-Written Commands" for details. Default = 0000, no user
     command table is defined.

$AC14-$AC15 - Line Buffer Pointer
     These locations contain the address of the next character in the
     Line Buffer to be processed. See documentation of the routines
     INBUFF, NXTCH, GETFIL, GETCHR, and DOCMND in the section "User-
     Callable System Routines" for instances of its use.

$AC16-$AC17 - Escape Return Register
     These locations contain the address to which to jump if a RETURN is
     typed while output has been stopped by an Escape Character. See the
     FLEX User's Guide, TTYSET, for information on Escape processing.
     See also the documentation for the routine PCRLF in the section
     called "User-Callable System Routines".

$AC18 - Current Character
     This location contains the most recent character taken from the
     Line Buffer by the NXTCH routine. See documentation of the NXTCH
     routine for additional details.

$AC19 - Previous Character
     This location contains the previous character taken from the Line
     Buffer by the NXTCH routine. See documentation of the NXTCH routine
     for additional details.

$AC1A - Current Line Number
     This location contains a count of the number of lines currently on
     the page. This value is compared to the Line Count value to
     determine if a full page has been printed.

$AC1B-$AC1C - Loader Address Offset
     These locations contain the 16-bit bias to be added to the load
     address of a routine being loaded from the disk. See documentation
     of the System Routine LOAD for details. These locations are also
     used as scratch by some system routines.

$AC1D - Transfer Flag
     After a program has been loaded from the disk (see LOAD
     documentation), this location is non-zero if a transfer address was
     found during the loading process. This location is also used as
     scratch by some system routines.

$AC1E-$AC1F - Transfer Address
      If the Transfer Flag was set non-zero by a load from the disk (see
      LOAD documentation), these locations contain the last transfer
      address encountered. If the Transfer Flag was set zero by the disk
      load, the content of these locations is indeterminate.

$AC20 - Error Type
      This location contains the error number returned by several of the
      File Management System functions. See the "Error Numbers" section
      of this document for an interpretation of the error numbers.

$AC21 - Special I/O Flag
      If this byte is non-zero, the PUTCHR routine will ignore the TTYSET
      Width feature and also ignore the Escape Character. The routine
      RSTRIO clears this byte. Default = 0.

$AC22 - Output Switch
      If zero, output performed by the PUTCHR routine is through the
      routine OUTCH. If non-zero, the routine OUTCH2 is used. See
      documentation of these routines for details.

$AC23 - Input Switch
      If zero, input performed by GETCHR is through the routine INCH. If
      it is non-zero, the routine INCH2 is used. See documentation of
      these routines for details.

$AC24-$AC25 - File Output Address
      These bytes contain the address of the File Control Block being
      used for file output. If the bytes are zero, no file output is
      performed. See PUTCHR description for details. These locations are
      set to zero by RSTRIO.

$AC26-$AC27 - File Input Address
      These bytes contain the address of the File Control Block being
      used for file input. If the bytes are zero, no file input is
      performed. The routine RSTRIO clears these bytes. See GETCHR for
      details.

$AC28 - Command Flag
      This location is non-zero if DOS was called from a user program via
      the DOCMND entry point. See documentation of DOCMND for details.

$AC29 - Current Output Column
      This location contains a count of the number of characters
      currently in the line being output to the terminal. This is
      compared to the TTYSET Width Count to determine when to start a new
      line. The output of a control character resets this count to zero.

$AC2A - System Scratch

$AC2B-$AC2C - Memory End
>   These two bytes contain the end of user memory. This location is set during system boot and may be read by programs requiring this information.

$AC2D-$AC2E - Error Name Vector
>   If these bytes are zero, the routine RPTERR will use the file ERRORS.SYS as the error file. If they are non-zero, they are assumed to be the address of an ASCII string of characters (in directory format) of the name of the file to be used as the error file. See the description of RPTERR for more details.

$AC2F - File Input Echo Flag
>   If this byte is non-zero (default) and input is being done through a file, the character input will be echoed to the output channel. If this byte is zero, the character retrieved will not be echoed.

$AC30-$AC4D - System Scratch

$AC4E-$ACBF - System Constants

$ACC0-$ACD7 - Printer Initialize
>   This area is reserved for the overlay of the system printer initialization subroutine.

$ACD8-$ACE3 - Printer Ready Check
>   This area is reserved for the overlay of the system "check for printer ready" subroutine.

$ACE4-$ACF7 - Printer Output
>   This area is reserved for the overlay of the system printer output character routine. See Printer Routine descriptions for details.

$ACF8-$ACFF - System Scratch

USER-CALLABLE SYSTEM ROUTINES

Unless specifically documented otherwise, the content of all registers should be presumed destroyed by calls to these routines. All routines, unless otherwise indicated, should be called with a JSR instruction.

$AD00 (COLDS) Coldstart Entry Point

> The BOOT program loaded from the disk jumps to this address to initialize the FLEX system. Both the Disk Operating System (DOS) portion and the File Management System portion (FMS) of FLEX are initialized. After initialization, the FLEX title line is printed and the STARTUP file, if one exists, is loaded and executed. This entry point is only for use by the BOOT program, not by user programs. Indiscriminate use of the Coldstart Entry Point by user programs could result in the destruction of the diskette. Documentation of this routine is included here only for completeness.

$AD03 (WARMS) Warmstart Entry Point

> This is the main re-entry point into DOS from user programs. A JMP instruction should be used to enter the Warmstart Entry Point. Here, the system stack and the Escape Return Register are reset. At this point, the main loop of DOS is entered. The main loop of DOS checks the Last Terminator location for a TTYSET end-of-line character. If one is found, it is assumed that there is another command on the line, and DOS attempts to process it. If no end-of-line is in the Last Terminator location DOS assumes that the current command line is finished, and looks for a new line to be input from the keyboard. If, however, DOS was called from a user program through the DOCMND entry point, control will be returned to the user program when the end of a command line is reached.

$AD06 (RENTER) DOS Main Loop Re-entry Point

> This is a direct entry point into the DOS main loop. None of the Warmstart initialization is performed. This entry point must be entered by a JMP instruction. Normally, this entry point is used internally by DOS and user-written programs should not have need to use it. For an example of use, see "Printer Driver" section for details.

$AD09 (INCH) Input Character
$AD0C (INCH2) Input Character

> Each of these routines inputs one character from the keyboard,
> returning it to the calling program in the A-register. These
> routines are alterable only through a custom I/O package or through
> the "Auto Block" in the SWTPc FLEX 2.0 version. The GETCHR routine
> normally uses INCH but may be instructed to use INCH2 by setting
> the "Input Switch" non-zero (see Memory Map). The user's program
> may change the jump vector at the INCH address to refer to some
> other input routine such as a routine to get a character from paper
> tape. The OUTCH2 address should never be altered. The Warmstart
> Entry Point resets the INCH jump vector to the same routine as
> INCH2 and sets the Input Switch to zero. RSTRIO also resets these
> bytes. User programs should use the GETCHR routine, documented
> below, rather than calling INCH, because INCH does not check the
> TTYSET parameters.

$AD0F (OUTCH) Output Character
$AD12 (OUTCH2) Output Character

> On entry to each of these routines, the A-register should contain
> the character being output. Both of these routines output the
> character in the A-register to an output device. The OUTCH routine
> usually does the same as OUTCH2; however, OUTCH may be changed by
> programs to refer to some other output routine. For example, OUTCH
> may be changed to drive a line printer. These routines are
> alterable only through the custom I/O package or the "Auto Block"
> in the SWTPc FLEX 2.0 version. The routine PUTCHR, documented
> below, calls one of these two routines, depending on the content of
> the location "Output Switch" (see Memory Map). The Warmstart Entry
> Point resets the OUTCH jump vector to the same routine as OUTCH2,
> and sets the Output Switch to zero. RSTRIO also resets these
> locations. User routines should use PUTCHR rather than calling
> OUTCH or OUTCH2 directly since these latter two do not check the
> TTYSET parameters.

$AD15 (GETCHR) Get Character

> This routine gets a single character from the keyboard. The
> character is returned to the calling program in the A-register. The
> Current Line Number location is cleared by a call to GETCHR.
> Because this routine honors the TTYSET parameters, its use is
> preferred to that of INCH. If the location "Input Switch" is non-
> zero, the routine INCH2 will be used for input. If zero, the byte
> at "File Input Address" is checked. If it is non-zero, the address
> at this location is used as a File Control Block of a previously
> opened input file and a character is retrieved from the file. If
> zero, a character is retrieved via the INCH routine. The X and B
> registers are preserved.

$AD18 (PUTCHR) Put Character

This routine outputs a character to a device, honoring all of the
TTYSET parameters. On entry, the character should be in the A-
register. If the "Special I/O Flag" (see Memory Map) is zero, the
column count is checked, and a new line is started if the current
line is full. If an ACIA is being used to control the monitor
terminal, it is checked for a TTYSET Escape Character having been
typed. If so, output will pause at the end of the current line. If
the location "Output Switch" is non-zero, the routine OUTCH2 is
used to send the character. If zero, the location File Output
Address is checked. If it is non-zero the contents of this location
is used as an address of a File Control Block of a previously
opened for write file, and the character is written to the file. If
zero, the routine OUTCH is called to process the character.
Normally, OUTCH sends the character to the terminal. The user
program may, however, change the address portion of the OUTCH entry
point to go to another character output routine. The X and B
registers are preserved.

$AD1B (INBUFF) Input into Line Buffer

This routine inputs a line from the keyboard into the Line Buffer.
The TTYSET Backspace and Delete characters are checked and
processed if encountered. All other control characters except
RETURN and LINE FEED, are ignored. The RETURN is placed in the
buffer at the end of the line. A LINE FEED is entered into the
buffer as a space character but is echoed back to the terminal as a
Carriage Return and Line Feed pair for continuation of the text on
a new line. At most, 128 characters may be entered on the line,
including the final RETURN. If more are entered, only the first 127
are kept, the RETURN being the 128th. On exit, the Line Buffer
Pointer is pointing to the first character in the Line Buffer.
Caution: The command line entered from the keyboard is kept in the
Line Buffer. Calling INBUF from a user program will destroy the
command line, including all unprocessed commands on the same line.
Using INBUF and the Line Buffer for other than DOS commands may
result in unpredictable side-effects.

$AD1E (PSTRNG) Print String

This routine performs a carriage return and line feed and then
prints a string of characters. On entry, the X-register should
contain the address of the first character of the string to be
printed. The string must end with an ASCII EOT character ($04).
This routine honors all of the TTYSET conventions when printing the
string. A carriage return and line feed are output before the
string. The B register is preserved.

$AD21 (CLASS) Classify Character

This routine is used for testing if a character is alphanumeric (i.e. a letter or a number). On entry, the character should be in the A-register. If the character is alphanumeric, the routine returns with the carry flag cleared. If the character is not alphanumeric, the carry flag is set and the character is stored in the Last Terminator location. All registers are preserved by this routine.

$AD24 (PCRLF) Print Carriage Return and Line Feed

In addition to printing a carriage return and line feed, this routine checks and honors several TTYSET conditions. On entry, this routine checks for a TTYSET Escape Character having been entered while the previous line was being printed. If so, the routine waits for another TTYSET Escape Character or a RETURN to be typed. If a RETURN was entered, the routine clears the Last Terminator location so as to ignore any commands remaining in the command line, and then jumps to the address contained in the Escape Return Register locations. Unless changed by the user's program, this address is that of the Warmstart Entry Point. If, instead of a RETURN, another TTYSET Escape Character was typed, or it wasn't necessary to wait for one, the Current Line Number is checked. If the last line of the page has been printed and the TTYSET Pause feature is enabled, the routine waits for a RETURN or a TTYSET Escape Character, as above. Note that all pausing is done before the carriage return and line feed are printed. The carriage return and line feed are now printed, followed by the number of nulls specified by the TTYSET Null Count. If the end of the page was encountered on entry to this routine, an "eject" is performed by issuing additional carriage return, line feeds, and nulls until the total number of blank lines is that specified in the TTYSET Eject Count. The X register is preserved.

$AD27 (NXTCH) Get Next Buffer Character

The character in location Current Character is placed in location Previous Character. The character to which the Line Buffer Pointer points is taken from the Line Buffer and saved in the Current Character location. Multiple spaces are skipped so that a string of spaces looks no different than a single space. The Line Buffer Pointer is advanced to point to the next character unless the character just fetched was a RETURN or TTYSET End-of-Line character. Thus, once an end-of-line character or RETURN is encountered, additional calls to NXTCH will continue to return the same end-of-line character or RETURN. NXTCH cannot be used to cross into the next command in the buffer. NXTCH exits through the routine CLASS, automatically classifying the character. On exit, the character is in the A-register, the carry is clear if the character is alphanumeric, and the B-register and X- register are preserved.

$AD2A (RSTRIO) Restore I/O Vectors

This routine forces the OUTCH jump vector to point to the same routine as does the OUTCH2 vector. The Output Switch location and the Input Switch location are set to zero. The INCH jump vector is reset to point to the same address as the INCH2 vector. Both the File Input Address and the File Output Address are set to zero. The A-register and B-register are preserved by this routine.

$AD2D (GETFIL) Get File Specification

On entry to this routine, the X-register must contain the address of a File Control Block (FCB), and the Line Buffer Pointer must be pointing to the first character of a file specification in the Line Buffer. This routine will parse the file specification, storing the various components in the FCB to which the X-register points. If a drive number was not specified in the file specification, the working drive number will be used. On exit, the carry bit will be clear if no error was detected in processing the file specification. The carry bit will be set if there was a format error in the file specification. If no extension was specified in the file specification, none is stored. The calling program should set the default extension desired after GETFIL has been called by using the SETEXT routine. The Line Buffer Pointer is left pointing to the character immediately beyond the separator, unless the separator is a carriage return or End of Line character. If an error was detected, Error number 21 is stored in the error status byte of the FCB. The X register is preserved with a call to this routine.

$AD30 (LOAD) File Loader

On entry, the system File Control Block (at $A840) must contain the name of a file which has been opened for binary reading. This routine is used to load binary files only, not text files. The file is read from the disk and stored in memory, normally at the load addresses specified in the binary file itself. It is possible to load a binary file into a different memory area by using the Loader Address Offset locations. The 16-bit value in the Loader Address Offset locations is added to the addresses read from the binary file. Any carry generated out of the most significant bit of the address is lost. The transfer address, if any is encountered, is not modified by the Loader Address Offset. Note that the setting of a value in the Loader Address Offset does not modify any part of the content of the binary file. It does not act as a program re-locator in that it does not change any addresses in the program itself, merely the location of the program in memory. If the file is to be loaded without an offset, be certain to clear the Loader Address Offset locations before calling this routine. On exit, the Transfer Address Flag is zero if no transfer address was found. This flag is non-zero if a transfer address record was encountered in the binary file, and the Transfer Address locations contain the last transfer address encountered. The disk file is closed on exit. If a disk error is encountered,

an error message is issued and control is returned to DOS at the Warmstart Entry Point.

$AD33 (SETEXT) Set Extension

On entry, the X-register should contain the address of the FCB into which the default extension is to be stored if there is not an extension already in the FCB. The A-register, on entry, should contain a numeric code indicating what the default extension is to be. The numeric codes are described below. If there is already an extension in the FCB (possibly stored there by a call to GETFIL), this routine returns to the calling program immediately. If there is no extension in the FCB, the extension indicated by the numeric code in the A-register is placed in the FCB File Extension area. The legal codes are:

```
 0 - BIN
 1 - TXT
 2 - CMD
 3 - BAS
 4 - SYS
 5 - BAK
 6 - SCR
 7 - DAT
 8 - BAC
 9 - DIR
10- PRT
11- OUT
```

Any values other than those above are ignored, the routine returning without storing any extension. The X register is preserved in this routine.

$AD36 (ADDBX) Add B-register to X-register

The content of the B-register is added to the content of the X-register. The content of the B-register is destroyed on exit.

$AD39 (OUTDEC) Output Decimal Number

On entry, the X-register contains the address of the most significant byte of a 16-bit (2 byte), unsigned, binary number. The B-register, on entry, should contain a space suppression flag. The number will be printed as a decimal number with leading zeroes suppressed. If the B-register was non-zero on entry, spaces will be substituted for the leading zeroes. If the B-register is zero on entry, printing of the number will start with the first non-zero digit.

$AD3C (OUTHEX) Output Hexadecimal Number

On entry, the X-register contains the address of a single binary byte. The byte to which the X-register points is printed as 2 hexadecimal digits. The B and X registers are preserved.

-13-

$AD3F (RPTERR) Report Error

On entry to this routine, the X-register contains the address of a File Control Block in which the Error Status Byte is non-zero. The error code in the FCB is stored by this routine in the Error Type location. A call to the routine RSTRIO is made and location Error Vector is checked. If this location is zero, the file ERRORS.SYS is opened for random read. If this location is non-zero, it is assumed to be an address pointing to an ASCII string (containing any necessary null pad characters) of a legal File name plus extension (string should be 11 characters long). This user provided file is then opened for random read. The error number is used in a calculation to determine the record number and offset of the appropriate error string message in the file. Each error message string is 63 characters in length, thus allowing 4 messages per sector. If the string is found, it is printed on the terminal. If the string is not found (due to too large of error number being encountered) or if the error file itself was not located on the disk, the error number is reported to the monitor terminal as part of the message:

        DISK ERROR #nnn

Where "nnn" is the error number being reported. A description of the error numbers is given elsewhere in this document.

$AD42 (GETHEX) Get Hexadecimal Number

This routine gets a hexadecimal number from the Line Buffer. On entry, the Line Buffer Pointer must point to the first character of the number in the Line Buffer. On exit, the carry bit is cleared if a valid number was found, the B-register is set non-zero, and the X-register contains the value of the number. The Line Buffer Pointer is left pointing to the character immediately following the separator character, unless that character is a carriage return or End of Line. If the first character examined in the Line Buffer is a separator character (such as a comma), the carry bit is still cleared, but the B-register is set to zero indicating that no actual number was found. In this case, the value returned in the X-register is zero. If a non-hexadecimal character is found while processing the number, characters in the Line Buffer are skipped until a separator character is found, then the routine returns to the caller with the carry bit set. The number in the Line Buffer may be of any length, but the value is truncated to between 0 and $FFFF, inclusive.

$AD45 (OUTADR) Output Hexadecimal Address

On entry, the X register contains the address of the most significant byte of a 2 byte hex value. The bytes to which the X register points are printed as 4 hexadecimal digits.

$AD48 (INDEC) Input Decimal Number

> This routine gets an unsigned decimal number from the Line Buffer.
> On entry, the Line Buffer Pointer must point to the first character
> of the number in the Line Buffer. On exit, the carry bit is cleared
> if a valid number was found, the B-register is set non-zero, and
> the X-register contains the binary value of the number. The Line
> Buffer Pointer is left pointing as described in the routine GETHEX.
> If the first character examined in the buffer is a separator
> character (such as a comma), the carry bit is still cleared, but
> the B-register is set to zero indicating that no actual number was
> found. In this case, the number returned in X is zero. The number
> in the Line Buffer may be of any length but the result is truncated
> to 16 bit precision.

$AD4B (DOCMND) Call DOS as a Subroutine

> This entry point allows a user-written program to pass a command
> string to DOS for processing, and have DOS return control to the
> user program on completion of the commands. The command string must
> be placed in the Line Buffer by the user program, and the Line
> Buffer Pointer must be pointing to the first character of the
> command string. Note that this will destroy any as yet unprocessed
> parameters and commands in the Line Buffer. The command string must
> terminate with a RETURN character ($D hex). After the commands have
> been processed, DOS will return control to the user's program with
> the B-register containing any error code received from the File
> Management System. The B-register will be zero if no errors were
> detected. Caution: do not use this feature to load programs which
> may destroy the user program in memory. An example of a use of this
> feature of DOS is that of a program wanting to save a portion of
> memory as a binary file on the disk. The program could build a SAVE
> command in the Line Buffer with the desired file name and
> parameters, and call the DOCMND entry point. On return, the memory
> will have been saved on the disk.

USER-WRITTEN COMMANDS

The programmer may write his own commands for DOS. These commands may be either disk-resident as disk files with a CMD extension, or they may be memory-resident in either RAM or ROM.

MEMORY-RESIDENT COMMANDS :

A memory-resident command is a program, already in memory, to which DOS will transfer when the proper command is entered from the keyboard. The command which invokes the program, and the entry-point of the program, are stored in a User Command Table created by the programmer in memory. Each entry in the User Command Table has the following format:

```
FCC  'command' (Name that will invoke the program)
FCB  0
FDB  entry address (This is the entry address of the program)
```

The entire table is ended by a zero byte. For example, the following table contains the commands DEBUG (entry at $3000) and PUNT (entry at $3200):

```
FCC  'DEBUG' Command Name
FCB  0
FDB  $3000 Entry address for DEBUG
FCC  'PUNT' Command name
FCB  0
FDB  $3200 Entry address for PUNT
FCB  0 End of command table
```

The address of the User Command Table is made known to DOS by storing it in the User Command Table Address locations (See Memory Map).

The User Command Table is searched before the disk directory, but after DOS's own command table is searched. The DOS command table contains only the GET and MON commands. Therefore, the user may not define his own GET and MON commands.

Since the User Command Table is searched before the disk directory, the programmer may have commands with the same name as those on the disk. However, in this case, the commands on the disk will never be executed while the User Command Table is known to DOS. The User Command Table may be deactivated by clearing the User Command Table Address locations.

DISK-RESIDENT COMMANDS

A disk-resident command is an assembled program, with a transfer address, which has been saved on the disk with a CMD extension. The ASMB section of the FLEX User's Guide describes the way to assign a transfer address to a program being assembled.

Disk commands, when loaded into memory, may reside anywhere in the User RAM Area; the address is determined at assembly time by using an ORG statement. Most commands may be assembled to run in the Utility Command Space (see Memory Map). Most of the commands supplied with FLEX run in the Utility Command Space. For this reason, the SAVE command cannot be used to save information which is in the Utility Command Space or System FCB space as this information would be destroyed when the SAVE command is loaded. The SAVE.LOW command is to be used in this case. The SAVE.LOW command loads into memory at location $100 and allows the saving of programs in the $A100 region.

The System FCB area is used to load all commands from the disk. Commands written to run in the Utility Command Space must not overflow into the System FCB area. Once loaded, the command itself may use the System FCB area for scratch or as an FCB for its own disk I/O. See the example in the FMS section.

GENERAL COMMENTS ABOUT COMMANDS

User-written commands are entered by a JMP instruction. On completion, they should return control to DOS by jumping (JMP instruction) to the Warmstart Entry Point (see Memory Map).

Processing Arguments.

User-written commands are required to process any arguments entered from the keyboard. The command name and the arguments typed are in the Line Buffer area (see Memory Map). The Line Buffer Pointer, on entry to the command, is pointing to the first character of the first argument, if one exists. If there are no arguments, the Line Buffer Pointer is pointing to either an end-of-line character or a carriage return. The DOS routines NXTCH, GETFIL, and GETHEX should be used by the command for processing the arguments.

Processing Errors.

If the command, while executing, receives an error status from either DOS or FMS of such a nature that the command must be aborted, the program should jump to the Warmstart Entry Point of DOS after issuing an appropriate error message. Similarly, if the command should detect an error on its own, it should issue a message and return to DOS through the Warmstart Entry Point.

EXAMPLES OF USING DOS ROUTINES

1. Setting up a file spec in the FCB can be done in the following manner. This example assumes the Line Buffer Pointer is pointing to the first character of a file specification, and the desired resulting file spec should default to a TXT extension.

```
LDX  #FCB     Point to FCB
JSR  GETFIL   Get file spec into FCB
BCS  ERROR    Report error if one
LDAA #1       Set extension code (TXT)
JSR  SETEXT   Set the default extension
```

The user may now open the file for the desired action, since the file spec is correctly set up in the FCB. Refer to the FMS examples for opening files.

2. The following examples demonstrate some simple uses of the basic I/O functions provided by DOS.

```
LDAA #'A      Setup an ASCII A
JSR  PUTCHR   Call DOS out character

LDX  #STRING  Point to string
JSR  PSTRNG   Print CR & LF + string
```

The above simple examples are to show the basic mechanism for calling and using DOS I/O routines.

THE FILE MANAGEMENT SYSTEM

The File Management System (FMS), forms the communication link between the DOS and the actual Disk Hardware. The FMS performs all file allocation and removal on the disk. All file space is allocated dynamically, and the space used by files is immediately reusable upon that file's deletion. The user of the FMS need not be concerned with the actual location of a file on the disk, or how many sectors it requires.

Communication with the FMS is done through File Control Blocks. These blocks contain the information about a file, such as its name and what drive it exists on. All disk I/O performed through FMS is "one character at a time" I/O. This means that programs need only send or request a single character at a time while doing file data transfers. In effect, the disk looks no different than a computer terminal. Files may be opened for either reading or writing. Any number of files may be opened at any one time, as long as each one is assigned its own File Control Block.

The FMS is a command language whose commands are represented by various numbers called Function Codes. Each Function Code tells FMS to perform a specific function such as open a file for read, or delete a file. In general, making use of the various functions which the FMS offers, is quite simple. The index register is made to point to the File Control Block which is to be used, the Function Code is stored in the first byte of the File Control Block, and FMS is called as a subroutine (JSR). At no time does the user ever have to be concerned with where the file is being located on the disk, how long it is, or where its directory entry is located. The FMS does all of this automatically.

Since the file structure of FLEX is a linked structure, and the disk space is allocated dynamically, it is possible for a file to exist on the disk in a set of non-contiguous sectors. Normally, if a disk has just been formatted, a file will use consecutive sectors on the disk. As files are created and deleted, however, the disk may become "fragmented". Fragmentation results in the sectors on the disk becoming out of order physically, even though logically they are still all sequential. This is a characteristic of "linked list" structures and dynamic file allocation methods. The user need not be concerned with this fragmentation, but should be aware of the fact that files may exist whose sectors seem to be spattered all over the disk. The only result of fragmentation is the slowing down of file read times, because of the increased number of head seeks necessary while reading the file.

THE FILE CONTROL BLOCK (FCB)

The FCB is the heart of the FLEX File Management System (FMS). An FCB is a 320 byte long block of RAM, in the user's program area, which is used by programs to communicate with FMS. A separate FCB is needed for each open file. After a file has been closed, the FCB may be re-used to open another file or to perform some other disk function such as Delete or Rename. An FCB may be placed anywhere in the user's program area (except page zero) that the programmer wishes. The memory reserved for use as an FCB need not be preset or initialized in any way. Only the parameters necessary to perform the function need be stored in the FCB; the File Management System will initialize those areas of the FCB needed for its use.

In the following description of an FCB, the byte numbers are relative to the beginning of the FCB; i.e. byte 0 is the first byte of the FCB.

DESCRIPTION OF AN FCB

Byte 0 Function Code

    The desired function code must be stored in this byte by the user before calling FMS to process the FCB. See the section describing FMS Function Codes.

Byte 1 Error Status Byte

    If an error was detected during the processing of a function, FMS stores the error number in this byte and returns to the user with the CPU Z-Condition Code bit clear, i.e. a non-zero condition exists. This may be tested by the BEQ or BNE instruction.

Byte 2 Activity Status

    This byte is set by FMS to a "1" if the file is open for read, or "2" if the file is open for writing. This byte is checked by several FMS function processors to determine if the requested operation is legal. A Status Error is returned for illegal operations.

The next 12 bytes (3-14) comprise the "File Specification" of the file being referenced by the FCB. A "File Specification" consists of a drive number, file name, and file extension. Some of the FMS functions do not require the file name or extension. See the documentation of the individual function codes for details.

Byte 3 Drive Number

    This is the hardware drive number whose diskette contains the file being referenced. It should be binary 0 to 3.

The next 24 bytes (4-27) comprise the "Directory Information" portion of the FCB. This is the exact same information which is contained in the diskette directory entry for the file being referenced.

Bytes 4-11 File Name

       This is the name of the file being referenced. The name must start with a letter and contain only letters, digits, hyphens, and/or underscores. If the name is less than 8 characters long, the remaining bytes must be zero. The name should be left adjusted in its field.

Bytes 12-14 Extension

       This is the extension of the file name for the file being referenced. It must start with a letter and contain only letters, digits, hyphens, and/or underscores. If the extension is less than 3 characters long, the remaining bytes must be zero. The extension should be left adjusted. Files with null extensions should not be created.

Byte 15 File Attributes

       At present, only the most significant 4 bits are defined in this byte. These bits are used for the protection status bits and are assigned as follows:

              BIT 7 = Write Protect
              BIT 6 = Delete Protect
              BIT 5 = Read Protect
              BIT 4 = Catalog Protect

       Setting these bits to 1 will activate the appropriate protection status. All undefined bits of this byte should remain 0!

Byte 16 Reserved for future system use

Bytes 17-18 Starting disk address of the file

       These two bytes contain the hardware track and sector numbers, respectively, of the first sector of the file.

Bytes 19-20 Ending disk address of the file

       These two byes contain the hardware track and sector numbers, respectively, of the last sector of the file.

Bytes 21-22 File Size

       This is a 16-bit number indicating the number of sectors in the file.

Byte 23 File Sector Map Indicator

    If this byte is non-zero (usually $02), the file has been created
    as a random access file and contains a File Sector Map. See the
    description of Random Files for details.

Byte 24 Reserved for future system use

Bytes 25-27 File Creation Date

    These three bytes contain the binary date of the files creation.
    The first byte is the month, the second is the day, and the third
    is the year (only the tens and ones digits).

Bytes 28-29 FCB List Pointer

    All FCBs which are open for reading or writing are chained
    together. These two bytes contain the memory address of the FCB
    List Pointer bytes of the next FCB in the chain. These bytes are
    zero if this FCB is the last FCB in the chain. The first FCB in the
    chain is pointed to by the FCB Base Pointer. (See Global
    Variables).

Bytes 30-31 Current Position

    These bytes contain the hardware track and sector numbers,
    respectively, of the sector currently in the sector buffer portion
    of the FCB. If the file is being written, the sector to which these
    bytes point has not yet been written to the diskette; it is still
    in the buffer.

Bytes 32-33 Current Record Number

    These bytes contain the current logical Record Number of the sector
    in the FCB buffer.

Byte 34 Data Index

    This byte contains the address of the next data byte to be fetched
    from (if reading) or stored into (if writing) the sector buffer.
    This address is relative to the beginning of the sector, and is
    advanced automatically by the Read/Write Next Byte function. The
    user program has no need to manipulate this byte.

Byte 35 Random Index

    This byte is used in conjunction with the Get Random Byte From
    Sector function to read a specific byte from the sector buffer
    without having to sequentially skip over any intervening bytes. The
    address of the desired byte, relative to the beginning of the
    sector, is stored in Random Index by the user, and the Get Random
    Byte From Sector function is issued to FMS. The specified data byte
    will be returned in the A-register. A value less than 4 will

access one of the linkage bytes in the sector. User data starts at an index value of 4.

Bytes 36-46 Name Work Buffer

These bytes are used internally by FMS as temporary storage for a file name. These locations are not for use by a user program.

Bytes 47-49 Current Directory Address

If the FCB is being used to process directory information with the Get/Put Information Record functions, these three bytes contain the track number, sector number, and starting data index of the directory entry whose content is in the Directory Information portion of the FCB. The values in these three bytes are updated automatically by the Get Information Record function.

Bytes 50-52 First Deleted Directory Pointer

These bytes are used internally by FMS when looking for a free entry in the directory to which to assign the name of a new file.

Bytes 53-63 Scratch Bytes

These are the bytes into which the user stores the new name and extension of a file being renamed. The new name is formatted the same as described above under File Name and File Extension.

Byte 59 Space Compression Flag

If a file is open for read or write, this byte indicates if space compression is being performed. A value of zero indicates that space compression is to be done when reading or writing the data. This is the value that is stored by the Open for Read and Open for Write functions. A value of $FF indicates that no space compression is to be done. This value is what the user must store in this byte, after opening the file, if space compression is not desired. (Such as for binary files). A positive non-zero value in this byte indicates that space compression is currently in progress; the value being a count of the number of spaces processed thus far. (Note that although this byte overlaps the Scratch Bytes described above, there is no conflict since the Space Compression Flag is used only when a file is open, and the Scratch Bytes are used only by Rename, which requires that the file be closed). In general, this byte should be 0 while working with text type files, and $FF for binary files.

Bytes 64-319 Sector Buffer

These bytes contain the data contained in the sector being read or written. The first four bytes of the sector are used by the system. The remaining 252 are used for data storage.

FILE MANAGEMENT SYSTEM - Entry Points

$B400 - FMS Initialization

    This entry point is used by the DOS portion of FLEX to initialize
    the File Management System after a coldstart. There should be no
    need for a user-written program to use this entry point. Executing
    an FMS Initialization at the wrong time may result in the
    destruction of data files, necessitating a re-initialization of the
    diskette.

$B403 - FMS Close

    This entry point is used by the DOS portion of FLEX at the end of
    each command line to close any files left open by the command
    processor. User-written programs may also use this entry point to
    close all open files; however, if an error is detected in trying to
    close a file, any remaining files will not be closed. Thus the
    programmer is cautioned against using this routine as a substitute
    for the good programming practice of closing files individually.
    There are no arguments to this routine. It is entered by a JSR
    instruction as though it were a subroutine. On exit, the CPU Z-
    Condition code is set if no error was detected (i.e. a "zero"
    condition exists). If an error was detected, the CPU Z-Condition
    code bit is clear and the X-register contains the address of the
    FCB causing the error.

$B406 - FMS Call

    This entry point is used for all other calls to the File Management
    System. A function code is stored in the Function Code byte of the
    FCB, the address of the FCB is put in the X-register, and this
    entry point is called by a JSR instruction. The function codes are
    documented elsewhere in this document. On exit from this entry
    point, the CPU Z-Condition code bit is set if no error was detected
    in processing the function. This bit may be tested with a BEQ or
    BNE instruction. If an error was detected, the CPU Z-Condition code
    bit is cleared and the Error Status byte in the FCB contains the
    error number. Under all circumstances, the address of the FCB is
    still in the X-register on exit from this entry point. Some of the
    functions require additional parameters in the A and/or B-
    registers. See the documentation of the Function codes for details.
    The X and B registers are always preserved with a call to FMS.

GLOBAL VARIABLES

This section describes those variables within the File Management System which may be of interest to the programmer. Any other locations in the FMS area should not be used for data storage by user programs.

$B409 - $B40A FCB Base Pointer

These locations contain the address of the FCB List Pointer bytes of the first FCB in the chain of open files. The address in these locations is managed by FMS and the programmer should not store any values in these locations. A user program may, however, want to chain through the FCBs of the open files for some reason, and the address stored in these locations is the proper starting point. Remember that the address is that of the FCB List Pointer locations in the FCB, not the first word of the FCB. A value of zero in these locations indicates that there are no open files.

$B40B - $B40C Current FCB Address

These locations contain the address of the last FCB processed by the File Management System. The address is that of the first word of the FCB.

$B435 Verify Flag

A non-zero value in this location indicates that FMS will check each sector written for errors immediately after writing it. A zero value indicates that no error checking on writes is to be performed. The default value is "non-zero".

FMS FUNCTION CODES

The FLEX File Management System is utilized by the user through function
codes. The proper function code number is placed, by the user, in the
Function Code byte of the File Control Block (FCB) before calling FMS
(Byte 0). FMS should be called by a JSR to the "FMS Call" entry. On entry
to FMS, the X-register should contain the address of the FCB. On exit
from FMS, the CPU Z-Condition code bit will be clear if an error was
detected while processing the function. This bit may be tested by the BNE
and BEQ instructions. Note: In the following examples, the line "JSR FMS"
is referencing the FMS Call entry at $B406.

Function 0 - Read/Write Next Byte/Character

        If the file is open for reading, the next byte is fetched from the
        file and returned to the calling program in the A-register. If the
        file is open for writing, the content of the A-register on entry is
        placed in the buffer as the next byte to be written to the file.
        The Compression Mode Flag must contain the proper value for
        automatic space compression to take place, if desired (see
        Description of the FCB, Compression Mode Flag for details). On
        exit, this function code remains unchanged in the Function Code
        byte of the FCB; thus, consecutive read/writes may be performed
        without having to repeatedly store the function code. When reading,
        an End-of-File error is returned when all data in the file has been
        read. When the current sector being read is empty, the next sector
        in the file is prepared for processing automatically, without any
        action being required of the user. Similarly, when writing, full
        sectors are automatically written to the disk without user
        intervention.

        Example:

                If reading -
                LDX   #FCB   Point to the FCB
                JSR   FMS   Call FMS
                BNE   ERROR   Check for error
                The character read is now in A.

                If writing -
                LDAA   CHAR   Get the character
                LDX   #FCB   Point to the FCB
                JSR   FMS   Call FMS
                BNE   ERROR   Check for errors
                The character in A has been written

Function 1 - Open for Read

        The file specified in the FCB is opened for read-only access. If
        the file cannot be found, an error is returned. The only parts of
        the FCB which must be preset by the programmer before issuing this
        function are the file specification parts (drive number, file name,
        and file extension) and the function code. The remaining parts of
        the FCB will be initialized by the Open process. The Open process
        sets the File Compression Mode Flag to zero, indicating a text
        file. If the file is binary, the programmer should set the File
        Compression Mode Flag to $FF, after opening the file, to disable
        the space compression feature. On exit from FMS, after opening a
        file, the function code in the FCB is automatically set to zero
        (Read/Write Next Byte Function) in anticipation of I/O on the file.

        Example:

```
            LDX    #FCB     Point to the FCB
            [ Set up file spec in FCB ]
            LDAA   #1       Set open function code
            STAA   0,X      Store in FCB
            JSR    FMS      Call FMS
            BNE    ERROR    Check for errors
            The file is now open for text reading

            To set for binary - continue with the following
            LDAA   #$FF     Set FF for sup. flag
            STAA   59,X     Store in suppression flag
```

Function 2 - Open for Write

        This is the same as Function 1, Open for Read, except that the file
        must not already exist in the diskette directory, and it is opened
        for write-only access. A file opened for write may not be read
        unless it is first closed and then re-opened for read-only. The
        space compression flag should be treated the same as described in
        "Open for Read". A file is normally opened as a sequential file but
        may be created as a random file by setting the FCB location File
        Sector Map byte non-zero immediately following an open for write
        operation. Refer to the section on Random Files for more details.
        The file will be created on the drive specified unless the drive
        spec is $FF in which case the file will be created on the first
        drive found to be ready.

        Example:

```
            LDX    #FCB     Point to FCB
            [ Setup file spec in FCB ]
            LDAA   #2       Setup open for write code
            STAA   0,X      Store in FCB
            JSR    FMS      Call FMS
            BNE    ERROR    Check for errors
            File is now open for text write.
            For binary write, follow example in Read open.
```

Function 3 - Open for Update

    This function opens the file for both read and write. The file must
    not be open and must exist on the specified drive. If the drive
    spec is $FF, all drives will be searched. Once the file has been
    opened for update, four operations may be performed on it; 1.
    sequential read, 2. random read, 3. random write, and 4. close
    file. Note that it is not possible to do sequential writes to a
    file open for update. This implies that it is not possible to
    increase the size of a file which is open for update.

Function 4 - Close File

    If the file was opened for reading, a close merely removes the FCB
    from the chain of open files. If the file was opened for writing,
    any data remaining in the buffer is first written to the disk,
    padding with zeroes if necessary, to fill out the sector. If a file
    was opened for writing but never written upon, the name of the file
    is removed from the diskette directory since the file contains no
    data.

    Example:
        LDX   #FCB   Point to FCB
        LDAA   #4   Setup close code
        STAA   0,X   Store in FCB
        JSR   FMS   Call FMS
        BNE   ERROR   Check for errors
        File has now been closed.

Function 5 - Rewind File

    Only files which have been opened for read may be rewound. On exit
    from FMS, the function code in the FCB is set to zero, anticipating
    a read operation on the file. If the programmer wishes to rewind a
    file which is open for writing so that it may now be read, the file
    must first be closed, then re-opened for reading.

    Example:

        Assuming the file is open for read:
        LDX   #FCB   Point to FCB
        LDAA   #5   Setup rewind code
        STAA   0,X   Store in FCB
        JSR   FMS   Call FMS
        BNE   ERROR   Check for errors
        File is now rewound & ready for read

Function 6 - Open Directory

This function opens the directory on the diskette for access by a program. The FCB used for this function must not already be open for use by a file. On entry, the only information which must be preset in the FCB is the drive number; no file name is required. The directory entries are read by using the Get Information Record function. The Put Information Record function is used to write a directory entry. The normal Read/Write Next Byte function will not function correctly on an FCB which is opened for directory access. It is not necessary to close an FCB which has been opened for directory access after the directory manipulation is finished. The user should normally not need to access the directory.

Function 7 - Get Information Record

This function should only be issued on an FCB which has been opened with the Open Directory function. Each time the Get Information Record function is issued, the next directory entry will be loaded into the Directory Information area of the FCB (see Description of the FCB for details of the format of a directory entry). All directory entries, including deleted and unused entries are read when using this function. After an entry has been read, the FCB is said to "point" to the directory entry just read; the Current Directory Address bytes in the FCB refer to the entry just read. An End-of-File error is returned when the end of the directory is reached.

Example:

```
     To get the 3rd directory entry -
     LDX    #FCB   Point to FCB
     LDAA   DRIVE  Get the drive number
     STAA   3,X    Store in the FCB
     LDAA   #6     Setup open dir code
     STAA   0,X    Store in FCB
     JSR    FMS    Call FMS
     BNE    ERROR  Check for errors
     LDAB   #3     Set counter to 3
LOOP LDAA   #7     Setup get rec code
     STAA   0,X    Store in FCB
     JSR    FMS    Call FMS
     BNE    ERROR  Check for errors
     DECB          Decrement the counter
     BNE    LOOP   Repeat til finished
     The 3rd entry is now in the FCB
```

Function 8 - Put Information Record

>This function should only be issued on an FCB which has been opened
with the Open Directory function. The directory information is
copied from the Directory Information portion of the FCB into the
directory entry to which the FCB currently points. The directory
sector just updated is then re-written automatically on the
diskette to ensure that the directory is up-to-date. A user program
should normally never have to write into a directory. Careless use
of this function can lead to the destruction of data files,
necessitating a re-initialization of the diskette.

Function 9 - Read Single Sector

>This function is a low-level interface directly to the disk driver
which permits the reading of a single sector, to which the Current
Position bytes of the FCB point, into the Sector Buffer area of the
FCB. This function is normally used internally within FLEX and a
user program should never need to use it. The Read/Write Next Byte
function should be used instead, whenever possible. Extreme care
should be taken when using this function since it does not conform
to the usual conventions to which most of the other FLEX functions
adhere.

>Example:

```
LDX    #FCB    Point to FCB
LDAA   TRACK   Get track number
STAA   30,X    Set current track
LDAA   SECTOR  Get sector number
STAA   31,X    Set current sector
LDAA   #9      Setup function code
STAA   0,X     Store in FCB
JSR    FMS     Call FMS
BNE    ERROR   Check for errors
The sector is now in the FCB
```

Function 10 ($0A hex) - Write Single Sector

>This function, like the Read Single Sector function, is a low-level
interface directly to the disk driver which permits the writing of
a single sector. As such, it requires extreme care in its use. This
function is normally used internally by FLEX, and a user program
should never need to use it. The Read/Write Next Byte function
should be used whenever possible. Careless use of the Write Single
Sector Function may result in the destruction of data,
necessitating the re-initialization of the diskette. The disk
address being written is taken from the Current Position bytes of
the FCB; the data is taken from the FCB Sector Buffer. This
function honors the Verify Flag (see Global Variables section for a
description of the Verify Flag), and will check the sector after
writing it if directed to do so by the Verify Flag.

Function 11 ($0B hex) - Reserved for future system use

Function 12 ($0C hex) - Delete File

> This function deletes the file whose specification is in the FCB
> (drive numbers, file name, and extension). The sectors used by the
> file are released to the system for re-use. The file should not be
> open when this function is issued. The file specification in the
> FCB is altered during the delete process.

> Example:

```
        LDX    #FCB    Point to FCB
        [ setup file spec in FCB ]
        LDAA   #12     Setup function code
        STAA   0,X     Store in FCB
        JSR    FMS     Call FMS
        BNE    ERROR   Check errors
        File has now been deleted
```

Function 13 ($0D hex) - Rename File

> On entry, the file must not be open, the old name must be in the
> File Specification area of the FCB, and the new name and extension
> must be in the Scratch Bytes area of the FCB. The file whose
> specification is in the FCB is renamed to the name and extension
> stored in the FCB Scratch Bytes area. Both the new name and the new
> extension must be specified; neither the name nor the extension can
> be defaulted.

> Example:

```
        LDX  #FCB  Point to FCB
        [ setup both file specs in FCB ]
        LDAA  #13  Setup function code
        STAA 0,X   Store in FCB
        JSR  FMS   Call FMS
        BNE ERROR  Check for errors
        File has been renamed
```

Function 14 ($0E hex) - Reserved for future system use.

Function 15 ($0F hex) - Next Sequential Sector

On entry the file should be open for either reading or writing (not
update). If the file is open for reading, this function code will
cause all of the remaining (yet unread) data bytes in the current
sector to be skipped, and the data pointer will be positioned at
the first data byte of the next sequential sector of the file. If
the file is open for write, this operation will cause the remainder
of the current sector to be zero filled and written out to the
disk. The next character written to that file will be placed in the
first available data location in the next sequential sector. It
should be noted that all calls to this function code will be
ignored unless at least one byte of data has either been written or
read from the current sector.

Function 16 ($10 hex) - Open System Information Record

On entry, only the drive number need be specified in the FCB; there
is no file name associated with this function. The FCB must not be
open for use by a file. This function accesses the System
Information Record for the diskette whose drive number is in the
FCB. There are no separate functions for reading or changing this
sector. All references to the data contained in the System
Information Record must be made by manipulating the Sector Buffer
directly. This function is used internally within FLEX; there
should be no need for a user-written program to change the System
Information Record. Doing so may result in the destruction of data,
necessitating the re-initialization of the diskette. There is no
need to close the FCB when finished.

Function 17 ($11 hex) - Get Random Byte From Sector

On entry, the file should be open for reading or update. Also, the
desired byte's number should be stored in the Random Index byte of
the FCB. This byte number is relative to the beginning of the
sector buffer. On exit, the byte whose number is stored in the
Random Index is returned to the calling program in the A-register.
The Random Index should not be less than 4 since there is no user
data in the first four bytes of the sector.

Example:

```
To read the 54th data byte of the current sector -
LDX    #FCB    Point to the FCB
LDAA   #54+4   Set to item + 4
STAA   35,X    Put it in random index
LDAA   #17     Setup function code
STAA   0,X     Store in FCB
JSR    FMS     Call FMS
BNE    ERROR   Check for errors
Character is now in acc. A
```

Function 18 ($12 hex) - Put Random Byte in Sector

> The file must be open for update. This function is similar to Get
> Random Byte except the character in the A accumulator is written
> into the sector at the data location specified by Random Index of
> the FCB. The Random Index should not be less than 4 since only
> system data resides in the first 4 bytes of the sector.

> Example:

```
        To write into the 54th data byte of the current sector-
        LDX    #FCB   Point to the FCB
        LDAA   #54+4  Set to item + 4
        STAA   35,X   Put it in Random Index
        LDAA   #18    Setup Function Code
        STAA   0,X    Store in FCB
        LDAA   CHAR   Get character to be written
        JSR    FMS    Call FMS
        BNE    ERROR  Check for errors
        Character has been written
```

Function 19 ($13 hex) - Reserved for future system use

Function 20 ($14 hex) - Find Next Drive

> This function is used to find the next online drive which is in the
> "ready" state. Due to hardware limitations, the minifloppy versions
> of FLEX may perform this command differently than the full size
> floppy versions. The functioning of the full size versions is as
> follows. If the drive number in the FCB is hex FF, the search for
> drives will start with drive 0. If the drive number is 0, 1, or 2,
> the search will start with drive 1, 2, or 3 respectively. If a
> ready drive is found, its drive number will be returned in the
> drive number byte of the FCB and the zero bit in the condition
> codes will be set on return. If no ready drive is found, the zero
> bit will be cleared (implying a Not equal to Zero condition) and
> error #16 (Drives Not Ready) will be set.

> The minifloppy versions function as follows. If called with a Drive
> Number in the FCB of hex FF, the function will return with 0 as the
> drive number in the FCB. If called with a 0, it will return with
> the drive number set to 1. In both cases the zero bit in the
> condition codes would be set on return. If called with a drive
> number of 1 or higher, the drive number is left unchanged, the zero
> bit is cleared (implying a Not equal to Zero condition), and error
> #16 (Drives Not Ready) is set.

Function 21 ($15 hex) - Position to Record N

This is one of the 2 function codes provided for random file accessing by sector. The desired record number to be accessed should be stored in the FCB location Current Record Number (a 16 bit binary value). The file must be open for read or update before using this function code. The first data record of a file is record number one. Positioning to record 0 will read in the first sector of the File Sector Map. After a successful Position operation, the first character read with a sequential read will be the first data byte of the specified record. An attempt to position to a nonexistent record will cause an error. For more information on random files, see the section titled 'Random Files'.

Example:

```
To position to record #6 -
LDX    #FCB Point to the FCB
LDAA   #6 Set position
STAA   33,X Put in FCB
CLR    32,X Set M.S.B to 0
LDAA   #21 Setup Function Code
STAA   0,X Store in FCB
JSR    FMS Call FMS
BNE    ERROR Check for errors
Record ready to be read
```

Function 22 ($16 hex) - Backup One Record

This is also used for random file accessing. This function takes the Current Record Number in the FCB and decrements it by one. A Position to the new record is performed. This has the effect of back spacing one full record. For example, if the Current Record Number is 16 and the Backup One Record function is performed, the file would be positioned to read the first byte of record #15. The file must be open for read or update before this function may be used. See 'Random Files' section for more details.

RANDOM FILES

The FLEX disk operating system supports random files. The random access technique allows access by record number of a file and can reach any specified sector in a file, no matter how large it is, in a maximum of two disk reads. With a small calculation using the number of data bytes in a sector (252), the user may also easily reach the Nth character of a file using the same mechanism.

Not all files may be accessed in a random manner. It is necessary to create the file as a random file. The default creation mode is sequential and is what all of the standard FLEX Utilities work with. The only random file in a standard FLEX system is the ERRORS.SYS file. FLEX uses a random access technique when reporting error messages. A file which has been created as a random access file may read either randomly or sequentially. A sequential file may only be read sequentially.

To create a random file, the normal procedure for opening a file for write should be used. Immediately following a successful open, set the File Sector Map location of the FCB to any non-zero value and proceed with the file's creation. It only makes sense to create text type files in the random mode. As the file is built, the system creates a File Sector Map. This File Sector Map (FSM) is a map or directory which tells the system where each record (sector) of the file is located on the disk. The FSM is always two sectors in length and is assigned record number 0 in the file. This implies that a data file requiring 5 sectors for the data will actually be 7 sectors in length. The user has no need for the FSM sectors and they are automatically skipped when opening a file for read. The FMS uses them for the Position and Backup function code operations.

The directory information of a file states whether or not a file is a random file. If the File Sector Map byte is non-zero, the file is random, otherwise it is sequential only. It should be noted that random files can be Copied from one disk to another without losing its random properties, but it can not be appended to another file.

FLEX ERROR NUMBERS

1 - ILLEGAL FMS FUNCTION CODE ENCOUNTERED
      FMS was called with a function code in the Function Code byte of
      the FCB that was too large or illegal.

2 - THE REQUESTED FILE IS IN USE
      An Open for Read, Update, or Write function was issued on an FCB
      that is already open.

3 - THE FILE SPECIFIED ALREADY EXISTS

      a. An Open for Write was issued on an FCB containing the
      specification for a file already existing in the diskette
      directory.

      b. A Rename function was issued specifying a new name that was the
      same as the name of a file already existing in the diskette
      directory.

4 - THE SPECIFIED FILE COULD NOT BE FOUND

      An Open for Read or Update, a Rename, or a Delete function was
      requested on an FCB containing the file specification for a file
      which does not exist in the diskette directory.

5 - SYSTEM DIRECTORY ERROR - REBOOT SYSTEM
      Reserved for future system use.

6 - THE SYSTEM DIRECTORY SPACE IS FULL
      This error should never occur since the directory space is self
      expanding, and can never be filled. Only disk space can be filled
      (error #7).

7 - ALL AVAILABLE DISK SPACE HAS BEEN USED

      All of the available space on the diskette has been used up by
      files. If this error is returned by FMS, the last character sent to
      be written to a file did not actually get written.

8 - READ PAST END OF FILE
      A read operation on a file encountered an end-of-file. All of the
      data in the file has been processed. This error will also be
      returned when reading a directory with the Get Information Record
      function when the end of the directory is reached.

9 - DISK FILE READ ERROR
      A checksum error was encountered by the hardware in attempting to
      read a sector. DOS has already attempted to re-read the failing
      sector several times, without success, before reporting the error.
      This error may also result from illegal track and sector addresses
      being put in the FCB.

10 - DISK FILE WRITE ERROR
     A checksum error was detected by the hardware in attempting to
     write a sector. DOS has already tried several times, without
     success, to re-write the failing sector before reporting the error.
     This error may also result from illegal track and sector numbers
     being put in the FCB. A write-error status may also be returned if
     a read error was detected by DOS in attempting to update the
     diskette directory.

11 - THE FILE OR DISK IS WRITE PROTECTED
     An attempt was made to write on a diskette which has been write-
     protected by use of the write-enable cutout in the diskette or to a
     file which has the write protect bit set.

12 - THE FILE IS PROTECTED - FILE NOT DELETED
     The file attempted to be deleted has its delete protect bit set and
     can not be deleted.

13 - ILLEGAL FILE CONTROL BLOCK SPECIFIED
     An attempt was made to access an FCB from the open FCB chain, but
     it was not in the chain.

14 - ILLEGAL DISK ADDRESS ENCOUNTERED
     Reserved for future system use.

15 - AN ILLEGAL DRIVE NUMBER WAS SPECIFIED
     Reserved for future system use.

16 - DRIVES NOT READY
     The drive does not have a diskette in it or the door is open. This
     message cannot be issued for mini floppys since there is no means
     of detecting such a state.

17 - THE FILE IS PROTECTED - ACCESS DENIED
     Reserved for future system use.

18 - SYSTEM FILE STATUS ERROR
     a. A read or Rewind was attempted on a file which was closed, or
     open for write access.
     b. A write was attempted on a file which was closed, or open for
     read access.

19 - FMS DATA INDEX RANGE ERROR
     The Get Random Byte from Sector function was issued with a Random
     Byte number greater than 255.

20 - FMS INACTIVE - REBOOT SYSTEM
     Reserved for future system use.

21 - ILLEGAL FILE SPECIFICATION
     A format error was detected in a file name specification. The name
     must begin with a letter and contain only letters, digits, hyphens,
     and/or underscores. Similarly with file extensions. File names are
     limited to 8 characters, extensions to 3.

22 - SYSTEM FILE CLOSE ERROR
     Reserved for future system use.

23 - SECTOR MAP OVERFLOW - DISK TOO SEGMENTED
     An attempt was made to create a very large random access file on a
     disk which is very segmented. All record information could not fit
     in the 2 sectors of the File Sector Map. Recreating the file on a
     new diskette will solve the problem.

24 - NON-EXISTENT RECORD NUMBER SPECIFIED
     A record number larger than the last record number of the file was
     specified in a random position access.

25 - RECORD NUMBER MATCH ERROR - FILE DAMAGED
     The record located by the FMS random search is not the correct
     record. The file is probably damaged.

26 - COMMAND SYNTAX ERROR - RETYPE COMMAND
     The command line just typed has a syntax error.

27 - THAT COMMAND IS NOT ALLOWED WHILE PRINTING
     The command just entered is not allowed to operate while the system
     printer spooler is activated.

28 - WRONG HARDWARE CONFIGURATION
     This error usually implies insufficient memory installed in the
     computer for a particular function or trying to use the printer
     spooler without the hardware timer board installed.

                              DISK DRIVERS

The following information is for those users who wish to write their own
disk drivers to interface with some other disk configuration than is
supplied by the vendor. Neither the vendor nor Technical Systems
Consultants is in a position to write disk drivers for other
configurations, nor do these companies guarantee the proper functioning
of FLEX with user-written drivers.

The disk drivers are the interface routines between FLEX and the hardware
driving the floppy disks themselves. The drivers released with the FLEX
System are designed to interface with the Western Digital 1771 Floppy
Disk Formatter/Controller chip.

The disk drivers are located in RAM at addresses $BE80 - $BFFF. All disk
functions are vectored jumps at the beginning of this area. The disk
drivers need not handle retries in case of errors; FLEX will call them as
needed. If an error is detected, the routines should exit with the disk
hardware status in the B-register and the CPU Z-Condition code bit clear
(issue a TST B before returning to accomplish this). FLEX expects status
responses as produced by the Western Digital 1771 Controller. These
statuses must be simulated if some other controller is used. All drivers
should return with the X-register unchanged. All routines are entered
with a JSR instruction.

The following set of routines is common to all versions of 6800 FLEX. The
SWTPc FLEX 2.0 version has only these routines, while some versions have
more as specified elsewhere.

$BE80 - Read
      Entry - (X) = FCB Sector Buffer Address
              (A) = Track Number
              (B) = Sector Number
      The sector referenced by the track and sector numbers is to be read
      into the Sector Buffer area of the indicated FCB.

$BE83 - Write
      Entry - (X) = FCB Sector Buffer Address
              (A) = Track Number
              (B) = Sector Number
      The content of the Sector Buffer area of the indicated FCB is to be
      written to the sector referenced by the track and sector numbers.

$BE86 - Verify
      Entry - (No parameters)
      The sector just written is to be verified to determine if there are
      CRC errors.

$BE89 - Restore
      Entry - (X) = FCB Address
      Exit - CC, NE, & B=$B if write protected
              CS, NE, & B=$F if no drive
      A Restore Operation (also known as a Seek to Track 00) is to be

performed on the drive whose number is in the FCB.

$BE8C - Drive Select
      Entry - (X) = FCB Address
      The drive whose number is in the FCB is to be selected.

$BE8F - Check Drive Ready
      Entry - (X) = FCB Address
      Exit - NE & CS if drive not ready
              EQ & CC if drive ready
      This routine is setup for FLEX systems where it is possible to
      check the drive whose number is in the FCB for a ready status after
      selecting that drive and delaying long enough for the drive motor
      to come up to speed (approx. 2 seconds). In the FLEX 2.0 versions
      for SWTPc and SSB, this is not possible due to hardware
      limitations. In these cases, this routine should not delay and
      should simply return a drive ready status if the drive number in
      the FCB is 0 or 1 or a drive not ready status for any other drive
      number.

$BE92 - Quick Check Drive Ready
      This routine is the same as Drive Check Ready except the 2 second
      delay is not done. This assumes the drive motor is already up to
      speed. For FLEX 2.0 versions, there is no difference in the two and
      this routine can simply be a jump to the Check Drive Ready routine.

Diskette Initialization

The NEWDISK command is used to "initialize" a diskette for use by the FLEX Operating System. The initialization process writes the necessary track and sector addresses in the sectors of a "soft-sectored" diskette such as is used by FLEX. In addition, the initialization process links together all of the sectors on the diskette into a chain of available sectors.

The first track on the diskette, track 0, is special. None of the sectors on track 0 are available for data files, they are reserved for use by the FLEX system. The first two sectors contain a "boot" program which is loaded by the "D" command of the SWTBUG monitor or by whatever comparable ROM based bootstrap is in use. The boot program, once loaded, then loads FLEX from the diskette. Another sector on track 0 is the System Information Record. This sector contains the track and sector addresses of the beginning and ending sectors of the chain of free sectors, those available for data files. The rest of track 0 is used for the directory of file names.

After initialization, the free tracks on the diskette have a common format. The first two bytes of each sector contain the track and sector number of the next sector in the chain. The next two bytes are used to store the logical record number of the sector in the file. The remaining 252 bytes are zero. Initially, all record number bytes are zero. When data is stored in a file, the two linkage bytes at the beginning of each sector are modified to point to the next sector in the file, not the next sector in the free chain. The sectors in the diskette directory on track 0 also have linkage bytes similar to those in the free chain and data files.

A FLEX diskette is not initialized in the strict IBM standard format. In the standard format, the sectors on the diskette should be physically in the same order as they are logically, i.e. sector 2 should follow sector 1, 3 follow 2, etc. On a FLEX diskette, the sectors are interleaved so that there is time, after having read one sector, to process the data and request the next sector before it has passed under the head. If the sectors are physically adjacent, the processing time must be very short. The interleaving of the sectors allows more time for processing the data. The phenomena of missing a sector because of long processing times is called "missing revolutions", and results in very slow running time for programs. The FLEX format reduces the number of missed revolutions, thus speeding up programs.

DESCRIPTION OF A DIRECTORY SECTOR

Each sector in the directory portion of a FLEX diskette contains 10 directory entries. Each entry refers to one file on the diskette. In each sector, the first four bytes contain the sector linkage information and the next 12 bytes are not used. When reading information from the directory using the FMS Get Information Record function, these 16 bytes are skipped automatically as each sector is read; the user need not be concerned with them.

Each entry in the directory contains the exact same information that is stored in the FCB bytes 4-27. See the description of the File Control Block (FCB) for more details.

A directory entry which has never been used has a zero in the first byte of the file name. A directory entry which has been deleted has the leftmost bit of the name set (i.e. the first byte of the name is negative).

DESCRIPTION OF A DATA SECTOR

Every sector on a FLEX diskette (except the two BOOT sectors) has the following format:
      Bytes 0-1 Link to the next sector
      Bytes 2-3 File Logical Record Number
      Bytes 4-255 Data

If a file occupies more than one sector, the "link to the next sector" portion contains the track and sector numbers, respectively, of the next sector in the file. These bytes are zero in the last sector of a file, indicating that no more data follows (an "end-of-file" condition). The user should never manually change the linkage bytes of a sector. These bytes are automatically managed by FMS. In fact, the user need not be concerned at all with sector linkage information.

DESCRIPTION OF A BINARY FILE

A FLEX binary file may contain anything as data; all ASCII characters are
allowed. Each binary file is composed of one or more binary records.
There may be more than one binary record in a single sector.

A binary record looks as follows: (byte numbers are relative to the start
of the record, not the beginning of a sector)

        Byte 0 Start of record indicator ($02, the ASCII STX)
        Byte 1 Most significant byte of the load address
        Byte 2 Least significant byte of the load address
        Byte 3 Number of data bytes in the record
        Byte 4-n The binary data in the record

The load address portion of a binary record contains the address where
the data resided when it was written to the file with the FLEX SAVE
command. When the file is loaded for execution or use, it will be put in
the same memory areas from which it was SAVED.

A binary file may also contain an optional transfer address record. This
record gives the address in memory of the entry point of a binary
program. The format of a transfer address record is as follows:

        Byte 0 Transfer Address Indicator ($16, ASCII ACK)
        Byte 1 Most significant byte of the transfer address
        Byte 2 Least significant byte of the transfer address

If a file contains more than one transfer address record (caused by
appending binary files which contain transfer addresses), the last one
encountered by the load process is the one that is used, the others are
ignored.

When reading or writing a binary file through the File Management System
from a user program, the calling program must process the record
indicator bytes and load addresses itself; FLEX does not supply or
process this information for the user.

DESCRIPTION OF A TEXT FILE

A text file (also called an "ASCII file" or "coded file") contains only printable ASCII characters plus a few special-purpose control characters. There is no "load address" associated with a FLEX text file as there is with FLEX binary files. It is the responsibility of the program which is reading the text file to put the data where it belongs.

The only control character which FLEX recognizes and processes in a FLEX text file are:

$0D (ASCII CR or RETURN)
     This character is used to mark the end of a line or record in the file.

$00 (ASCII NULL)
     Ignored by FLEX; if encountered in the file, it is not returned to the calling program.

$18 (ASCII CANCEL)
     Ignored by FLEX; if encountered in the file, it is not returned to the calling program.

$09 (ASCII HT or HORIZONTAL TAB)
     This is a flag character which indicates that a string of spaces has been removed from the file as a space-saving measure. The next byte following the flag character is a count of the number of space removed (2-127). The calling program sees neither the flag character nor the count character. The proper number of spaces are returned to the user program as successive characters are requested by the Read Next Byte function. When writing a file, the spaces are automatically deleted as the user program sends them to the File Management System using the Write Next Byte function. The data compression is, therefore, transparent to the calling program.(The above discussion is only valid if the file is open for Text operations. If open for Binary, the compression flag and count get passed exactly as they appear in the file.)

WRITING UTILITY COMMANDS

Utility commands are best prepared by the use of an assembler. FLEX reserves a block of memory in which medium size utilities may be placed. This memory starts at hex location $A100 and extends through location $A6FF. The system FCB at location $A840 may also be used in user written utilities for either FCB space or temporary storage. No actual code should reside in this FCB space since it would interfere with the loading of the utility (FLEX is using that FCB while loading utilities).

An example will be given to demonstrate some of the conventions and techniques which should be used when writing utilities. The example, which can be found on the following pages, is a simple text file listing utility. Its syntax is:

    LIST,[<FILE SPEC>]

The default extension on the file spec is TXT. The utility will simply display the contents of a text file on the terminal, line for line.

The following is a section by section description of the LIST utility. The first section of the source listing is a set of EQUATES which tell the assembler where the various DOS routines reside in memory. These equates represent the addresses given in this manual for "User Callable DOS System Routines".

The next two sections are also equates, the first to the FMS entry points, and the second references the system FCB. The actual program finally starts with the ORG statement. In this program, we will make use of the Utility Command space located at $A100, therefore, the ORG is to $A100.

One of the conventions which should be observed when writing DOS utilities is to always start the program with a BRA instruction. Following this instruction should be a 'VN FCB 1' which defines the version number of the utility. The 1 should of course be set to whatever the actual version number is. In this example, the version number is 1. This convention allows the FLEX VERSION Utility to correctly identify the version number of a command.

Moving down the program to the label called 'LIST2', the program needs to retrieve the file specification and get it into the FCB. Pointing X to the FCB, we can make use of the DOS resident subroutine called 'GETFIL' to automatically parse the file spec, check for errors, and set the name in the FCB correctly. If all goes well in GETFIL, the carry should be clear, otherwise there were errors in the file spec and this fact needs reported. If the carry is set, control is passed to the line with the label 'LIST9'. At this point, the error message is reported and control is returned to FLEX.

If the file spec was correct, and the carry was clear after the return from GETFIL, we want to set a default file name extension of TXT. The DOS subroutine named SETEXT will do exactly that. First it is necessary

to put the code for TXT in the A accumulator (the code is 1). X needs to be pointing to the FCB which it still is. The '1' is also put in the FCB for the future open operation. The call is made to SETEXT and the file name is now correctly set up in the FCB. Note that no errors can be generated by a call to SETEXT.

Now that we have the file spec, it is necessary to open the requested file for read. X is still pointing to the FCB so it is not necessary to reset. The FMS Function Code for 'open a file for read' is 1 which was previously put in the FCB location 0. A call to FMS is now made in an attempt to open the file. Upon return, if the Z-condition code is set, there were no errors. If there was an error, the 'BNE LIST9' will take us to the code to report the error. This section of code is the desired way to handle most FMS caused disk errors. The first thing to do is call the DOS routine RPTERR which will print the disk error message on the monitor terminal. Next, all open disk files should be closed. This can be easily accomplished by a call to the FMS close entry (FMSCLS). Finally, return control back to DOS by jumping to the WARM START entry. If the file opened successfully, control will be transferred to the line with the label 'LIST4'. At this time it is desirable to fetch characters one at a time from the file, printing them on the monitor terminal as they are received. Since line feeds are not stored in text files (carriage returns mark the end of lines, but the next line will follow immediately), each carriage return received from the file is not output as is, but instead a call to the DOS routine 'PCRLF' is made to print a carriage return and a line feed. As each character is received from the file (by a call to FMS at label LIST4), the error status is checked. If an error does occur, control is transferred to "LIST6". Since FLEX does not store an End of File character with a file, the only mechanism for determining the end of a file is by the End of File error generated by FMS. At 'LIST6', the error status is checked to see if it is 8 (end of file status). If it is not an 8, control is transferred to the error handling routine described above. If it is an End of File, we are finished listing the file so it must now be closed. The FMS Function Code for closing a file is 4. This is loaded into A and stored in the FCB. Calling FMS will attempt to close the file. Upon return, errors are checked, and if none found, control is transferred back to DOS by the jump to 'WARMS'.

This example illustrates many of the methods used when writing utilities. Many of the DOS and FMS routines were used. The basic idea of file opening and closing were demonstrated, as well as file I/O. The methods of dealing with various types of errors were also presented. Studying this example until it is thoroughly understood will make writing your own disk commands and disk oriented programs an easy task.

```
                    *
                    * SIMPLE TEXT FILE LIST UTILITY
                    *

                    * COPYRIGHT (C) 1978 BY
                    *
                    * TECHNICAL SYSTEMS CONSULTANTS, INC.

                    * DOS EQUATES

AD03                WARMS   EQU     $AD03           DOS WARMS START ENTRY
AD2D                GETFIL  EQU     $AD2D           GET FILE SPECIFICATION
AD18                PUTCHR  EQU     $AD18           PUT CHARACTER ROUTINE
AD24                PCRLF   EQU     $AD24           PRINT CR & LF
AD33                SETEXT  EQU     $AD33           SET DEFAULT NAME EXT
AD3F                RPTERR  EQU     $AD3F           REPORT DISK ERROR

                    * FMS EQUATES

B406                FMS     EQU     $B406
B403                FMSCLS  EQU     $B403

                    * SYSTEM EQUATES

A840                FCB     EQU     $A840           SYSTEM FCB

                    * LIST UTILITY STARTS HERE

A100                        ORG     $A100

A100 20 01          LIST    BRA     LIST2           GET AROUND TEMPS

A102 01             VN      FCB     1               VERSION NUMBER

A103 CE A8 40       LIST2   LDX     #FCB            POINT TO FCB
A106 BD AD 2D               JSR     GETFIL          GET FILE SPEC
A109 25 34                  BCS     LIST9           ANY ERRORS?
A10B 86 01                  LDA A   #1              SET UP CODE
A10D A7 00                  STA A   0,X             SAVE FOR READ OPEN
A10F BD AD 33               JSR     SETEXT          SET TXT EXTENSION
A112 BD B4 06               JSR     FMS             CALL FMS - DO OPEN
A115 26 28                  BNE     LIST9           CHECK FOR ERROR
A117 CE A8 40       LIST4   LDX     #FCB            POINT TO FCB
A11A BD B4 06               JSR     FMS             CALL FMS - GET CHAR
A11D 26 0E                  BNE     LIST6           ERRORS?
A11F 81 0D                  CMP A   #$D             IS CHAR A CR?
A121 26 05                  BNE     LIST5
A123 BD AD 24               JSR     PCRLF           OUTPUT CR & LF
A126 20 EF                  BRA     LIST4           REPEAT
A128 BD AD 18       LIST5   JSR     PUTCHR          OUTPUT THE CHARACTER
A12B 20 EA                  BRA     LIST4           REPEAT SEQUENCE
```

```
A12D A6 01     LIST6    LDA A  1,X           GET ERROR STATUS
A12F 81 08              CMP A  #8            IS IT EOF ERROR?
A131 26 0C              BNE    LIST9
A133 86 04              LDA A  #4            CLOSE FILE CODE
A135 A7 00              STA A  0,X           STORE IN FCB
A137 BD B4 06           JSR    FMS           CALL FMS - CLOSE FILE
A13A 26 03              BNE    LIST9         ERRORS?
A13C 7E AD 03           JMP    WARMS         RETURN TO FLEX

A13F BD AD 3F  LIST9    JSR    RPTERR        REPORT ERROR
A142 BD B4 03           JSR    FMSCLS        CLOSE ALL FILES
A145 7E AD 03           JMP    WARMS         RETURN TO FLEX

                        END    LIST
```

THE DOS LINK UTILITY

The LINK Utility provided with FLEX is a special purpose command. Its only function is to inform the "disk boot", which is on track 0, where the program resides which is to be loaded during the boot operation. Normally, LINK is used to set the pointer to the DOS program. Since DOS may reside anywhere on the disk, LINK takes the starting disk address of the file and stores it in a pointer in the boot sector. When the boot program is later executed, it simply takes this disk address, and loads the binary file which resides at that location. The load process is terminated upon the receipt of a transfer address record. At this time, control is transferred to the program just loaded by jumping to the address specified in the transfer address record. If the 'linked' program is ever moved on the disk, then it must be re-linked so the boot knows the new disk address.

LINK may be used in some specialized applications. One is the development of custom operating systems. The user may write his own operating system, link it to the boot, and use it exactly as FLEX is used now. It may also be desirable for special disks to boot in specialized programs rather than the operating system. If this is done, remember that unless the DOS is loaded during the boot process, there will not be any disk drivers or File Management System resident in memory.

PRINTER ROUTINES

There are two printer related programs provided with FLEX. One is the P Utility, the other is the PRINT.SYS file which is the actual set of printer drivers (initialize printer and output character). The P command source listing is provided on the following pages and should be self explanatory. Below you will find the requirements of the PRINT.SYS file. No source listing is provided here since one is given in the "FLEX User's Manual".

"PRINT.SYS' FILE REQUIREMENTS

The PRINT.SYS file needs to provide the system with three basic printer routines, one for printer port initialization, one for printer status, and one for output character to printer routine. The P routine and the system printer spooler use these routines to communicate with the printer. A source listing of the provided routines are included in the "FLEX User's Manual" and will not be duplicated here. The three routines and their requirements are listed here.

PINIT   ($ACC0-ACD7) This routine should initialize the printer port.
         No registers need preserved.

PCHK   ($ACD8-ACE3) This routine should check to see if the printer can
         accept another character. Return Negative CC status if can
         accept, Plus if can not. Preserve A, B, and X.

POUT   ($ACE4-ACF7) This routine should output the character in A after
         calling PCHK to verify the printer can accept the character.
         Preserve B and X.

THE SYSTEM PRINTER SPOOLER

FLEX contains a printer spooler module. It requires the installation of an interval timer board for operation. Essentially, the spooler is a multi-tasking system, with the output to printer function being a low priority task. Any requested disk service will cause the printer task to temporarily halt until the disk has been used. It should be noted that the SWI CPU vector is adjusted in this task scheduler. The PRINT command is used to activate the spooler which in turn prints the files (if any) in the print queue. Exact details of the spooling operation are not available at this time.

```
                          *
                          * "P" UTILITY COMMAND
                          *
                          * THE P COMMAND INITIALIZES A PORT AND
                          * CHANGES THE OUTCH JUMP VECTOR IN FLEX
                          *

                          * COPYRIGHT (C) 1978 BY
                          *
                          * TECHNICAL SYSTEMS CONSULTANTS, INC.

                          * EQUATES

  0010            INDEX   EQU     $0010
  A840            FCB     EQU     $A840
  AD30            LOAD    EQU     $AD30
  B406            FMS     EQU     $B406
  B403            FMSCLS  EQU     $B403
  AD06            RENTER  EQU     $AD06
  0004            NFER    EQU     $4
  AC09            PAUSE   EQU     $AC09
  AD1E            PSTRNG  EQU     $AD1E
  AD3F            RPTERR  EQU     $AD3F
  AD03            WARMS   EQU     $AD03
  AC11            LSTTRM  EQU     $AC11
  AC02            EOL     EQU     $AC02
  ACC0            PINIT   EQU     $ACC0
  ACE4            POUT    EQU     $ACE4
  AD0F            OUTCH   EQU     $AD0F
  ACFC            PR1     EQU     $ACFC


  A100                    ORG     $A100


  A100 20 01      P       BRA     P1        BRANCH AROUND TEMPS


  A102 01         VN      FCB     1         VERSION NUMBER


  A103 B6 AC FC   P1      LDA A   PR1       CHECK SYSTEM PROCESS REG
  A106 27 09              BEQ     P12       IS IT BUSY?
  A108 CE A8 40           LDX     #FCB      POINT TO FCB
  A10B C6 1B             LDA B   #27       SET BUSY ERROR
  A10D E7 01              STA B   1,X       STUFF IN FCB
  A10F 20 43              BRA     P3        GO REPORT ERROR
  A111 B6 AC 11   P12     LDA A   LSTTRM    GET LAST TERMINATOR
  A114 81 0D              CMP A   #$D       IS IT A CR?
  A116 27 45              BEQ     P8
  A118 B1 AC 02           CMP A   EOL       IS IT EOL CHARACTER
  A11B 27 40              BEQ     P8
  A11D 7F AC 09           CLR     PAUSE     DISABLE THE PAUSE FEATURE
```

- continued -

```
A120 B6 AC E4              LDA A   POUT        GET 1ST BYTE OF SPACE
A123 81 39                 CMP A   #$39        IS IT RTS?
A125 26 13                 BNE     P15         IF NOT - THEN LOADED
A127 CE A8 40              LDX     #FCB        POINT TO FCB
A12A 86 01                 LDA A   #1          OPEN FILE FOR READ
A12C A7 00                 STA A   0,X
A12E BD B4 06              JSR     FMS         CALL FMS
A131 26 13                 BNE     P2          CHECK FOR ERRORS
A133 86 FF                 LDA A   #$FF        SET FOR BINARY READ
A135 A7 3B                 STA A   $3B,X       SET COMPRESSION FLAG
A137 BD AD 30              JSR     LOAD        CALL FLEX'S LOADER
A13A BD AC C0  P15         JSR     PINIT       GO INITIALIZE PORT
A13D CE AC E4              LDX     #POUT       GET OUTPUT ADDRESS
A140 FF AD 10              STX     OUTCH+1     STUFF IN FLEX
A143 7E AD 06              JMP     RENTER      RETURN TO FLEX

A146 A6 01     P2          LDA A   1,X         GET ERROR CODE
A148 81 04                 CMP A   #NFER       IS IT "NO SUCH FILE"?
A14A 26 08                 BNE     P3
A14C CE A1 62              LDX     #NOPST      POINT TO MESSAGE
A14F BD AD 1E  P25         JSR     PSTRNG      GO PRINT IT
A152 20 03                 BRA     P4

A154 BD AD 3F  P3          JSR     RPTERR      REPORT ERROR
A157 BD B4 03  P4          JSR     FMSCLS      CLOSE ALL FILES
A15A 7E AD 03              JMP     WARMS       RETURN TO FLEX

A15D CE A1 78  P8          LDX     #ERSTR      POINT TO STRING
A160 20 ED                 BRA     P25         GO PRINT IT

A162 22        NOPST       FCC     '"PRINT.SYS" NOT FOUND'
A177 04                    FCB     4
A178 22        ERSTR       FCC     '"P" MUST BE FOLLOWED BY A COMMAND'
A199 04                    FCB     4

               * THE FOLLOWING CODE IS LOADED INTO
               * THE SYSTEM FCB WHEN THE P COMMAND IS
               * LOADED INTO MEMORY.
               * IT PRESETS THE FILE NAME IN THE FCB.

A843                       ORG     $A843

A843 FF                    FCB     $FF
A844 50                    FCC     'PRINT'
A849 00                    FCB     0,0,0
A84C 53                    FCC     'SYS'

                           END     P
```

INTERRUPTS IN FLEX

FLEX makes extensive use of interrupts during printer spooling. Anytime there are files in the PRINT Queue (as a result of using the PRINT command) the timer is activated (MP-T board in I/O slot #4 -in SWTPc compatible systems). This timer is initialized to output interrupts every 10 milliseconds. These are IRQ type interrupts and FLEX sets the IRQ vector to point to its IRQ routine. When the PRINT Queue is empty, the timer is shut off and no interrupts are generated. The SWI instruction is also used quite extensively in FLEX. The SWI vector in RAM is set by FLEX to point to its SWI routine. Because of the SWI and IRQ use, the MON command will not permit leaving FLEX while there is a file in the PRINT Queue.

All FLEX utilities, the Editor, the Assembler, the Text Processor, and BASIC are interruptable programs. When writing your own programs, if they are to be used while printing with the PRINT command (files in the print queue), they should be written to be interruptable as well. At no time should the IRQ or SWI vectors be changed in a utility which is to be run while printing. In general, good programming practice will yield interruptable programs.

SYSTEM MEMORY MAP

The following memory map shows the location of user RAM and several major sections of the FLEX operating system. All addresses are in hexadecimal.

```
ADDRESS         DESCRIPTION

0000 - 7FFF     User RAM (Some of the lower end of this area is used
                      by certain utilities such as NEWDISK.)
A000 - A07F     Stack Area (SP is initialized to A07F)
A080 - A0FF     Input Buffer
A100 - A6FF     Utility Command Area
A700 - A83F     Scheduler & Printer Spooler
A840 - A97F     System FCB
A980 - ABFF     System Files Area
AC00 - B3FF     DOS
B400 - BE7F     FMS
BE80 - BFFF     Disk Drivers
```