APPENDICES

APPENDIX A

Understanding and learning to use the marked item and fixed item files specified in chapter 8 has proved difficult to the normal NEWDOS/80 user; therefore appendices A and B have been included to provide examples and more explanation in an effort to ease this difficulty. Nothing in appendix A or B is to be construed as overriding the specifications provided in chapter 8; the two appendices are provided simply and exclusively for examples and elaboration.

Appendix A was written by a user trying to cope with chapter 8 and is basically his understanding of marked item and fixed item files.

Appendix B is the NEWDOS/80 author's attempt to provide example programs of the 5 file sub-types: MF, MU, MI, FF and FI.

File Positioning

File Position (fp) is an operand in all NEWDOS/80 GETS and PUTS, and is specified in section 8.4.1. When omitted, a null operand is assumed. The fp operand otherwise commonly consists of a special character, occasionally followed by other special characters and/or expressions. One form of the fp operand consists of nothing more than a numeric expression. In the forms, which follow, special characters are to be used as shown. In those forms showing a prefixing special character adjoining some other character string, the special character does not necessarily have to be contiguous with the rest of the expression; it may be separated from it by a blank or space.

fp Value Meaning

(null)

If the file is an MU, MF or FF type file, and the REMRA is valid, the file is advanced to the next sequential record; in any other case, the current file position is not changed and processing continues from the position left at the termination of data transfer of the previous GET/PUT. Open leaves REMRA marked invalid for all file types, and sets current file position equal to 0 (except for mode "E", which causes current file position to be set equal to the FPDE's EOF value). The first sequential access for record segmented files always starts at current file position.

*****

The current file position is not changed. This specification allows the continuation of processing of a particular record by a GET or PUT. It is primarily used to continue processing a record already partially read or written. For MU, MF and FF type files, it cannot be used to advance the file to the next sequential record, even though the file is actually already positioned at that record, having exhausted the bytes of the current record. To sequentially advance to the next record, use fp = (null).

#

If the REMRA is valid, the file is positioned to process that record again; an error condition is raised if the REMRA is invalid. For MU, MF and FF type files, this specification allows the reprocessing of the record currently being processed, from the beginning, perhaps with different variable names or expressions in the IGELs. For MI and FI type files, it allows the reprocessing of the same data item group as was processed by the immediately preceding GET/PUT.

$

If the REMBA is valid, the file is positioned to begin processing at again it the point where the previous GET or PUT was at the end of its file positioning phase; an error condition is raised if the REMBA is invalid. This specification allows the reprocessing of a particular group of data by a GET/PUT, and is primarily used to reposition a file for partial record I/O. It functions in the same fashion for all NEWDOS/80 file types.

%

This specification performs a "pseudo FIELD" operation. No data transfer takes place; the filearea FCB is not changed; the file does not have to be open when this fp is used. It is used with FF and FI files to allocate user data strings of fixed sizes from the BASIC string storage area in high memory.

&

This specification is used only with PUTs, and has no effect on file positioning. It does however cause the current contents of a filearea buffer to be written to the diskette. It should be used whenever the data in the buffer is particularly sensitive. It may be used specifying the FAN of a PRINT file.

&&

This specification is similar to &, except that in addition the file's EOF is updated from the FCB to the FPDE. PUT fan,&& allows the programmer to force the EOF update to the FPDE without having to do a CLOSE.

!rba

Using this form of fp specification causes GET/PUT processing to begin at the specified location in the file where rba is a BASIC expression evaluating to a RBA value. For MU, MF and FF type files, the system checks to make sure that a record begins at the specified location. In the case of a MU file, the RBA value must point to an SOR item. This form of fp specification demands the greatest amount of care and premeditation on the programmer's part, as if it is used incorrectly, especially with FF and FI type files, it can be most disastrous. It is just about the only way to randomly access data stored in MI, MU and FI type files.

!%

This specification is basically the same as the !rba form except that the current EOF value is used as the RBA. It is commonly used to position a file for extension - that is, to add records/data to the end of the file. To extend a file it must be opened with mode "R"; mode "D" will yield an error if extension is attempted.

!$rba

This specification allows the programmer to position the file for the next data transfer for that particular filearea,-without regard to the specific access technique or verb used for the transfer; no data transfer to user data areas occurs with this specification. No IGEL may be referred to or included in the GET/PUT using this specification. The positioning resulting from the use of this specification doesn't become effective until the next INPUT/PRINT or GET/PUT, and then only if no additional positioning is specified. It can be used to position a file for random access in a program which uses a subroutine containing a single GET/PUT having a (null) fp to do all file access; such a program could process sequential groups of records randomly distributed throughout a file.

!$%

The basic function of this specification is identical to !$rba, except that it uses the current EOF value as the RBA. The GET/PUT using this specification must not refer to or include an IGEL. Again, the file position resulting from this specification doesn't take effect until the next INPUT/PRINT operation, or the next GET/PUT (if another fp isn't specified).

!#rba

Used only with PUT, this specification sets the filearea's EOF value equal to the value rba. For the real EOF value of the file to be altered, that is, the one in the FPDE, the filearea must either be closed or a PUT && statement executed. The EOF value provided must be rational for the file type involved. For MF and FF files it must be an integral multiple of the file's standard record length.

rn (Record Number)

This specification is the same as the one supported by TRSDOS; rn is a numeric BASIC expression which evaluates to an integer value from 1 to 32767, inclusive. The specified record number is converted to an RBA which is then used in the same functional manner as !rba.

As mentioned above, certain forms for fp change REMBA, REMRA or EOF. For your convenience, the fp forms and their effects on these fields are summarized in the following decision table.

fp	REMBA	REMRA	EOF
(null)	1	1	6
*	1	2	6
#	3	4	6
$	4	4	6
%	4	4	4
&	4	4	4
&&	4	4	4
!RBA	1	1	6
!%	1	1	6
!$RBA	5	5	4
!$%	5	5	4
!#RBA	4	4	1
RN	1	1	6

Meanings of codes in the matrix:

1 -- The field is set to the RBA resulting from that fp value.

2 -- If REMRA is invalid at the beginning of the statements execution, or it is an MI or FI file, the field is set to the RBA resulting from the fp value. In other words, it is set if the current file position is at the beginning of a record, otherwise it is unchanged.

3 -- The field is set equal to REMRA.

4 -- The field is not changed.

5 -- The field is set to an invalid value.

6 -- For output/update files, the field is changed if a PUT extends the file.

Altogether, there are four areas in an FCB relevant to 'file positioning. These are:

Current File Position
: This single field can be looked at as being 3 different values, depending upon where the GET/PUT is in its processing:

  GPP1
  : The file position at the start of GET/PUT execution. Unless the file has been closed and re-opened, it is the same value left as GPP3 from the last GET/PUT for that filearea.

  GPP2
  : The resulting RBA value after positioning has been done, and prior to any data transfer. GPP2 is the value saved as REMBA and REMRA whenever these values are set.

  GPP3
  : The RBA value after the last byte of data transfer, if any, real or bypassed, has been accomplished.

REMRA
: For MU, MF, FF and field item type files, it contains the RBA value of the beginning of the record in process. For MI, FT and INPUT/PRINT files, it is equal to REMBA. See GPP2 above.

REMBA
: The RBA value where the previous data-transferring GET/PUT began its data transfer. If the file is record-segmented, and REMBA is at the start of a record, REMRA is set equal to REMBA. See GPP2 above.

EOF
: The RBA value of the last byte of data in the file, plus 1. For MU, MF, FF and field item type files, it effectively points to the next sequential record to be written to the file. For MI, FI and INPUT/PRINT files, it effectively points to the next sequential byte to be written to the file.

The general method of managing the various fp values in the FCB goes as follows:

The file is moved from the current file position (GPP1) to the requested position, if necessary. This may include writing an updated buffer back to the diskette, computing the new sector address, and reading that sector into the buffer.

The RBA resulting from the requested positioning is placed in the current file position (GPP2).

REMBA is set equal to the current file position (GPP2).

If the file is an INPUT/PRINT file, is user-segmented, or is record-segmented and the current file position (GPP2) points to the start of a record, REMRA is set equal to the current file position (GPP2).

Data transfer, if any is requested, is done. The current file position (GPP3) contains the RBA of the byte following the last one transferred.

If the file has been extended, or the fp = INS, EOF is set to the appropriate value of the two.

OPEN

Any file must be opened before the data in it can be processed. The OPEN verb itself establishes an I/O link between the file and the applications program. The link's control information is maintained in the filearea (which contains a FCB). Once opened, the data in the file is made available to the program by means of INPUTS or GETS; data is placed on the file via PRINTS or PUTS. When the processing of the data is complete, the file should be closed, thus breaking the I/O link between the file and the program.

NEWDOS/80 supports five OPEN modes: "I" for sequential input (INPUT verb), "0" for sequential output (PRINT verb), "R" for random access input/output (GET or PUT verbs), "E" for sequential output starting at the current EOF for existing or new files (the "E" could be read as "extend"), and "D" for random access files which the user does not want expanded/lengthened with PUTS beyond the current EOF.

NEWDOS/80 BASIC marked item and fixed item file support allows the GET and PUT verbs to be used with all five modes. The general form of the NEWDOS/80 OPEN verb is:

1. OPEN m,fan,filespec
2. OPEN m,fan,filespec,lrec1
3. OPEN m,fan,filespec,ft
4. OPEN m,fan,filespec,ft,lrec1

where: m
is an expression evaluating to a string equal to "I", "0", "R", "E" or "D". It specifies the mode of access to be used for the file, as well as the initial positioning of the file.

fan
: is the number of the filearea to be opened.

filespec
: is an expression evaluating to the name of the file to be opened. The expression itself can be a string literal or constant.

ft
: is an expression evaluating to a string equal to "FI", "FF", "MI", "MU" OR "MV". It identifies a particular NEWDOS/80 sub-file type, which will all be explained shortly. Whenever ft is used in an OPEN statement, GETS and PUTS are the only way to transfer data from and to the file. INPUTS and PRINTS must not be used. Neither may the BASIC FIELD statement be used. All GETs or PUTS used to transfer data must specify either an IGELSN or contain the IGEL itself. The applications program must not alter or directly reference the data in the filearea in any way. Two ft values require the specification of lrec1 in the OPEN statement; a third ft allows its optional specification.

lrec1
: is an expression evaluating to an integer value between 1 and 256

for field item files and between 1 and 4095 for marked item and fixed item files. It must be specified for all record-segmented files (except field item files where 256 is assumed if lrec1 is not specified), and specifies the exact length of all records in the file for field item, "FF" and "MF" files, or the optional maximum lrec1 for file type "MU".

Note that the standard forms of BASIC OPEN have not changed (formats 1 and 2), thus allowing existing field-file and print/input file oriented applications to continue to function. The extensions to the standard forms identify the file as a NEWDOS/80 file, and define the file type and access technique used to retrieve and manipulate the data in it.

Of all the file types supported by NEWDOS/80, the easiest one to use and understand is ''MU''. It defines a file, which contains marked items, and is segmented into records of varying lengths. The length of a record-is defined as the difference between the RBA of the record's SOR and the RBA of the next record's SOR or the RBA of the file's EOF, whichever follows. The record length need not be specified in the OPEN statement; but if it is provided, it specifies the maximum record length allowed in the file.

A record in an "MU" file can be updated with another record of the same or shorter length than it was originally created with, but it cannot be lengthened. When a record is updated with a record, which is shorter than the original record, the new record is padded on the right with fill items (bytes of hex '00') to the end of the original record. This shorter record can later be replaced with one, which is longer, as long as the new one is not longer than the record originally written to the file.

The "MU" file type is intended to replace BASIC's sequential input/output files accessed via INPUT/PRINT verbs. Its greatest strength is that no special delimiters have to be provided by the programmer to separate two contiguous string items (in BASIC sequential file support, a comma must be PRINTed between the strings for the INPUT to be able to separate them). A secondary benefit of "MU", and all other NEWDOS/80 BASIC files too, is that numeric values are stored on the diskette in their internal form. That is, for example, a double precision value is written as an 8-byte item, rather than an up-to-14 character item requiring conversion back to internal (8-byte) form on input. Don't forget that in the case of marked item files, such as "MU", a double precision item actually requires nine bytes due to the prefixing control character. If an lrec1 is specified in the OPEN statement, it sets the maximum record length allowable for the file, and must allow for all control bytes (including SOR items) in each record.

The next most simple forms of ft to use are "MF" and "FF". Both identify a file as record-segmented, and having records of fixed length. They both imply that all records have the same internal data structure, but do not guarantee that condition. The OPEN statement must specify the exact logical record length of all records in the file. In the case of "MF", the marking control bytes must also be accounted for in the length (note that an "MF" file doesn't use SOR items at the start of each logical record since BASIC knows where each record starts). Each GET/PUT checks the IGEL's data length against the lrec1 specified at OPEN time, and raises an error condition if the IGEL's length is greater.

The most difficult forms of ft to use are "MI" and "PI". They specify that the file is record segmented entirely under user control. The "lrec1" must not be specified in the OPEN statement for these file types. These forms allow a file to contain a very complex data relationship, without BASIC's knowledge of the users data structure. That is, BASIC cannot advance from one user record to another.

CLOSE

The CLOSE verb breaks the I/O link set up by the OPEN verb between the BASIC application program and a file. Its general format has not been modified by NEWDOS/80.

Depending on the file's mode and type, the contents of the filearea buffer may be written to the diskette by this verb. For output and random-access files the file's directory entry is updated to reflect the current EOF value stored in the filearea's FCB.

GET

In field item (TRSDOS random) file processing, the GET statement is used to read a particular record into the filearea's buffer. The FIELD statement is then used to adjust the data pointers of string variables to address the buffer itself. This method of data access causes the file to be termed a field item file in NEWDOS/80 since all the other file types may also be used randomly.

In addition to continued support of field item files, NEWDOS/80's GET statement is used in marked item and fixed item file processing to transfer data from a file to user-specified variables, define the variables themselves, or position a file for later operations. The actual transfer of data from the diskette to the buffer occurs only as needed by the BASIC's determination of the IGEL data requirements in relation to the data currently in the buffer.

The general form of the GET statement is:

1. GET fan (a null fp is assumed)
2. GET fan,fp
3. GET fan,fp,igelsn
4. GET fan,fp,,igel

Formats 1 and 2 are used for field item files and are compatible with TRSDOS BASIC. They naturally may also be used in NEWDOS/80 BASIC application programs.

Formats 3 and 4 are unique to NEWDOS/80 BASIC. They must be used in data transfer GET whenever the filearea is open for marked item or fixed item file operations. Format 2's usefulness has been expanded by the addition of several new fp specifications unique to NEWDOS/80.

Format 3 specifies the location of the IGEL containing the data names, which are to contain the data at the completion of the GET; format 4 contains the IGEL as an integral part of the GET statement itself.

In NEWDOS/80, no function in the IGEL or the fp parameter may reference a filearea, even if that filearea is the same as that used by the GET or PUT statement.

At the successful completion of a GET statement, the filearea is left positioned at:

a. the next byte of the file for fixed item files.
b. the next item in the file for marked item files.
c. the next 256 byte record for field item files.

If an error is encountered during the processing of a GET statement, the filearea is reset to its status and content prior to the execution of the statement. After correction of the error, the GET statement may be executed again. The contents of the variables named in the IGEL are entirely unpredictable when an error is detected, and should not be used unless the GET has been re-executed successfully.

When a GET statement refers to or contains an IGEL, successive file items are transferred to successive variables named in the IGEL.

For fixed item files:

String variables of the IGEL are filled with the number of bytes specified in the expression prefix. As a result, the length of the variable is made equal to the value of the prefix.

Numeric variables of the IGEL are filled with the number of bytes corresponding to that item's internal form. (Integer items are two bytes long, single precision items are four, and double precision are eight.)

Prior to the first GET which transfers data to user variables, a GET using ft = % may be issued. The file referenced by the fan need not necessarily be open when this GET is issued, as the purpose of this GET is to perform the pseudo FIELD function for fixed item file operations. As the IGEL items are processed, numeric variables are left unchanged, (len)$ and (len)# items are ignored, and string variables have their length set to the value of the expression prefix, and are truncated or padded on the right with blanks as necessary. If a string variable exists at the time the pseudo FIELD is issued and its contents/value doesn't reside in the BASIC string area, its contents are moved there. This is done in an attempt to ensure that enough string space exists for continued operation, as the subsequent data transfer GETS will actually move data to the variable, rather than simply changing the variables data pointer. Once referred to by a pseudo FIELD operation, string variables should have their contents changed only by LSET or RSET to ensure that the variables lengths do not change. In NEWDOS/80 version 1, the pseudo FIELD function was required before any PUTS to a fixed item file; in version 2 this is not required and many programs using fixed item files will elect not to use the pseudo FIELD function at all.

If the file is record segmented and there are fewer bytes in the record from the current file position at the start of data transfer of the item than are requested by the IGEL item, a "RECORD OVERFLOW" error condition is raised.

For marked item files:

A null IGEL expression causes the corresponding file item to be skipped.

The expression prefix of a string variable is used to limit the number of characters actually transferred to the variable. If the file item is shorter than what the expression prefix allows, the length of the string variable is set to that of the file item. If the file item is longer than what the expression prefix allows, the file item is truncated on the right to that length, as would be done by an LSET.

SOR and fill items are skipped as they are encountered.

If the file item type and the IGEL item type are incompatible, a "TYPE MISMATCH" error is raised. If for example, the file item type were single precision and the IGEL item type were string, the error would be

raised. If however, the IGEL item type were integer, no error would be raised unless the file item's value exceeded what was legal for integer items.

If the file is record-segmented, and there are fewer items remaining in the record from the current file position at the start of data transfer of the item than are in the IGEL, a "RECORD OVERFLOW" error is raised.

Two special forms of fp may be used to set the file position for subsequent processing, regardless of the type of processing normally done for the file. These are fp = !$rba and fp = !$%. Use of either of these forms cause REMRA and REMBA to be marked invalid. Use of either of these ft values in a format 3 or 4 GET is invalid, as no actual data transfer takes place.

More than one GET may be used to retrieve successive file items from a single record. This technique is called partial record I/O. The first item in a record could, for example, identify the record as containing a name and address, a transaction number and amount, or an invoice number and expected ship date. The first byte could be read by itself and used to transfer control within the program to the appropriate routine to handle the data, which follows.

Partial record I/O as an access technique can be readily used with fixed item files and field item files. In field item files, the technique calls for reFIELDing when the new record is not the same type as the previous record. In marked item files, items to be bypassed in a record are simply left as null items in the GET's IGEL. In fixed item files, the length of the fields to be bypassed must be determined, and that sum be specified as the length prefix of a (len)$ IGEL item, in order to position the record to the proper byte to be transferred. The real strength of partial-record I/O with fixed item files is that as little as a single field imbedded within a record can be updated independently of all other data in the record; with marked item files, all items beyond the one to be updated would first have to be read, then re-written with the item being updated to maintain their content. The primary benefit of partial record I/O is that several record formats can reside in a single file and only as much data need be transferred as necessary to identify the particular format.

PUT

In field item file processing, the programmer executes, if not done previously, a FIELD statement to define the variables' buffer overlaying main memory positions. Next, the values for those variables are moved into them using LSET or RSET statements. Lastly, the record is written (or buffered) using the PUT statement.

For marked item and fixed item file processing, the contents of BASIC variables are written (or buffered) using the PUT statement without the need of moving the data first to special encoded variables. Instead an IGEL is used to specify during the PUT which variables are to have their contents sent to the file.

Remember, no IGEL expression or the-fp expression may contain functions that reference a filearea.

The general form of the PUT statement is:

1. PUT fan (a null fp is assumed)
2. PUT fan,fp
3. PUT fan,fp,igelsn
4. PUT fan,fp,,igel

Formats 1 and 2 are used in field item file operations and are compatible with TRSDOS BASIC. They naturally may continue to be used in application programs running under control of NEWDOS/80.

Formats 3 and 4 are unique to NEWDOS/80 BASIC. One or the other or both must be used whenever data is transferred to the file during marked item or fixed item file processing. Format 3 specifies the location of the IGEL containing the expressions to be sent to the file; format 4 contains the IGEL itself as a part of the PUT statement. Format 2 PUTS may be interspersed with formats 3 or 4 to achieve the necessary file positioning for subsequent data transfer.

At the successful completion of a PUT statement, the filearea is left positioned at:

a. the next byte of the file for fixed item files.
b. the next item in the file for marked item files.
c. the next 256-byte record for field item files.

If an error is encountered during the processing of a PUT statement, the filearea is reset to its status and positioning prior to the execution of the PUT statement. The data in the file as a result of the error is completely unpredictable, and will most likely cause errors on a subsequent GET. This situation occurs only during the updating of existing records; if possible and practical, a PUT should be issued later in an attempt to correct the error. In an effort to reduce the possibility of damage to the file when the file is opened using the "R" or "D" mode, NEWDOS/80 BASIC processes the IGEL twice in its entirety, once to catch errors in IGEL specification, and again to actually transfer the data to the buffer.

When a PUT statement refers to or contains an IGEL, the contents of successive IGEL expressions are transferred to the filearea buffer and become file items.

For fixed item files:

A string variable or expression may have a length different than the one allowed by the expression prefix in the IGEL. Strings which are shorter have the corresponding file item padded on the right with blanks; strings which are longer have the corresponding file item truncated on the right in the manner used by LSET. In other words, the expression prefix value determines exactly how many bytes are to be moved to the file item.

A record overflow error condition is raised if the logical record length is exceeded. During whole-record I/O, if the sum of all item lengths in the IGEL exceeds the LRECL, the error is raised. During partial-record I/O, if the sum of all item lengths in the IGEL exceeds the number of bytes left in the record, the error is raised.

Prior to the first PUT statement which actually transfers user data to the buffer, a PUT using ft = % may be issued. The file referred to by the fan need not necessarily be open at the time of this PUT, as its purpose is to perform the pseudo FIELD function. As the IGEL items are processed, numeric items are left unchanged, (len)$ and (lend items are ignored, and string expressions have their length set equal to the value of the expression prefix, and are truncated or padded on the right with blanks as necessary. If the string variable exists at the time of the pseudo FIELD PUT and the string itself doesn't reside in the BASIC string space, it is moved there. Once referred to by a pseudo FIELD PUT statement, string variables should have their contents changed only by LSET or RSET statements to ensure that their lengths do not change. In NEWDOS/81 version 1, this pseudo FIELD function was required before any PUTS to a fixed item file; in version 2 this is no longer required and many programs using writing to fixed item files will elect not to use the pseudo FIELD function at all. The pseudo FIELD function is left in existence for the programmer who wants to assure IGEL related string variables maintain the required length at all times.

For marked item files:

SOR and fill items are inserted into the filearea buffer as dictated by the file's ft, the PUT's fp and the IGEL data length versus the file's record length.

Nearly anything syntactically legal on the right hand side of a LET expression's equal sign is legal as an expression in an IGEL referenced by a PUT statement, excepting that a filearea may not be referenced in such an expression. Specifically excluded from appearing in any IGEL expression are LOC, LOF, EOF and any other expression, which references a fan.

When a string expression in an IGEL has a length prefix, the prefix determines the maximum number of characters to be written to the file. If the string is shorter than the expression prefix allows, the string

is written to file as is. If the string is longer, the corresponding file item it is truncated on the right as would be done by an LSET operation.

Strings require either one or two marking bytes, depending on the number of bytes in the string. If the string has from 0 to 127 bytes in it, it requires only one marking byte to describe it on file. If it has 128 bytes or more, then two marking bytes are required to describe it. All these marking bytes must be allowed for when specifying an lrec1 at open time.

Numeric IGEL expressions are placed in the buffer in their internal BASIC form: 2 bytes for integers, 4 bytes for single-precision numbers and 8 bytes for double-precision numbers. Don't forget that each individual file item has a marking byte associated with it, and that the correct lengths of the item types just mentioned are 3, 5 and 9 bytes.

Numeric literals and expressions in the IGEL are first converted to the most compact internal BASIC data type that preserves their precision before being sent to the file. For example, the numeric literal 3.14159 would be sent to file as a single precision number (5 bytes including marking byte); the value resulting from LEN(A$) minus LEN(B$) would be sent to file as an integer number (3 bytes including marking byte).

Two or more PUT statements may be used to output all the items of a record. The number of bytes actually comprising a single logical record cannot exceed the lrec1 value specified in the OPEN statement, or the system maximum of 4095 bytes.. Any attempt to exceed either of these limits results in a "RECORD OVERFLOW" error.

In the case of MU and MF type files opened for random access updating purposes, the record existing on file, from the current file position at the beginning of data transfer for the PUT, to the record's end (defined by the next SOR, or EOF) is replaced in its entirety. If the cumulative IGEL data length is less then the file record's remaining length, the IGEL data is sent to the file and padded out with fill items to completely fill the file record. Be very careful when operating in this mode, because if the PUT's IGEL defines fewer items than exist in the file record at the time of update, the excess file items are eliminated; later GET statements will encounter problems if they expect the original number of items to be present in that record.

Items in a MI type file cannot be updated as the system has no idea where the user's record ends, and therefore cannot pad to the end of the record as it does for MU and MF files.

For both fixed item and marked item files:

The filearea's buffer is actually written to the diskette when:

The last byte of the buffer is filled with data from the IGEL, and more data has yet to be moved.

A PUT statement with an fp of "&" or "&&" is executed, causing the buffer to be written to the diskette in its current state.

The file is closed, explicitly by fan, or implicitly by a general (non-specific) CLOSE.

If the data in the file be especially critical, the programmer should consider the use of PUT statements with the fp of "&". This will cause the filearea's buffer to be written to the diskette without disturbing the current file positioning. If there is no data in the buffer waiting to be written to the diskette, this particular PUT statement will be ignored. Should some other filearea used by the program require the data in this filearea to be disk-resident, the fp of "&" must be used. Don't overlook the fact that an fp of "&" is used only in a format 2 PUT; any data to be written to file must first have been placed there by a format 3 or 4 PUT. The use of fp = & is not restricted to marked item or fixed item files - it may be used with field item files or print/input disk files also.

Everything said above for the PUT fan,& statement also applies to the PUT fan,&& statement which, in addition, writes the file EOF from the filearea's control information back to the file's directory entry.

Two special forms of fp may be used to set the file position for subsequent processing normally done for the file, regardless of the actual type of processing involved: GET, PUT, INPUT or PRINT. These are fp = !$rba and FP = !$%. Use of either of these forms causes REMRA and REMBA to be marked invalid. The file is positioned so that the next GET/PUT/INPUT/PRINT verb begins processing either at rba or EOF, if no further fp is specified. No data movement occurs using these fp values, as they are allowed only in a format 2 PUT.

A PUT statement using an fp of !#rba causes the file's EOF to be set to the RBA value rba. Don't forget that the EOF value is not written to the file's FPDE until a CLOSE or a PUT fan,&& statement is executed. The EOF may be changed many times in this fashion before it is made final. An error condition is raised if the OPEN statement's mode was "D", and the RBA exceeds the current EOF value. This fp value may only be used in a format 2 PUT.

As was the case with GET for sequential input, the PUT statement can be used in a sequential output mode. A marked item or fixed item file can be created sequentially with PUT statements after having been opened with mode "O", and later read sequentially with GETS after having been opened with mode "I". The same file can be updated randomly by use of GET and PUT statements when the open mode is "R" or "D". Single data fields in FF and FI type files can be updated using partial record I/O access techniques.

Should a particular data file be especially sensitive, and require read-only random access, the use of open mode "R" is not required; open mode "I" may be used instead. The use of this particular mode will cause any PUT attempted to get a "BAD FILE MODE" error.

LOF

The function of the LOF statement is to return to the programmer the record number of the last record of the file. Its general format is:

```
LOF(fan)
```

The fan specifies the number of the filearea for which the last record number is being requested. If the file is empty, a zero is returned. LOF naturally may be used only with field item, MY and FF type files.

LOC

The LOC function, in TRSDOS BASIC, returned to the programmer, the record number last accessed via GET/PUT for a specified filearea. In NEWDOS/80 BASIC, its function has been expanded to allow the programmer to find the file location of a group of items, records or the files' EOF, or determine if the current file position is exactly at or beyond the file's EOF. Its general formats are a follows:

1. LOC(fan) performs essentially the same as in TRSDOS
2. LOC(fan)$
3. LOC(fan)%
4. LOC(fan)!
5. LOC(fan)#

where fan specifies the filearea number containing the requested information.

Format 1 (no suffix) is the one used in TRSDOS BASIC. For field item files (as are supported by that BASIC) and MF and FF files, it returns the number of the record most recently read or written via GET/PUT. If the file has not been accessed, a value of zero is returned, except in the case of a file opened using mode "E", where the record number of the last record in the file is returned. If the file being referenced is not made of fixed-length records, a "BAD FILE MODE" error condition is raised.

Format 2 ("$" suffix) is used to provide a true/false indication of the relationship of the filearea's positioning to the file's EOF. It returns a -1 (BASIC IF statement 'true') or a 0 (BASIC IF statement 'false') as follows:

For record-segmented (fixed item, MU, MF and FF type) files:

If the REMRA is valid, and the RBA of the start of the next record (not necessarily the current file position!) is equal to or greater than the EOF value, a 'true' value is returned; otherwise a 'false' value is returned.

If the REMRA is invalid and the RBA of the current file position is equal to or greater than the EOF value, a 'true' value is returned; otherwise, a 'false' value is returned.

For user-segmented (MI and FI type) files, and for print/input files:

If the RBA of the current file position is equal to or greater than the EOF value, a 'true' value is returned; otherwise, a 'false' value is returned.

Format 3 ("%" suffix) returns to the programmer the file location of the current file EOF in RBA format. This value can be used in the development of indices to the file, where the indexing item is built prior to the data record being added to the file at the EOF location. Using this form of LOC allows indices to be created during the sequential creation of the prime data file.

Format 4 ("!" suffix) returns the RBA value of the next logical record for field item, MU, MF and FF type files, if the REMRA is valid. In all other cases (including print/input files), it returns the RBA value of the current file position. For record segmented files, the value returned can be used to create an indexing item for the sequential record before the data record has been written to the file. For user-segmented and print/input files, the value returned can be used to create an indexing item for the group of data items prior to writing them to the file. For the indexing value to really be good, a PUT with a null fp, or a PRINT, must be used to write the data; nearly all other fp forms will cause the RBA value returned to be different from the actual location of the data. As with format 3, this form can be used to create indices as a sequential file is being written.

Format 5 ("#" suffix) returns the current REMRA in RBA format. A "BAD FILE MODE" error condition is raised if the REMRA is invalid, due for example, to the use of an FP m !$%. This too can be used for all file types to create indexing items for records or groups of data after, however, the record or data group has been written.

By using the values returned by LOC(fan)%, LOC(fan)! or LOC(fan)#, the programmer is able to build indices to either records (record-segmented files) or groups of items (user-segmented files and print/input files). The values, returned can be included in records/file items and later used to position the filearea via fp types !rba or !$rba.

MU FILES

The MU file type is the easiest of all NEWDOS/80's file types to implement. When it was originally conceived, it was intended as a replacement for TRSDOS's sequential file support. In TRSDOS, sequential files could not be updated; in NEWDOS/80 all but print/input and MI type files can be updated.

The MU type file is segmented into records of varying lengths and each record is detectable by the system. This attribute relieves the programmer of the need to be aware of the size of each record. The programmer can impose a smaller record size maximum than the system's maximum of 4095 bytes by specifying a lrec1 value in the OPEN statement. Any record exceeding the maximum record length will cause a "RECORD OVERFLOW" error condition.

Besides being record-segmented, the file items in a MU file are all marked. The marking bytes occupy space on the file, and must be included in any record length calculations along with the SOR byte, which marks the beginning of each record. These marking bytes identify the type of data, which follows the byte, and in the case of strings, tells the system the length of the string. Strings may be 0 to 255 bytes long, just as in BASIC; strings of 128 to 255 bytes require 2 marking bytes instead of the 1 required by all other items. Numeric items are stored on the disk in their internal form: integers as 2 bytes, single-precision items as 4 bytes, and double-precision items as 8 bytes. Don't forget that as marked-file items these lengths must be increased by 1 to 3, 5, and 9 bytes respectively.

Even though the numeric items are stored in their internal forms on the disk in all the NEWDOS/80 file types, BASIC's CVx and MKx do not (indeed, must not) be used to perform a pseudo-string conversion in order to cause this form of data storage to occur; CVx and MKx must still be used to accomplish this form of data storage for field item files, as was the case with TRSDOS BASIC.

A MU file can be created by specifying "0" as the mode in the OPEN statement; the file will be created using the data in successive PUTS without regard to the file's existence at the time of the open. AMU file may also be created using mode "R" in the OPEN statement only if the file did not exist prior to the open. A third method of creating a MU file is to use mode "E" in the OPEN statement for a previously non-existent file, or an existing file, which is empty.

A existing MU file can be expanded sequentially by specifying mode "E" in the OPEN statement. As noted above, if the file is empty, it will effectively be created rather than expanded/extended. An alternate method of sequentially expanding a MU file is to specify mode "R" in the OPEN statement. In this mode if non-null fp's are specified, the system writes padding bytes from the current EOF to the specified beginning of the new record. Any PUT to a file position less than the EOF causes an updating action to occur, not an extension of the file.

A MU file may be accessed sequentially by specifying "I" as the mode in the OPEN statement; use of this mode prevents accidental updates from occurring. The file may also be accessed randomly when opened with mode "I". If the file is non-existent at the time of the open, an error condition is raised. A MU file may also be accessed sequentially by specifying "R" or "D" as the mode

in the OPEN statement. Using these modes, if the file was non-existent prior to the open, any GET issued without a prior PUT and subsequent repositioning will cause an error condition to be raised.

A MU file may be updated by specifying mode "R" or mode "D" in the OPEN statement. The use of mode "D" precludes the expansion of the file. In either of these modes, anything from an entire record to a single item may be updated, depending upon the fp values used and the contents of the IGEL.

To understand the workings of the system on a MU type file, we'll do the following things. First, we'll create a MU file using a very simple, short BASIC program. Then, by working in the so-called calculator mode, we'll access the file and update it. To create the file, enter and RUN the following BASIC program:

```
10 CLEAR 250
20 OPEN "O", 1, "MU/DAT", "MU"
30 PUT 1,,,"ABCDEF","2ND STRING";
40 PUT 1,,,STRING$(120,"*")+"0123456789";
50 I%=2:I!=4:I#=8
60 PUT 1,,,I$,I%,I!,I#;
70 CLOSE
```

Save the program with an appropriate name just in case you need it later.

Now, notice that the program uses the simplest form of IGEL in statements 30 and 40; the values to be written to the file are in the IGEL proper. The PUT at 60 references the four different BASIC data types: string, integer, single precision and double precision. Notice also that no lrec1 specification was in the OPEN statement. This allows the records to be as much as 4095 bytes long.

Run the program to create the file named "MU/DAT". For study purposes, run the SUPERZAP program using DFS to read the sector written by MUFILE.

The first byte of the sector is a hex 70. This is the SOR byte. All records in a MU file start with this byte. Be aware that not all hex 70's are start of record bytes, however, that particular bit configuration can occur in numeric values as well as in strings where it is a lower-case "p".

The second byte is a marking byte identifying the next 6 bytes as a string. Adding 6 to the displacement of the first byte of the string will give us the displacement of the marking byte for the second string (a hex 8A). It defines a string 10 bytes long. If you now count to the 11th byte down from that marking byte, the SOR byte for the second record will be found (at displacement hex 13). The following marking byte (a hex 71) identifies a string of greater than 127 bytes long; the byte following that marking byte contains the length, and is not a part of the string data itself. A little hex arithmetic at this point will show that the SOR byte of the third record will be found at displacement hex 98. The marking byte following that SOR identifies a string zero bytes long: a null string. The next marking byte (hex 72) identifies the following 2 bytes as an integer number. Following the integer is a marking byte (hex 73) identifying the next 4 bytes as a single precision number. Following that number is a marking byte (hex 74) identifying the next 8 bytes as a double precision number. At this point

(displacement hex AB) we've exhausted the data we actually wrote to the disk; any data, which follows, is unpredictable.

Now that we've seen how data is stored in a MU file, as well as any other marked item file for that matter, we'll access the data using GETS in the "calculator mode" and analyze the results. Later, we'll introduce a few errors. Before going any further, return to the BASIC READY state, enter CLEAR 50 and NEW, and type in the following three-line program (this will save steps later).

```
10 PRINT LOC(1)$; "$ EOF TEST   "; LOC(1)%; "% EOF RBA"
20 PRINT LOC(1)!; "! NEXT RCD RBA   ";
30 IF LOC(1)! = 0 THEN PRINT ELSE PRINT LOC(1)#; "# REMRA"
```

The purpose of the program is to display the file positioning values available to us. For the sake of clarity, the first character of the string identifies the LOC suffix used to get the value displayed and the remainder of the string a mnemonic associated with that particular LOC function. You may want to save this program also, as it will be used in experiments with all the other file types later.

The first thing to do now is to open the file for input. Type in:

```
OPEN "I", 1, "MU/DAT", "MU"
```

Now enter "GOTO 10" to run the program entered a moment ago. (You must use GOTO rather than RUN because RUN closes any open files.) The system will respond with:

```
0 $EOF TEST  171 % EOF RBA
0 ! NEXT RCD RBA
```

Notice that the REMRA value isn't printed. That's because the value hasn't been set yet, and is marked as invalid by the system. Because the program we entered isn't too smart, it simply checks for a zero next record value, rather than attempting to be sensitive to the actual validity of the REMRA.

Now we'll read the first record in its entirety. Type in:

```
GET 1,,,A$,B$; : PRINT A$, B$ : GOTO 10
```

The system will respond with:

```
ABCDEF 2ND STRING
0 $ EOF TEST   171 % EOF RBA
19 ! NEXT RCD RBA   0 # REMRA
```

Notice that the two EOF related values have not changed, but that the next record RBA has. It now contains the decimal displacement of the SOR byte of the second record. This is the normal action of the GET on a record-segmented file. Notice also that the REMRA has now appeared, and that it has a value of zero. Remember that for record segmented files the REMRA contains the RBA of the latest record involved in the GET or PUT for that filearea, unless its has been marked invalid due to the use of OPEN or !$RBA.

Now we'll go back and read the first record again in its entirety by using the fp value which causes file positioning back to the REMRA value. To prove the record has been read a second time, we'll reverse the order of the variable names. Type in:

```
GET 1,#,,B$,A$; : PRINT A$, B$ : GOTO 10
```

The system will respond with:

```
2ND STRING   ABCDEF
0 $ EOF TEST   171 % EOF RBA
19 ! NEXT RCD RBA   0 # REMRA
```

Again, the EOF values have not changed. This time, however, neither have the other two values. This is because the file's next record pointer was changed to the REMRA value prior to the data transfer. The next record pointer was then moved to the REMRA, followed by the transfer of the data to the named variables. The same general method is followed when !rba is specified for the fp.

Let's get daring now, and ignore the contents of the next record (the one with the 120 asterisks in it), and at the same time position ourselves to process the third record. Type in:

```
GET 1,,,; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   171 % EOF RBA
152 ! NEXT RCD RBA   19 # REMRA
```

Nothing really surprising there; again, in the case where no file positioning was specified in the GET, the next record RBA was moved to the REMRA. With the lack of variable names in the IGEL, no data transfer occurred, and the file was left positioned to the record's first item.

Now let's try some of partial record I/O. We'll start by transferring only the string from the third record, and leave ourselves positioned so that the next transfer will begin at the integer. Type in:

```
GET 1,,,A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```
(blank line)
-1 $ EOF TEST   171 % EOF RBA
171 ! NEXT RCD RBA   152 # REMRA
```

Several things should be noted here. Since the file was left positioned to the 2nd record's 1st item by the previous GET and the GET in this example specified fp = (null), the file was automatically advanced to the beginning of the third record's 1st item by this GET's file positioning phase. Then the 3rd record's first item was read, and the file was left positioned to the 3rd record's second item. We've started processing the last record in the file. The system hasn't told us that, but has made the information available to us through the LOC(fan)$ statement. in common sequential data processing

situations, the EOF status of a file is tested as a function of the GET logic, and transfer of control is made to an end-of-data routine specified by the programmer. As no provision has been made for the specification of such a routine in NEWDOS/80, the EOF status of the file must be tested immediately prior to the GET statement attempting to transfer the next record's contents into memory, and appropriate action taken if the EOF condition is found to be true.

Notice that the LOC(fan)! value is the same as the EOF RBA value, even though we transferred only the first item of the last record. This is because in the case of record-segmented files, the function returns the RBA of the next record. Only when it is used on a user-segmented file does it return current file position. If you've gone back to chapter 8, you've seen that there's no way to get the current file position back from the system. There isn't, nor is there a way to get the REMBA either. Somebody out there will probably find a way via PEEKS and so on, but the fact remains that BASIC itself doesn't have provision for telling you simply and directly.

To show that we are indeed positioned at the record's 2nd item, the integer, we'll read just that field. Type in:

```
GET 1,*,,I; : PRINT I : GOTO 10
```

The system will respond with:

```
2
-1 EOF TEST   171 EOF RBA
171 NEXT RCD RBA   152 REMRA
```

Did you notice the variable type of "I"? It's single precision, but the file item transferred to it was an integer. The changing of type between a file item and a variable is allowed, so long as it is allowed in BASIC.

Now let's go back and transfer the integer and the single precision items using the REMBA to position the file before the transfer. Type in:

```
GET 1,$,,K,J; : PRINT J; K : GOTO 10
```

The system will respond with:

```
4  2
-1 $ EOF TEST   171 % EOF RBA
171 ! NEXT RCD RBA   152 # REMRA
```

The REMBA was set to the file's RBA at the start of the previous GET. Regardless of the number of fields transferred or bypassed, the starting byte RBA is remembered. Again, none of the LOC functions has changed.

To prove that the REMBA hasn't changed with the multiple file item transfer, let's transfer the integer and the double precision items next. Type in:

```
GET 1,$,,J,,I; : PRINT I; J : GOTO 10
```

The system will respond with:

```
8  2
-1 $ EOF TEST   171 % EOF RBA
171 ! NEXT RCD RBA   152 # REMRA
```

Notice that by omitting a variable name in the IGEL in the position where the single precision file item occurs, that the file item is bypassed. Again, both file items have their types changed as they are moved to the variables.

Now we'll try some RBA positioning to see how that works. Type in:

```
GET 1,l0,,A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```
ABCDEF
0 $ EOF TEST   171 % EOF RBA
19 ! NEXT RCD RBA  0 # REMRA
```

The use of a specific RBA provided by the programmer, whether it's a number as in this example, or some variables contents, or an expression, causes the RBA to be moved to the next record pointer just as the REMRA is moved there when "#" is used for fp. The sequence of actions is the same from that point on for the two fp's just mentioned.

Let's try the other RBA positioning technique. Type in:

```
I=152 : GET 1,!$I : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   171 % EOF RBA
152 ! NEXT RCD RBA
BAD FILE MODE
```

Hey! Was that supposed to happen? You bet! Both the REMRA and REMBA were tagged as invalid by the system due to the fp type used. It does nothing more than set the next record pointer. No data transfer occurs.

Now we'll try it again, but this time with a "later" data transfer. Type in:

```
GET 1,!$19 : GET 1,,,A$ : PRINT A$ : GOTO 10
```

The system will respond with:

```
OUT OF STRING SPACE
```

Another error? Why? Because when we started this session, we did a CLEAR 50, and the string we're trying to transfer is 130 bytes long. Don't forget that NEWDOS/80 doesn't change the string variable's pointer to point to the buffer, but moves the string to the BASIC string space at the top of memory as if a LET statement had been executed. Now type in:

```
GET 1,,,(10)A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```
**********
0 $ EOF TEST   171 % EOF RBA
152 ! NEXT RCD RBA   19 # REMRA
```

NOTE: the same file item was inputted as for the previous GET. Due to the error that occurred, the filearea, but not the data, was restored to what it was at the beginning of that previous GET. Note that only the first 10 asterisks of the 120 in the file item were transferred to A$.

That just about exhausts the fp's we can use. The ones not covered yet are fairly well explained in chapter 8. It is time now to try some updating of records, both in whole and in part. Before we can do that however, the file must be opened for input and output. Type in:

```
CLOSE : OPEN "R", 1, "MU/DAT", "MU" : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   171 $ EOF RBA
0 ! NEXT RCD RBA
```

That is just as it was after the open for input only. The mode we just specified allows the file to be expanded (which we will do shortly). If we wanted to not allow the ability to expand the file beyond its existing EOF, we would have specified mode "D".

First, let's simply replace the first record on the file with a single field. Type in:

```
PUT 1,,,"RECORD REPLACED"; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   171 % EOF RBA
19 ! NEXT RCD RBA   0 # REMRA
```

Notice that the next record pointer is pointing to the second record, just as if a GET were issued.

Now, let's replace the double precision value in the third record with 3 times its complement. Type in:

```
I=152 : GET 1,!I,,,,;
GET 1,*,,D#; : PUT 1,$,,3*-D#; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   171 % EOF RBA
171 ! NEXT RCD RBA   152 # REMRA
```

Again, the system is ready to process the next record even though it's positioned at EOF. We can't transfer any information from this file position, but can write additional new records to the file.

To demonstrate this, type in:

```
PUT 1,,,"THIS IS THE FOURTH RECORD"; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   198 % EOF RBA
198 ! NEXT RCD RBA   171 # REMRA
```

It's easy to see that the file has been extended. You should be aware that the new EOF hasn't yet been recorded in the FPDE in the directory. If there were to be a power outage at this point, our little example file would show no change from when we first opened it for update. We could ensure that the file has the new data recorded in it by doing a PUT using the fp of &. That would only write the buffer to the file. To update the FPDE's EOF value, either a CLOSE or a PUT fan,&& must be done. A CLOSE will also write out an updated buffer, if any.

Now let's go back to the second record and replace its single file item with several smaller ones. We'll do this using a couple of PUTS. Type in:

```
PUT 1,!19,,"ITEM 1",3.14159*2;
PUT 1,*,,"ITEM 3",4,10D2;
PUT 1,*,,"LAST ITEM RECORD 2"; : GOTO 10
```

The system will respond with:

```
$ EOF TEST   198 % EOF RBA
152 ! NEXT RCD RBA   19 # REMRA
```

Once again, the next record pointer has the RBA of the record following the one we're processing, and the REMRA has the RBA of the record last processed. Note that all three PUT statements wrote items into the same record.

To show that the record has been updated type in:

```
GET 1,#,,A$,I,B$,J,K,C$; : PRINT A$,B$,C$,I,J,K : GOTO 10
```

The system will respond with:

```
ITEM 1   ITEM 3   LAST ITEM IN RECORD 2
6.28318 4 1000
0 $ EOF TEST   198 % EOF RBA
152 ! NEXT RCD RBA   19 # REMRA
```

That's pretty conclusive, isn't it? If we were to try to GET more data using the fp = *, we would find a "RECORD OVERFLOW" error staring back at us. We could, if we wanted to, add more data to this particular record, just as long as we didn't exceed its total original length of 131 bytes.

The only thing remaining to be done is to update the EOF value on disk. To do this, simply type in:

```
CLOSE
```

It should be noted, we could have used the statement:

```
PUT 1,&&
```

to update the EOF into the directory without closing the file. We could then have continued processing the file.

Once again, let's examine the file using SUPERZAP. Now you'll find SOR bytes at displacements 0, 13, 98 and AB. Examine The first record closely. The string marking byte (hex 8F) shows a length of 15 bytes. Adding hex F to the starting displacement of the string yields a result of hex 11. Looking at that displacement, you'll find the first of two bytes of hex 00. These are fill bytes which are skipped by the system as GETS are processed. If we were to try to retrieve two strings from the first record, as were there before our little updating session, we'd get a "RECORD OVERFLOW" error in response as there is now only one string item in the record. The system pads out a logical record with fill items when it finds that the data being written to the record has fewer bytes in it than were in the record to start with.

In the second record, starting at displacement hex 13, you'll find the SOR byte followed by a marking byte defining a string of 6 bytes. Counting down to the 7th byte from that marking byte, you'll find a marking byte defining a single precision numeric value. Five bytes further on you'll come across a marking byte defining another 6 byte long string. Seven bytes down from that byte is a marking byte defining an integer. The third byte beyond that is a double precision number marking byte. Nine bytes from there is the marking byte for the last item in the record, a hex 8A, defining a 10 byte long string. The remainder of the record following the string to displacement hex AB is filled with fill bytes. If it became necessary to replace record 2 with totally new data, the new record could take as many as 133 bytes, SOR byte included. All that is there right now would be replaced if the proper fp's were used.

The remainder of the record should be quite self-explanatory. The only differences between its first contents and now are the double precision number at displacement hex A2, and the new fourth record starting at AB and having its last byte at C5.

This discussion doesn't show all that can be done with MU files, of course. It is intended to show many of the abilities built into NEWDOS/80 BASIC file support. For those of you with data base experience, the partial-record I/O should look somewhat familiar. It is, after all, one of many data base abilities to update a single field in a record. Granted, NEWDOS/80 doesn't have the built-in file item security that data bases have; that is something you'll have to build into your systems as you see fit. But for now, you'll have to agree that NEWDOS/80's abilities are far superior to anything else available on the market.

For those of you getting into file processing for the first time, don't be daunted by the apparent complexities of the methods available to you. The best thing that you can do is to continue on with exercises similar to what we've just done here. As you practice, the concepts will seem to become easier to understand and work with.

MF FILES

Now that you've experimented with the MU file type and feel somewhat more comfortable with some of NEWDOS/80's capabilities, we'll go on now to experiment a little bit with the MF file type. Returning to chapter 8 you'll find that an MF file type is made up of marked items, and is record-segmented with all records having the same length. In other words, it is a marked item, fixed length record file. The length of the record is defined to the system by the lrec1 operand of the OPEN statement.

Like the MU file type the MF file type can be updated with new data items on a record by record basis. The updating data need not be the same data type or length as the original data, nor does there have to be the same number of items in the updated record as there were to start with. You must be mindful of the file position being used during the updating of an MF file, just as you were with the MU file. The update can start in the middle of the record just as easily as at the beginning; the same fp controls are available to you for MF files as there were for MU files. Don't lose sight of the fact that when updating marked item files, all bytes from the current file position to the end of the record are re-written, whether you had really intended that to happen or not.

We'll use the same technique to experiment with the MF file as we used for the MU file. First we'll have to create a file for use as the experimental base. Enter the following BASIC program, and save it in case you need it again later.

```
10 OPEN "0", 1, "MF/DAT", MF", 20
20 PUT 1,,,"STRING1", "STR 2", "STR3";
30 PUT 1 "MAXIMUM STRING (19)";
40 I!=4 : I#=8 : I%=2
50 PUT 1,,,I#,I!,I%;
60 PUT 1,,,I#*10,I!*100,I%*1000;
70 CLOSE
```

Now run the program to create the file. When its done, run SUPERZAP using DFS to display sector 0 of the file just created. The first thing you'll notice is that there is no SOR byte at the beginning of the sector. That's because only MU files use them to define the start of records which are all presumed to have different lengths; other record-segmented file types have fixed length records so the system "knows" where each record begins. In the first byte is a marking byte describing a 7 byte long string. At displacement 8 is the marking byte describing a 5 byte long string, and at displacement E one describing a 4 byte string. Progressing down to displacement 13, where the next marking byte should be, you'll find a padding byte (00 hex). Remember that the records in the file we created are 20 (14 hex) bytes long. We wrote 3 items of 7, 5 and 4 bytes length respectively giving an aggregate byte count of 19; one fill byte is used to complete the 20 byte record.

The second record starts at displacement 14, where you'll find a marking byte describing a 19 (13 hex) byte long string. The one item is the entire record. The third record starts in displacement 28. You'll find marking bytes located at 28, 31 and 36 describing a double precision item, a single precision item and an integer respectively. This record has an aggregate data length of 17 bytes, and thus requires 3 padding bytes, which you'1l find in displacements

39 through 3B inclusive. The fourth and last record we wrote has a data structure identical to that of the third record. Its marking bytes are located at displacements 3C, 45 and 4A; its padding bytes are in displacements 4D through 4F. The data beyond 4F is unpredictable. It is in fact whatever was in the sector before we created the file.

Return to BASIC and retrieve the location displaying program originally used when experimenting with MU files. It should read:

```
10 PRINT LOC(1)$; "$ EOF TEST   "; LOC(1)%; "% EOF RBA"
20 PRINT LOC(1)!, "! NEXT RCD RBA   ";
30 IF LOC(1)!=0 PRINT ELSE PRINT LOC(1)#; "# REMRA"
```

We'll use this program in the same way we did fox the MU file experiments to show the results of GETS and PUTS on file position. The experiments we'll go through won't be as thorough as the ones done for the MU file. Instead they'll touch on the major differences between the two file types.

To start with, we'll open the file and examine the results of the LOC statements. Type in:

```
OPEN "I", 1, "MF/DAT", "MF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   80 % EOF RBA
0 ! NEXT RCD RBA
```

Except for the EOF RBA, the results are the same as for the MU file. The system is ready to process the record starting at displacement 0, the first logical record.

Now type this in:

```
GET 1,,,,A$,B$; : PRINT A$, B$ : GOTO 10
```

The system will respond with:

```
STR 2    STR3
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

Notice that the last two items of the record were transferred. This is due to the null where the first variable name would normally reside (after the third comma).

From the current file position we can go back and transfer again the first two items of the record by using REMRA positioning. Type in:

```
GET 1,#,,A$,B$; : PRINT A$,B$ : GOTO 10
```

The system will respond with:

```
STRING1   STR 2
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

Nothing overly tricky there. As with MU files, we can continue processing the same record.

To do just that, type in:

```
GET 1,*,,C$; : PRINT C$ : GOTO 10
```

The system will respond with:

```
STR3
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

The fp "*" value tells the system to continue processing from where it left off on the preceding GET or PUT; in other words, from the current file position. If the GET had asked for two or more items, record overflow error would have occurred as the record, at that point, contained only one more item.

Now let's try processing the fourth logical record without first processing the second or third. Type in:

```
GET 1,!(4-1)*20,,J,K,L; : PRINT J; K; L : GOTO 10
```

The system will respond with:

```
80   400   2000
-1 $ EOF TEST   100 % EOF RBA
100 ! NEXT RCD RBA   80 # REMRA
```

Notice that the expression used in the !rba type fp specifies a value equal to 60. The numbers themselves represent the logical record number we really wanted, minus 1, times the record length. !rba positioning in a MF file, or a FF file too, is quite simple, as you can see. Just as something for you to do on your own, try the same statement as you just entered, using the rn form of fp instead of the !RBA form.

To do this, you should have changed the PUT statement to be:

```
GET 1,4,,J,k,l;
```

Now let's try some simple random updates to the records and check the results. Prepare the file for this by typing in:

```
CLOSE : OPEN "R", 1, "MF/DAT", "MF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   80 % EOF RBA
0 ! NEXT RCD RBA
```

Those are exactly the same results as when we opened the file for input. Again, no big surprise there.

As a starting point, let's replace the first record. Type in:

```
PUT 1,,,I$; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

The responses show that the first logical record has been processed. You should be aware that even though the next record RBA shows a value of 20, the current file position is in fact equal to 1 as the above PUT replaced the entire contents of the record with a null string (an 80H marker byte only) and 19 bytes of zeroes, then repositioned the file back to the byte following the null string. If we were to write to the current file position using fp = *, the PUT's first marking byte would be placed in the second byte of the file.

Just for fun, let's add two fields to the record we just updated. Type in:

```
PUT 1,*,,"2",2; : GOTO 10
```

The system will respond:

```
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

We'll see the results of this last update in a moment.

Now, let's add two more records to the end of the file. Type in:

```
PUT 1,!%,,"RCD 5"; : PUT 1,,,"RECORD 6"; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   120 % EOF RBA
120 ! NEXT RCD RBA   100 # REMRA
```

The numbers indicate that the file is now six records long.

Close the file now, and enter the SUPERZAP program; use the DFS function to display sector zero of the file again. The records in the file begin at displacements 0, 14, 28, 3C, 50 and 64 respectively. The marking byte at displacement 0 describes a null string; the one at 1 a string 1 byte long and the one at 3 an integer. Notice that the remainder of the record has been padded with fill (00 hex) characters. The contents of the fifth and sixth

records should need no explanation. You should notice that the data beyond the sixth record was not modified by our little updating session. The system ignores this area of the sector as it is file space at and beyond the file's EOF and therefore not really part of the file.

As short as this session was in comparison to the one for MU files, you should now be aware that MF files are not at all hard to manage. Depending upon your own leanings, an individual record can be retrieved for update by either lrba positioning as shown in the example, or by using the record number itself On fp positioning).

MI FILES

Now we come to the last of the marked item files - the MI file type. Its most important differences from the MU and MF file types are:

1. MI files cannot be updated.
2. MI files have no system-recognizable record lengths.

These differences restrict this file type to being used for compact reference file only, as they can only be written to or extended, and later read again. Also, to get to any specific data group or item in a random-access fashion, !rba positioning (or its logical equivalents) must be employed.

Because you've seen marked item files in some detail by now, the experimental files accesses we've employed to this point will be quite limited and intended to amplify the differences in structure and access methods rather than similarities.

To start with, retrieve the program we used to create the MF file and change it to read as follows:

```
10 OPEN "O", 1, "MI/DAT", "MI"
20 PUT 1,,,"STRING1","STR 2","STR3";
30 PUT 1,,,"MAXIMUM STRING (19)";
40 I!=4 : I#=8 : I%=2
50 PUT 1,,,I#,I!,I%;
60 PUT 1,,,I#*10,I!*100,I%*1000;
70 CLOSE
```

Note that only line 10 of the program is changed from the MF file example.

Save the program if you wish, and run it. A user-segmented file will be created containing some 73 bytes of rather unlikely-looking data. Now exit BASIC and enter SUPERZAP, and use the DFS function to display sector zero of the file just created.

You'll see that there aren't any SOR marking bytes or padding items in the sector. There aren't any records in so far as BASIC is concerned, just a string of data items. The data in the file and its structure and organization are entirely the responsibility of the programmer. All you'll see in the sector is a series of contiguous marked data items. Good data design on the programmer's part demands that there be some rational, coherent data structure for the data items to be at all usable.

All there is in the file we created is unrelated data items. To access them sequentially would require the intimate knowledge we have: there are four strings and six numeric items. To access them randomly requires that we know the specific RBAs of the marking bytes. Otherwise at best, a "BAD FILE DATA" error will be raised by the system; at worst, it will return incoherent data.

Now, let's examine the SUPERZAP dump of the sector. The string marking bytes occur at displacements 0, 8, E and 13. The first set of numeric items have their marking bytes at 27, 30 and 35; the second set at 38, 41 and 46. We'll use these numbers (displacements, all in hex) in just a moment to access the data. By the way, the EOF RBA is 49.

Return to DOS BASIC at this time, and load the same location printing program as you used for MU and MF files. This program will aid in showing the lack of logical record support afforded to MI files by the system.

As usual, the file must be opened for access. Type in:

```
OPEN "I", 1, "MI/DAT", "MI" : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   73 % EOF RBA
0 ! NEXT RCD RBA
```

As with other file types, the input mode open positions the system so that the next byte to be processed is the first byte in the file, if a (null) fp is used.

To show a different positioning resulting from open, and to extend the files besides, type in:

```
CLOSE : OPEN "E", 1, "MI/DAT", "MI" : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   73 % EOF RBA
73 ! NEXT RCD RBA
BAD FILE MODE IN 30
```

This last message is due to the fact that the location printing program tries to print the REMRA value when it has just been marked invalid by the system as a result of the open itself. (The location printing program tries to display REMRA because the next record RBA is non-zero.)

The file is now in an output mode. To prove this we'll extend the file by three integer items. Type in:

```
PUT 1,,,-1,-2,-3; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   82 % EOF RBA
82 ! NEXT RCD RBA   73 # REMRA
```

Notice that the EOF RBA is 9 bytes higher in the file, and that the REMRA has the original EOF RBA value. In MI processing the REMRA is always set to the same value as the REMBA; they both equal the file position at the beginning of the GET or PUT data transfer.

Now, let's go back and reference a few of the data items. Type in:

```
CLOSE : OPEN "R", 1, "MI/DAT", "MI"
GET 1,!19,,A$; : PRINT A$ : GOTO 10
```

The system will respond with:

```
MAXIMUM STRING (19)
0 $ EOF TEST   82 % EOF RBA
39 ! NEXT RCD RBA   19 # REMRA
```

The REMRA reflects the starting RBA of the GET, and the next record RBA points to the first of the numeric items. If no overriding fp were specified, that is where the next GET would start examining items for transfer.

To show this, type in:

```
GET 1,,,,J%,K#,I!; : PRINT J%; K#; I! : GOTO 10
```

The system will respond:

```
4  2  80
0 $ EOF TEST   82 % EOF RBA
65 ! NEXT RCD RBA   39 # REMRA
```

Notice that once again all the items in the IGEL are of a different numeric type than the file items being transferred to them. One of the marked item file's intrinsic powers is this numeric type conversion.

To show that in an MI file the REMRA and REMBA are the same, we'll have to do the same basic thing twice, with the appropriate fp characters. First type in:

```
GET 1,#,,I,J,K; : PRINT I; J; K : GOTO 10
```

Then enter:

```
GET 1,$,,I,J,K; : PRINT I; J; K : GOTO 10
```

In both cases, the system will respond with:

```
8  4  2
0 $ EOF TEST   82 % EOF RBA
56 ! NEXT RCD RBA   39 # REMRA
```

Q.E.D. Don't lose sight of the fact that this REMRA equals REMBA relationship is true at all times for field item, MI and FI files, and for MU, MF and FF files only when the GET/PUT data transfer starts at the beginning of a logical record.

Now, to show that an MI file can be extended after having been opened with mode "R", type in:

```
      PUT 1,1%,,15,-15; : GOTO 10
```

The system will respond with:

```
      -1 $ EOF TEST 88   % EOF RBA
      88 ! NEXT RCD RBA   82 # REMRA
```

The EOF has been extended by 6 bytes as expected. The file is left positioned to continue adding data to the end of the file if fp = (null) or * are employed.

This about exhausts the experiments we can perform on MI files. On your own, you can try to update a single existing item. (You'll get a "BAD FILE MODE" error -- chapter 8 specifies that MI files cannot be updated.) If you are unsure of what will happen if an MI file has been opened with some mode, and a certain fp is specified in a GET/PUT, create the situation with a small file from BASIC's calculator mode and try it - it's the surest way to find out what does happen.

FF FILES

The fixed item file is different from the marked item file in several respects. To start with, it has no marking bytes for each item or record; all item description is taken from the IGEL, not the file. Because of this, if you describe a string item of 20 bytes to be read, that's exactly what will happen, even if the data written to the file originally was numeric. Also, it is required that numeric items written to file are read back as the same type; otherwise file synchronization is lost.

A second major difference is that fixed item files can be updated using true partial-record I/O. That is to say, a single field in a fixed item record may be updated without affecting any surrounding fields, whereas, in a marked item file, the field to be changed and all other fields to the end of the record had to be written.

A third significant difference is that the expressions in the IGEL cannot be anything more than variable names, with mandatory (len) prefixes for string items. This is due to the indeterminate type/length of an item resulting from an expression.

Fixed item files come in two types: FF files, in which all records have the same length, and FI files which have no BASIC detectable records. For the moment, we'll concern ourselves with only the FF type file.

As with the marked item discussions, we'll create an FF file, then experiment with it in "calculator mode". Enter the following program and save it if you wish. Then run it to create the FF file.

```
10 CLEAR 100
20 OPEN "O", 1, "FF/DAT", "FF", 20
30 PUT 1,%,40 : GOTO 50
40 (20)I$;
50 LSET I$="ABCDEFIJK"
60 PUT 1,,,(20)I$;
70 LSET I$="12345678901234567890"
80 PUT 1,,,(20)I$;
90 I%=2:I!=4:I#=8
100 PUT 1,,,(4)I$,I%,I!,I#;
110 I%=I%*10 : I!=I!*100 : I#=I#*1000
120 PUT 1,,,(4)I$,I%,I!,I#;
130 CLOSE
```

You will have noticed that this program is a little different than those used for the marked item files. For one thing, the string items are written from variables rather than literals in the IGEL proper. Additionally, no expressions as such were used to place numeric data on the file. In assigning values to the variable I$, LSET was used instead of the (implied) LET. This latter was done to preserve the length of I$ set up in the pseudo FIELD operation done in lines 30 and 40. This pseudo FIELD operation is not required if your program can live with the fact that variables providing string data to the file are NOT padded on the right while variables receiving data from the file are padded. Note too that all string items in the IGELS

have length prefixes. It's these prefixes that actually determine how many bytes of string data are to be transferred to/from the file, not the pseudo FIELD operation (refer to lines 100 and 120). Remember that the file string items are padded or truncated on the right as necessary to meet the length prefix's demands.

If we had elected NOT to use the pseudo FIELD function, the program could have been written:

```
10 CLEAR 1000
20 OPEN "O", 1, "FF/DAT, "FF",20
50 I$="ABCDEFIJK"
60 PUT 1,,,(20)I$;
70 I$="12345678901234567890"
80 PUT 1,,,(20)I$;
and so on
```

Now run SUPERZAP, and use the DFS function to examine the sector just written. You'll see that the first record (hex 11 bytes long) has the nine data bytes we had intended to transfer padded to 20 bytes with 14 blanks (the blank padding is due to the use of LSET in the first encoding above and due to the PUT in the second). The second record has no padding - the string item we wrote was twenty bytes long in the first place (had it been longer, the LSET in the first encoding would have truncated the variable I$ on the right and the PUT in the second would have truncated the file item). The third and fourth records have identical formats: a four byte string, a two-byte integer value, a four-byte single precision value, an eight-byte double-precision value and two padding bytes. Again notice that there are no marking bytes to describe the type of the file item. The file's EOF is at displacement 80 (hex 50). Any bytes in the sector at or beyond this displacement were unmodified by the running of the program as those bytes are not part of the file.

Reload the program that was used to display the results of the LOC function as was used in the MU file experiments - we'll use it once again to demonstrate how file position is maintained.

To demonstrate the file's position after open, type in the following:

```
OPEN "I", 1, "FF/DAT", "FF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   80 % EOF RBA
0 ! NEXT RCD RBA
```

As expected, the system is positioned to process the next (first) record on the file.

To transfer the first record, type in:

```
GET 1,,,(20)A$; : PRINT LEN(A$); A$ : GOTO 10
```

The system will respond with:

```
20 ABCDEFIJK
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

As you can see, 20 bytes were transferred to the variable named in the IGEL. We could just as easily have transferred a part of the record if we had wanted.

Just to show how this can be done, we'll assume that the record consists of 3 6-byte items and transfer them individually in separate GETS. Of course we'll have to use some special fp values to accomplish this task. Type in:

```
GET 1,#,,(6)A$; : GET 1,*,,(6)B$; : GET 1,*,,(6)C$;
PRINT LEN(A$); A$ : PRINT LEN(B$); B$ : PRINT LEN(C$); C$ : GOTO 10
```

The system will respond:

```
6 ABCDEF
6 IJK
6
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

Here we've read 3 fields from the same record using as many GET statements to do it. This shows one of the freedoms of partial record I/O.

Another of the freedoms available to you is the ability to skip over bytes in a record to get to the ones you really want. We'll do that now with the second record. Type in:

```
GET 1,,,(12)$,(4)A$; : PRINT LEN(A$); A$ : GOTO 10
```

The system will respond with:

```
4 3456
0 $ EOF TEST   80 % EOF RBA
40 ! NEXT RCD RBA   20 # REMRA
```

The 12 bytes we skipped could just as easily have been 6 integers as a 12 byte ASCII string. The point being made is that the system neither knows nor cares what data types or items are being skipped, only that Men) bytes are being skipped.

Now we'll make a slight error in processing the fourth record - we'll forget for a moment that was written with a 4-byte string at the start. Type in:

```
GET 1,4,,I%,I!,I#; : PRINT I%; I!; I# : GOTO 10
```

The system will respond with:

```
12849  0  0
-1 $ EOF TEST   80 % EOF RBA
80 ! NEXT RCD RBA   60 # REMRA
```

Certainly not what we wrote! It does point out the need for consistent record description within FF (and FI, for that matter) files. Unlike a marked item file, in which this error would have been detected and reported, the fixed item processing demands that whatever is at the current file position be transferred to the named variable; no checks are done or can be done to prevent this type of error. (The reason for the zero values showing in the display for the single and double-precision numbers is that their exponent bytes were zero).

You'll notice that we're now also positioned at EOF, or at least apparently so. In fact the current file position, in so far as the system is concerned, is the 15th byte of the record. The LOC(fan)! returns the RBA of the start of the next sequential record to be processed; that is, the one which would be processed with an fp = (null).

Just to show that we are positioned at the 15th byte, type in:

```
FOR I=1 TO 6 : GET 1,*,,(1)A$; : PRINT ASC(A$); : NEXT
```

The system will respond with:

```
0  0  122  141  0  0
```

A little decimal-to-hex conversion will show the non-zero values to be the most significant mantissa byte and exponent byte respectively of the double precision number originally written as record 4.

Now let's go back and process record 4 correctly. Type in:

```
GET 1,4,,(4)A$,I%,I!,I#; : PRINT I$;I%,I!,I# : GOTO 10
```

The system will respond with:

```
1234 20 400 8000
-1 $ EOF TEST   80 % EOF RBA
80 ! NEXT RCD RBA   60 # REMRA
```

Just like it was written in the first place. You've noticed, of course, that the 4th record was processed using the rn form of the fp specification. When developing indices to fixed-length record files (FF, MF and field item) a couple of bytes can be saved by using integer items containing record numbers for the indices instead of single precision items containing the RBA value returned from some LOC function. Random access to a fixed-length record file is just as reliable using rn positioning as using RBA positioning, and perhaps a little easier to understand when examining an index item's contents.

Now let's close the file and open it for some examples of updating. Type in:

```
CLOSE : OPEN "R", 1, "FF/DAT", "FF", 20 : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   80 % EOF RBA
0 ! NEXT RCD RBA
```

What we'll concentrate on is partial-record I/O. It's in this area that the fixed item files really have it over marked item files. Let's assume that the first and second records in our file have the same format: 4 5-byte long items each. Now let's update the 2nd item in the 1st record, and the last in the 2nd record. Type in:

```
A$="2ND" : PUT 1,1,,((2-1)*5)$,(5)A$; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   80 % EOF RBA
20 ! NEXT RCD RBA   0 # REMRA
```

Now, to update the 2nd record, type in:

```
A$="LAST" : PUT 1,2,,((4-1)*5)#,(5)A$; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   80 % EOF RBA
40 ! NEXT RCD RBA   20 # REMRA
```

You may have noticed that positioning to the field in the record was done by computing the number of bytes to be skipped. In the 2nd record, the positioning to the last 5-byte field didn't simply skip over the preceding 15 bytes, but nulled them out in the process. We'll see the effects of this later.

Now let's update the integer items in the 3rd and 4th records. Type in:

```
I%=-50 : PUT 1,3,,(4)$,I%; : PUT 1,4,,(4)$,I%;
GET 1,3,,(4)J$,J%,J!,J#; : PRINT J$,J%;J!;J# : GOTO 10
```

The system will respond with:

```
1234      -50  4  8
0 $ EOF TEST   80 % EOF RBA
60 ! NEXT RCD RBA   40 # REMRA
```

The first line of the response shows that our update affected only the one field we wanted to mess with - the other fields in the record were not modified. In MF and MU files having similar records (allowing for marking bytes), the integer, single-precision and double-precision values would all have had to be specified in the IGEL in order to have updated just the integer. That statement isn't quite complete: the single and double-precision numbers would have to have been read first to maintain the correct values; also, they could have been written back individually using the various fp values. Again, here in an FF file, we had only to skip over the bytes we wanted to, and write the single field to be modified.

While in the "R" mode, any NEWDOS/80 file may be extended. To show this feature, type in:

```
PUT 1,6,,J%,J%,J%; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   100 % EOF RBA
100 ! NEXT RCD RBA   80 # REMRA
```

In this example we entirely skipped over the 5th record of the file. The system, in order to maintain the necessary record orientation, created and wrote a 5th record containing only nulls before writing the 6th record as we specified.

Now close the file, run SUPERZAP, and use the DFS function as before to examine the 1st sector of the file. You'll notice that the 2 string records (numbers 1 and 2) have been updated as required, that the integer items in the Ad and 4th records both read CEFF, that the 5th record (displacements 50-63 hex inclusive) is all nulls, and that the 6th record contains 3 repetitions of CEFF hex, followed by 14 nulls.

FI FILES

We now come to the last of NEWDOS/80's unique file types: the FI file. Like the MI type file, it is a user segmented file; and like the FF type file, it is made of fixed items, rather than marked items. Unlike the MI type file, the FI file can be updated. This attribute makes it a little more powerful in its application than the MI type file.

As we have done with each file type up to this point, we'll create a file by running a BASIC program, then experiment with that file from the BASIC calculator mode. To create the file, enter, save and run the following program.

```
10 OPEN "O", 1, "FI/DAT", "FI"
20 A$="1ST STRING" : B$="STR 2"
30 I%=2 : I!=4 : I#=8
40 PUT 1,,,(15)A$,(6)B$,I%,I!,I#;
50 I%=I%*-1000 : I!=I!*-100 : I#=I#*-10
60 PUT 1,,,(15)B$,(6)A$,I%,I!,I#;
70 CLOSE
```

The first thing you'll have noticed is that we never did a pseudo FIELD operation Up = X) as we did for the FF type file. This is because it isn't absolutely necessary. BASIC will allocate the strings on GETS as it needs to, and the file support will pad/truncate. the string file items as needed to make them fit the length specified by the IGEL item prefix.

The second thing to notice is that the PUTS both put out data groups having identical formats: a 15-byte string, a 6-byte string, an integer, a single precision value and a double precision value. In a larger file, such a consistent data group format would make it eligible for an FF type file, as all data groups would have the same length and structure.

Load the program used to print the LOC function results used in all previous experiments, and type in:

```
OPEN "R", 1, "FI/DAT", "FI" : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   70 % EOF RBA
0 ! NEXT RCD RBA
```

As expected, the file is positioned so that the first GET or PUT will begin processing at the first byte of the file if fp = (null) or * is specified. This would be the case for all open modes except "E", which would position the file to the EOF RBA.

Knowing the data structure of the two groups that we wrote makes it reasonably easy to access the second group via RBA positioning. Type in:

```
GET 1,135,,(15)A1$,(6)A2$,I%,I!,I#;
PRINT A1$, A2$, I%; I1; I# : GOTO 10
```

The system will respond with:

```
STR           1ST ST     -2000 -400 -80
-1 $ EOF TEST   70 % EOF RBA
70 ! NEXT RCD RBA   35 # REMRA
```

By processing all the data in the second data item group, the file is positioned at EOF as the LOC(fan)$ shows.

FI files can be extended in the same manner as FI files. Let's do just that, and leave an area of 10 bytes between the current file position and the new data. Type in:

```
L=LOC(1)! : J=EXP(1) : PUT 1,,,(10)#,J; : GOTO 10
```

The system will respond with:

```
-1 $ EOF TEST   84 % EOF RBA
84 ! NEXT RCD RBA   70 # REMRA
```

The variable L contains the file location of the 10 padding bytes we wrote. We'll use that in a minute. Notice that in the PUT statement, the variable J was not suffixed by a type character. J has the default type of single precision floating point and 4 bytes were written into the file. Though using explicit type suffix characters in IGELs is not required as it was for NEWDOS/80 version 1, it is highly recommended that you do so.

Now, let's go back and put something in that padding area we just wrote. We'll use RBA positioning again to get to that area of the file. Type in:

```
A$="ABCDEFG" : PUT 1, ! L, , (4) A$, L! ; : GOTO 10
```

The system will respond with:

```
0 $ EOF TEST   84 % EOF RBA
78 ! NEXT RCD RBA   70 # REMRA
```

Pay special attention to the spacing on that last PUT statement. It shows the freedom you're allowed when entering a program. Depending on your own leanings, the spacing may or may not make the program more readable. Feel free to use spacing or not as you see fit.

Now let's go back and read some of the data we've written to the file. Type in:

```
GET 1,158,,I!,I#,(4)B$,K!;
PRINT I!; I#; B$; K! : GOTO 10
```

The system will respond with:

```
-400 -80 ABCD 70
0 $ EOF TEST   84 % EOF RBA
78 ! NEXT RCD RBA   58 # REMRA
```

We read the data with the proper data types for what was located at the starting file position, so the results of the PRINT are normal. Needless to say, if we were off by even 1 byte in our positioning the results would have been rather different.

To show the disaster which could befall the unwary programmer, lets malposition the file and repeat the last transfer. Type in:

```
GET 1,157,,I!,I#,(4)B$,K!;
PRINT I!; I#; B$; K! : GOTO 10
```

The system will respond with:

```
2.36125E+21 2147483648.000008 xABC 6.01858E-36
0 $ EOF TEST   88 % EOF RBA
77 ! NEXT RCD RBA   57 # REMRA
```

Nasty, isn't it? But the system did just what we told it to do - because it's an FI type file, it couldn't protect us from ourselves.

If you've gotten this far, you should have a fair idea of how to manipulate marked item and fixed item files. As the support for TRSDOS BASIC's field item files and print/input files has not changed in NEWDOS/80, excepting for allowing field item files to have a standard record length of other than 256, no experiments were provided here to familiarize you with them. If you haven't done so lately, go back and read chapter 8. You'll see that indeed the support for the old TRSDOS file types has been extended. With the experience you've received here, you'll be able to generate your own experiments for those extensions. Good luck to you!

APPENDIX B

The purpose of this appendix is to give some examples of marked item and fixed item file usage and to give different explanations than were offered in chapter 8 and Appendix A. Chapter 8 contains the specifications for the I/O enhancements to BASIC. This appendix hopes to give enlightenment but is not the specifications; chapter 8 is!!!

Throughout this appendix (as well as the whole manual) we shall refer to the file types by their short names. The reader should refer now to the glossary in chapter 10 for the definitions of MI file, MU file, MF file, FI file and FF file. This appendix will also refer to other terms such as IGEL, RBA, REMRA, REMBA, etc. which are defined in the glossary or in chapter 8.

Most of the examples given in this appendix deal with MU files and FF files since these two types will be the most commonly used by the programmers.

We have tried to make the examples as much alike as possible or practical to make it easier for the reader to spot the differences.

Since we are basically interested in demonstrating the use of the files (an exception is the demonstration of the uses of CMD"O", BASIC's in-memory sort), we do not provide the routines which actually use or generate the data to be read from or sent to the files. The programmer is assumed to provide these routines if he/she wishes to use these examples in live situations.

In all these examples, each named variable corresponding to a file item is suffixed with an explicit type symbol ($, %, 1 or #) (See line 120 of Example 3). This is done so that the reader will know exactly what type of data is being read from or written to the file. We STRONGLY recommend that in your own IGELs that you do the same; otherwise it is quite possible that you can severely damage a file by the implied type not being as you thought you remembered it. Use of explicit type symbols was required in version 1 for IGELs used in fixed item file processing, but that is not so in version 2. An example of an IGEL that does not use explicit type symbols is to change two lines in Example 7 to be:

```
10 CLEAR 2000: DEFSTR N,S: DEFINT A,I: DEFSNG F: DEFDBL D
120 (20)NM,AN,AM!,DT,(15)ST,IG,FP,DP;
```

Remember, we STRONGLY recommend the suffixing of type symbols to the variable names in IGELs.

The operation of a GET or a PUT proceeds in two phases:

1. The file positioning phase. In this phase, the file is positioned according to the second parameter, the file positioning parameter, of the GET or PUT statement. At the end of this phase, for certain types of positioning parameters, the file location values REMRA and REMBA are saved for possible future use when the subsequent positioning parameter for that filearea is # or $ (see section 8.10).

2. The data transfer phase. In this phase, data is transferred between the file and the variables named in the IGEL.

Example 1. Write records sequentially to a MU file.

MU files are intended as an alternative to print/input files. AMU file tends to use less disk space than a print/input file, can be updated with some restrictions, can be indexed into via the !rba positioning parameter, and does not need the ;","; character sequence to separate strings during writes to the file.

```
10 CLEAR 2000
20 OPEN "O",1,"XXX/DAT:1","MU"
30 GOSUB 10000    'build data for record
40 IF RN% = 0 THEN CLOSE: END       'end of run
50 PUT 1,,,NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
60 GOTO 30
```

The file is opened for sequential output of records whose individual lengths vary depending upon the size of the two strings contained in each record.

The file positioning parameter in the PUT statement is null, indicating that each execution of that PUT writes the next sequential record.

The programmer supplies the routine at 10000 to generate the data for the records. If no more records are to be created, set RN% = 0. Otherwise set RN% not 0 and put into the 8 variables NM$, AN%, AM!, DT#, ST$, IG%, FP! and DP# the data that is to be transmitted to the file.

The IGEL in this example is contained within the PUT statement and consists of 8 expressions (in this case, all named variables). The variable or expression associated with each record item is separated from its neighbor by a comma. The IGEL is terminated by a semicolon.

The full contents of each of the strings NM$ and ST$ is sent to the file, with each preceded by one or two string marker bytes. The second marker byte is used for strings 128 to 255 characters in length.

Each of the integers AN% and IG% is represented in the file as 3 bytes, a 72H marker byte followed by the 2 bytes of the binary integer value in the same format as used by BASIC.

Each of the single precision floating point numbers AM! and FP! is represented in the file as 5 bytes, a 73H marker byte followed by the 4 byte binary single precision floating point value in the same format as used by BASIC.

Each of the double precision floating point numbers DT# and DP# is represented in the file as 9 bytes, a 74H marker byte followed by the 8 byte binary double precision floating point value in the same format as used by BASIC.

Using the IGEL in the PUT statement to compute the minimum and maximum record lengths, the minimum length of a record in this file will be 37 bytes (both strings are null and including the SOR byte) and the maximum length of a record will be 549 bytes (both strings have 255 characters).

Example 2. Read records sequentially from a MU file.

```
10 CLEAR 2000
20 OPEN "I",1,"XXX/DAT:1","MU"
30 IF EOF (1) THEN END
40 GET 1,,,NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
50 GOSUB 10000    'process the record's data
60 GOTO 30
```

This example is the opposite of Example 1, using the same IGEL and named variables. The data records of the file are successively read and processed. The programmer supplies the routine at 10000 to do what he/she wishes with the data.

The file positioning parameter in the GET statement is null, meaning that each execution of that GET reads the next sequential record in the file.'

The EOF(1) function returns a true condition when the file position of the next record is exactly at file EOF.

Example 3. Sequentially read and update the records of a MU file.

```
10 CLEAR 2000
20 OPEN "R",1,"XXX/DAT:1","MU"
30 IF EOF(1) THEN END
40 GET 1,,120     'read the next sequential record
50 GOSUB 10000    'update the record's data
60 PUT 1,#,120    'rewrite the record back to the file
70 GOTO 30
120 NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
```

The same file created in Example 1 is used in this example. The file is opened for both input and output operations. Records are read sequentially from the file into the BASIC variables, zero or more of those variables are updated by the programmer supplied routine at statement 10000. Upon return from that routine, the record is written back to the file.

The file positioning parameter in the PUT statement is the character #. For this example, at the start of each execution of that PUT statement, the file is repositioned back to the start of the record read by the GET, causing the PUT to update that record.

Both the GET and the PUT statement use the same IGEL which is located at text line 120. This IGEL is identical to that used in examples 1 and 2 except that instead of being contained within the GET or PUT statement, it is contained in a separate text line with the GET and PUT statements containing that line number as their third parameter.

An error will be declared if the PUT statement finds the new length of a record exceeds the length originally assigned to that record during Example 1. This will only occur when the sum total of the file space used by the record's string items exceeds that of what the strings originally occupied plus any null space included in the original record (insertable using the (lend function, see section 8.4.3.4). The numeric values may be updated

without concern as they always occupy the same amount of file space. Thus, if a string item is to be updated in a MU file, the string's resulting length should not be increased; it can be, but be careful.

Example 4. Read in, sort in memory and write back out a MU file.

```
10 CLEAR 10000: DEFINT I
15 DIM NM$(200),AN%(200),AM!(200),DT#(200)
17 DIM ST$(200),IG%(200),FP!(200),DP#(200),IX%(200)
18 DIM IX%(200)
20 IX=0: OPEN "I",1,"XXX/DAT:1","MU"
30 IF EOF(1) THEN 80
40 IX=IX+1: IF IX > 200 THEN PRINT "TOO MANY RECORDS": END
50 GET 1,,60: GOTO 30
60 NM$(IX),AN%(IX),AM!(IX),DT#(IX),       'IGEL 1st line
70 ST$(IX),IG%(IX),FP!(IX),DP#(IX);       'IGEL last line
80 IF IX = 0 THEN PRINT "EMPTY FILE": END
90 CMD"O",IX,*IX%(1),AM!(1),NM$(1)
100 CLOSE: OPEN "O",1,"XXX/DAT:1","MU"
110 IY = IX: FOR IZ = 1 TO IY
120 IX = IX%(IZ): PUT 1,,60
130 NEXT IZ: CLOSE: END
```

The MU file XXX/DAT:1 of 1 to 200 records is read into 8 arrays.

The records are then indirectly sorted at line 90 using BASIC's array sort using the IX% array as the integer indirect array. The sorting criteria is ascending order, first by the AM! values and then by the NM$ values. During the sort, the integer IX% array is set up to contain sequentially in the sorted order the index values into the other arrays.

It should be noted that the sort changed nothing in the record arrays AM! and NM$. The IX% array was initialized to point each successive element to successive elements of the AM! array. As the sort proceeded, the elements in the IX% array were moved around to conform to the sort order.

It should further be noted that though the file records span across 8 arrays, the sort saw only the two of them (AM! and NM$) that provided the sort data.

After the sort the records of the file are written out in sorted order. Since the same file was used to store the sorted records, the user must be sure to preserve a backup copy of the original file in case an error occurs during the output of the sorted records.

This example demonstrates that IGELs can contain array named variables and that an IGEL may span multiple text lines (lines 60 and 70).

Example 5. Write records sequentially to a FF file.

FF files are intended as an alternative to field item files (TRSDOS random files). The FIELD statement is not used with FF files; though the user may wish to use the pseudo FIELD function specified in section 8.11. LSET and RSET are not used in FF file processing to set up the variables making up the record, though if the user has set the strings to the specified lengths via the pseudo FIELD function, he/she may wish to use LSET or RSET to maintain a string variable at that length. LSET and RSET must never be used for numeric variables. For FF files, MKD$, MKI$, MKS$, CVD, CVI and CVS are not used.

Each string variable in the IGEL must be prefixed with the length the string item will have in the file, and regardless of the number of characters in the variable's string at the time of the PUT, the corresponding string in the file will be truncated on the right or padded on the right with spaces to make up the required number of characters. During the PUT, the string variable is NOT changed; only the file item is.

```
10 CLEAR 2000
20 OPEN "O",1,"XXX/DAT:1","FF",63
30 GOSUB 10000    'build data for record
40 IF RN% = 0 THEN CLOSE: END       'all done
50 PUT 1,,,(20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
60 GOTO 30
```

The file is opened for sequential output of records each 63 bytes long.

The file positioning parameter in the PUT statement is null, indicating that each execution of that PUT writes the next sequential record.

The programmer supplies the routine at 10000 to generate the data fox the records. If no more records are to be generated, set RN% = 0. Otherwise set RN% non-zero and load the 8 variables NM$, AN%, AM!, DT#, ST$, IG%, FP! and DP# with the data that is to be transmitted to the file.

The IGEL is contained within the PUT statement and consists of 8 named variables. The name variable associated with each record item separated from its neighbor by a comma. The IGEL is terminated by a semicolon.

Each of the strings NM$ and ST$ is represented in the file by the number of characters specified by the variable's prefix in the IGEL. For each PUT executed by this example, the current contents of NM$ are sent to the file. If the NM$ string has more than 20 characters, the excess characters on the right are dropped from the file item, not from NM$. If the NM$ string has less than 20 characters, the file item, not NM$, is padded on the right with spaces to make the file item 20 characters long. The same concept holds in restricting to 15 characters the file item associated with the ST$ string.

Each of the integers AN% and IG% is represented in the file by 2 bytes in the same format as used by BASIC.

Each of the single precision floating point numbers AM! and DF! is represented in the file by 4 bytes in the same format as used by BASIC.

Each of the double precision floating point numbers DT# and DP# is represented in the file by 8 bytes in the same format as used by BASIC.

Example 6. Read records sequentially from a FF file.

```
10 CLEAR 2000
20 OPEN "I",1,"XXX/DAT:1","FF",63
30 IF EOF(1) THEN END
40 GET 1,,,(20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
50 GOSUB 10000    'process the record's data
60 GOTO 30
```

This example is the opposite of Example 5, using the same IGEL and named variables. The data records of the file are successively read and processed. The programmer supplies the routine at 10000 to process the data.

The file positioning parameter in the GET statement is null, meaning that each execution of that GET reads the next sequential record in the file.

After each record read, AM$ contains a 20 character string and ST$ contains a 15 character string.

Example 7. Sequentially read and update the records of a FF file.

```
10 CLEAR 2000
20 OPEN "R",1,"XXX/DAT:1","FF",63
30 IF EOF(1) THEN END
40 GET 1,,120     'read the next sequential record
50 GOSUB 10000    'update the record's data
60 PUT 1,#,120    'rewrite the record back to the file
70 GOTO 30
120 (20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
```

The same file created in Example 5 is used here. The file is opened for both input and output operations. Records are read sequentially from the file into the BASIC variables, zero or more of those variables are updated by the programmer supplied routine at statement 10000. Upon return from that routine, the record is written back to the file.

The file positioning parameter for the PUT statement is the character A At the start of each execution of the PUT in this example, the file was repositioned back to the start of the file record read by the GET (in more complicated words, to the REMRA), thus causing the PUT to write that record.

Both the GET and the PUT use the IGEL starting at line 120.

After each GET, NM$ contains a 20 character string and ST$ contains a 15 character string. During the programmer supplied processing, the lengths of one or both of the strings may change. During the PUT, the file item corresponding to AM$ is set again to a length of 20 characters, with space padding or truncation taking place on the right as necessary. The same

concept applies to the 15 character file item corresponding to ST$. Remember, AM$ and ST$ are not changed by the PUT.

Example 8. Randomly read and optionally update the records of a FF file.

```
10 CLEAR 2000
20 OPEN "D",1,"XXX/DAT:1","FF",63
30 GOSUB 10000          'determine which record to read
40 IF RN% = 0 THEN END  'end if no more
50 GET 1,RN%,120        'read that record
60 GOSUB 15000          'optionally update the record's data
70 IF RN% <> 0 THEN PUT 1,RN%,120   'if required, write the record
80 GOTO 30
120 (20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
```

This example is similar to Example 7 excepting that the record reads are done randomly and the programmer can elect not to update the record.

The file created in Example 5 is used here. For each record, the processing is as follows:

The programmer supplied routine at line 10000 determines which file record is to be looked at next. On return, RN% contains the desired record number; if RN% = 0, the run is ended.

The record is read, using RN% as the file positioning parameter to specify which record is wanted.

The programmer supplied routine at line 15000 looks at the record's data and optionally changes 1 or more variables associated with the record. On exit from the routine, RN% is set to zero if the record is not to be updated; otherwise RN% is unchanged.

If RN% is not zero, the record is rewritten to the file using RN% as the file positioning parameter. Note, the file positioning parameter for the PUT could have been the character

Note the use of "D" rather than "R" as the 1st parameter in the OPEN statement. "R" could have been used, but "D" prevents the file from being extended if for some reason RN% was changed before the PUT statement to be a record number beyond the range of the file.

Example 9. Sequentially write records to a MU file and sequentially write records to a FF file that serve as an index into the MU file.

There are many cases where the user has a huge file with each record having strings of varying lengths, does not want the string padding or truncation that is done by field item or fixed item files and yet still wants to be able to randomly access the file, and to a limited extent be able to update that file. Using a MU file as the main file and an FF file as an index file, the user can achieve these objectives.

```
10 CLEAR 2000
20 DIM AN%(4000),RB!(4000)    'two arrays to hold index data
30 OPEN "O",1,"XXX/DATA","MU" 'open the main data file
40 RC% = 0
50 GOSUB 10000                'create next record's data
60 IF RN% = 0 THEN 105        'done with main file
70 RC%=RC%+1: IF RC% > 4000 THEN PRINT "FILE T00 LARGE": GOTO 105
80 RB!(RC%) = LOC(1)!         'save RBA of next record
90 PUT 1,,,NM$,AN%(RC%),AM!,DT#,ST$,IG%,FP!,DP#;
100 GOTO 50
105 CLOSE
110 IF RC% = 0 THEN PRINT "NO DATA RECORDS": END
120 CMD"O",RC%,AN%(1),RB!(1)        'sort index data
130 OPEN "O",1,"XXX/NDX:1","FF",6   'open index file
140 FOR X = 1 TO RC%
150 PUT 1,,,AN%(X),RB!(X);          'write index record
160 NEXT X: CLOSE: END
```

This example could have been programmed to write alternately one record to each of the two files. However, since both are on the same drive, the drive arm would be constantly in motion and execution would take, or at least seem to take, forever. Therefore, the index file's data is stored into arrays to be written out after the main file has been completely written.

For this example the AN% array is assumed to hold account numbers and for each main data file record, the account number is unique.

The program proceeds as follows:

For each main data file record:

The programmer supplied routine at 10000 set RN% = 0 if no more main data file records are to be created. Otherwise it sets RN% non-zero and creates the record's data by storing the values in the proper variables, including the account number into its array.

The RBA of where that record is to be placed in the main data file is determined by the line 80 LOC(1)1 function and is stored into the RBA array RBI.

The record is written to the main data file.

The two arrays AN% and RBI are directly sorted. Since this is a direct sort, both arrays are physically arranged in the sort order, which is in ascending order of account number. Note, though ascending order of RBA is the secondary sort criteria, since the account numbers are unique, the RBA values are never checked.

The index file is created by writing the index records sequentially from the arrays, which are in ascending order of account number.

Exactly the same results would have been attained had text lines 80 and 9 above been written as:

```
80 PUT 1,,,NM$,AN%(RC%),AM!,DT#,ST$,IG%,FP!,DP#;
90 RB! = LOC(1)#        'RBA where the record was placed
```

Example 10. Randomly read and optionally update the records of an indexed MU file.

```
10 CLEAR 2000
20 DIM AN%(4000),RB!(4000)
30 RC% = 0: OPEN "I",1,"XXX/NDX:1","FF",6       'open index file
50 IF EOF(1) THEN 100
60 RC% = RC% + 1
70 IF RC% > 4000 THEN PRINT "INDEX TOO LARGE": END
80 GET 1,,,AN%(RC%),RB!(RC%);       'read index data into arrays
90 GOTO 50
100 CLOSE: IF RC% = 0 THEN PRINT "NO ACCOUNTS": END
110 OPEN "D",1,"XXX/DATA","MU"      'open main data file
120 GOSUB 10000   'determine which account record wanted
130 IF RN% = 0 THEN END       'if req, end the run
140 FOR X = 1 TO RC%
150 IF RN% = AN%(X) THEN 170  '.search index for account #r match
160 NEXT X: PRINT "BAD ACCOUNT NUMBER": GOTO 120
170 GET 1,!RB!(X),300         'read the account record
180 IF AN% <> RN% THEN PRINT "BAD DATA FILE": END
185 GOSUB 15000   'display data, receive back updates
190 IF RN% <> 0 THEN PUT 1,#,300    'if required, re-write record
200 GOTO 120
300 NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;
```

It is assumed that you, the programmer, have an application that is basically data retrieval for display to the terminal operator and in some cases, update information is received from the terminal operator to alter information already in the main data file.

The two files created in Example 9 are used in this example. The index file is opened first and its contents read into the two arrays AN% and RB!

For each record, the processing is as follows:

The programmer supplied routine at 10000 queries the terminal operator to determine the account number. On return from that routine RN% contains the account number; if zero, the run is to end.

The account number array AN% is searched for the matching account number. If not found, an error message is displayed.

The account record is read from the main data file, using the account record's RBA value from the RB! array as the file positioning

parameter. RN% is then compared against the AN% value from the file and if not equal, an error message is displayed and the run terminated.

The programmer supplied routine at 15000 displays the account data to the terminal operator, and, if required, accepts back the new data for the fields being updated. If the record is not to be updated, set RN% = 0.

On return, the record is re-written to the file if RN% is non-zero. The PUT uses file positioning parameter # which repositions the file to the start of the record (again, in other words, the # causes file positioning to REMRA).

Example 11. Sequentially write different type records to a MU file.

A programmer's application may use different types of records. Normally, the individual record types would be stored in different files, but if a record of one type corresponds to one or more records of one or more other record types the programmer may want to store all of these associated records together in the file in order to minimize the time needed to access them. MU, MI and FI files readily allow this mixing of record types whereas field item, MF and FF files do so only if the records are all the same length.

An example of this mixture of record types might be an insurance account which can have the account main record, one or more records of the individuals covered by the account, one or more records of payments and one or more records of claims, each of the 4 record types having different formats.

In the example below, three different record types are used; the main record and two subsidiary types. There is no correlation between these three record types and anything in the real world; all we are trying to do is demonstrate how a MU file can contain the multiple type records.

In the example below, each set of records consists of one type 1 record and a mixture of zero or more type 2 and 3 records.

```
10 CLEAR 2000
20 OPEN "O",1,"XXX/DAT:1","MU"
30 GOSUB 10000    'create type 1 record's data
40 IF RN% = 0 THEN CLOSE: END       'all done
50 PUT 1,,,"1",NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;  'write type 1 record
60 GOSUB 11000    'create type 2 or 3 record's data
65 IF X$ <> "2" THEN 90
70 PUT 1,,,"2",SA$,SB$,LN$,PD!;     'if type 2 record, write it
80 GOTO 60
90 IF X$ <> "3" THEN 30
100 PUT 1,,,"3",SJ$,DF#,IP%,IA%,FG!;      'if type 3 record, write it
110 GOTO 60
```

The programmer supplied routine at 10000 sets RN% = 0 if no more records are to be created. Otherwise it sets RN% non-zero and creates the type 1 record's data. For each set of records, there is one and only one type 1 record.

The programmer supplied routine at 11000 creates the type 2 or type 3 record's data, whichever comes next. On exit from that routine X$ contains the record type flag or if neither "2" or "3", it indicates the end of the series of records. The type 2 and type 3 records are intermixed on the file following the associated type 1 record. For a given type 1 record, there need not be any type 2 or 3 records.

Each PUT statement contains its own IGEL. Note, in each of these IGELs the first entry is an expression rather than a named variable. It could have been a named variable containing the record type character, but expressions were used instead to demonstrate that the IGELs used for writing to marked item files can contain expressions as well as named variables, hence the reason why its called an IGEL (item group expression list) instead of an IGVL (item group variable list).

Example 12. Sequentially read and optionally update records from a MU file containing multiple record types.

The file created in Example 11 is used here. In this example we will demonstrate the partial record read feature of marked item files (also available for fixed item files). The first three, the 5th and 7th items of the type 1 record will be read, type 2 records will be skipped, and type 3 records will be processed entirely, including optionally being updated.

```
10 CLEAR 2000
20 OPEN "D",1,"XXX/DATA","MU"
30 IF EOF(1) THEN END
40 GET 1,,,RT$;   'read record type character
50 IF RT$ <> "1" THEN PRINT "BAD RECORD TYPE": END
60 GET 1,*,,NM$,AN%,,DT#,,IG%;      'read selected type 1 record items
70 IF EOF(1) THEN END
80 GET 1,,,RT$;         'read next record type character
90 IF RT$="2" THEN 70   'bypass type 2 records
100 IF RT$ <> "3" THEN 50
110 GET 1,*,200         'read in rest of type 3 record
120 GOSUB 11000         'process type 3 record's data
130 IF RN% <> 0 THEN PUT 1,$,200  'if required, re-write type 3 record
140 GOTO 70
200 SJ$,DF#,IP%,IA%,FG!;
```

The GET statements at lines 40 and 80 read only one item of the next record, and the file is left positioned at the 2nd item of the record. Both REMRA and REMBA point to the record's 1st item.

The GET statements at lines 60 and 110 start where the line 40 or 80 GET left off. The line 60 and 110 GETS do NOT advance to the next record before inputting data. However, before inputting the data REMRA is set to point to the positioning point which is the 2nd item of the record. Thus, if the line 130. PUT is executed, the PUT's reposition parameter will reposition the file to REMBA which is pointing at the record's 2nd item. REMRA is not changed by the line 60 or line 110 GET statements.

The line 60 GET reads the 2nd, 3rd, 5th and 7th items of the type 1 record and leaves the file positioned at the 8th item. The 4th and 6th items were skipped, and the next execution of the line 80 GET will skip the remainder of the type 1 record items (the 8th and 9th).

The remainder of the item in the type 2 records are skipped over by the next execution of the line 80 GET.

The line 110 GET reads the rest of the type 3 record, the 2nd through 6th items.

The programmer supplied routine at line 11000 does whatever processing is needed for the type 3 record, using its data plus that extracted from the type 1 record. If the type 3 record is to be updated, RN% is set non-zero; otherwise it is set equal to 0.

On return for that routine, if RN% <> 0 the type 3 record is re-written to the file. Note that only the 2nd through the last items were written back to the file. The first item was not changed as the file positioning done for the line 130 PUT was to the REMBA position which was the file position existing during the line 110 GET immediately after its positioning was done and before any items were inputted.

Remember that in updating a MF and MU record, once an item is written back to that record, all items following it in the record must also be written back if those items are to remain part of the record. It is not necessary they all be written by the same PUT statement, but they must all be written; for each PUT that updates only part of a MU file record fills with null bytes all of the record's bytes, if any, following the last item written.

For record segmented files, EOF compares the location of next record against the EOF. For the type 3 record processing above, the position where the line 130 PUT leaves off and the position of the next record are the same. For the type 1 and 2 records, the GET statements left the file positioned to the 8th and 2nd items respectively. In these cases, EOF computes the position of the next record and uses that value in its compare against EOF.

Example 13. Sequentially write records to a MF file (marked item file of fixed length records).

A programmer's application may require full update capability for a file that contains strings. Since a MU file cannot guarantee record update success when strings are being lengthened, the programmer must go to either field item file, a FF file or lastly, a MF file. The relative merits of the field item and the FF file have already been discussed, so we will concern ourselves here only with the relative merits of the FF and MF files.

The advantage of MF files is that string items are not padded with blanks to fill out the item to the maximum length allowed it. Each string item is written with the number of characters it needs, up to but not exceeding the maximum length allowed it. Then, at the end of the record, if unused record space exists, the record is filled out with null bytes which during the read of a MF file, the program never sees. Though most of the time padding bytes do not bother the programmer in comparing a string item from the file with another string, there are times when it creates a real inconvenience compared to the cost of the extra disk space involved.

The disadvantage of MF files over FF files is that MF files use more disk space due to the inclusion of the item marker bytes. In this example, the record size is 13% greater than the corresponding FF file in Example 5 though both contain the same data, excepting that the MF file strings are not padded.

```
10 CLEAR 2000
20 OPEN "O",1,"X-XX/DAT:1","MF",71
30 GOSUB 10000    'create data for record
40 IF RN% = 0 THEN CLOSE: END       'no more records
45 IF LOF(1) < RN% THEN PRINT "BAD RECORD #: GOTO 30
50 PUT 1,,,(20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;   'write record
60 GOTO 30
```

The file is opened for sequential output of records each 71 bytes long. According to the IGEL at line 50, this 71 bytes allows for items of 21, 3, 5, 9, 16, 3, 5 and 9 bytes respectively (remember, each item starts with a marker byte).

The programmer supplies the routine at line 10000 to set RN% = 0 if no more records are to be created. Otherwise it sets RN% non-zero and loads the data for the new record into the 8: variables NM$, AN%, AM!, DT#, ST$, IG%, FP! and DP#.

The representation of the items on the disk is the same as described in Example 1 for the MU file excepting that SOR items are not used and that both string items are limited to a maximum number of characters, 20 for NM$ and 15 for ST$. If at the time the file item is written, either string variable has a length greater than the maximum allowed for the file item, then the excess characters on the right are not transmitted to the file item.

Strictly speaking, it is not a requirement that string expressions in the IGEL used at line 50 above be prefixed with a maximum string length value. The IGEL of Example 1, line 50 could have been used. However, by not specifying a maximum string length value for any one string item, full update capability cannot be guaranteed for the record.

Note the use of the LOF function at line 45 to check if the requested record is within the file.

Example 14. Randomly read and optionally update records of a MF file.

```
10 CLEAR 2000
20 OPEN "D",1,"XXX/DAT:1","MF",71
30 GOSUB 10000    'determine which record to read
40 IF RN% = 0 THEN CLOSE: END       'end if no more
50 GET 1,RN%,120  'read that record
60 GOSUB 15000    'optionally update the record's data variables
70 IF RN% <> 0 THEN PUT 1,#,120     'if required, rewrite the record
80 GOTO 30
120 (20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
```

The file is opened for random reading and writing. The first parameter of the OPEN statement is "D" to prevent extension of the file.

The programmer supplies the routine at line 10000 to determine which record is to be processed next. On exit from that routine, RN% contains the record number except that if RN% = 0, then the run is to end.

The record is then read. The resulting length of the string NM$ is 0 to 20 characters and of string ST$ is 0 to 15 characters depending on what the corresponding file item actually had in it. Remember, no padding with spaces was done during the item write and none is done during the GET.

The programmer supplies the routine at line 15000 to query the data in the variables NM$, AN%, AM!, DT#, ST$, IG%, FP! and DP#. If the record is not to be updated, it sets RN% = 0. Otherwise, it changes some or all of those variables.

On return from this routine, if RN% is not zero, the record is written back to the file. If one or more of the string variables have a new length, then the corresponding file item assumes that new length.

Example 15. Sequentially write to a MI file.

MI and FI files are one long series of items. If the programmer logically groups items into records, BASIC knows nothing of it since a record length is not specified at OPEN, such as is done for field item, MF and FF files, nor is there a record marker, such as the SOR (start of record) byte for MU files and the EOL (end of line) byte for print/input files. Not knowing anything about the programmer's possible logical record segmentation of MI and FI files, BASIC cannot automatically advance to the next record such as was done by the GET statement at line 80 in Example 12 where the remainder of a type 1 or type 2 record was bypassed.

We will use the code of Example 11 with one change, to generate a MI file consisting of 3 record types (remember, BASIC knows nothing of records in MI and FI files). Changing line 20 of Example 11 to:

```
20 OPEN "O",1,"XXX/DAT:1","MI"
```

we can generate the file used in Example 16 below.

Example 16. Sequentially read a MI file.

The file created in Example 15 is used here. Though the programmer knows the file contains records of 3 types, BASIC does not. Therefore, to advance to the next record, the program must read the previous record completely, though it need not do so all in one GET statement.

A MI file cannot be updated. This is restriction is made because of the impossibility of handling strings whose lengths change.

```
10 CLEAR 2000
20 OPEN "I",1,"XXX/DAT:1","MI"
30 IF EOF(1) THEN END   'exit if empty file
40 GET 1,,,RT$;   'read record type character
50 IF RT$ <> "1" THEN PRINT "BAD RECORD TYPE": END
60 GET 1,,,NM$,AN%,AM!,DT#,ST$,IG%,FP!,DP#;     'read type 1 record
70 IF EOF(1) THEN END
80 GET 1,,,RT$    'read next record type character
90 IF RT$ = "2" THEN 150
100 IF RT$ <> "3" THEN 50
110 GET 1,*,,SJ$,DF#,IP%,IA%,FG!;   'read type record
120 GOSUB 11000   'process using type 1 and 3 record data
140 GOTO 70
150 GET 1,,,SA$,SB$,LN$,PD1;: GOTO 70
```

Remember, though the comments in the above program discuss records, the logical segmenting of the file into records is known only to the programmer and not to BASIC.

Note that line 60 above used a null positioning parameter where line 60 in Example 12 used the * positioning parameter. In Example 12, the * was used because file positioning was to stay where it was and not advance to the next record. However, since for MI and FI files, BASIC knows nothing of records, the null and the * positioning parameters work exactly the same which is to leave the file positioned where it is. Thus, in lines 40, 60, 80, 110 and 150 above, the positioning parameters null and * could have been used interchangeably. Since it is easier to type a null than an *, the null will tend to be used. Remember though, for MU, MF and FF file processing, there is a difference between the meaning of null and the meaning of *.

Example 17. Sequentially write records to a FI file and sequentially write index records at the end of that file to indexing into the main records.

The FI file is a very flexible file. It allows the programmer the capability of the FF file while allowing records of different lengths. Remember, as if the MI file, BASIC knows nothing of the programmer's segmenting of a FI file into records. To the programmer though, the FI file can be an assortment of all kinds of records. This example and Example 18 will use a FI file composed of 5 different logical record types: the three record types used in examples 11, 12, 15 and 16, the index records used in the FF file in examples 9 and 10, and another 2 byte record type unique to the current file.

In this example, we will assume the record type 1's AN% item is an account number and that the account number is unique for each type 1 record. The file is first written to contain all the records of the first 3 types. The RBA and the account number of each type 1 record is saved in two BASIC arrays. After all the data records are written, a 1 byte record is written to indicate the end of the data records. Next, the two arrays are sorted into ascending account number order. Index records are then written to the file. Lastly, the number of index records is written to the file.

This example is a cross between Example 9 and Example 11 and most of the comments there apply here, excepting that you are dealing with one FI file instead of a MU file and a FF file.

```
10 CLEAR 2000
20 DIM AN%(4000),RB!(4000)          'arrays for index data
30 OPEN "O",1,"XXX/DAT:1","FI"      'open the combined data/index file
40 RC% = 0
50 RT$ = "1": GOSUB 10000     'create next type 1 record
60 IF RN% = 0 THEN 170        'done with main data
70 RC%=RC%+1: IF RC% > 4000 THEN PRINT "FILE T00 LARGE": GOTO 170
80 RB!(RC%) = LOC(1)!         'RBA where type 1 record will be stored
90 PUT 1,,,(1)RT$,(20)NM$,AN%(RC%),AM!,DT#,(15)ST$,IG%,FP!,DP#; write
type 1 rec
100 GOSUB 15000   'create type 2 or 3 record
110 IF RT$ = "2" THEN 150
120 IF RT$ <> "3" THEN 50
130 PUT 1,,,(1)RT$,(40)SJ$,DF#,IP%,IA%,FG!;     'write type 3 record
140 GOTO 100
150 PUT 1,,,(1)RT$,(3)SA$,(32)SB$,(14)LN$,PD!;  'write type 2 record
160 GOTO 100
170 IF RC% = 0 THEN PRINT "NO DATA RECORDS": END
175 RT$="0": PUT 1,,,(1)RT$;  'flag end of main data
180 CMD"O",RC%,AN%(1),RB!(1)  'sort index data
190 FOR X = 1 TO RC%
200 PUT 1,,,AN%(X),RB!(X);    'write index record
210 NEXT X
220 PUT 1,,,RC%;        'write number of index records
230 CLOSE: END
```

An advantage of writing the index records in with the data records is that only one file is used for both, thus avoiding problems in backup and copy of keeping a data and an index file in synchronization with each other. In examples 9 and 10 we could have used only one file, storing the index again on the back end of the MU file.

Since the file is a fixed item type, all named string variables in the IGELs must be prefixed with the length the file item is to have. Truncation or padding with spaces on the right takes place as the string data is moved to the file item.

As with Example 9, exactly the same results would have been attained had text lines 80 and 90 above been written as:

```
80 PUT 1,,,(20)NM$,AN%(RC%),AM!,DT#,(15)ST$,IG%,FP!,DP#
90 RB!(RC%) = LOC(1)#   'RBA where the record was placed
```

Note at line 175 a one byte end-of-main data record is written. This separator is needed by Example 18.

Example 18. Randomly read and optionally update the data records of an indexed MI file.

The file created in Example 17 is used here. The last two bytes of the file are read to determine the number of index records (and type 1 records). The index records are then read into two arrays. Then selected data record groups are read from the file and optionally the DF# and IA% items of the type 3 records are updated back to the file.

This example is a cross between Example 10 and Example 12 excepting that both the index and the data are contained within the FI file, only two items of the type 3 record are updated, and other differences as noted below.

```
10 CLEAR 2000
20 DIM AN%(4000),RB!(4000)    'two arrays for index data
36 OPEN "D",1,"XXX/DAT:1","FI"
40 X! = LOC(1)% - 2     'compute RBA of file's last 2 bytes
50 GET 1,!X!,,RC%;      'read in count of index records records
60 GET 1,!$(X!-6*RC%)   'position file to 1st index record
70 FOR X = 1 TO RC%     'read index data into the two arrays
80 GET 1,,,AN%(X),RB!(X);
90 NEXT X
100 GOSUB 10000         'determine which account to process
110 IF RN% = 0 THEN END 'run is completed
120 FOR X = 1 TO RC%    'search account # array for match
130 IF RN% = AN%(X) THEN 150
140 NEXT X: PRINT "BAD ACCOUNT NUMBER": GOTO 100
150 GET 1,!RB!(X),,(1)RT$,(20)NM$,AN%,AM!,DT#,(15)$,IG%,(12)$; 'type 1 rec
160 IF RT$ <> "1" THEN PRINT "BAD INDEX": END
170 IF RN% <> AN% THEN PRINT "BAD FILE DATA": END
180 GET 1,,,(1)RT$;     'read next record's type char
190 IF RT$ = "3" THEN 230
195 IF RT$ = "1" OR RT$ = "0" THEN 100    'test end of account
200 IF RT$ <> "2" THEN PRINT "BAD FILE DATA": END
210 GET 1,*,,(3)SA$,(32)SB$,(14)LN$,PD!;  'bypass type 2 record
220 GOTO 180
230 GET 1,,,(40)SJ$,DF#,(2)$,IA%,FG!;     'read type 3 record
240 GOSUB 11000   'process type 1 and 3 record data
250 IF RN% <> 0 THEN PUT 1,$,,(40)$,DF#,(2)$,IA%,(4)$; 'update type 3 rec
260 GOTO 180
```

At line 40, note the use of the LOC(1)% function to obtain the RBA of the file EOF. This is used to compute the RBA of the next to the last byte in the file as the files last 2 bytes contain the integer count of both the number of index records in the file and the number of type 1 records in the file. This integer value is read into RC% at line 50.

At line 60, using this index record count, the RBA of the 1st index record is computed and the file positioned to the start of the first index record. Note that the computation is done right in the GET statement's file positioning parameter. This may be done provided the computation itself does not reference a filearea. Also note that the file positioning parameter is of the !$rba type (see section 8.8.6), meaning that this GET is for file positioning only; an IGEL or IGELSN is not allowed, and no data transfer takes place.

The programmer supplied routine at line 10000 determines the account number for the group of records to be interrogated. If no more accounts are to be read, set RN% = 0; otherwise set RN% = to the account number.

At line 150 the file is positioned using the RBA associated with the account number in the index arrays. The type 1 record is then read. It has been arbitrarily decided that it is not necessary to know what is in the file items corresponding to ST$, FP! and DP# (see line 90 in Example 17). Therefore, these items can be bypassed. However, since this is fixed item file, it is necessary to inform BASIC of the number of files bytes of be skipped, using the (len)$ format (see section 8.4.3.3). The (15)$ causes the skip of the 15 file bytes that would normally be read into the string ST$. The (12)$ causes the skip of the 4 bytes that would normally be read into FP! and the 8 bytes that would normally be read into DP#. Lastly, if this IGEL was used in FF file processing, the (12)$ expression could have been dropped from the IGEL, as no other expressions follow it in the IGEL and the next GET with null positioning parameter will advance the file to the next record. In FI processing, the programmer must account for all of the record's space since BASIC knows nothing of his/her record structure, hence the (12)$ is required.

At line 210, though all we want to do is skip the type 2 record, we still must advance the file positioning as once again, BASIC knows not where this record ends. The same file position advancement could have been obtained with:

```
      120 GET 1,,,(53)$;      'bypass rest of type 2 record
```

The programmer supplied routine at 11000 processes the data from the type 1 and 3 records. If the type 3 record is not to be updated, set RN% = 0. Otherwise change either or both of DF# and IA% and set RN% non-zero.

The PUT statement at line 250 repositions the file to the REMBA position remembered by the GET at line 230. The type 3 record is then updated. Note that only 2 of the items, those corresponding to DF# and IAA, are replaced in the file (compare with line 130 of Example 12). The other items are skipped over and are not changed. In using the (len)$ or (lend expressions in the IGEL, the programmer must be certain to account for the proper number of bytes.

The use of the (len)$ type expression in the IGELs of lines 150 and 250 was done only to give examples of the (len)$ use. The user might prefer to just use the regular IGELs, changing those two lines to be:

```
      150 GET 1,*,,(1)RT$,(20)NM$,AN%,AM!,DT#,(15)ST$,IG%,FP!,DP#;
```

and

```
      250 IF RN% <> 0 THEN PUT 1,$,,(40)SJ$,DF#,IP%,IA%,FG!;
```

APPENDIX C

NEWDOS/80 VERSION 2.5 --- THE NEWDOS/80 VERSION 2 HARD DISK SYSTEM

The NEWDOS/80 Version 2 modified for hard disk operations is called both the NEWDOS/80 Version 2 Hard Disk Operating System and NEWDOS/80 Version 2.5. The difference between the regular NEWDOS/80 Version 2 and Version 2.5 is the inclusion in 2.5, via patches, of code to handle the hard disk. This documentation for NEWDOS/80 Version 2.5 is considered as Appendix C to the regular NEWDOS/80 Version 2 manual and should be inserted into that manual after Appendix B.

NEWDOS/80 Version 2.5 is NOT offered as a stand alone DOS; the regular NEWDOS/80 Version 2 must be PURCHASED and REGISTERED either prior to or at the time of purchase of NEWDOS/80 Version 2.5.

As usual with NEWDOS/80, the user should study this document carefully before attempting to do anything with the NEWDOS/80 Version 2.5 Hard Disk Operating System.

The basic NEWDOS/80 Version 2.5 supports Apparat's and Tandy's hard disks for either the TRS-80 Model I or III. Special Version 2.5 system diskettes may later be made available for other types of hard disks.

You may have already been using your hard disk under LDOS or another DOS and have valued user files on the lard disk. If this is so, you are already a serious hard disk user and cannot afford to lose valued data just because you are switching DOSs. You Faust:

CAREFULLY plan your move.

Backup up those files from hard disk to diskette using that DOS's offload program. This is insurance in case the conversion to NEWDOS/80 fails; you can reformat the hard disk(s) for the other DOS and reload the files.

Move NEWDOS/80 Version 2.5's HDBACKUP/CND program over to that DOS (see the NND parameter discussion in section 7).

Use HDBACKUP under that DOS to again offload your files to another set of diskettes. Remember to use the SAVE, INCLUDE and NND parameters. Also remember, for 5 million bytes of data, the HDBACKUP SAVE function will need 27 pre-formatted single sided, double density 40 track (720 sector) diskettes.

Initialize the hard disks for NEWDOS/80 Version 2.5 using HDFMTAPP, PDRIVE and FORMAT.

Use HDBACKUP under NEWDOS/80 Version 2.5 to RESTORE the files from diskettes to the hard disks. Carefully plan this move; you may decide to use more than one RESTORE from the same backup to get the various files where you want them.

1. OVERVIEW

1. A data file may contain as many as 16 million bytes.

2. A hard disk data volume can be up to 65535 sectors and contain up to 246 user files.

3. PDRIVE allows a maximum of eight active slots with a maximum of 4 floppy data volumes or eight hard disk volumes active at one time.

4. The capacity to support hard disk drives of over 100 million bytes exists, though currently only the Apparat and Tandy hard disk drives are supported.

5. A hard disk drive is divided into one or more drive sections.

6. A hard disk section is divided into one or more data volumes. The data volume is what is defined by PDRIVE. A data volume may not span multiple hard disk drives or drive sections.

7. 48K of RAN is required. The hard disk modifications for NEWDOS/80 Version 2.5 have preempted computer main memory 0F900H - 0FFFFH. Programs that execute in that area must no longer be used.

8. Aside from the main memory limitation above, most programs that work with NEWDOS/80 Version 2 will work with NEWDOS/80 Version 2.5. However, if any program assumes certain volume sizes (i.e. 350 sectors on the Model I or 720 sectors on the Model III) or a certain location and size of the directory, that program will have to be modified. Basically, programs that use standard file I/O and observe HIMEM should be OK.

9. The hard disk system can operate either from floppy drive 0 (in which case floppies retain their old drive numbers) or from the hard disk (in which case, floppy drives 0 - 3 become drives 4 - 7 respectively). Section 5 steps the user through the shift of the system from a floppy to a hard disk volume.

10. Program HDFMTAPP/CMD is used to magnetically format Apparat and Tandy hard disks.

11. Program HDBACKUP/CMD is used to selectively save hard disk files of any size onto floppies or to selectively restore them to hard disk. This is NEWDOS/80 Version 2.5's hard disk backup facility and must be used to make backup copies of valued data files. Further, the program HDBACKUP/CMD can be transferred to another DOS (see NND parameter in section 7) so that the program can be used under that DOS to offload user files to diskette preparatory to changing the hard disk to operate under NEWDOS/80 Version 2.5 on to onload user files from diskette to hard disk should the user wish to take the hard disk back to the other DOS.

12. Program EXTPDRIV/BAS can be used to set PDRIVE slot definitions from definitions stored in an ASCII text file. Since hard disk data volume specifications are both difficult and critical, it is recommended they be permanently built in a text file via SCRIPSIT or CHAINBLD and then activated when needed via EXTPDRIV.

13. Parameter HDS has been added to DOS command PDRIVE to define hard disk volumes.

14. Hard disk volumes defined under Model III NEWDOS/80 can be used under Model I NEWDOS/80 and vice versa. The files on these volumes are NOT useable interchangeably if they were NOT useable interchangeably when those files were on diskettes.

2. COMMENTS and RESTRICTIONS:

1. The user must be knowledgeable of NEWDOS/80 Version 2 and all subsequent information issued via the zaps prior to attempting to use the Hard Disk system. All hard disk discussion herein assumes this knowledge. This document is intended only as supplementary information to the regular NEWDOS/80 Version 2 manual and its subsequent zaps.

2. This document does NOT provide information about your hard disk. That information must be obtained from the source where you purchased or otherwise obtained your hard disk. The information NEWDOS/80 needs to know about your hard disk drive is (1) the number of recording surfaces (or number of I/O heads), (2) the number of tracks per surface (or the number of cylinders), (3) the number of 256 bytes sectors actually formatted on each track, and (4) track-to-track stepping rate code.

3. NEWDOS/80 Version 2 was not designed to operate with hard disks. All of the changes creating Version 2.5 have been done by patching the standard Version 2, with the exception that SYSO/SYS has been extended five sectors. This patching to Version 2 provides a minimum hard disk operating system and each specially purchased hard disk system diskette will operate with one and only one type of hard disk drive. If another type of hard disk drive has exactly the same interface to the computer, then it can also work with a particular hard disk system diskette (example, both Tandy's and Apparat's hard disks for the Model I and Model 3 have the sane software interface, therefore either (but not both at the same time) can be used with the same Version 2.5 Hard Disk system diskette). The standard issue NEWDOS/80 Version 2.5 hard disk system supports Apparat's and Tandy's hard disks for either the TRS-80 Model I or III.

4. This system REQUIRES 48K of RAM. To implement hard disk code, main memory from F900H to FFFFH has been taken by DOS and is not available to the users. Any user programs that use this area MUST either no longer be used or be modified to no longer use the F900H to FFFFH main memory area. HIMEM is set to 0F8FFH by DOS automatically and programs that observe HIMEM should be all right.

5. The number of active PDRIVE slots (formerly called drives) has been expanded from 4 to 8, allowing a maximum of 4 floppy volumes, 8 hard disk volumes or a combination thereof. The number of actual active

PDRIVE slots is still controlled by SYSTEM option AL. If a ?DRIVE active slot is to be unused, defining it as a hard disk volume and setting the PDRIVE HDS sub-parameter vsc1 to 0 will cause PDRIVE to accept the definition in an active slot and NEWDOS/80 to treat the slot as DEVICE NOT AVAILABLE whenever it attempts to use that slot (drive).

6. Two floppy drives are desirable, though only one is required. The Version 2.5 Hard Disk System comes on a standard 40 track, double density diskette on the Model III and a standard 35 track, single density diskette on the Model I. Since the capacity of the Model I diskette is too small to contain all the files for the hard disk system as cell as those of the nor.-hard disk system, some of the files (or program modules) of the regular NEWDOS/80 are not present on the hard disk system diskette. When needed, you may copy these modules over from the regular NEWDOS/80 system diskette.

7. The user may elect to run using a floppy system diskette (a copy of the Version 2.5 Hard Disk System diskette) or he/she ray move the NEWDOS/80 Version 2.5 Hard Disk System onto a hard disk volume.

1. If the system is being run from floppy drive 0 (the normal drive 0 for the computer), the floppy drives 0 - 3 use PDRIVE slots 0 through 3 respectively, just as they do in the regular NEWDOS/80 Version 2. Under the floppy hard disk system, PDRIVE slots 1 - 7 may be defined as hard disk volumes and accessed by user programs as drives 1 - 7 respectively.

2. If the system is being run from hard disk, the floppy drives 0 -3 use PDRIVE slots 4 through 7 respectively, and the floppy drives 0 - 3 are known to the system and user programs as drives (or slots) 4 through 7 (though you may use any or all of drives (or slots) 0 through 7 for hard disk volumes).

***** Warning, when the system is being run from the hard disk, access to the floppy drives is slots 4 - 7 meaning that all slots between the system volume in slot 0 and the slot being used by the floppy drive MUST be valid definitions even though you are using only one hard disk volume. PDRIVE will allow and DOS will ignore a slot defined for a null hard disk volume (having the HDS sub-parameter vsc1 equal 0), thus allowing access to the floppy drives.

8. The DOS command FORMAT or the format portion of COPY do NOT actually format the hard disk; instead of formatting, the message, INITIALIZING SECTORS, is displayed and the sectors are written with a standard pattern. To actually format the Apparat or Tandy hard disk, use the HDFMTAPP/CAD program provided. To format another hard disk, you must use a program provided by the hard disk retailer (NOT provided by Apparat).

9. This hard disk upgrade does NOT support the standard LDOS TRS-80 hard disk data volumes as directory concepts slightly differ, though those volumes can be read via SUPERZAP by expert users (provided the hard disk is divided into sections properly, spg1 value is 16, the ddsl1 value is 76, the ddsa1 value is 32, and the gpl1 value is 2 for one surface volumes, 4 for two surfaces, 6 for three surfaces and 8 for four surfaces)(changing HIT sector rel byte 1FH from 00H to 16H allows DIR to work and many other functions marginally)(you are on your own processing LDOS volumes under NEWDOS; don't call Apparat when you get into trouble). Basically, when shifting from one DOS to another, the user must off-load the hard disk files to floppies under that DOS using the NEWDOS/80's HDBACKUP program and bring them back in under the other DOS using NEWDOS/80's HDBACKUP program (the NND parameter must be used when the DOS is other than NEWDOS/80).

10. A number of user programs read and interpret the directory. If that program was reading the directory as the DIR/SYS file, observing the protected sector error code and observing EOF, there should be no problem. If the program was using the DDSL value in the data volume 1st sector to compute the directory location, the program will fail unless the data volume has spg1 = 5. If the program was assuming the location and size of the directory, it will most probably fail!!!

11. A data volume must not exceed 65535 sectors. Aside from the space used by BOOT/SYS and DIR/SYS on that data volume, all the remaining space may be allocated to one file, over 16 million bytes. The sector range assigned to one data volume must NOT overlap that of any other data volume; it is the user's responsibility, through careful PDRIVE definition of the data volumes, to avid this overlap, which can be quite disastrous.

12. A hard disk drive is logically divided into one or more data volumes via judicious use of the PDRIVE HDS parameter. Though a data volume is limited to a maximum of 65535 sectors, a hard disk drive is limited ONLY by its actual capacity AND the limitations that Sectors Per Track (SPT or spg1) must be less than 256, Tracks Per Cylinder (TPC)(or RSC (Recording Surface Count)) must be less than 256, Tracks Per Surface UPS or tps1) must be less than 65536, and TPS times TPC must be less than 65536.

13. A hard disk physical drive's space may be divided into drive sections. Normal NEWDOS/80 Version 2.5 operations DO NOT require this. However, if your division of the hard disk is to be such that part of the hard disk is to be used for data volumes of another DOS (such as LDOS) which assigns data volumes in units of one or more entire recording surfaces, it is necessary to sectionalize your hard disk under NEWDOS/80. This is done by setting the PDRIVE HDS sub-parameter ssc1 value to the number of recording surfaces assigned to that drive section and by setting, the sfs1 value to the relative number of the first recording surface assigned to that drive section. For a given drive, no two sections may share the same recording surface, and no data volume may have space assigned from more than one drive section.

14. For DOS command COPY, the =tc1 parameter is not legal if the SOURCE is a hard disk data volume. For FORMAT and COPY, the =tc2, DDSL and DDGA parameters are not legal if the DESTINATION is a hard disk

data volume. For FREE and the header of DIR, to avoid ambiguity, a track count of 0 is displayed if the data volume is on a hard disk.

15. The SUPERZAP displays may look awkward as they were not designed to handle over 9999 sectors. However, they do work, excepting that TRK and SOT values are not displayed for sectors on hard disk. The DTS main menu function is not allowed for hard disk volumes.

16. Format 5 COPY (full diskette COPY) requires that SOURCE and DESTINATION have the same GPL and SPG values and, if the destination is on a hard disk, the same ddsl1 and ddsa1 value. Otherwise format 6 COPY (Copy By File) must be used.

17. Hard disk volumes defined under Model III NEWDOS/80 can be used under Model I NEWDOS/80 and vice versa if NEWDOS/80 supports the drive for the Model I and III. The files on these volumes are NOT useable interchangeably if they were NOT useable interchangeably when those files were on diskettes (such as system program and most user, non-BASIC program files). If you intend to use a hard disk with both your Model I and your Model III (though not at the same time) and intend to run the system from that hard disk, you should create two system volumes on the hard disk, one for the Model I and one for the Model III.

18. ****** Errors may occur in DIRCHECK and SUPERZAP if DFG (MINI-DOS) or 123 (DEBUG) are used during the program's execution and the target drive is not explicitly re-specified after conclusion of MINI-DOS or DEBUG. After MINI-DOS or DEBUG in SUPERZAP, it is recommended that you return to the main menu or do the 'J' display function; for DIRCHECK, respond Y or N to the menu.

3. CHANGES TO PDRIVE for hard disk operation.

No existing parameters in PDRIVE have been changed (so floppies are defined exactly as before), and one parameter, the HDS parameter, has been added to accommodate the hard disks.

The TRS-80 diskette directory was originally intended for 35 or 40 track diskettes of 350 to 400 sectors. In NEWDOS/80 Versions 1 and 2, the directory was modified somewhat to allow for a maximum of 222 user files instead of 62 and allow a maximum of 1536 granules instead of 192. To get these extra granules, the granule lockout table was eliminated from the GAT sector and number of granules per lump (GPL) was expanded from the old implied value of 2 to a user specified value with a maximum value of 8. At 5 sectors per granule, this allowed for 7680 sectors (1,966,080 bytes) per data volume.

However, with hard disks, we really want the capability of allowing a volume to be up to 65535 sectors and a file to be not much less than that. In order to retain the same directory structure but increase the number of sectors for a data volume, we have changed the number of Sectors Per Granule from the old implied value of 5 to a user specified value of not more than 255. Theoretically, this should allow for 1536 * 255 = 391,680 sectors, but there is another governing restraint, that of the NEXT and EOF fields of the directory FPDE and the file's FCB. These fields allow for a maximum of 65535 sectors (if wrap around is to be avoided). Normally this restriction limits the size of a file, but actually this restriction limits a data volume's size

since NEWDOS/80 has a special use of the FCB that allows sector I/O directly to a data volume, bypassing the file concept altogether. Therefore, the NEWDOS/80 version 2.5 hard disk system limits a data volume to 65535 sectors (16,776,960 bytes). Since each volume has a BOOT/SYS file and a DIR/SYS file, the maximum size of a user file is somewhat less than 65535 sectors.

Though 5, 10 or 15 million byte hard disk can be treated by NEWDOS/80 as one data volume, it is generally desirable to divide a hard disk into more than one data volume. NEWDOS/80 allows the user great flexibility in this, admittedly at a cost of complexity (as usual with NEWDOS/80's PDRIVE which many users are still uncomfortable with). A PDRIVE slot definition actually specifies a data volume, not a floppy drive or a hard disk drive or a hard disk drive section. The specifications for the drive and, optionally, drive section are simply part of the specifications of a data volume.

The definition of hard disk data volumes is more difficult and more critical than for floppy diskette data volumes. The user is solely responsible of assuring that a hard disk sector is NOT shared by two or more data volumes. As an aid to the user, the BASIC program EXTPDRIV/BAS has been provided to search an ASCII text file for a specified definition and assign the definition to a specified PDRIVE slot. Using SCRIPSIT or CHAINBLD, the user can carefully and permanently build his/her hard disk data volume definitions (actually just the HDS parameters), and later, when a particular data volume is needed in a particular PDRIVE slot, EXTPDRIV can be used to effect this assignment.

Further, NEWDOS/30 Version 2.5 does NOT maintain a table of bad hard disk sectors. If your hard disk has bad sectors, you must either operate that drive with a sufficiently reduced SPT (sectors per track) value or you must define data volumes such that the bad sectors at not included within any data volume.

Since a hard disk data volume's definition has more values than for a floppy data volume, and we want to limit each slot's definition to one line on the display, we have decided to combine all 12 values of a hard disk data volume specification into one parameter, the SIDS (Hard Disk Specification) parameter. The 12 values are called sub-parameters; ALL 12 MUST be given EACH time the HDS parameter is used, and all must be in the exact order specified. The specification of the LIDS parameter is:

HDS=(hddn1,tps1,sfs1,ssc1,spt1,tsr1,vfs1,vsc1,spg1,gpl1,ddsl1,ddsa1)

where:

1. hddn1 means Hard Disk Drive Number and is the relative number (0 3) of the drive on the hard disk cable with 0 being the first drive. hddn1 specifies which physical hard disk drive the data volume is on.

2. tps1 means Tracks Per Surface and is the number of tracks per recording surface (also the number of cylinders) for the hard disk drive. Each recording surface of the drive has tps1 number of tracks. tps1 is an integer from 0 to 65536. For Apparat hard disks, tps1 = 306. For Tandy 5 Meg hard disks, tps1 = 153.

   LDOS 5.1.3 appears unable to support the tps1 value of 306 used with Apparat's hard disk. However, an Apparat 10 Meg hard disk (with tps1 =

306 and RSC = 4) can be used as a 5 Meg hard disk with LDOS 5.1.3 where implied values of tps1 = 153 and RSC = 4 are used.

3. sfs1 means Section First Surface and is the relative number of the first surface of the hard disk drive assigned to the drive section containing the data volume. sfs1 is an integer between 0 and RSC-1. If you are not sectioning your hard disks, sfs1 will always be 0.

RSC means Recording Surface Count and is the number of recording surfaces for the hard disk. Another term for the number of recording surfaces is TPC (Tracks Per Cylinder). For Apparat 5, 10 and 15 Meg hard disks, CSC is 2, 4 and 6 respectively. For Tandy 5 Meg hard disks, RSC is 4.

4. ssc1 means Section Surface Count and is the number of consecutive surfaces of the hard disk drive assigned to the drive section containing the data volume. ssc1 is an integer between 1 and RSC with the sum of sfs1 and ssc1 not greater than RSC. If you are not sectioning your hard disks, ssc1 will always equal RSC.

5. spt1 means Sectors Per Track and is the number of 256 byte sectors on each track of the hard- disk drive which in turn is the number of sectors formatted on each track by the format program supplied with your hard disk for Apparat and Tandy hard disks, this is the HDFMTAPP program). spt1 is an integer from 1 to 255. Normally, Apparat and Tandy hard disk drives have 32 sectors per track; however, if during HDFMTAPP formatting of the hard disk, a track is found with more than one error sector, it will be necessary to format the hard disk with less titan 32 sectors per track unless you intend to define data volumes to bypass the bad sectors; remember, NEWDOS/80 does NOT maintain a hard disk bad sector table.

6. tsr1 means Track Stepping Rate and is a code used by DOS to send track-to-track stepping rate information to the hard disk controller wizen it is necessary to move the disk arm which contains the read/write heads. tsr1 is an integer between 0 and 255. Apparat hard disk use tsr1 = 0. Tandy 5 Meg hard disks require tsr1 = 6.

7. vfs1 means Volume First Sector and is the relative sector number within the drive section of the data volume's first sector (the data volume's relative sector 0). vfs1 is an integer between 0 and 16,777,215 with an effective upper limit of one less than the number of sectors assigned to the drive section (if a hard disk is not sectioned, the hard disk is one in the same as its one section). If vfs1 = 0, then the data volume's sector range starts with the first sector of the drive section; further, if both vfs1 and sfs1 are 0, the data volume's range starts with the drive's 1st sector.

8. vsc1 means Volume Sector Count and is the number of consecutive sectors of the drive section, beginning with sector vfs1, assigned to this data volume. vsc1 is an integer between 0 and 65535 but the sum of vfs1 and vsc1 must not exceed the number of sectors assigned to the drive section (which is tps1 * ssc1 * spt1). If vsc1 is simply the asterisk character instead of an integer, PDRIVE will assign all of the drive section's remaining sectors to the data volume.

********** IMPORTANT. PDRIVE will accept a vsc1 value of 0, meaning a null data volume, and it will allow the data volume definition into an active slot (provided the definition has no other errors). If vsc1 is 0, DOS will generate a DEVICE NOT AVAILABLE error whenever a slot is selected that contains this data volume. This is needed as a way of filling in PDRIVE slot definitions so that access can be made to slots 4 - 7 for floppy diskette operations when running the system from hard disk and not all of slots 1 to 3 are defined for valid hard disk data volumes.

The sub-parameters hddn1, tps1, sfs1, ssc1, spt1, vfs1 and vsc1 combine to define a unique range of hard disk sectors assigned to the data volume. No sector in this range may be shared by another data volume defined by PDRIVE; it is the user's responsibility to avoid this conflict. Otherwise, the same sector can end up being used for two different purposes.

9. spg1 means Sectors Per Granule and is the number of sectors in each allocation granule for this data volume. spg1 is an integer between 1 and 255. If spg1 = 0 is specified, PDRIVE will compute the lowest spg1 above 4 that will suffice for the gpl1 value specified and the number of sectors assigned to the data volume (vsc1).

When DOS assigns disk space to a file, it does so in minimum units called granules; so the spg1 value defines the minimum number of sectors allocated to a file and also is one more than the maximum number of sectors that a file will have allocated beyond its needs. Generally, it is desirable to have a small spg1 value, but the smaller the spg1 value, the smaller the maximum size a data volume may be. In the regular NEWDOS/80 Version 2, a SPG value of 5 was implied and always used, except in some of the COPYs to and from special TRSDOS diskettes. If full diskette COPY (not CBF) compatibility is wanted with the floppies, spg1 = 5 must be used as that is the standard in the NEWDOS/80 Version 2 floppy world.

****** Warning, when a NEWDOS/80 system volume is being COPY'ed using CBF and the destination spg1 value is less than the source spg1 value, DISKETTE GAT OVERFLOW error may occur. The only alternative is to copy the system from the hard disk system diskette and use a destination spg1 greater than 4.

10. gpl2 means Granules Per Lump and is the maximum number of allocation granules for each byte in the data volume directory's Granule Allocation Table in the GAT sector (the first sector of the directory). gpl1 is an integer between 2 and 8. GPL = 2 is the standard for the old Model I TRSDOS 2.3, and the NEWDOS/80 Version 2 master diskettes use GPL = 2. However, any data volume, whether hard disk or floppy, with more than 1920 sectors, should use a larger GPL under the criteria that it is better to increase the GPL that the SPG. It is recommended that if GPL = 2 is not used, then use GPL = 8. Though the other values are legal, don't use them unless you are attempting compatibility with another DOS.

A lump???? For NEWDOS/80 Version 2, we wanted to eliminate the one-to-one correspondence between a byte in the GAT table and a diskette (or hard disk) track or cylinder so that granules could flow across track and cylinder boundaries. A granule's allocation state is

handled by one bit in the GAT, and we wanted to use all eight bits in each GAT byte to extend the number of granules the GAT could account for. However, the old TRSDOS 2.3 standard was to use only the right two bits of each GAT byte; so we couldn't arbitrarily force all directories to start using all 8 bits. Yet, we wanted to allow use of all 8 bits; so we had to come up with a name for a byte in the GAT as distinct from anything else. Under the assumption that if a number of sectors is a granule, then a number of granules could be called a lump, we defined a lump to be simply a byte in the Granule Allocation Table in the data volume directory's first sector, and that's all it is.

11. ddsl1 means Default Directory Starting Lump and means the relative number of the lump whose 1st sector is the beginning of the data volume's directory. ddsl1 is an integer between 1 and 191, though no guarantee is given that a particular value will work. The standard ddsl1 value in the 35/40 track single sided, single density diskette world was and is 17, and your master NEWDOS/80 system diskette uses that value. If ddsl1 = 0 is specified, NEWDOS/80 will compute a ddsl1 value somewhere near the middle of the data volume, but not greater than 80, as it is assumed more data will exist near the beginning of the volume than at the end.

All Model I and Model III DOSS put the directory somewhere in the middle of the data volume. Since NEWDOS/80 runs with a variety of diskette and hard disk capacities, NEWDOS/80 allows the user to specify where the directory is to be put. The ddsl1 value is this specification. NEWDOS/80 stores the ddsl1 value in 3rd byte of the first sector of BOOT/SYS (also the first sector of the data volume) during data volume format (either FORMAT or COPY) so that DOS (and clever users) can find the directory. NEWDOS/80 senses it has lost the directory location when it reads a directory sector that is not protected. It then goes to the 3rd byte of the volume's 1st sector for the ddsl1 value and computes the directory location. The standard DDSL value for the 35 and 40 track single density diskettes was 17, but as diskettes have increased in capacity and hard disks have appeared, starting the directory at lump 17 placed it too close to the start of the data volume. For dual sided 80 track, double density diskettes with GPL=8, it was common to put the directory at lump 35.

In the diskette world, DDSL has meaning only when a diskette is formatted as NEWDOS/80, at all other times, can find the directory when it wants to. However, in the hard disk world, since we can't write directory sectors with address marks different from the other sectors, NEWDOS/80 cannot tell when it should go to the volume's first sector, get the ddsl1 value, and re-compute the location of the directory. Therefore, the ddsl1 value is used by NEWDOS/80 at all. tines to know where a hard disk volume directory is. If you change the ddsl1 value at a time other then just before the hard disk volume is formatted, NEWDOS/80, without realizing it, will process non-directory data as directory data.

12. ddsa1 means Default Directory Sector Allocation and specifies the number of sectors to be used for the directory. ddsa1 is an integer from 10 to 33. This ddsa1 value is different than the DDGA value used by PDRIVE for floppy diskette definitions. Do not confuse the two. The change from DDGA to DDSA was necessitated by the fact that SPG for the hard disks is no longer a standard 5 sectors per granule. A ddsa1 value

of 10, 15, 20, 25 and 30 is compatible with older configurations that used DDGA=2, 3, 4, 5 or 6 respectively. A ddsa1 value of 33 allows a data volume to have a maximum of 246 user files. Unless a hard disk data volume is to be small or compatibility with diskettes is to be maintained, it is recommended that ddsa1 = 33 be used.

The ddsa1 value for hard disk is more important than the DDGA value is for floppies. The DDGA value is used only at diskette format time. The ddsa1 value, along with the ddsl1 value, is the only way the DOS sector I/O routines know if a sector is part of the hard disk data volume directory or not; therefore, if the ddsa1 value is to be chanted, it must be changed only before a hard disk data volume is formatted. The ddsl1 and ddsa1 values are the ONLY way the NEWDOS/80 sector I/O routines know that a given hard disk sector is a directory sector.

EXAMPLES:

****** Remember, when parameter HDS is specified, all 12 sub-parameters must be supplied in the correct order.

1. PDRIVE,0,1,HDS=(0,306,0,2,32,0,0,2880,5,8,35,33)
specifies a 2880 sector data volume with 5 sectors per Granule, 8 granules per lump, a 33 sector directory positioned at the start of lump 35. TLC first 2880 sectors of the first drive section of hard disk drive 0 will be allocated to this volume. The drive section consists of the first 2 recording surfaces of the drive, which may or may not be all that the drive has. Each recording surface has 306 tracks. Each track has 32 sectors and the drive's stepping rate code is 0. This data volume cam he accessed by user programs as drive 1.

2. PDRIVE,0,2,HDS=(1,153,1,3,32,6,2000,10000,0,8,0,33)
specifies a data volume on hard disk drive 1 that has 153 tracks per surface and 32 sectors per track. The drive section consists of the 2nd, 3rd and 4th recording surfaces. The data volume consists of 10,000 sectors beginning with the drive section's relative sector 2,000. PDRIVE will compute the sectors per granule and use 8 granules per lump. PDRIVE will compute the position of the 33 sector directory within the volume. User programs will access this data volume as drive 2.

3. PDRIVE,0,1,HDS=(0,153,0,4,32,6,0,*,0,8,0,33) specifies a data volume that occupies all 19,584 sectors of the first four recording surfaces of hard disk drive 0. The vsc1, spg1 and ddsl1 values are computed by PDRIVE. User programs will access this data volume as drive 1.

4. PDRIVE,0,3,HDS=(0,153,0,1,32,6,0,0,5,2,17,33)
specifies a null data volume (vsc1 value is 0). NOTE, all other sub-parameters must be valid. If a user program attempts I/O via drive 3, DEVICE NOT AVAILABLE, error will occur. However, the FREE command and any other DOS functions that search the various drives will ignore drive 3.

EXAMPLES OF PDRIVE COMBINATIONS:

1. Settings to exactly overlay the standard LDOS values on a single Tandy 5 Meg drive where each of 4 volumes has one surface.

```
HDS=(0,153,0,1,32,6,0,4896,5,8,61,33)
HDS=(0,153,1,1,32,6,0,4896,5,8,61,33)
HDS=(0,153,2,1,32,6,0,4896,5,8,61,33)
IDS=(0,153,3,1,32,6,0,4896,5,5,61,33)
```

This divides the hard disk drive into 4 drive sections, each containing one data volume. If you assign the 4 definitions to PDRIVE slots 0 - 3 respectively, you must have moved the NEWDOS/80 system to hard disk as described in section 5. However, if you assign these definitions to slots 4 - 7 and have previous file data from LDOS operation, you can look at that data via SUPERZAP (if you are interested), and you can look at the directory starting at relative sector 2432.

2. The user has one Apparat 5 Meg drive, fundamentally wants all his/her user files accessible via drive 1 with a small amount of work space on drive 2. The user wants to run using a hard disk system volume for drive 0 and to be able to access to his two floppies via slots 4 and 5. With SYSTEM option AL = 6, the definitions for slots 0 - 5 will be as follows:

```
HDS=(0,306,0,2,32,6,6,720,5,8,17,10)
HDS=(0,306,0,2,32,6,720,16864,11,8,80,33)
HDS=(0,306,0,2,32,6,17584,2000,5,8,25,33)
HDS=(0,1,0,1,1,0,0,0,5,8,17,10)  a dummy definition
TI=A,TD=E,TC=40,SPT=18,TSR=0,GPL=2,DDSL=17,DDGA=2
TI=A,TD=E,TC=40,SPT=18,TSR=0,CPL=2,DDSL=17,DDGA=2
```

Note that the 8th sub-paraneter (vsc1) of HDS is the number of sectors assigned to the data volume (NOT the ending sector number). Slot 3 has been defined as a dummy (the vsc1 value = 0) to allow FREE to get to slots 4 and 5.

3. The user has two Apparat 10 Meg drives and wants the system volume on hard disk, 3 hard disk data volumes with slot 1 to contain all the space of the 2nd drive. The definitions for slots 0 - 7 could be:

```
HDS=(0,306,0,4,32,0,0,5595,5,8,69,33)
HDS=(1,306,0,4,32,0,0,39168,26,8,94,33)
HDS=(0,306,0,4,12,0,5595,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,11190,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,16785,5595,5,8,11,33)
HDS=(0,306,0,4,32,0,22380,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,27975,5595,5,8,69,33)
HDS=(0,306,0,4,32,0,33570,5595,5,8,69,33)
```

4. FORMATTING YOUR HARD DISKS.

Hard disks must be formatted before they can be used with NEWDOS/80 Version 2.5 or any other DOS. Some hard disk manufacturers format their hard disks before shipping the drive and have internal coding to bypass error sectors automatically, and if this is the case, you may bypass this section on hard disk formatting.

NEWDOS/80 Version 2.5 does not maintain an error sector table and assumes the consecutive sectors that it can read from a hard disk are error free. Bad (error) sectors must be hidden from NEWDOS/80. One way to do this is to reduce the number of data sectors per track, allowing HDFMTAPP to write a dummy sector over the bad spot on the track. Another way is to later define (via PDRIVE) the data volumes such that the bad sectors are not part of any data volume.

NEWDOS/80 DOS commands FORMAT or COPY with format do not actually format a hard disk. The actual formatting must be done either by a stand alone program or by a program that operates under NEWDOS/80 but does all of its own I/O to the hard disk. NEWDOS/80 Version 2.5 provides the program HDFMTAPP to format Apparat's and Tandy's hard disks for the Model I or III. The format program for other types of hard disk drives must be supplied to the user by that hard disk drive retailer.

Formatting a hard disk destroys all information on that hard disk. If you must re-format a hard disk, be sure to extract as much valued information from that hard disk (you may use program HDBACKUP) as you can before re-formatting.

Though we recommend that you format the hard disk drive before use with NEWDOS/80 so that you will be made aware of all the error sectors, a previous format done for another DOS (such as done during the LDOS 5.1.3 hard disk initialization) can suffice if there were no error sectors or you know where they are for bypassing in your definition of data volumes using PDRIVE, and if you know the parameters needed for PDRIVE's HDS parameter. If you elect to do this, then bypass the rest of this section (on HDFMTAPP). An example where you might want to do this is where you wish to share the hard disk between one or more existing LDOS volumes and one or more NEWDOS/80 volumes, thus allowing both LDOS and NEWDOS/80 to use the hard disk (though not both at the same time and not the same data volumes).

To format an Apparat or Tandy Model I or III hard disk, assure the hard disk drive is properly connected to the computer and power is on; then execute the DOS command HDFMTAPP, proceeding as follows:

1. Reply the relative hard disk drive number. This is the same number as hddn1 in the PDRIVE HDS parameter.

2. Reply the relative number of the first surface to be formatted. When formatting an entire hard disk drive, this value is 0.

3. Reply the number of recording surfaces to be formatted. When formatting an entire hard disk drive, the value is the number of recording surfaces the hard disk has (the RSC or TPC values discussed earlier). For Apparat 5, 10 and 15 Meg hard disks, this value is 2, 4 and 6 respectively. For Tandy 5 Meg drives, this value is 4.

4. Reply the number of tracks per surface UPS or tps1) for this drive. This is the same as the number of cylinders the drive has. For Apparat hard disks, this value is 306. For Tandy 5 Meg hard disk drives, this value is 153.

5. Reply the relative number of the first cylinder (the first track on a surface) to be formatted. When formatting an entire hard disk drive, this value is 0.

6. Reply the number of cylinders (number of tracks on each surface) to be formatted. When formatting an entire hard disk drive, this value is the same as given in #4 above.

7. Reply the track stepping rate code. Use a value of 15 here as we are not too concerned with a slow stepping rate during formatting.

8. Reply your intended data sectors per track. The normal value here is 32. The tracks supposedly have a capacity for 33 sectors per track, but test have shown that many parity errors occur. Specifying 32 sectors per track does allow for one error sector per track to be automatically specially encoded so that NEWDOS/80 will never see it.

9. Reply the sector interleave count. We recommend a value of 21 if there are to be 32 sectors per track. This value allows time for the DOS I/O routine, the transfer of the bytes on the cable to/from the drive's buffer, the actual read/write of the sector by the drive, and 1 to 2 milliseconds for the user program to invoke the I/O for the next sequential sector. This value of 21 is also optimal for the HDBACKUP program, which is too slow as it is. Values 19 and 20 will work, but allow much less time for the user program to turn the I/O around. Values 22 to 30 allow the user more turn around time but slowly decrease the number of I/Os per second that can be done. Values 0 - 18 allow too little time for the above functions and cause the hard disk to wait till the next revolution (16.7 ms) for the next sector.

10. Reply N if you wish to restart the specifications again at step 1 above. Reply Y if the program is to start the format.

11. Once started, the formatting will proceed, blinking an asterisk in the display upper right corner to indicate progress. If a track cannot be formatted with the required number of sectors, an error will be displayed giving the cylinder, head and number of error sectors above and beyond the number implicitly allowed in step 8 above. A track that has some error sectors and some good data sectors will have the good sectors numbered from track relative sector 0 consecutively on up with the higher numbered sectors for that track simply not there.

12. During HDFMTAPP execution, holding down the up-arrow key causes the program to terminate and the right-arrow key causes the program to pause. After right-arrow, pressing ENTER causes the program to continue. This pause/cancel function is useable only through the keyboard matrix, not via remote terminals.

13. When the format is complete, the number of tracks with too many errors will be displayed. If there are any such tracks, you SHOULD reformat the hard disk using a lesser sectors per track value. ;lark the resulting sectors per track value spt1 on a label on the hard disk to remind you of what spt1 value MUST be used in all PDRIVE definitions for data volumes on that drive. HOWEVER, when only a small number of consecutive tracks have all the error sectors, you may decide to leave the error sectors alone and define your volumes (via PDRIVE) in such a way as to assure that the error tracks at not assigned to any volume (i.e., ending one volume on the last sector of the first good track preceding the bad track range arid starting another volume on the first sector of the first good track following the bad track range). If the error tracts are assigned to a volume, NEWDOS/80 will give SECTOR NOT FOUND error when ever I/O is attempted to the a bad, non-existent sector. NEWDOS/8O does not maintain any bad track or bad sector tables.

14. If all tracks have beer, formatted with the required number of sectors, the hard disk is now ready for use by NEWDOS/80.

It is possible, due to the extensive specifications, for the HDFMTAPP program to format just one track on the hard disk. This may be of interest to a few users when a track has apparently gone bad and an attempt is to be made to reformat just that one track.

5. MOVING NEWDOS/80 VERSION 2.5 TO THE HARD DISK.

Usually, you want to have slots 0 to 3 as hard disk volumes and still have access to your two floppy disc drives. For this, it is necessary to operate using the NEWDOS/80 Version 2.5 system volume, which must be volume 0, from the hard disk. This section steps you through setting up NEWDOS/80 Version 2.5 to run from the hard disk. The hard disk is assumed previously formatted.

1. Be sure you know how to use the DOS command PDRIVE, especially with the Hard Disk Specification parameter HDS.

2. Mount a copy of the NEWDOS/80 Version 2.5 hard disk system diskette in floppy drive 0. This will be known as the system diskette as different from the hard disk system volume which will be on the hard disk.

3. Choose one of the system diskette's PDRIVE active slots whose number is greater than one. For this example slot 2 will be used (the SYSTEM option AL must be at least 3). If you choose a different slot number, then use that number in place of 2 in the following discussion.

4. Using PDRIVE,0,2,A,HDS=----- define floppy system diskette PDRIVE slot 2 with the specifications wanted for the hard disk system volume.

5. Execute the DOS command:

```
COPY, 0,2,,FMT,CBF,USD
```

and respond to the requests for SOURCE and DESTINATION diskettes (even though the destination is on a hard disk). GAT OVERFLOW error may occur if the spg1 value for the destination is less that that of the source; in which case you must increase the destination spg1 value.

6. Execute PDRIVE,2 to see the hard disk system volume's specifications for the 10 slots defined on that volume. Note that the definition for slot 2 has been duplicated in slot 0. This was done as a normal part of the COPY done above. Don't confuse the specifications of PDRIVE,2 which refers to system control data on drive 2, the intended hard disk system volume, with those of PDRIVE,0 which refers to system control data on drive 0, the floppy system diskette.

7. Using PDRIVE,2,----- define the PDRIVE specifications as you intend for that volume to be used as the system volume (drive 0). Since PDRIVE,2,2 has been duplicated as PDRIVE,2,0 in anticipation of that hard disk volume becoming the system volume, you MUST now redefine the PDRIVE,2,2 slot for another volume or by setting its vsc1 value to 0, causing slot 2 to be undefined. The specifications for PDRIVE,2 slots 0 - 3 must be for hard disks volumes only. Definitions for the floppies must be in slots 4 - 7 which correspond to the old drives 0 - 3 respectively. If one or more of the slots 4 - 7 are not used for floppies, then they may be used for hard disk volumes, thus allowing a maximum of 8 hard disk volumes to be active at any one time. Do not go on to the next step until all PDRIVE,2 slots have been defined as you will want them to be in the system operating from the hard disk, though it is not necessary to change any of them except slot 2 and you should not change slot 0. Remember, you cannot use PDRIVE parameter A when doing PDRIVE,2 definitions as that volume is not the current system volume.

8. Using SYSTEM,2,AL=xxx, specify the number of PDRIVE,2 slots to be active. xxx must be between 1 and 8, and must be at least 5 if any floppies are to be used.

9. The hard disk system volume now has the correct specifications, but we need a hard disk boot diskette (also known in this section as the boot diskette) to enable RESET (also known as BOOT), which must start on floppy drive 0, to switch to the hard disk system volume. This diskette must contain at least BOOT/SYS, DIR/SYS and SYS0/SYS, and must have its PDRIVE slot 0 defined exactly as for the hard disk system volume. So we proceed to do this.

10. If the system diskette's PDRIVE,0,1 specification is not identical to that for PDRIVE,0,0, then make them so by executing the command:

```
PDRIVE,0,1=0,A
```

11. Assign an otherwise unused diskette as the hard disk boot diskette and label it as such. Mount the boot diskette in floppy drive 1.

12. At this point, the system diskette is in floppy drive 0, the hard disk boot diskette in floppy drive 1, and the hard disk system volume is on the hard disk. Execute the DOS command:

```
FORMAT,1,,,,Y
```

13. When done, execute to DOS command:
    COPY,SYS0/SYS:2,:1
to move a copy of SYS0/SYS, the resident DOS, from the hard disk system volume to the hard disk boot diskette. Since it is the first file placed on the boot diskette, aside from BOOT/SYS and DIR/SYS, it will automatically be placed in the proper place for RESET.

14. When done, execute: PDRIVE,1,0=2 to move the proper hard disk system volume specification to the boot diskette's PDRIVE slot 0.

15. At this point, you may want to change the PDRIVE,0,2 and PDRIVE,0,1 definitions back to what they were before steps 3 and 10 above. This step is optional.

16. Remove the system diskette from drive 0. Move the hard disk boot diskette from drive 1 to drive 0 and press RESET. Computer execution will read the boot sector and then the resident DOS, SYS0/SYS, from the boot diskette in floppy drive 0 and then shift to the hard disk. You may now take the hard disk boot diskette out of drive 0 or leave it in, in which case it may be accessed via the PDRIVE slot 4 (used for floppy drive 0 when the hard disk system is in use) if PDRIVE,0,4 is defined for a floppy. The diskette can be accessed by user programs as drive 4.

You may use the hard disk boot diskette as a normal data diskette by copying data files on to it. Remember though, it is the hard disk system's boot diskette and its SYS0/SYS is the resident DOS that is loaded into main memory at RESET time and remains there until the next RESET.

********* WARNING. A backup up of a hard disk boot diskette will not transfer its booting-up-the-hard-disk capability unless the backup is done using format 5 COPY with the BDU option.

The hard disk system volume is drive (slot) 0 when operating the system from the hard disk. The hard disk system volume does NOT have to be positioned at the beginning or a hard disk drive; in steps 4 and 5 above, you are allowed to place the hard disk system volume where you wish on the hard disk.

The file SYS0/SYS on the hard disk boot diskette MUST remain exactly identical to the SYS0/SYS on the hard disk system: volume. If you alter one, you MUST alter the other. This is necessary because the hard disk system: thinks its own SYS0/SYS is in the resident DOS area (4000H - 4CFFH and 0F900H - 0FFFFH) at all times when actually it is the SYS0/SYS from the hard disk boot diskette.

If you only have one floppy drive, then the following changes must be made to the above procedure:

1. Step 10 above is excluded.

2. In step 11, do not mount the boot diskette into drive 1.

3. In step 12, change the command to be FORMAT,0,,,,Y and perform diskette mounts as requested where the SYSTEM diskette is the system: diskette and the DESTINATION diskette is the boot diskette.

4. In step 13, change the command to be COPY,$SYS0/SYS:2,:0 Perform the diskette mounts as requested where the SYSTEM diskette is the system diskette, SOURCE diskette is the hard disk system volume and the DESTINATION diskette is the boot diskette.

5. Replace step 14 with the following action. Enter SUPERZAP and at the menu, reply CDS. Remove the system diskette from floppy drive 0, and mount the boot diskette in floppy drive 0. Reply Y. Reply 2,2 for the source drive and relative sector. Reply 0,2 as the destination drive and relative sector. Reply 1 as the sector count. Press ENTER to return to menu. Remount the system diskette in floppy drive 0. Reply EXIT to exit SUPERZAP and return to DOS READY.

6. DEFINING PDRIVE SLOTS FROM A VOLUME DEFINITION FILE.

The definition of hard disk volumes via PDRIVE is more difficult and more critical than for floppy disk volumes. Therefore, it is recommended that the user carefully plan out his/her allocation of hard disk space amongst the various volumes and store the definitions (the HDS parameter part) into an ASCII text file (called a data volume definition file) created and updated by using either CHAINBLD or SCRIPSIT or both. Do this very, very, very carefully as you can create havoc amongst your data if two or more data volumes share the same hard disk sectors. Under NEWDOS/80 Version 2.5, you have great flexibility in assignment of hard disk space to data volumes, but with this flexibility comes complexity of definition.

Each record within the data volume definition file must start with a unique but arbitrarily assigned identification integer. Following the integer must be a comma followed by the intended PDRIVE definition excluding the initial part of the PDRIVE command (the PDRIVE,dn1,dn2, portion) and the ,A (for activation) as these parts of the PDRIVE command will be supplied by the EXTPDRIV/BAS program.

Since each definition record within the data volume definition file starts with an integer, you may imbed comments within the file as you like provided the comment record does not start with an integer.

It is strongly recommended that you keep copies of the data volume definition file on floppy diskettes in case that file on your hard disk becomes unusable. Remember, this is your master copy of the hard disk space layout!

Assuming that you have carefully constructed your data volume definition file, you may assign one or more of these definitions to the various PDRIVE slots when needed by running the BASIC program EXTPDRIV/BAS.

1. The program will ask for the filespec of your volume definition file and then open it.

2. The program will ask for the identification integer of the definition to be used. Respond with an EXACT copy of the integer that starts that definition's record in the file. The program will then search the file for the record.

3. When found, the program will ask for the two numbers needed for the PDRIVE,dn1,dn2,--- function. Respond with the two numbers separated by a comma. The first number, dn1, (usually 0) specifies which data volume contains the system control information, which will be changed by the PDRIVE command. The second number, dn2, specifies which PDRIVE slot definition is to be changed.

4. The program will then ask if slot definitions are to be activated within the resident DOS (i.e., the ,A PDRIVE parameter). Reply Y if so; N if not.

5. The program will then build the appropriate PDRIVE command and execute the command via DOS-CALL. You will see the PDRIVE results displayed.

6. The program will then ask if there is another definition from the same file to be applied. If you reply Y, the program returns to step 2 above. If you reply N, the program ends.

EXAMPLES of data volume definition file records:

1. 103,HDS=(0,153,0,4,32,6,0,2880,5,8,35,33)

2. 91,HDS=(1,153,0,4,32,6,1000,2000,0,8,0,33)

3. 44,TI=A,TD=E,TC=40,SPT=18,TSR=0,GPL=2,DDSL=17,DDGA=2

7. BACKING UP HARD DISKS TO DISKETTE:

Copies of user data stored on hard disk must be kept elsewhere in case the hard disk crashes, a program malfunctions or a user goofs. Users MUST, from time to time, make backup copies of valued data, the frequency of backup depending upon how often the data changes and how valuable the data is.

NEWDOS/80 Version 2.5 provides the HDBACKUP (hard disk back up) function as a way of saving files from the hard disk(s) to floppy diskettes, and a way of restoring one, some or all of those files back onto the hard disk(s).

HDBACKUP saves by file rather than by full volume contents. It uses this considerably slower technique because over 50% of the restores that are eventually done involve only a selected set of files and not a full media or data volute. Restores to a hard disk don't have to be the result of a hard disk failure but more frequently are due to user mistakes or user program malfunction logically damaging or destroying certain files, and the restore should allow only the damaged files and their interrelated files to be restored, leaving unchanged all other files on the hard disk(s) involved. Unfortunately, saving by file requires more administrative consideration than does saving by entire volume contents; so we hope the greater flexibility will be worth it.

For purposes of HDBACKUP discussion, a backup is the content of the one or acre diskettes used to contain the files copied from data volumes during the execution of the HDBACKUP program's SAVE function. These diskettes must be preformatted and, after being used by SAVE, cannot be read/written using standard DOS functions; however, they can be read/written using SUPERZAP disk (not file) mode.

In this discussion of the HDBACKUP function, a data volume refers to one of the active hard disk data volumes defined via PDRIVE.

HDBACKUP/CMD is the program that (1) creates a backup containing specified files from the various defined data volumes (as defined by PDRIVE) of your system, (2) lists which files are contained within a backup and (3) restores specified files from a backup to the various defined data volumes of your system. HDBACKUP is the method under NEWDOS/80 Version 2.5 of backing up your files from hard disk or diskette and, if necessary, restoring one, some or all of those files back to the hard disk or diskette. Under the SAVE parameter, HDBACKUP creates a backup that spans one or more diskettes. Under the LIST parameter, HDBACKUP lists the filespecs of and errors associated

with the files contained in the specified backup. Via the RESTORE parameter, HDBACKUP copies specified files from the backup to specified data volumes of your system.

The HDBACKUP SAVE function saves a file's contents, not its attributes. Except for the file name, name extension, data volume number and, if NND not specified, the logical record length, no other attributes of the file are saved such as passwords, protection level, etc. SYSTEM files are not SAVEd. The user is responsible for backing 4p system files to regular diskettes using the COPT' command; normally it is sufficient to simply maintain, copies of your original NEWDOS/80 Version 2.5 Hard Disk System diskette and your regular NEWDOS/80 Version 2 System diskette. If the NND parameter is specified, system files included in the INCLUDE list are copied, but are no longer marked as system files.

Provided the NND parameter is specified, the HDBACKUP function is designed to attempt to run with TRSDOS-like DOSS other than NEWDOS/80 Version 2.5. Via the NED parameter, you must inform the HDBACKUP/CMD program of certain values for that DOS.

The HDBACKUP program requires passwords be disabled, as standard file OPENs are done without passwords in the filespecs. If passwords cannot be disabled in the current system, the passwords must be taken off the files being backed up. SYSTEM option AA=N disables passwords in NEWDOS/80.

The HDBACKUP program requires, unless the NND parameter is specified, that all volume directories be named DIR/SYS.

Usually after the user has responded to a request, HDBACKUP displays an * to indicate that it is no longer waiting for an operator response.

HDBACKUP blinks an * in the upper right corner of the display screen to let you know that is preceding in an orderly fashion. The speed of the blinks will vary due to the different functions.

The RESTORE function of HDBACKUP takes a very long time to initialize (in one test of 3444 files, it took 30 minutes). This extra initialization (1) performs KILLs if RENEW specified, (2) creates all new files, (3) CLOSES the files to store the new EOF and release any excess disk space on the data volume, (4) if NND not specified, writes the last sector of each file to allocate any needed disk space and (5) if NND not specified, updates the logical record length in the directory.

The HDBACKUP/CMD program expects the diskettes used for the backup to already be formatted. The program will write over the entire diskette; after SAVE, the diskette will not have a directory. The program will not tolerate a bad sector when writing to the backup diskettes. If a sector is bad, you have three options: (1) retry the write, (2) cancel the entire SAVE function, or (3) restart the SAVE function at the beginning of the current backup diskette. If you choose option 3, you will be asked for the current backup volume again; you should then (and not before) mount a different previously formatted diskette (remember to label it properly) and place the other diskette in your bad diskette collection.

The HDBACKUP command sequence is:

```
HDBACKUP
fc1
PRINT
NND=(filespec1,r/n,spg1,gpl1,spv1)
BSN=list1
TITLE=title1
DATE=date1
TIME=time1
SVL=list2
RVL=list3
SLOW
SKIP
RENEW
MAXERRS=ec1
TEST
INCLUDE
EXCLUDE
*END
```

HDBACKUP invokes the HDBACKUP/CMD program. HDBACKUP must be the only parameter on the first command line (the command line used by DOS to invoke the program). This program then displays the cursor and waits for the user to input subsequent command lines. Command parameters are processed until the *END parameter is encountered. There must not be extraneous characters within a command line. A command line may contain multiple parameters separated by commas, but a parameter must be fully contained within a command line. A command line is limited to 79 characters in NEWDOS/80 and 63 characters for most other DOSs.

The user will generally build the command lines and the file specifications for INCLUDE or EXCLUDE into a CHAIN (aka DO) file as it is strongly recommended that HDBACKUP commands be constructed very carefully. Though CHAINBLD will work, it is recommended you build your chain file via a word processor, storing the resulting file as an ASCII file.

> **** Warning, be sure that the chain file has no extraneous characters after the end-of-line character for the SEND statement; otherwise subsequent responses needed for the HDBACKUP execution will receive bad data.

The TEST parameter was included to allow the user a 'dry' run to test the workability of the command parameters. If you don't know what your are doing, gain some familiarly by using the TEST parameter before doing a live run. Remember, you can't test a RESTORE until you have a backup to test with.

> **** Warning, SAVE with TEST does write backup control information on the backup's 1st diskette; be sure that diskette is intended for a backup.

fc1 fc1 must be the first parameter after HDBACKUP. fc1 specifies the function to be performed which is one of the following:

1. SAVE Anew backup is created having title, date and time as specified by the TITLE, DATE and TIME parameters. The files specified, either explicitly or implicitly, are copied from the specified data volumes to as many backup diskettes as necessary. Parameters BSN, SVL and *END are required. Optional parameters are PRINT, NND, TITLE, DATE, TIME, SLOW, SKIP, MAXERRS, TEST, INCLUDE and EXCLUDE. If one of TITLE, DATE or TIME is not specified, the HDBACKUP program will ask for that parameter. If NND is specified, INCLUDE must be specified.

2. LIST This function lists the files contained within the specified backup and includes their associated error sector numbers. Required parameters are BSN and *END. Optional parameters allowed are PRINT, NND, TITLE, DATE and TIME. The listing starts with the backup's name, date, time, file count and error count. Then for each file in the backup's table of contents, the following are listed:

1. The filespec for the file.

2. If the file has been deleted from the backup table of contents, '***** DELETED *****' is displayed and steps 3 - 6 are bypassed.

3. The file's EOF value in xxx/yyy format where xxx is the relative sector within the file and yyy the relative byte within the sector.

4. The file's logical record length, 1 - 256.

If NND specified during the SAVE that made this backup, the record length may or may not be correct if the file's record length prior to the SAVE was not 256. This occurs under NND as HDBACKUP does not get the record length from the directory but records whatever record length appears in the FCE after OPEN. Normal NEWDOS/80 operations do not use the file's record length from the directory, but many users want it correct anyway. If a file's logical record length was changed during the SAVE and RESTORE, the user may correct it by using the LRL parameter of ATTRIB (see regular NEWDOS/80 Version 2 ZAP 007 (Model I) or ZAP 004 (Model III).

5. The location within the backup of the file's header sector, expressed as a backup volume number and a relative sector within that volume. This is of interest only to those viewing/updating the backup via SUPERZAP. Volumes (diskettes) of a backup are numbered consecutively from 1, not 0.

6. If the file has any error sectors, they are listed each in the decimal format:

sssss/ee/vvv/rrrrr

where:

1. sssss is the sector's relative number within the file.

2. ee is the DOS error code.

3. vvv is the number of the backup volume containing the error sector

4. rrrrr is the sector's relative number within the backup volume.

3. RESTORE The files specified, either implicitly or explicitly, are copied from the backup to the specified data volumes. Required parameters are BSN, RVL and *END. Optional parameters allowed are PRINT, NND, TITLE, DATE, TIME, SLOW, SKIP, TEST, RENEW, INCLUDE and EXCLUDE.

PRINT This parameter informs the HDBACKUP program that display information is to be sent to the printer as well as the display. If PRINT is not specified, only the display will be used. If PRINT is specified, the program will display WAITING ON PRINTER, and then, if the printer is not ready, the program will hang.

NND=(filespec1,r/n,spg1,gpl1,spv1) This option specifies that the Disk Operating System (the DOS) is not WEWDOS/80 Version 2.5, though it can be. If NND is specified, the following hold:

1. SLOW is implied.

2. For SAVE, INCLUDE is required.

3. NND must be specified immediately after fcl and before BSN.

4. For RESTORE, the pre-allocation of needed file space during initialization is not done; an out-of-space error will not be detected until the file is actually restored.

5. file logical record lengths recorded in table of contents during SAVE or in the data volume directory during RESTORE may be wrong if they were not 256.

HDBACKUP is designed to run with NEWDOS/80 Version 2.5, but users initially may have their hard disk data under a different operating system, thus creating a dilemma, as NEWDOS/80 cannot process directories for other DOSs. Recognizing this as potentially a serious problem, an attempt (via the NND parameter) has been made to allow HDBACKUP to run under another DOS using faked extents in the FCB used for backup diskette I/O. This attempt will not work with a DOS that determines a diskette's characteristics from the diskette itself (as HDBACKUP writes over the entire backup diskette) or which automatically changes a drive's specification when an error is encountered. So far, the only successful tests have been (1) with Tandy's Model III Hard Disk Operating System (LDOS 5.1.3) using single sided, double density, 40 track drives as the backup drives specified in the BSN parameter with NND=(TEMPFILE:0,N,6,3,720), and (2) with Tandy's Model I Hard Disk Operating System (LDOS 5.1.3) using single sided, single density, 35 track drives as the backup drives specified in the BSN parameter with NND=(TEMPFILE:0,N,5,2,350). Apparat does not plan to test under the other DOSs or other configurations, and Apparat reserves the right to withdraw the NND parameter and all support for it at any time and without notice.

If using HDBACKUP with the NND parameter does not work with your other DOS, the user will have to find some other way of offloading the files from hard disk under the other DOS and reloading them under NEWDOS/80 Version 2.5.

***** Warning!!! Before using HDBACKUP to offload files under a DOS that is not NEWDOS/80 and then reloading the files to hard disk under NEWDOS/80, the user should offload the valued files to diskettes using the other DOS's normal backup procedures. This provides the user with a second backup source should the conversion to NEWDOS/80 fail.

When using the NND option, certain extra information MUST be provided to the HDBACKUP program. If you don't know what these values are, call the distributor for that DOS; don't call Apparat.

filespec1 is the filespec of new or existing file that HDBACKUP can write one sector to in order to determine a correct FCB to be used for backup diskette I/O. HDBACKUP will write garbage into that one sector and will not CLOSE the file. The file filespec1 must be for a file within a volume that is already mounted when HDBACKUP begins execution; further, for some DOSs, it may be necessary that the file be on a diskette with the same spg1, gpl1 and spv1 characteristics specified in this NND parameter. The diskette can be mounted on a drive specified in BSN below as HDBACKUP will conclude its use of file filespec1 before it asks for the first backup diskette.

r/n is one character, either R or N. R is specified if the EOF field of FCBs (the File Control Block in main memory, not the directory FDEs) for this DOS are in Relative Byte Address format (such as all NEWDOS versions and Model III TRSDOS 1.3). N is specified if the EOF field of the FCBs for this DOS are in Next Record Address format (such as LDOS (regular and hard disk), Model I TRSDOS 2.3 and Model III TRSDOS 1.1)

********* The choice of R or N is critical. Choosing the wrong value will cause every file not ending on a sector boundary to be assigned the wrong EOF in the backup, thus making the file one sector too long or too short. Further, reportable errors may occur.

Once again, the NEWDOS author apologizes for having brought Relative Byte Addressing to the TRS-80 world (the FCBs, not the directories) with the NEWDOS release in March, 1979, thus causing the confusion between RBAs and NRAs (Next Record Addressing). NRA was the standard at that time and has remained the LDOS standard (TRSDOS on the Model III changed to RBAs in July, 1982). NEWDOS shifted to and remains with RBAs because that method is the more reliable method for arbitrary random disk I/O.

spg1 is the number of sectors per granule for this DOS for the backup diskettes that will be mounted on floppy drive(s) specified in BSN below. (LDOS Hard Disk System uses spg1 = 5 for single density 5 inch diskettes and spg1 = 6 for double density).

gpl1 is the number of granules per lump for this DOS for the backup diskettes that will be mounted on the floppy drive(s) specified in BSN below. This is also known as granules per cylinder and is the

number of bits per byte used in the GAT sector to account for granule allocation. LDOS Hard Disk System uses gpl1 = 2 for single sided single density 5 inch diskettes, gpl1 = 4 for double sided single density, gpl1 = 3 for single sided double density diskettes and 6 for double sided double density.

spv1 is the number of sectors per backup diskette. This is the total number of sectors on a diskette (720 for single sided, double density 40 track 5 inch diskettes, 350 for single sided, single density 35 track 5 inch diskettes, 1440 for double sided, double density 40 track 5 inch diskettes). Whatever the number, the DOS must be capable of doing I/O for that number of sectors per diskette.

HDBACKUP/CMD may be moved to another DOS via the following steps:

1. Under NEWDOS/80 Version 2.5, execute LMOFFSET. Respond D. Respond HDBACKUP/CMD. Respond new load address = 7000. Respond N to request appendage. Record the new start, end and entry address values displayed (will be used in step 4 below). Respond <ENTER> to indicate load point not being changed again. Respond N to keep DOS enabled. Respond D. Respond XXX/CMD:0 to write the modified module back to disk. Respond N. Respond PI again. You should now be back at DOS READY.

2. Execute the DOS command LOAD,XXX/CMD:0. This loads the load-offsetted HDBACKUP program created in step 1 into main memory from where it will be written to the other DOS's diskette in step 4 below.

3. Load the other DOS diskette into drive 0 and press RESET to bring up that DOS. Be sure that this DOS does not clear user memory upon coming up.

4. Use the DUMP command for that DOS to store onto that DOS's disk the HDBACKUP/CMD program loaded into main memory in step 2. The DUMP command will need the start, end and entry addresses recorded in step 1. See that DOS's manual for explanation of the DOS command DUMP. For LDOS, this command will be:

DUMP HDBACKUP/CMD:0 (START=X'start',END=X'end',TRA=X'entry')

where start, end and entry are the hexadecimal addresses recorded in step 1 above.

5. If that DOS's DUMP does not allow the filespec HDBACKUP/CMD:0, use what it will allow and then change the file's name via RENAME.

6. The HDBACKUP program is now ready for execution on that DOS.

BSN=list1 The Backup Slot Number specifies either one or two slot numbers (if two, list1 must be enclosed in parenthesis) of the slots (PDRIVE active volumes) to be used for reading/writing the backup diskettes.. These slots must be defined in PDRIVE as floppy disk drives. None of the backup slot numbers may be included in the volume numbers listed in the SVL or RVL parameters. If two slot numbers are specified, they must have the same PDRIVE definition. If only one slot is specified, all backup diskettes will be mounted as needed using that one drive. If two slot numbers are specified, the backup's volume 1 is left mounted on the first drive throughout the HDBACKUP function and the second drive is used for the other volumes. Since backup volume 1 is frequently referred to or updated during the SAVE or RESTORE, assigning two slots (drives) greatly reduces operator actions. If you only have two drives, run the system from the hard disk so that floppy drive 0 is free to be used as a backup drive.

TITLE=title1 title1 is the 0 to 48 printable character title of the backup. For SAVE, this title is assigned to the backup; if not specified in the command lines, the program will ask for it. For LIST and RESTORE, an error will be displayed if TITLE is not specified or title1 does not match that of the backup; the user may elect to use the backup anyway. Where TITLE is specified in a command line, it must be the last parameter of that line as title1, even if over 48 characters, is considered to be the rest of the line; the excess characters are ignored. During SAVE, when a backup diskette is first asked for, the program will reject the diskette if it has been used for a previous backup with the same title, date and time (as it may really be an earlier volume of this backup).

DATE=date1 date1 is the backup's date in nun/dd/yy format. For SAVE, this date is assigned to the backup; if DATE is not specified, the operator will be asked for it. For LIST and RESTORE, an error will be displayed if DATE is not specified or date1 does not match the backup's date, but the user may elect to use the backup anyway.

TIME=time1 time1 is the backup's time in hh:mm:ss format. For SAVE, this time is assigned to the backup; if TIME is not specified, the operator will be asked for it. For LIST and RESTORE, an error will be displayed if TIME is not specified or time1 does not match the backup's time, but the user may elect to use the backup anyway.

SVL=list2 This Save Volume List parameter is required for and used only if the function is SAVE. list2 specifies the volume(s) whose files are to be copied to the backup during SAVE. If list2 has more than one sub-parameter, list2 must be enclosed in parenthesis. list2 consists of one or more sub-parameters, separated by comas, of the type:

> vn1 specifies the number of an active slot whose data volume files, as restricted by INCLUDE or EXCLUDE, are to be copied to the backup. vn1 may have integer values 0 to xxx, where xxx is one less than the SYSTEM AL parameter. vn1 must not equal a slot number specified in the BSN parameter.

RVL=list3 This Restore Volume List parameter is required for and used only if the function is RESTORE. This parameter specifies (1) volume numbers whose files in the backup, as restricted by INCLUDE or EXCLUDE, are to be restored and (2) optionally, the data volume to receive the files of another volume.

If list3 has more than one sub-parameter, list3 must be enclosed in parenthesis. list3 consists of one or more sub-parameters, separated by commas, of the type:

vr2=vn1 The files contained in the backup for volume vn1, as restricted by INCLUDE or EXCLUDE, are copied to volume vn2. Volume numbers in the INCLUDE or EXCLUDE list refer to vn1, not vn2. If vn2 and vn1 are the same volume number, the vn2=vn1 sub-parameter may be written as Just vn1. vn2 may have integer values 0 to xxx where xxx is one less than the SYSTEM; AL parameter. vn2 must not equal a slot number specified in the BSN parameter.

SLOW This option can only be used with NEWDOS/80 and specifies that the HDBACKUP program is NOT to use its faster diskette I/O when reading/writing the backup (not the data volumes) diskettes. SLOW is implied by NND. Normally, NEWDOS/g0 Version 2.5 uses a faster mode of backup diskette I/O in the hope of increasing the speed of SAVE and RESTORE by 20-40. SLOW should be specified only if the fast I/O appears to actually run slower than normal diskette I/O. You can study this by timing the time to read or write a backup diskette, preferably a volume other than backup volume 1.

SLIP During HDBACKUP processing when an error is encountered and the operator would normally have a 'SKIP' option allowing processing to continue, if the SKIP command parameter was specified, the SKIP option will automatically be assumed. Normally, this option will not be specified; however, there are times when a SAVE or RESTORE must accomplish what it can despite errors. For example, if part of a hard disk has gone bad and the disk is to be sent to the repair shop where it may or may not retain its current data, it may be important to assure that whatever data can be retrieved, is retrieved with the problem of restructuring bad files addressed later.

RENEW This option is used only with RESTORE. During HDBACKUP initialization after the files to be restored have been determined, a KILL is issued to the destination volume for each file that is to be restored. If the file did not exist on the destination volume, the KILL does nothing. Normal RESTORE initialization will then recreate the files on the destination volumes. The purpose of RENEW is to reallocate file space in, hopefully, less fragmented units (which can increase the efficiency of programs using these files); RENEW should only be used when all, or almost all, files of a data volume are being restored.

MAXERRS=ec1 ec1 is the number of errors the backup is to provide for in its error table. The default value is 640 with 6400 the maximum ec1 value allowed. MAXERRS is used only in the SAVE function.

TEST This option allows initialization processing to occur, including backup control information writes. When the initialization is done, HDBACKUP terminates with 'TEST COMPLETED' error. TEST allows the user to test the command parameters, including the INCLUDE or EXCLUDE lists.

*** Warning, TEST with SAVE writes control information to the backup's first diskette; this is necessary for a good test.

INCLUDE and EXCLUDE INCLUDE and EXCLUDE are mutually exclusive keywords. Each must terminate the current command input line. Subsequent command lines until but not including the *END command line compose a file list with each

line specifying either a volume number preceded by a colon (i.e., :3) or the filespec, without passwords, of a file to be INCLUDEd or EXCLUDEd. The number of volume numbers or filespecs allowed in a file list is limited by computer main memory constraints but is over 1500.

If the command line consist solely of a volume number, then all files for that volume are INCLUDED or EXCLUDED.

All volume numbers in the INCLUDE or EXCLUDE list must refer to a vnl volume number specified in the appropriate SVL or RVL parameter.

INCLUDE and EXCLUDE are optional keywords (except for SAVE with NND). If neither is specified, HDBACKUP will assume inclusion of all the files for the vn1 volumes specified in the SVL or RVL parameter.

INCLUDE Only the files specified in the file list are included in the SAVE or RESTORE. If a file in the list does not exist, an error comment will be listed, and the operator given the option of bypassing the file.

EXCLUDE The files specified in the file list are excluded from the SAVE or RESTORE; all other files of the vn1 volumes specified in the SVL or RVL parameter are copied. If a file in the list does not exist on the specified data volume (SAVE) or the backup (RESTORE), no indication is given to the operator.

*END This required parameter ends the HDBACKUP command specification.

INTERNAL STRUCTURE OF THE BACKUP:

For those users interested, this section will show the structure of a backup. Some users may find this description helpful in repairing a backup using SUPERZAP.

Each volume (diskette) of a backup has a volume header sector as the diskette's first sector. The header sector for volume 1 is the most important and is used by RESTORE and LIST to access backup control information. The headers for the other volumes contain roughly the same information, and are used during RESTORE to verify that you have mounted the correct volume and by SAVE to verify that you don't mount as a new volume for this SAVE a volume that has already been used in the SAVE. The user must remember that a file's sectors can span many backup volumes and must allow for the volume header records when computing where a particular sector of a particular file is within the backup. The contents of the backup volume header sector are:

1. 48 byte backup title.
2. 8 byte backup data in mm/dd/yy format.
3. 8 byte backup time in hh:mm:ss format.
4. 2 byte count of sectors for the table of contents.
5. 1 byte count of sectors for the error table.
6. 2 byte count of files in the table of contents.
7. 2 byte count of number errors allowed during SAVE.
8. 2 byte count of sectors per backup volume.

9. 2 byte value = this diskette's volume number.
******** valid only for volume 1:
10. 1 byte of control bits:
bit 7 = 1, the SAVE is complete.
bits 6 - 0, undefined and reserved, must be 0.
11. 2 byte count of errors in error table.
12. 2 byte count of volumes for this backup.
13. 3 byte backup total sector count.
14. remainder of sector's bytes are 00H.

On backup volume 1, the table of contents sectors immediately follow the volume header sector. Each sector contains eight 32 byte file entries of the form:

1. 3 byte file name, padded on right with blanks.
2. 3 byte file Filename extension, padded on right with blanks.
3. 1 byte data volume number.
4. 3 byte file EOF in FBA format.
5. 1 byte logical record length (0=256). Not necessarily valid if NND specified during SAVE.
6. 3 byte relative sector number within the backup of the file's header sector.
7. 2 byte relative entry number of this entry within the table of contents.
8. 1 byte control bits:
bit 7 = 1, this table of contents entry is used.
bit 6 = 1, this file is active.
Bit 7-6 = 10, file has been deleted from the backup. Actually some of it may still be there, but LIST and RESTORE ignore it.
bits 5 - 0, undefined and reserved, must be 0.

9. The remainder of the 32 byte entry are bytes 00H.

The error table sectors immediate follow the table of contents. Each sector has 64 entries of the form:

1. 1 byte containing the DOS error code plus 40H. If the byte is 00H, the error has either been corrected by the user or he/she wants it ignored.

2. 3 byte relative sector number within the backup of the file sector in error. If the error is corrected or to be ignored, this value must be set to 0.

The remainder of the backup is file data with each file's sectors preceded by a file header record. If a file's EOF is zero, then only the file's header record will appear. The user must remember that where a file's sectors flow onto the next backup volume, the first sector on that volume with be the volume's header sector, not a file sector. The format of a file header is:

1. The first 22 bytes are an exact copy of the first 22 bytes of the table of contents entry for this file, but with none of the changes to the-t entry after it was initially created. During RESTORE, these 22 bytes of the file header must match the 22 bytes from the table of contents.

2. Each of the remaining bytes of the file header sector contains the ones complement of its relative location in the sector. This makes it easier to recognize a file header should it be necessary to search for it.

HDBACKUP EXAMPLES:

1. HDBACKUP
   SAVE,BSN=(4,5),SVL=(0,1,2,3),*END

This is a copy of user files from hard disk data volumes 0, 1, 2 and 3 to a backup whose diskettes will be mounted on the floppy drives associated with slots 4 and 5 (assumed defined for floppy drives 0 and 1 respectively), with backup volume 1 remaining on slot 4's drive and the other backup volumes requested on slot 5's drive as needed. The user must have on hand enough pre-formatted diskettes for the needs of the backup. Since slots 4 and 5 are the access to floppy drives 0 and 1, we know that the hard disk system is being run from the hard disk.

2. HDBACKUP
   RESTORE,BSN=(4,5),RVL=(0,1,2,6=3),*END

This is a copy of user files from a backup to data volumes 0, 1, 2 and 6. The backup diskette volumes will be mounted on the drives for slots 4 and 5 as described in the above example. All files in the backup are copied, but the files that originally came from volume 3 are actually written to volume 6 instead.

3. HDBACKUP
   SAVE,BSN=1,SVL=(2,3,4,5,6,7)
   EXCLUDE
   XXX/DAT:4 YYY/DAT:6,*END

This is a copy of all user files from hard disk data volumes 2, 3, 4, 5, 6 and 7 to backup diskettes which will all be mounted as needed on the floppy drive 1. File XXX/DAT of volume 4 and file YYY/DAT of volume 6 will not be copied to the backup. Since BSN=1 was used for the backup floppy drive, we know the system is being run from a system diskette in floppy drive 0.

4. HDBACKUP
   LIST,BSN=1,PRINT,*END

The contents of the backup's table of contents is listed on both the display and the printer.

5. HDBACKUP
   SAVE,BSN=(4,5),SVL=(1,2),INCLUDE
   ACCTPYBL/DAT:1
   ACCTRVBL/DAT:1
   PAYROLL/DAT:2
   INVENTRY/DAT:2
   *END

A backup is made consisting only of the 4 files specified in the INCLUDE list. In this installation, the burden of making backups of valued files has been placed on the individual users, in this case, accounting.

```
6.    HDBACKUP
      SAVE,NND=(TEMPFILE:0,N,6,3,720),BSN=(4,5)
      SVL=(0,1,2,3),INCLUDE
      FILE001:0
      FILE002:0
      FILE003:1
      and so on through
      FILE999:4
      *END
```

In this example, the HDBACKUP/CMD program has been previously moved to the LDOS Hard Disk Operating System (in the manner described at the NND discussion). The HDBACKUP runs under LDOS 5.1.3 and dumps the specified files from volumes 0, 1, 2 and 3 to 4 backup whose diskettes have all been preformatted as single sided, double density, 5 inch 40 track (with spg = 6, gpl = 3 and spv = 720). After the hard disk has been reinitialized for NEWDOS/80 Version 2.5, the HDBACKUP program under NEWDOS/80 (without the NED parameter) can be used to RESTORE the files from the backup to the hard disk.

When NND is specified for a SAVE, such as above, an INCLUDE list must be used to inform the HDBACKUP program of which files to copy to the backup, as the HDBACKUP program does not read the non-NEWDOS/80 directories.

This example could be used for single sided, single density 35 track backup diskettes under LDOS 5.1.3 on the TRS-80 Model I by using replacing the NED parameter with NND=(TEMPFILE:0,N,5,2,350). If 40 track diskettes are used, replace 350 with 400.

The TEMPFILE:0 filespec used in this example is just our choice of a filespec for this example; you are free to use any filespec you wish so long as it conforms to the specifications given for the END parameter.

Index