MINIFLEX© VER. 1.Ø

ADVANCED PROGRAMMER'S GUIDE

SOUTHWEST TECHNICAL PRODUCTS CORPORATION
219 W. RHAPSODY — SAN ANTONIO, TEXAS 78216

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 the mini-floppy version of FLEX (sometimes referenced as "mini-FLEX"). Most of the features of the larger versions of FLEX are fully operational on the mini version. 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.

Copyright Notice

Disclaimer

This product is intended for use only as described in this document and the FLEX User's Guide. Technical Systems Consultants cannot be responsible for the proper functioning of undocumented features or parameters. The user is also urged to abide by the warnings and cautions issued in this document lest valuable data or diskettes be destroyed.

Important Note

Although every effort has been made to make the supplied software and its documentation as accurate and functional as possible, Southwest Technical Products Corporation and Technical Systems Consultants will not assume responsibility for any damages incurred or generated by such material. Also, Southwest Technical Products Corporation and Technical Systems Consultants reserve the right to make changes in such material at any time.

Patching "FLEX"

It is not possible to patch FLEX. Technical Systems Consultants cannot be responsible for any destructive side-effects which may result from attempts to patch FLEX.

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 putting tape over the write-enable cutout on the diskette. 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 communciation 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.

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.

$7000 - $707F -- 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 back-space character do not appear in the buffer, nor does the backspace character itself appear. The carriage return signaling the end of the keyboard input is, however, put in the buffer. This buffer is also used to hold the STARTUP file during a coldstart (boot) operation.

$7080 -- 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).

$7081 -- TTYSET Delete Character
This is the character DOS recognizes as the entry delete character. It is user definable thru the TTYSET utility. Default =$18, a control X.

$7082 -- 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 (:).

$7083 -- 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.

$7084 -- 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.

$7085 -- 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.

$7086 - 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.

$7087 -- TTYSET Duplex Transmission Mode
A zero value indicates Full Duplex; a non-zero value, Half Duplex. Default = 0, Full Duplex.

$7088 -- 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

$7089 -- 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.

$708A -- TTYSET Escape Character
The Escape character causes DOS to pause after an output line. Default = $1B, ASCII ESC.

$708B -- System Drive Number
This is the number of the disk drive from which commands are loaded. Default = 0.

$708C -- Working Drive Number
This is the number of the default disk drive referenced for non-command files. Default = 0.

$708D - $7090 -- Reserved for system scratch and future system use.

$7091 -- 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".

$7092 - $7093 -- 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.

$7094 - $7095 -- 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.

$7096 - $7097 -- 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".

$7098 - $7099 -- System Scratch

$709A -- Previous 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.

$709B -- 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.

$709C - $709D -- 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.

$709E -- 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.

$709F - $70A0 -- 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.

$70A1 -- ACIA Flag

This location is non-zero if the console terminal port (Port 1) has an MP-S ACIA serial interface. Zero in this location implies that a MP-C control Interface (PIA driven) is in that port.

$70A2 -- 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.

$70A3 -- Output Switch

If zero, output performed by the PUTCHR routine is through the routine OUTCH. If nonzero, the routine OUTCH2 is used. See documentation of these routines for details.

$70A4 -- System Scratch

$70A5 -- 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.

$70A6 -- 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.

$70A7 - $70B4 -- System Scratch

$70B5 - $70FF -- System Constants

User-Callable System Routines

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

$7100 (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.

$7103 (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 is reset to $A07F, the monitor (SWTBUG® /MIKBUG®) program counter ($A048) is reset to $7103, as well as the Escape Return Register. 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.

$7106 (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 the "Printer Driver" section for details.

$7109 (INCH) Input Character

This routine inputs one character from the keyboard, returning it to the calling program in the A-register. The address portion of this entry point is set to the SWTBUG /MIKBUG Input Character routine. It is not possible to patch this address to refer to some other routine. User programs should use the GETCHR routine, documented below, rather than calling INCH, because INCH does not check the TTYSET parameters. Registers B and X are preserved.

$710C (OUTCH) Output Character
$7136 (OUTCH2) Output Charcter

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 prin-ter. OUTCH2 is never changed, and always points to the SWTBUG®/MIKBUG® Output Cha-racter routine. This address' may not be patched to refer to some other output routine. 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 re-sets the OUTCH jump vector to the same routine as OUTCH2, and sets the Output Switch to zero. User routines should use PUTCHR rather than calling OUTCH or OUTCH2 directly since the latter two do not check the TTYSET parameters. Registers B and X are preserved.

$710F (GETCHR) Get Character

This routine gets a single character from the keyboard, echoing it back via PUTCHR, if necessary, in accordance with the TTYSET parameters. 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. Registers B and X are preserved.

$7112 (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. 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 Swtich" is non-zero, the routine OUTCH2 is used to send the character. 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. Registers B and X are preserved.

$7115 (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, are ignored. The RETURN is placed in the buffer at the end of the 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 sideeffects.

$7118 (PSTRNG) Print String

This routine is similar to the PDATA routine in SWTBUG® and MIKBUG® . 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. Register B is preserved.

$711B (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.

$711E (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 feed, and nulls until the total number of blank lines is that specified in the TTYSET Eject Count.

$7121 (NXTCH) Get Next Buffer Character

The character to which the Line Buffer Pointer points is taken from the Line Buffer and saved in the Previous Character location. 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.

$7124 (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 is set to zero. The A-register and B-register are preserved by this routine.

$7127 (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.

$712A (LOAD) File Loader

On entry, the X-register must contain the address of a File Control Block which has been opened for reading the desired file. 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 relocator in that it does not change any addresses in the program itself, merely the location of the program in memory. 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.

$712D (SETEXT) Set Extension

On entry, the X-register should contain the address of the FCB into which the default extension is to be stored of 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

Any values other than those above are ignored, the routine returning without storing any extension.

$7130 (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.

$7133 (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.

$7136 (OUTCH2) (See OUTCH)

$7139 (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 across this routine.

$713C (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, and 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.

$713F (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.

$7142 (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 featrue 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-residentas 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                     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 instructions supplied with the TSC 6800 Assembler 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. Small 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 $7600 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 in 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 Meory 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 of FCB
JSR     GETFI L Get file spec into FCB
BCS     ERROR   Report error if one
LDAA    #1      Set extension code (TXT)
LDX     #FCB    Point to FCB
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 open 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 192 byte ($C0 hex) long block of RAM in the user's program area which is used by programs to communciate 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 that the programmer wishes except in page zero (0 - 256). 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 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 (null). 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.

Bytes 15-16 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 bytes 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.

Bytes 23-27 Reserved for future system use

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 if it is still in the buffer.

Bytes 32-33 Reserved for future use.

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 conjuntion 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. The value of Random Index may not exceed 127. 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 a 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 64-191 Sector Buffer

These bytes contain the data in the sector being read or written. The first four bytes of the sector are used by the system. The remaining 124 are used for data storage.

File Management System -- Entry Points

$7800 -- 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.

$7803 -- 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.

$7806 -- 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/orB-registers. See the documentation of the function codes for details. The X and B registers are always preserved with a call to the FMS.

Global Variables

This section describes those variables within the File Managment 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.

$7809 -- $780A 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 the 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.

$780B -- $780C 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.

$782D 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 $7806.

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 using this function are the 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 the FMS, after opening a file, the function code in the FCB is automatically set to zero (Read/Write Next Byte Function) in anticipation of Input/Output 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. It is not possible to open a file for both read access and write access simultaneously. The space compression flag should be treated the same as described in "Open for Read".

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, follow example in Read open.
```

Function 3 -- Reserved for future system use

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 the 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 and 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 director. Careless use of the Put Information Record 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 Postion 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. On return from FMS, the B-register is zero if no error was detected. If the B-register is non-zero on exit, a non-recoverable error was detected and the B-register contains the hardware status returned by the disk driver, not a F LEX error number. The error code is not stored in the Error Status byte by this function, nor are any of the pointers in the FCB updated. 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 exteme 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 form the FCB Sector Buffer. On return, the B-register is zero if no error was detected. 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. If the B-register is non-zero on exit, an unrecoverable error was detected, and the B-register contains the hardware status returned by the driver, not a FLEX error number. The error status is not stored in the Error Status byte of the FCB, nor are any of the pointers in the FCB updated.

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) -- Reserved for future system use

Function 16 ($10 hex) -- Get Random Byte From Sector

On entry, the file should be open for reading. 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. A byte number larger than 127 will result in an error being returned. 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    #16         Setup function code
STAA    0,X         Store in FCB
JSR     FMS         Call FMS
BNE     ERROR       Check for errors
Character is now in acc. A
```

FLEX Error Numbers

1 -- Illegal Function Code

FMS was called with a function code in the Function Code byte of the FCB that was too large or illegal.

2 -- File Busy or FCB Already in Use

An Open for Read or Open for Write Function was issued on an FCB that is already open.

3 -- File 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 -- File Does Not Exist

An Open for Read, 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 -- Directory Error

Reserved for future system use.

6 -- Too Many Files

The diskette directory has no more room for new file names. The directory can hold a maximum of 75 file names.

7 -- Disk Full

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 ge t written.

8 -- 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 -- 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 ($0A hex) -- 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 ($0B) -- Write Protected

An attempt was made to write on a diskette which has been write protected by the placing of tape over the write-enable cut in the diskette.

12 ($0C) -- Delete Protected
Reserved for future system use.

13 ($0D) -- Illegal File Control Block
An attempt was made to access an FCB from the open FCB chain, but it was not in the chain.

14 ($0E) -- Illegal Disk Address
Reserved for future system use.

15 ($0F) -- Drive Number Error
Reserved for future system use.

16 ($10) -- Not Ready
The drive does not have a diskette in it.

17 ($11) -- Access Denied
Reserved for future system use.

18 ($12) -- 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 ($13) -- Index Range Error
The Get Random Byte from Sector function was issued with a Random Byte number greater than 127.

20 ($14) -- FMS Inactive
Reserved for future system use.

21 ($15) -- Illegal File Name
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.

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 $7F00 - $7FFF. 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.

$7F00 -- 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.

$7F03 -- 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.

$7F06 -- Verify

Entry -- (No parameters)

The sector just written is to be verified to determine if there are CRC errors.

$7F09 -- 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 ~ is to be performed on the drive whose number is in the FCB.

$7F0C -- Drive Select

Entry -- (X) = FCB Address

The drive whose number is in the FCB is to be selected.

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 togehter all of the sectors on the diskette into a chain of avialable 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. 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 address 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 reserved for future system use. The remaining 124 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, all tracks start with sector number 1. On a FLEX diskette, all tracks except track 0 start with sector 1. Track 0 starts with sectors 0 and 1, but there is no sector 2. Sector 3 immediately follows sector 1 in the logical ordering of the sectors. This is due to the requirements of the SWTBUG monitor's "D" command. 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 5 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 four bytes are not used. When reading information from the directory using the FMS Get Information Record function, these 8 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 F LEX diskette (except the two BOOT sectors) has the following format:

Bytes 0-1	Link to the next sector
Bytes 2-3	Reserved for future system use
Bytes 4-127	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 address 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 responsiblity 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 F LEX, if encountered in the file, it is not returned to the calling program.

$18 (ASCII CANCEL)
Ignored by F LEX; 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 spaces 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. 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 $7600 and extends through location $773F. The system FCB at location $7740 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 $7600, therefore, the ORG is to $7600.

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 reporting. If the carry is set, control is passed to the line with the label 'LIST8'. At this point, X is made to point to the error string 'ILLST'. Calling the DOS routine PSTRNG will print this string after first having output a carriage return and line feed.

Finally, control is returned to DOS by a JMP to WARMS since there is nothing else to be done in the utility.

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 so it is reset. 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. Once again X is made to point to the FCB. The FMS Function Code for 'open a file for read' is 1. Therefore, we load the accumulator with 1 and store it in the FCB Function Code byte (FCB+0). A call to the 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 LIST7' will take us to the code which will determine what kind of error exists. At 'LIST7', the error status is retrieved from the FCB and compared to 4. This is the error number which is generated if the requested file could not be found on the disk. If this is the error which was generated, this program will print a message to that effect and then return to DOS.

If the error reported by FMS was not a 4, control is transfered to 'LIST9'. 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 number 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 transfered 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 the 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 transfered 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 the FMS will attempt to close the file. Upon return, errors are checked, and if none is 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

7103             WARMS  EQU    $7103     DOS WARM START ENTRY
7127             GETFIL EQU    $7127     GET FILE SPECIFICATION
7112             PUTCHR EQU    $7112     PUT CHARACTER ROUTINE
7118             PSTRNG EQU    $7118     PRINT STRING ROUTINE
711E             PCRLF  EQU    $711E     PRINT CR & LF
712D             SETEXT EQU    $712D     SET DEFAULT NAME EXT
713C             RPTERR EQU    $713C     REPORT DISK ERROR

                 * FMS EQUATES

7806             FMS    EQU    $7806     FMS CALL ENTRY
7803             FMSCLS EQU    $7803     FMS CLOSE ENTRY

                 * SYSTEM EQUATES

7740             FCB    EQU    $7740     SYSTEM FCB

                 * LIST UTILITY STARTS HERE *

7600                    ORG    $7600

7600 20 01       LIST   BRA    LIST2     GET AROUND TEMPS

7602 01          VN     FCB    1         VERSION NUMBER

7603 CE 77 40    LIST2  LDX    #FCB      POINT TO FCB
7606 BD 71 27           JSR    GETFIL    GET FILE SPEC
7609 25 4B              BCS    LIST8     ANY ERRORS?
760B CE 77 40           LDX    #FCB      RESTORE POINTER
760E 86 01              LDA A  #1        SETUP TXT EXTENSION
7610 BD 71 2D           JSR    SETEXT    SET DEFAULT EXTENSION
7613 CE 77 40           LDX    #FCB      POINT TO FCB AGAIN
7616 86 01              LDA A  #1        OPEN FOR READ CODE
7618 A7 00              STA A  0,X       STORE IN FCB
761A BD 78 06           JSR    FMS       CALL FMS - DO OPEN
761D 26 28              BNE    LIST7     CHECK FOR ERRORS
761F CE 77 40    LIST4  LDX    #FCB      POINT TO FCB
7622 BD 78 06           JSR    FMS       CALL FMS - GET CHAR
7625 26 0E              BNE    LIST6     ANY ERRORS?
7627 81 0D              CMP A  #$D       IS CHARACTER A CR?
7629 26 05              BNE    LIST5
762B BD 71 1E           JSR    PCRLF     OUTPUT A CR & LF
762E 20 EF              BRA    LIST4
```

```
 7630 BD 71 12    LIST5   JSR     PUTCHR     OUTPUT THE CHARACTER
 7633 20 EA               BRA     LIST4      REPEAT SEQUENCE

 7635 A6 01       LIST6   LDA A   1,X        GET ERROR STATUS
 7637 81 08               CMP A   #8         IS IT EOF ERROR?
 7639 26 20               BNE     LIST9
 763B 86 04               LDA A   #4         CLOSE FILE CODE
 763D A7 00               STA A   0,X        STORE IN FCB
 763F BD 78 06            JSR     FMS        CALL FMS - CLOSE FILE
 7642 26 17               BNE     LIST9      ERRORS?
 7644 7E 71 03            JMP     WARMS      RETURN TO DOS

 7647 A6 01       LIST7   LDA A   1,X        GET ERROR STATUS
 7649 81 04               CMP A   #4         IS IT "NO FILE" ERROR?
 764B 26 0E               BNE     LIST9
 764D CE 76 64            LDX     #NOFST     POINT TO MESSAGE
 7658 BD 71 18    LIST75  JSR     PSTRNG     OUTPUT THE STRING
 7653 7E 71 03            JMP     WARMS      RETURN TO DOS

 7656 CE 76 71    LIST8   LDX     #ILLST     POINT TO MESSAGE
 7659 20 F5               BRA     LIST75

 765B BD 71 3C    LIST9   JSR     RPTERR     REPORT DISK ERROR
 765E BD 78 03            JSR     FMSCLS     CLOSE ALL FILES
 7661 7E 71 03            JMP     WARMS      RETURN TO DOS

                  * STRINGS FOR ERROR MESSAGES

 7664 4E          NOFST   FCC     'NO SUCH FILE'
 7670 04                  FCB     4

 7671 49          ILLST   FCC     'ILLEGAL FILE NAME'
 7682 04                  FCB     4

                          END     LIST

NO ERROR(S) DETECTED
```

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 load several locations in memory. The P command, when executed, loads PRINT.SYS as a binary file. During the load process, two locations need to be set. The first is at location $0010 At this location, the starting address of the printer initialization routine should be loaded. As an example, if the Printer Initialization routine actually resides at location $A04A, then the value $A04A should be loaded at address $0010 This address is referenced in the "P" command listing. The second address which needs to be set is location $710D. This should contain the address of the actual location of the printer output character routine. As an example, if the output routine resided in memory at location $A016, then the value $A016 should be loaded at address $710D. It should be noted that this is the address part of the OUTCH jump vector in the DOS jump table. This overstore operation simply is changing the jump address which FLEX uses for character output. This address is automatically restored to its original value upon DOS Warmstart. Finally, PRINT.SYS must also load the actual print drivers referenced by the addresses in $0010 and $710D. The output character routine should preserve the X and B registers. Refer to the source listing provided in the user's manual.

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

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

                  * EQUATES

0010              INDEX   EQU     $0010
7740              FCB     EQU     $7740
712A              LOAD    EQU     $712A
7806              FMS     EQU     $7806
7803              FMSCLS  EQU     $7803
7106              RENTER  EQU     $7106
0004              NFER    EQU     $4
7089              PAUSE   EQU     $7089
7118              PSTRNG  EQU     $7118
713C              RPTERR  EQU     $713C
7103              WARMS   EQU     $7103
7091              LSTTRM  EQU     $7091
7082              EOL     EQU     $7082

7600                      ORG     $7600

7600 20 01        P       BRA     P1          BRANCH AROUND TEMPS

7602 01           VN      FCB     1           THE VERSION NUMBER

7603 B6 70 91     P1      LDA A   LSTTRM      GET THE LAST TERMINATOR
7606 81 0D                CMP A   #$D         IS IT A CR?
7608 27 39                BEQ     P8
760A B1 70 82             CMP A   EOL         IS IT AN EOL CHARACTER?
760D 27 34                BEQ     P8
760F 7F 70 89             CLR     PAUSE       DISABLE THE PAUSE FEATURE
7612 CE 77 40             LDX     #FCB        POINT TO THE FCB
7615 86 01                LDA A   #1          OPEN THE FILE FOR READ
7617 A7 00                STA A   0,X
7619 BD 78 06             JSR     FMS
761C 26 0E                BNE     P2          CHECK FOR ERRORS
761E 86 FF                LDA A   #$FF        SET FOR BINARY READ
7620 A7 3B                STA A   59,X        SET COMPRESSION FLAG
7622 BD 71 2A             JSR     LOAD        CALL DOS'S BINARY LOADER
7625 DE 10                LDX     INDEX       GET THE JUMP VECTOR
7627 AD 00                JSR     0,X         DO THE INITIALIZATION
7629 7E 71 06             JMP     RENTER      RETURN TO DOS
```

```
 762C A6 01       P2      LDA A   1,X        GET THE ERROR CODE
 762E 81 04               CMP A   #NFER      IS IT "NO SUCH FILE"?
 7630 26 08               BNE     P3
 7632 CE 76 48            LDX     #NOPST     POINT TO MESSAGE STRING
 7635 BD 71 18    P25     JSR     PSTRNG     PRINT STRING
 7638 20 03               BRA     P4

 763A BD 71 3C    P3      JSR     RPTERR     CALL DOS'S REPORT ERROR ROUTINE
 763D BD 78 03    P4      JSR     FMSCLS     CLOSE ALL FILES
 7640 7E 71 03            JMP     WARMS      RETURN TO DOS

 7643 CE 76 5E    P8      LDX     #ERSTR     POINT TO MESSAGE
 7646 20 ED               BRA     P25

 7648 22          NOPST   FCC     '"PRINT.SYS" NOT FOUND'
 765D 04                  FCB     4
 765E 43          ERSTR   FCC     'COMMAND MUST FOLLOW "P"'
 7675 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.

 7744                     ORG     $7744

 7744 50                  FCC     'PRINT'
 7749 00                  FCB     0,0,0
 774C 53                  FCC     'SYS'

                          END     P

NO ERROR(S) DETECTED
```

FLEX REFERENCE SHEET

FLEX MEMORY LOCATIONS

$7000 - $707F	Line buffer
$7080	TTYSET Backspace Character
$7081	TTYSET Delete Character
$7082	TTYSET End of Line Character
$7083	TTYSET Depth Count
$7084	TTYSET Width Count
$7085	TTYSET Null Count
$7086	TTYSET Tab Character
$7087	TTYSET Duplex Mode
$7088	TTYSET Eject Count
$7089	TTYSET Pause Control
$708A	TTYSET Escape Character
$708B	System Drive Number
$708C	Working Drive number
$708D - $7090	System Scratch; future use
$7091	Last Terminator
$7092 - $7093	User Command Table Address
$7094 - $7095	Line Buffer Pointer
$7096 - $7097	Escape Return Register
$7098 - $7099	System Scratch
$709A	Previous Character
$709B	Current Line Number
$709C - $709D	Loader Address Offset
$709E	Transfer Flag
$709F - $70A0	Transfer Address
$70A1	ACIA Flag
$70A2	Error Type
$70A3	Output Switch
$70A4	System Scratch
$70A5	Command Flag
$70A6	Current Output Column
$70A7 - $70B4	System Scratch
$70B5 - $70FF	System Constants
$7809 - $780A	FCB BASE POINTER
$780B - $780C	CURRENT FCB ADDRESS
$782D	VERIFY FLAG

FMS COMMANDS

FUNCTION - (HEX)	FUNCTION
01	OPEN FOR READ
02	OPEN FOR WRITE
03	reserved/future use
04	CLOSE FILE
05	REWIND FILE
06	OPEN DIRECTORY
07	GET INFORMATION RECORD
08	PUT INFORMATION RECORD
09	READ SINGLE SECTOR
0A	WRITE SINGLE SECTOR
0B	reserved/future use
0C	DELETE FILE
0D	RENAME FILE
0E	reserved/future use
0F	reserved/future use
10	GET RANDOM BYTE FROM SECTOR

FLEX SUBROUTINES

$7100	COLDS	(coldstart entry)
$7103	WARMS	(warm start entry)
$7106	RENTER	(main loop re-entry)
$7109	INCH	(input character
$710C	OUTCH	(output character)
$710F	GETCHR	(prefered get character)
$7112	PUTCHR	(prefered output character)
$7115	INBUFF	(input to line buffer)
$7118	PSTRNG	(print string)
$711B	CLASS	(classify character)
$711E	PCRLF	(print C/R, L/F)
$7121	NXTCHR	(next character)
$7124	RSTRIO	(restore I/O vectors)
$7127	GETFIL	(parse file spec.)
$712A	LOAD	(file loader)
$712D	SETEXT	(set extension)
$7130	ADDBX	(add ACC-B to X)
$7133	OUTDEC	(output decimal number)
$7136	OUTCH2	(output character)
$7139	OUTHEX	(output hex number)
$713C	RPTERR	(report error)
$713F	GETHEX	(get hex number)
$7142	DOCMND	(call DOS)
$7800	FMS Initialization	
$7803	FMS Close	
$7806	FMS Call	

FILE CONTROL BLOCK SPECIFICATIONS

BYTE # (DECIMAL)	FUNCTION
0	FMS COMMAND
1	ERROR STATUS
2	ACTIVITY STATUS
3	DRIVE NUMBER
4 - 11	FILE NAME
12 - 14	EXTENSION
15 - 16	Reserved/future use
17 - 18	STARTING DISK ADDRESS
19 - 20	ENDING DISK ADDRESS
21 - 22	FILE SIZE
23 - 27	Reserved/future use
28 - 29	FCB LIST POINTER
30 - 31	CURRENT POSITION
32 - 33	Reserved/future use
34	DATA INDEX
35	RANDOM INDEX
36 - 46	NAME WORK BUFFER (internal)
47 - 49	CURRENT DIRECTORY ADDRESS
50 - 52	FIRST DELETED DIRECTORY POINTER
53 - 63	SCRATCH BYTES (for RENAME)
59	SPACE COMPRESSION FLAG
64 - 191	SECTOR BUFFER

1771 REFERENCE SHEET

Below is a short table showing the possible error combinations that a 1771 controller integrated circuit can return. For Example, if during a read single sector read operation an error is found and the B accumulator return a 08 hex (0000 1000), the controller discovered a CRC error.

	COMMANDS					
BIT SET	**All Type 1 Commands**	**Read Address**	**Read**	**Read Track**	**Write**	**Write Track**
7	Not Ready	Not Ready	Not Ready	Not Ready	Not Ready	Not Ready
6	Write Protect	0	Record Type	0	Write Protect	Write Protect
5	Head Engaged	0	Record Type	0	Write Fault	Write Fault
4	Seek Error	ID Not Found	Record Not Found	0	Record Not Found	0
3	CRC Error	CRC Error	CRC Error	0	CRC Error	0
2	Track 0	Lost Data	Lost Data	Lost Data	Lost Data	Lost Data
1	Index	DRQ	DRQ	DRQ	DRQ	DRQ
0	Busy	Busy	Busy	Busy	Busy	Busy

Type	**Command**	**Bits**							
		7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
I	Restore	0	0	0	0	h	V	r1	r0
I	Seek	0	0	0	1	h	V	r1	0
I	Step	0	0	1	u	h	V	r1	r0
I	Step In	0	1	0	u	h	V	r1	r0
I	Step Out	0	1	1	u	h	V	r1	r0
II	Read Command	1	0	0	m	h	E	0	0
II	Write Command	1	0	1	m	h	E	a1	a0
III	Read Address	1	1	0	0	0	1	0	0
III	Read Track	1	1	1	0	0	1	0	s
III	Write Track	1	1	1	1	0	1	0	0
IV	Force Interrupt	1	1	0	1	I3	I2	I1	I0

DC1 and DC2 Address Registers

DRIVE REGISTER	$8014
COMMAND REGISTER	$8018
TRACK REGISTER	$8019
SECTOR REGISTER	$801A
DATA REGISTER	$801B

h = Head Load flag (bit 3 of Type I)
h = 1, Load head at beginning
h = 0, Do not load head at beginning

V = Verify flag (bit 2 of type I)
V = 1, Verify on last track
V = 0, No verify

r1r0 = Stepping Motor Rate (bits 1 - 0 of type I)(2MHz clock)
r1r0 = 00, 6 ms steps
r1r0 = 01, 6 ms step
r1r0 = 10, 10 ms step
r1r0 = 11, 20 ms step

u = Update flag (bit 4 of type I)
u = 1, Update track register
u = 0, No update

b = Block Length flag (bit 3 of type II)
b = 1, IBM format (128 to 1024 bytes)
b = 0, Non-IBM format (16 to 4096 bytes)

E = Enable HLD & 10 ms delay (bit 2 of type II)
E = 1, Enable HLD, HLT & 10 ms delay
E = 0, Head is assumed engaged & no 10 ms delay

a1a0 = Data Address Mark (bits 1 - 0 of type II)
a1a0 = 00, FB (Data Mark)
a1a0 = 01, FA (Data Mark)
a1a0 = 10, F9 (Data Mark)
a1a0 = 11, F8 (Data Mark)

s = Synchronize flag (bit 0 of type III)
s = 0, Synchronize to AM
s = 1, Do not synchronize to AM

m = Multiple Record flag (bit 4 of tyep III)
m = 0, Single Record
m = 1, Multiple Records

In = Interrupt Condition flags (bit 3 - 0 of type IV)
I0 = 1, Not Ready to Ready Transition
I1 = 1, Ready to Not Ready Transition
I2 = 1, Index Pulse
I3 = 1, Every 10 ms

The above tables are for reference only – a complete data sheet on the 1771 disk controller (not available from SWTPC or TSC) should be consulted for detailed information