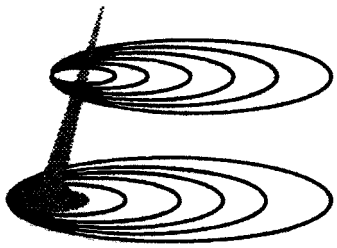
L B A S I C

USER'S GUIDE



by Galactic Software Ltd.

L B A S I C T A B L E O F C O N T E N T S

First Edition LBASIC Model I/III

I N T R O D U C T I O N T O L B A S I C

Contained on your LDOS Master Diskette is a program named LBASIC/CMD (LBASIC). As was noted in the GETTING STARTED portion of the manual, your computer contains two different types of memory, ROM (Read Only Memory) and RAM (Random Access Memory). Your computer, as received from your dealer, does contain a ROM BASIC. This ROM Basic does allow you some capabilities of programming in the Basic language. However, ROM Basic does not allow you to interface with your disk drives when programming, and hence does not fully utilize your TRS-80 disk system. For this reason, LBASIC has been included with your LDOS system. LBASIC is an extension of ROM Basic and resides in RAM. LBASIC utilizes commands found in ROM Basic, and adds commands to ROM Basic which will allow you to interface your Basic programs with the disk operating system. Because of this, programs and data files created under LBASIC may be stored on your disk drives. In addition, many LDOS functions may be performed when programming in LBASIC, without having to return to the "LDOS Ready" level.

This manual will detail all enhancements to ROM Basic which are contained in LBASIC. Commands which are inherent in ROM Basic will not be detailed in this manual. Refer to your Radio Shack owner's manual (Model I Level II Basic Manual or Model III Operation and Basic Language Reference Manual) for a complete description of ROM Basic commands.

One final point concerning the LBASIC manual. It is written as a reference manual only. All commands will be explained in terms of the function which they serve. In no way will this manual serve as a tutorial on implementation of these commands. There are many such books currently on the market that deal with using a "Microsoft compatible" disk Basic for generalized and specific applications. If you require tutorial aids for implementing LBASIC, contact your computer dealer for a list of such material.

E N T E R I N G L B A S I C

This is the syntax to be observed when entering LBASIC.

```
LBASIC (parm,parm,. . . ,parm) command

LBASIC *  used to re-enter LBASIC with the program and
          the variables intact.

The allowable parameters are as follows:

BLK=   parameter that specifies Blocked file mode,
       either ON or OFF. ON is the default.

FILES= parameter that specifies the maximum number of
       files LBASIC will be able to access (1 to 15).
       If not specified, 3 is assumed.

MEM=   parameter to set the highest memory address
       to be used by LBASIC. All memory above this
       address will be "protected". If not specified,
       all memory up to HIGH$ will be available.
       This parameter may be specified as either a
       decimal (MEM=nnnnn) or hexadecimal (MEM=X'xxxx')
       value.

EXT=   parameter used as a switch to turn on or off
       the default file extension "IBAS" used with
       the LBASIC commands LOAD, RUN, MERGE and SAVE.
       Either ON or OFF may be specified. If not
       specified, ON is assumed. See LBASIC - GENERAL
       INFORMATION for a detailed description.

HIGH   Model III parameter that sets the cassette baud
or     rate, either HIGH or LOW (HIGH=1500 and LOW=500).
LOW    The default is HIGH. ** If HIGH is used, the
       HITAPE command must be issued prior to entering
       LBASIC. **

command - This may be any valid LBASIC command
          which will execute immediately upon entering
          LBASIC, such as RUN"MYPROG/BAS", AUTO100, etc.

abbr: BLK=B, FILES=F, MEM=M, ON=Y, OFF=N, HIGH=H, LOW=L
      EXT=E
```

Any or all of the parameters may be specified when entering LBASIC. If no parameters are specified, the default values listed in the above syntax block will be assumed.

The "command" specification is also optional. If not specified, you will enter into LBASIC, and the following lines will appear on the screen:

```
LBASIC - Version 5.x.x - mm/dd/yy
(C) 19xx by Logical Systems Incorporated

Ready
```

The "Ready" prompt will indicate that LBASIC is ready to accept any command that you wish to give it.

If you have rebooted the system, or have performed an exit from LBASIC to the operating system (usually done by issuing a CMD"S" command), and wish to re-enter LBASIC, you may enter the command:

```
LBASIC *
```

at the LDOS Ready level. Doing so will cause LBASIC to be re-entered, and any program that was resident in memory prior to performing the exit to the LDOS Ready level will remain intact. Be aware of the fact that if LBASIC * is used to re-enter LBASIC from the LDOS Ready level, any commands which affect HIGH$, or any commands that utilize memory (such as BACKUP and COPY) may cause your LBASIC program to be overwritten with other information. For this reason, LBASIC * should only be used as a last resort. You may perform certain LDOS Library commands directly from LBASIC (using the CMD command). If a function cannot be performed from LBASIC using the CMD command, it is not advised to re-enter Basic using LBASIC * if you have exited back to LDOS to perform the command, as the integrity of your program will be suspect.

Example

One of the following commands may be given if you wish to enter LBASIC in the blocked file mode with 2 files open, having memory protected up to location 6l440 (X'F000'). Also, you wish to have the program MYPROG/BAS loaded upon entering LBASIC.

```
LBASIC (FILES=2,MEM=61440,BLK=ON,EXT=ON) LOAD"MYPROG/BAS"
LBASIC (F=2,M=X'F000') LOAD"MYPROG"
```

Issuing either of the above two commands will produce the same results. The second command above uses the abbreviations F and M for FILES and MEM, and also utilizes the default "ON" for the BLK and EXT parameters. Note that the extension for the program MYPROG/BAS need not be specified if EXT is ON. Also, realize that for either of the above commands, if HIGH$ is lower than 61440 (X'F000'), an "Out of Memory" error will occur, and you will be returned to the LDOS Ready prompt without entering LBASIC.

L B A S I C - G E N E R A L I N F O R M A T I O N

ABBREVIATED COMMANDS

Each of the following LBASIC commands may now be represented as single characters. When using a single character command, the effect will be identical to using the entire word. This abbreviated form is only acceptable when typed on a command line, not in a program line or JCL file.

A represents the command AUTO.

D represents the command DELETE.

E represents the command EDIT.

L represents the command LIST.

The following commands are implemented by pressing the indicated key as the first character in the command line. No carriage return is necessary; the indicated action will take place immediately. Note that any of the following single key commands must be the first character entered after the "Ready prompt" appears.

. (period) This will perform the same function as "LIST.<ENTER>", which will instruct LBASIC to list the currently active line.

,(comma) This will perform the same function as "EDIT.<ENTER>", which will instruct LBASIC to enter the "edit mode" for the currently active line.

<UP ARROW> This will cause LBASIC to display the next lower numbered line in the program.

<DOWN ARROW> This will cause LBASIC to display the next higher numbered line in the program.

<LEFT ARROW> This will cause LBASIC to display the first line of the program.

<RIGHT ARROW> This will cause LBASIC to display the last line of the program.

DEFAULT EXTENSIONS

LBASIC allows you to utilize the default extension of /BAS when issuing the LOAD, RUN, MERGE and SAVE commands. If the EXT parameter is set to ON (or not specified) when entering LBASIC, all filespecs used with the above commands that do not have extensions will be assigned the extension /BAS. If EXT is on and an extension is specified, the extension used in the filespec will override the default extension.

If EXT is ON and the file in question has no extension, it must be specified as "filename/" (i.e. the "/" will override the extension /BAS with no extension). If the EXT parameter is turned OFF when entering LBASIC, all file extensions will have to be specified.

FILE BLOCKING

LBASIC provides a Blocked file mode (which has often been misnamed Variable Length Files). This mode allows files with Logical Record Lengths (LRL) of less than 256 bytes to be created and accessed. Any record length from 1 to 256 bytes will be allowed, even if the record size is not evenly divisible into 256.

All blocking and de-blocking across "sector boundaries" will be performed by LDOS. In this way, user records can span across sectors to provide maximum disk storage capacity. If the LRL is not specified when OPENing a Random file, 256 will be assumed. Note that an LRL of 0 will signify a 256 byte LRL. Enhancements have also been made to the allowable methods of OPENing both Random and Sequential type files (See OPEN).

If the Blocked file mode is ON, each file declared when entering LBASIC will take 544 bytes of memory. If the Blocked mode is OFF, each file will take 288 bytes.

LBASIC OVERLAYS

Three overlays are present on a Master LDOS diskette. They are:

LBASIC/OV1 - This overlay contains the renumbering program associated with the LBASIC CMD"N" function. It may be killed if no renumbering will be done.

LBASIC/OV2 - This overlay contains the cross reference program associated with the LBASIC CMD"X" function. It may be killed if no cross referencing will be done.

LBASIC/OV3 - This overlay contains the error handling routine used with LBASIC, along with the sort routine used for the CMD"O" function. It MUST be present when using LBASIC.

PROGRAM PROTECTION

LBASIC programs may be protected with an "Execute only" password. This means that the program may be RUN, but not LOADed, LISTed, LLISTed, or otherwise examined. Any attempt to break the program execution and examine the program will cause the program to be erased from memory, and the message "Protection has cleared memory" will be displayed. The DEBUGger will also be disabled during program execution.

SINGLE STEPPING AN LBASIC PROGRAM

This new feature allows the LBASIC programmer to step through each program statement singly, with a "HOLD" after each step. To invoke this feature simply do a normal pause (<SHIFT @>), which will cause LBASIC to go into a wait state. While continuing to hold down the <SHIFT @> press the <SPACE BAR>) and the next LBASIC statement will execute. After execution of that statement the computer will immediately go into its wait state again. Holding down the <SPACE BAR> will execute statements at the normal keyboard repeat rate. If you press any key without holding down the <SHIFT @>, normal program execution will resume. Note that this feature also functions when listing a program.

TAPE ACCESS

Accessing information off of tape will vary depending on the type of machine you are using. You should follow these guidelines when storing and retrieving information from tape.

Model I users need to disable the interrupts prior to performing tape I/O, and must re-establish them after the input/output has been performed. To disable the interrupts, use the LBASIC command - CMD"T" -. To enable the interrupts, use the command - CMD"R" -. See the LBASIC Commands Section for more information on these two commands.

Model III users need to do one of several things, depending on the type of tape involved. If you are dealing with a 500 baud tape, you will need to specify the LOW parameter when entering LBASIC (Remember, if HIGH or LOW is not specified, the default will be HIGH). If you are dealing with a 1500 baud tape, you will need to establish the HITAPE utility. For more information on HITAPE, refer to the Utilities section of the LDOS manual.

L B A S I C C O M M A N D S

This section of the manual will detail commands found in LBASIC which are not included in ROM Basic. These commands will be listed in alphabetical order. For the novice, this type of grouping might be a bit confusing in terms of when and how these commands will be used. However, for the person who is somewhat versed in using a disk oriented Basic, this will be a very convenient way of locating information dealing with any LBASIC command.

&H - Hexadecimal Representation of a number

To represent a number in its hexadecimal format, you may use the characters - &H - as a prefix to the number. This may be useful when you wish to define an address for a user machine language subroutine (see DEFUSR).

One to four hexadecimal digits may follow the &H prefix. Hexadecimal digits consist of the numeric digits 0-9, as well as the alphabetic letters A-F. The number represented using the &H prefix will always be taken as two's complement notation.

Examples

```
A=&H11    (A would be set equal to the decimal number 17).
A=&HA9    (A would be set equal to the decimal number 169).
A=&HF000  (A would be set equal to the decimal number -4096).
```

&O - Octal Representation of a number

To represent a number in its octal format, you may use the characters - &O - (or just - & -) as a prefix to the number.

One to six octal digits may follow the &O prefix. Octal digits consist of the numeric digits 0-7. The number represented using the &O prefix will always be taken as two's complement notation. The largest octal number which may be represented is &O177777.

Examples

```
A=&O11    (A would be set equal to the decimal number 9).
A=&170000 (A would be set equal to the decimal number -4096).
```

CLOSE - Close any or all open disk files

The CLOSE command is used in conjunction with the OPEN command. After a file has been opened, it is capable of being read from and/or written to. To disable this read/write capability of a disk file, a CLOSE of the file must be done. In addition, CLOSE will update the Mod flag, Mod date and end of file marker in the directory record of that file (provided the file has been written to). See OPEN for more information on file access.

The syntax for the CLOSE command may be in one of the following formats:

```
CLOSE
CLOSE #,...,#
```

The CLOSE command issued by itself will close all open files. The CLOSE #,...,# command will close only those files that have been opened with the specified buffer number (where # represents the buffer number used to define a particular file in an OPEN statement).

If you issue any command which will perform a CLEAR (such as EDIT, CLEAR or RUN), a global CLOSE will automatically be performed for you. However, if you issue a CMD"S", CMD"A", or CMD"I" command, closing of any open files will not occur. For this reason, you should always make sure files have been closed prior to exiting back to the LDOS Ready prompt.

CMD - Perform an LDOS or Special Command

The CMD command allows you to perform certain LDOS library and utility commands without having to leave LBASIC. In addition, there are 13 distinct parameters that may be used in conjunction with the CMD command which will allow you to perform various different functions. The syntax used for the CMD command is as follows:

```
CMD"dos command"
CMD"x" (Where 'x' is the letter assigned to the special command).
```

We will first describe how to use the CMD command to issue an LDOS command, after which we will explain the use of the 13 distinct parameters with the CMD command.

CMD"dos command"

LDOS Library commands and Utilities that do not affect HIGH$ may be executed from LBASIC by use of the CMD"dos command". The following examples should illustrate implementation of this feature:

```
CMD"DIR :0"          - Will display a Directory of the disk in drive 0.
CMD"DEVICE"          - Will display the device table.
CMD"LIST DAT1/SCR"   - Will list the file DAT1/SCR.
CMD"BACKUP :0 :1"    - Will perform the designated Backup.
```

After the desired LDOS function has been completed, control will be returned to LBASIC. This type of CMD command will function whether it is called from LBASIC's command line or from within an LBASIC program. If performed from within an LBASIC program and an error occurs, or the CMD command is aborted with the break key prior to being completed, the appropriate error message will be displayed, or the message "System Command Aborted" will appear, and execution of the Basic program in question will be terminated. The command may also be contained within a string variable, such as the following format:

```
A$="DIR :0":CMD A$
```

Approximately 4K of free memory must be available for these types of CMD commands, or an "Out of Memory" error will occur.

CMD"A"

This command will perform an abnormal return to LDOS. Any active DO command will be cancelled.

CMD"B","switch"

This command will enable or disable the <BREAK> key, with "switch" being either ON or OFF. A string constant or string expression may be used to represent the "switch".

CMD"D"

Turns on and enters the system Debugger.

CMD"D","switch"

This command is similar to the CMD"D" command, with the following exceptions. The switch ON will turn on the system Debugger, but will remain in LBASIC. Pressing the <BREAK> key (or <CLEAR> <SHIFT> <D> keys if Minidos is active) will cause you to enter the Debugger. The switch OFF will turn off the Debugger.

CMD"E"

This command will return the last LDOS error message encountered. If no error has been encountered, the message "No Error" will appear. CMD"E" may be useful when you wish to pinpoint the exact nature of an error. LBASIC's error dictionary is not as extensive as that found in LDOS, hence various LDOS errors can produce the same LBASIC error message. Performing a CMD"E" will give you the exact error seen by LDOS. This may be of use when you get the LBASIC error message "Disk full or write protected" or "Disk I/O error".

CMD"I","dos command"

This command functions much the same as the CMD"dos command", with the exception that control will return to LDOS after the "dos command" has been executed. Dos command can be represented as a string constant or a string expression. If represented as a string constant, it must be contained within quotes.

CMD"L","filespec"

This command will load a Load Module Format file (a machine language program) into memory, much the same as the LOAD Library command does. Filespec may be represented as a string constant or a string expression. If represented as a string constant, it must be contained within quotes.

CMD"N"

This command provides you with a program line renumbering function. For the specific parameters involved with this command, please refer to page 47 at the end of the LBASIC section.

CMD"O",number of elements to sort,first element of array to sort

This command will allow you to sort a single dimensioned string array. The sort will start at the element specified, and will sort the number of elements specified. The number of elements to be sorted must not force the sort past the end of the array. In order to utilize the CMD"O" function, the module LBASIC/OV3 must be present on a disk in the system.

Example

```
CMD"O",15,A$(10)
X=15:Y=10:CMD"O",X,A$(Y)
```

Issuing either of the above commands will cause a sort to be performed on the A$ array. After the sort has been finished, elements 10-24 will be sorted in alphabetical order.

CMD"P",variable

This command will return the printer status in the variable specified. The variable may be any type, including a string. The value will have the bottom 4 bits stripped before being passed back to LBASIC.

CMD"R"

Model I - This command enables the interrupts. It should be performed after a CMD"T" command has been issued. For more information see the CMD"T" command.

Model III - This command will turn on the clock display.

CMD"S"

This command is the normal way to return to LDOS Ready from LBASIC.

CMD"T"

Model I - This command will disable the interrupts. It must be issued prior to performing tape I/O. After the tape I/O has been completed, the interrupts must be enabled with the CMD"R" command.

Model III - This command will turn off the clock display.

CMD"X"

This command provides you with a program cross reference function. For the specific parameters involved with this command, please refer to page 48 at the end of the LBASIC section.

CVD - Convert to Double Precision

This command is used to convert an 8 byte string into a double precision number. The 8 byte string should be a representation of a double precision number stored in compressed format. This command is used primarily to uncompress double precision values which have been retrieved from a disk file (in essence, it performs the opposite function of the MKD$ command). For more information on storing a double precision number in compressed format in a disk file, refer to the MKD$ command.

Example

```
A#=CVD(A$)
```

In the above example, assume that A$ is an 8 byte string which represents a compressed double precision number. After the above command is performed, A# will be set equal to the uncompressed number that A$ represents.

Realize that you are not limited in using CVD to assign a value to a variable. The value generated by a CVD command may be used directly (e.g. PRINT CVD(A$), or IF CVD(A$)<100000 THEN GOTO 1000).

CVI - Convert to Integer

The CVI command functions identically to the CVD command with the following exceptions. The CVI command will convert a two byte string into an integer. This two byte string should be a representation of an integer stored in compressed format. CVI performs the opposite function of the MKI$ command. The value returned from the CVI function will be an integer within the range of -32768 to +32767 inclusive. For more information, refer to the MKI$ command.

Example

```
A%=CVI(A$)
```

In the above example, assume that A$ is a 2 byte string which represents a compressed integer. After the above command is performed, AZ will be set equal to the uncompressed number that A$ represents.

CVS - Convert to Single Precision

The CVS command functions identically to the CVD command with the following exceptions. The CVS command will convert a four byte string into a single precision number. This four byte string should be a representation of a single precision number stored in compressed format. CVS performs the opposite function of the MKS$ command. For more information, refer to the MKS$ command.

Example

```
A!=CVS(A$)
```

In the above example, assume that A$ is a 4 byte string which represents a compressed single precision number. After the above command is performed, A! will be set equal to the uncompressed number that A$ represents.

DEF FN - Define Function

There are many intrinsic functions provided for you in ROM Basic and LBASIC (i.e. VAL, STR$, SIN, etc.). The DEF FN command allows you to define your own functions. This may be of use when performing lengthy calculations at different points in your program when you do not use the same variable names to perform these calculations.

The syntax for the DEF FN command is as follows:

```
DEF FNfunction name(parm,... ,parm)=expression
```

The "function name" is the name that you will assign to the function, and has the same restrictions as those imposed on a variable name. The function name must be of the same type as the value to be returned from the function.

The "(parm,.. .,parm)" is a list of variables to be passed to the function. The variable names used are local to the function, and act as dummy variables. They will have no effect on other variables in the program which have the same name. However, they must be of the same variable type as the variable represents in the function (i.e. string, integer, single precision, double precision). Also, if more than one variable is to be passed to the function, they must be passed in the same order as that defined in "(parm,.. .,parm)" (see example below).

The "expression represents how the variables passed to the function are to be worked on.

The example below will show how to define and invoke your own functions.

Example

This example will show how to create a function which will build a filespec. This function will be passed three variables; the filename, the file extension, and the drive specification. It will return a filespec in the form - filename/ext:d -. A DEF FN statement to create such a function might take on the following format:

```
DEF FNFS$(X$,Y$,Z%)=X$+"/"+Y$+":"+MID$(STR$(ZZ),2,1)
```

The function name is FS$, and is of string type, since a string value will be returned from the function.

Three values will be passed to the function. The first two values passed will be strings, while the third value will be an integer.

The function that will be performed is as follows. The first string passed to the function will have a '/' added onto the end of it, after which the extension, a ':', and the drivespec will be added to the string, respectively.

The following example will illustrate how to invoke the function, as well as changes that will occur to the variables involved.

```
X$="HELLO":F$="MYPROG":E$="BAS":G%=2
F1$=FNFS$(F$ ,E$ ,G%)
F2$=FNFS$(E$ ,F$ ,G%)
```

After execution of the above three lines, the following variables will be assigned the following values:

```
X$="HELLO"   F1$="MYPROG/BAS:2"   F2$="BAS/MYPROG:2"
F$="MYPROG"  E$="BAS"             G%=2
```

Note that the value of X$ does not change from the calling of this function. Also note the difference between F1$ and F2$. The order in which variables appear when invoking the function determines the value that will be returned from the function.

As a final note on DEF FN, the value returned from the function can be used directly, and does not have to be stored in a variable (e.g. PRINT FNFS$(F$,E$,G%)).

DEFUSR - Define the entry point to a user machine language subroutine

This command is used to define the starting address (entry point) of a user created machine language subroutine. A DEFUSR statement must be done prior to utilizing the machine language subroutine via the USR command. The syntax for the DEFUSR statement is:

```
DEFUSRn=xxxx
```

where n is a numeric constant (0-9) which is used to identify the machine language subroutine, and xxxx is the address which represents the entry point into the machine language subroutine.

The number assigned to the subroutine (n) must be the same as the number used to reference the subroutine with the USR statement.

The entry address to the subroutine may be a constant (i.e. a hexadecimal or decimal number), or it may be a numeric expression. Note that if the starting address is specified as a decimal number, and this address is greater than 32767, it must be specified as the address minus 65536.

Example

Suppose you have a machine language subroutine that has a starting address of &HF000 (61440), and you wish to reference this routine as machine language subroutine number 2. To define this subroutine, one of the following commands may be given:

```
DEFUSR2=&HF000
DEFUSR2=(6l440-65536)
DEFUSR2=(-4096)
```

EOF - Determine if "End of File" has been encountered

This command is used to determine if the end of file has been reached when inputting from an open disk file. It is used primarily in conjunction with sequential files, but can also be used with random files. EOF is a function, and will return a 0 (false) if the end of file has not been reached, or a -1 (true) if the end of file has been reached. It can be used with the IF statement, and will determine the outcome of the IF, as it will return either a logical true or a logical false.

The syntax for the EOF command is:

```
EOF(#)
```

where # is the buffer number used to open the file.

Example

Assume that you have created a sequential file named MYDATA, and wish to access the information in it, but you do not know the amount of data in the file. The following program lines will illustrate how to use EOF to determine when the last piece of data has been accessed.

```
1000 OPEN"I",1,"MYDATA"
1100 IF EOF(1) THEN PRINT"ALL DATA HAS BEEN ACCESSED":END
xxxx
xxxx  'lines used to input and process data
xxxx
1500 GOTO 1100
```

Notice that the EOF command is used prior to inputting any information. This will ensure that you will not try to input from an empty file, or after the end of file has been encountered. Either case would result in an "INPUT PAST END" error.

FIELD - Partition the buffer associated with a random file

The field statement is used to partition the buffer associated with an open random file. This partitioning allows you to break a record up into fields, where each field denotes a particular piece of information in that record. The fielding of a record determines the length of each piece of information in the record, and where this information will physically reside in the record.

The syntax used in the FIELD statement is:

```
FIELD#,aaa AS variable1,bbb AS variable2,...,nnn AS variableN
```

is the buffer number used in the associated OPEN statement. It may be a constant, or a numeric expression. The value of this number must be in the range of 1 to the total number of files allocated when entering LBASIC, inclusive, and must correspond to an open file.

aaa, bbb and nnn are numeric constants or expressions denoting the maximum length (in bytes) of the fielded variable. The value of these numeric constants or expressions must be in the range of 0 to 255) inclusive, as the length of a string cannot exceed 255 bytes. If denoted as numeric expressions, these values must be enclosed within parentheses.

variable1, variable2, and variableN are intermediate variables used to retrieve information from and pass information to the buffer. They must be string variables.

When information passes between the computer and the disk, a buffer is used as a temporary storage place for this information. Information is placed in this buffer with the LSET and RSET commands. Where this information is physically placed in the buffer is determined by the FIELD statement.

The field statement will allow you to break up the buffer into various "slots", assigning a variable name to each of these slots. When information is placed into or accessed from the buffer, it is done so by using the variable name which was assigned to each slot in the FIELD statement. The length of each of these slots is also determined by the FIELD statement. The total number of bytes to be fielded in a record must be less than or equal to the number of bytes that a record will contain.

The following example will illustrate how the FIELD statement is used.

Suppose that you wish to deal with a file that will contain records whose lengths will be 100 bytes. In each record, there will be 4 pieces of information (fields). Field 1 will be 20 characters long, and will represent the name of a person. Field 2 will be 10 characters long, and will represent an account number. Field 3 will be 30 characters long, and will represent address information. Field 4 will be 40 characters long, and will represent an account description. The following OPEN and FIELD statements will allow you to open such a file and field the buffer accordingly.

```
 OPEN"R",1,"MYFILE/DAT",100
FIELD1,20 AS NA$,10 AS AC$,30 AS AD$,40 AS DE$
```

Using the above lines in a program will produce the following results. A file by the name of MYFILE/DAT will be opened, and records in this file will have a length of 100 bytes. A buffer for this file will be set up in memory. The first 20 bytes of this buffer will represent name, and will be referenced by the variable NA$. The next 10 bytes of this buffer will represent the account number, and will be referenced by the variable AC$. The next 30 bytes will represent the address, and will be referenced by the variable AD$. The last 40 bytes will represent the description, and will be referenced by the variable DE$.

More than one field statement may correspond to the same buffer. Variable names used in a FIELD statement may only be used to pass information to or retrieve information from the buffer. Using fielded variables for any other purpose will break the link between the variable and the buffer, and the variable will not be connected to the buffer until the original FIELD statement is re-executed. For more information on passing information to and retrieving information from the disk, see OPEN, GET, PUT, LSET, RSET, MKI$, MKS$, MKD$, CVI, CVS and CVD.

GET - Retrieve a record from a random file

The GET command is used to retrieve information from a random file. The information that is retrieved is stored in the buffer that was used to open the file. The syntax for the GET command is:

```
GET#,r
GET#
```

where # is the buffer number used to open the file, and r is the record number you wish to retrieve. Both # and r may be numeric constants or numeric expressions. If the record number (r) is not specified, the computer will increment the current record number by one, after which it will perform a GET of the current record number. If no current record number has been established, the computer will perform a GET of record number one, and the current record number will be set equal to one.

Example

Suppose you have opened a file and fielded the corresponding buffer. The buffer number used is 3. One of the following GET commands may be used to retrieve the 17th record of the file.

```
GET3,17
N%=2:N1%=16:GETN%+1,N1%+1
```

After executing one of the above statements, record 17 of the file will be contained in the designated buffer, and information dealing with this record may now be accessed by referencing the variables used in the FIELD statement.

INPUT# - Input information from a sequential file.

The INPUT# statement is used to retrieve information from a sequential file. The syntax used with the INPUT# command is:

```
INPUT#n,variable1,.. ,variableN
```

where n is the buffer number used to open the file, and variable1,...,variableN are the variables used to store the information retrieved.

Sequential files are created by specifying an OPEN"O"/OPEN"E" command, followed by one or more PRINT# commands. After a sequential file has been created, the information in it may be accessed by using the OPEN"I" and

INPUT# commands. The INPUT# command can be thought of as performing a function similar to the INPUT command, the exception being that the information is not entered from the keyboard. Rather, it is retrieved from the disk. Like the INPUT command, INPUT# can only be executed from within a program, and cannot be executed from the Basic Ready prompt.

The variable types used in an INPUT# statement must be the same type of variable used when the information was written to the file via the PRINT# command. At least one variable must be specified with the INPUT# command. If multiple variables are specified with the INPUT# command, they must be separated by commas.

After execution of an INPUT# command, the variable(s) specified will be assigned values corresponding to the data retrieved from the disk. If you try to execute an INPUT# command after all of the data has been retrieved from the file, an INPUT PAST END error will be generated.

Example

Suppose a file called MYFILE/SEQ was created using the OPEN"O" and PRINT# commands, and this file contains the following pieces of data:

```
JONES
THOMAS
12
MALE
```

The following commands may be used to access this information:

```
OPEN"I",1,"MYFILE/SEQ"
INPUT#1,LN$,FN$,AG%
INPUT#1,SE$
```

After the execution of the first two commands, the file MYFILE/SEQ would have been opened for sequential input, the variable LN$ would have been assigned the value "JONES", the variable FN$ would have been assigned the value "THOMAS", and the variable AG% would have been assigned the value 12. Note that the last piece of data in the file ("MALE") would not have been accessed by either of the first two commands. However, after the third command (INPUT#1,SE$) has been executed, the variable SE$ would be assigned a value of "MALE".

INPUT# deals with data in a disk file in a special way. For more information on creating sequential files that are accessed by the INPUT# command, refer to OPEN ("0", "E" and "I") and PRINT#.

INSTR - Locate the position of a sub-string within a target string

The INSTR command allows you to search for a specified sub-string within a given target string, and returns the position number in the target string of where the sub-string was found. The syntax for the INSTR command is:

INSTR(starting position,target string,sub-string)

"starting position" is the point where you wish the search to begin in the target string (e.g. start the search from the third character in the target string). If not specified, starting position will default to 1.

"target string" is the string you wish to search.

"sub-string" is the string you wish to search for within the target string.

The starting position may be either a numeric constant or a numeric expression, and must represent an integer value in the range of 1 to 255, inclusive. The target string and sub-string may be either string constants or string expressions.

INSTR will begin the search of the target string for the sub-string from the starting position specified (if no starting position is specified, INSTR will begin the search from the first character of the target string), and will return a numeric value corresponding to the position in the target string of where the first occurrence of the sub-string is found. If the sub-string is not found in the target string, INSTR will return a 0. If the sub-string to be searched for is a null string, INSTR will return the starting position of the search, as the null string is a sub-set of any string.

Other occurrences may cause INSTR to return a zero. They are:

If the target string is a null string.

If the starting position is a number greater than the length of the target string.

The following example will illustrate the use of the INSTR command.

Example

Suppose you have the following lines in a program:

```
A~="ROY IS A BOY":B$="OY":C$="ROY":D$="oy":E$="ROYIS"
A%=INSTR(A$,C$)
B%=INSTR(2,A$,B$)
C%=INSTR(3,A$,B$)
D%=INSTR(2,A$,C$)
E%=INSTR(A$,D$)
F%=INSTR(A$,E$)
```

After executing the above lines, the following variables will have been assigned these values:

A%=1 B%=2 C%=11 D%=0 E%=0 F%=0

Note that the value of E% will be 0. This is because the sub-string ("oy") is in lower case, and there are no lower case letters in the target string. Also note that the value of F% will be $. This is because the string "ROYIS" does not appear in the target string (there is a space between the words ROY and IS in the target string).

KILL - Kill (Remove) a disk file from the directory

The KILL command will allow you to kill a file from a disk directory, making that file inaccessible, and freeing up the space on the diskette that the file consumed. The KILL command functions identically to the LDOS Library command "KILL". The syntax for the KILL command is:

```
KILL"filespec"
```

where filespec is any valid LDOS file specification. Filespec may be represented as a string constant or a string expression.

Realize that if the filespec given with the KILL command does not exist, you will get the error message FILE NOT FOUND.

Example

Suppose you wish to remove the file MYFILE/DAT from the diskette currently in drive 1, and free up the space consumed by that file. The following command will perform this function.

```
KILL"MYFILE/DAT:1"
```

Realize that after the kill is performed, you will no longer be able to access any information which was previously stored in the file. Also note that since the filespec is being represented as a string constant, it must be enclosed in quotes.

N O T E

When performing a KILL of a data file, the file in question must NOT be in an OPENed state. The KILLing of an open file may cause certain parts of the diskette in question to be totally inaccessible!

LINEINPUT - Input a line into one variable

The LINEINPUT command is very similar to the INPUT command. It will allow you to input information in from the keyboard to be stored in a variable. The differences between the LINEINPUT command and the INPUT command are as follows:

No question mark will appear when the input is taken.

Only one variable may be assigned a value.

All characters entered before <ENTER> is pressed will be assigned to the variable specified (i.e. commas and quotes may be input from the keyboard, and leading spaces are not ignored).

The syntax for the LINEINPUT command is:

```
LINEINPUT"prompting message";variable
```

The prompting message is optional; if used, it must be included within quotes, and must be separated from the variable by a semicolon. If the prompting message is not used, a semicolon cannot be used. As is the case with the INPUT command, LINEINPUT cannot be issued from the Basic Ready prompt.

Example

Suppose that you wish to input a person's name and title into a program, and you wish to separate the name from the title by use of a comma. Using the LINEINPUT command, you may now input the comma from the keyboard to be taken as part of the input The following LINEINPUT command may be used to accomplish this.

```
LINEINPUT"Enter Name, Title";A$
```

When the computer executes the above command, you will see the prompt "Enter Name, Title" appear, and there will be no question mark after the prompt. The computer will now be awaiting your input. If you answer this prompt by typing in the response "JOHN JONES, PRESIDENT" , A$ will be assigned all characters that you have typed in, prior to pressing the <ENTER> key.

LINEINPUT# - Input a line from a disk file into a variable.

The LINEINPUT# command will allow you to input a line from a disk file into a variable. It functions similarly to the LINEINPUT command, with the exception being that the input is taken from the disk, rather than the keyboard.

The syntax for the LINEINPUT# command is:

```
LINE INPUT#b ,variable
```

where b is the buffer number used when the file was opened, and variable is a string variable used to stored the retrieved information.

LINEINPUT# differs from INPUT# in several ways. As noted in the PRINT# command, INPUT# will read information in from the disk until it encounters a comma, a carriage return, the end of file, or the 255th character when dealing with string information. When using LINEINPUT#, commas will not be taken as delimiters of the string, and hence may be included in the input from disk. The LINEINPUT# of a variable will terminate when a carriage return, the end of file, or the 255th character of a string is encountered. As is the case with INPUT#, LINEINPUT# cannot be executed from the Basic Ready prompt.

Example

Assume the following data is stored in a disk file, and the file has been opened using buffer number 1 (<cr> represents a carriage return).

```
JOHN JONES , PRESIDENT , ABC CORPORATION<cr>
```

If the command LINEINPUT#1,A$ is used to input the above information, A$ would be assigned the value:

```
JOHN JONES , PRESIDENT , ABC CORPORATION
```

Realize that all of the characters (including the commas) would be read in and assigned to A$.

If the command INPUT#1,A$ were used instead of LINEINPUT#, the value of A$ would be "JOHN JONES", as INPUT# will read information until it encounters a comma. For more information on how data is stored on the disk in a sequential file, see PRINT#.

LOAD - Load a BASIC program into memory

The LOAD command allows you to retrieve a BASIC program that has been stored on disk, and place it in the computer's memory so that it may be executed or edited. The syntax for the LOAD command is:

```
LOAD"filespec",R
```

filespec may be represented as a string constant or a string expression. If represented as a string constant, filespec must appear within quotes.

The R parameter is optional; if used, the program to be loaded will be executed after it is loaded, and all open files will remain open. Performing a LOAD without the R option will cause any open files to be closed.

Loading a program will always overwrite any program in memory with the program to be loaded. Basic programs cannot be concatenated with the LOAD command (see MERGE for program concatenation). The LOAD command may be given from the BASIC Ready prompt, or can be issued from within a program. If issued from within a program, the program issuing the LOAD command will be overwritten by the program to be loaded, and execution will be terminated.

Example

```
LOAD"MYPROG/BAS"
```

After execution of this command, any program which was in memory will be replaced by the program MYPROG/BAS.

LOC - Get current record number

The LOC command is used primarily with random files, and will return a value corresponding to the current record number of the given file. The syntax for the LOC command is:

```
LOC(#)
```

where # represents the buffer number used to open the file in question. # may be either a numeric constant or a numeric expression, and must correspond to an open file.

When a tile is in an open state, the computer maintains some control information dealing with that file. One piece of information that is available to the user is the record number currently being dealt with. The LOC command will return the current record number that the computer has accessed. If no record in an open file has been accessed, LOC will return the value 0.

Example

Suppose you have opened a file using buffer number 2, and have fielded the buffer accordingly. If the following commands are executed:

```
GET2,17
A%=LOC(2)
```

the variable A% will be assigned the value 17.

LOF - Get last record number

The LOF command is used primarily with random files, and will return a value corresponding to the last record number of the given file. The syntax for the LOF command is:

```
LOF(#)
```

where # is the buffer number used to open the file in question. # may be either a numeric constant or a numeric expression.

The LOF command provides a means of determining the number of records that have been written to a random file. Note that if a file has been pre-created using the CREATE library command, LOF will return a number corresponding to the highest record number actually written to, not the number of records that have been pre-created.

Example

Suppose you have a file named MYFILE/DAT, and the highest record number written to is record number 43. If the file has been opened using buffer number 3, and has been fielded accordingly, the following command will result in the variable A% being set equal to 43.

```
A%=LOF(3)
```

LSET - Place data into the buffer assigned to an open file

The LSET command will allow you to place information in the buffer associated with a random file, prior to writing the information in the buffer out to disk. It is used primarily in conjunction with random files. The syntax for the LSET command is:

```
LSET fielded string variable=value
```

fielded string variable is the variable used in the FIELD statement that points to the location in the buffer where the data is to be placed.

value is the value that you wish to place in the buffer, and must be a string constant or a string expression.

When dealing with random files, the FIELD statement is used to set up and partition the buffer associated with the file. String variables are used in the FIELD statement to designate various slots for information storage and retrieval in the buffer. The LSET command allows you to place information in these slots in the buffer, prior to writing the information out to disk.

The LSET command will left-justify the information in the buffer. That is to say, if the length of the string to be placed in the buffer is less than the length allocated for the particular slot, trailing spaces will be inserted at the end of the string in the buffer. This will make the string in the buffer the same length as specified in the FIELD statement.

If the length of the string to be LSET into the buffer is greater than the fielded length, the left most part of the string will be placed in the buffer, and any characters to the right of the total allocated space will be truncated. See RSET to right-justify a string into the buffer.

The commands MKI$, MKS$, and MKD$ are also used in conjunction with the LSET statement. Because the buffer is fielded in terms of string Variables, only string values may be LSET into the buffer. The MKI$, MKS$, and MKD$ commands are used to change numeric data into compressed string representations of numbers, and will create strings of 2 bytes, 4 bytes, and 8 bytes respectively. When performing an LSET using the MKI$, MKS$ or MKD$ commands, the length of the fielded variable to be LSET must be at least 2 bytes, 4 bytes, or 8 bytes, respectively. For more information on commands that are used with LSET, refer to the commands MKI$, MKD$, MKS$, and FIELD, and the example below.

Example

Suppose you have a file called MYFILE/DAT, and have opened the file to have record lengths of 45 bytes. In addition, assume that the buffer corresponding to the file (buffer number 1) has been fielded with the following statement, and the variables listed below have been assigned the given values:

```
FIELD 1, 31 AS NA$, 2 AS A2$, 4 AS A4$, 8 AS A8$
NM$="JOHN JONES, PRESIDENT":A2%=92:A4!=23.79:A8#=123498.63
```

The LSET statements you may use to place these values into the buffer may look like this:

```
LSET NA$=NM$
LSET A2$=MKI$(A2%)
LSET A4$=MKS$(A4!)
LSET A8$=MKD$(A8#)
```

The values of the variables A2%, A4!, and A8# will be stored in the slots in the buffer pointed to by the variables A2$, A4$, and A8$, respectively. They will be stored as compressed string representations of the values the variables have been assigned.

The value of NM$ will be stored in the slot in the buffer pointed to by the variable NA$. Realize that since the length of NM$ is 21 characters, the last 10 characters of the slot in the buffer pointed to by NA$ will be spaces (CHR$(32)). If the length of NM$ would have been longer than 31 characters, the left-most 31 characters would have been placed in the buffer, and the remaining characters would have been truncated (in essence, ignored).

The LSET command will typically be used prior to performing a data write to a random file. For more information on performing a data write to a random file, see OPEN, FIELD and PUT.

MERGE - Merge a program from disk with current program in memory

The MERGE command will allow you to merge a program file stored on disk (in ASCII) with a program resident in memory, with the resultant program being stored in memory. The syntax for the MERGE command is:

```
MERGE"filespec"
```

where filespec represents a BASIC program stored on disk in ASCII (For more information on storing BASIC programs on disk in ASCII, see SAVE). Filespec may be represented as a string constant or a string expression. If represented as a string constant, filespec must be contained within quotes.

The MERGE command will read in (line by line) the program from disk, and merge these lines in with the existing program. Any line number in the program to be merged that does not exist in the program in memory will be added to the program in memory. Any line number in the program to be merged that does exist in the program in memory will overwrite the line in memory.

The MERGE command provides for an easy way to merge subroutines which are common to several different programs into these programs without always having to type in the subroutine. The following example will illustrate how the MERGE command functions.

Example

Suppose you have a program which is resident in memory, and this program consists of the following statements:

```
10 FOR L=1TO100
20 PRINT L
30 NEXT L
```

Assume also that you have a program named NYPROG/ASC stored in ASCII on disk, and this program Consists of the following statements:

```
5 DEFINT A-Z
10 FORL=1TO500
25 'THIS LINE HAS BEEN MERGED IN
40 GOTO 10
```

If you wish to merge the program MYPROG/ASC with the program currently in memory, you may do so by issuing the following command:

```
MERGE "MYPROG/ASC"
```

By giving the above command, the program resident in memory will be changed to the following:

```
5 DEFINT A-Z
10 FORL=1TO500
20 PRINT L
25 'THIS LINE HAS BEEN MERGED IN
30 NEXT L
40 GOTO 10
```

Before merging in a program, you should make sure that there is enough free memory for the program to be merged in. Also, note that the MERGE command is usually issued from the BASIC Ready prompt. However, if incorporated within a program, the MERGE will be done, but execution of the program will cease.

MID$= - Replace a portion of a string

The MID$= command will allow you to perform a character for character replacement of any characters within a string. MID$= is the only BASIC function which may be used on the left-hand side of the equal sign. The syntax for the MID$= command is:

```
MID$(string value,starting position,length)=replacement string
```

"string value" may be either a string constant or a string expression, and represents the target string for the replacement.

"starting position is the place in the string value where the replacement is to start. This may be either a numeric constant or a numeric expression.

"length" is the number of characters to be changed. This may be either a numeric constant or a numeric expression. The length parameter is optional; if omitted, the number of characters to be replaced will be determined by the replacement string.

"replacement string" is the string you wish to replace the specified portion of the current string with. This may be either a string constant or a string expression.

The MID$= command will perform a character for character replacement on a given string with the replacement string. It may not be used to lengthen or shorten an existing string. If the length parameter is not specified, the number of characters involved in the replacement will be determined by the length of the replacement string. If the length parameter differs from the length of the replacement string, one of several things may happen.

If the length parameter is less than the length of the replacement string, the length parameter will take precedence, and only the left-most number of characters as specified in the length parameter will be changed.

If the length parameter is greater than the length of the replacement string, the replacement string will take precedence, and only those characters specified in the replacement string will be changed.

If the parameters specified in the MID$= command would cause the original string to become larger, only those characters up to the end of the original string would be changed, and the length of the string would remain unchanged. In essence, the extra characters at the end of the replacement string would be ignored.

The following example should clarify how the MID$= command functions.

Example

Suppose you have a string variable A$ set equal to the value "THIS IS IT". The following MID$= commands would have these affects on A$.

```
MID$(A$,3,2)="AT"       ---> A$ would change to "THAT IS IT"
MID$(A$,6,2)="WAS"      ---> A$ would change to "THIS WA IT"
MID$(A$,3,8)="AT'S IT"  ---> A$ would change to "THAT'S ITT"
MID$(A$,9,3)="ALL"      ---> A$ would change to "THIS IS AL"
```

MKD$ - Change a numeric value into an 8 byte compressed string

The MKD$ command (MaKe Double precision string) will change a numeric value into an 8 byte string which is a compressed representation of the value. This command is used primarily with the LSET and RSET commands to place numeric data into the buffer associated with an open random file. The syntax for the MKD$ command is:

MKD$(numeric value)

where numeric value may be either a numeric constant or a numeric expression. Numeric value can represent any value which may be assigned to a double precision variable. Up to 16 significant digits will be maintained. To convert an 8 byte compressed string representation of a number back to a numeric value, use the CVD command.

Since only strings may be stored in the buffer associated with an open random file, there exists a need to change numeric data into a string form. MKD$ provides a way to change numeric data into a string. The string formed by MKD$ will always be 8 bytes in length, regardless of the actual value to be converted. The resultant string value obtained when performing an MKD$ command will be the compressed form of a number, contained in an 8 byte string. After a numeric value has been changed into an 8 byte compressed string, it may then be placed into a buffer via the LSET and RSET commands. (Note: This is not the same as the STR$ command, as the STR$ command produces an ASCII string, not a compressed string representation of a number.)

Example

Suppose you have opened and fielded a random file, and wish to place a double precision value into the buffer. The fielded variable you are dealing with is A8$, and the value you wish to place in the part of the buffer pointed to by A8$ is contained in the variable A8#. The following command will cause an 8 byte compressed string representation of the value stored in A8# to be written to the portion of the buffer pointed to by A8$.

```
LSET A8$=MKD$(A8#)
```

Note that the fielded length of the variable A8$ must be at least 8 bytes, and in most cases will be exactly 8 bytes.

MKI$ - Change a numeric value into a 2 byte compressed string

The MKI$ command (MaKe Integer string) will change a numeric value into a 2 byte string which is a compressed representation of the value. This command is used primarily with the LSET and RSET commands to place numeric data into the buffer associated with an open random file. The syntax for the MKI$ command is:

```
MKI$(numeric value)
```

where numeric value may be either a numeric constant or a numeric expression. Numeric value must be within the range of -32768 to +32767, inclusive. If numeric value is not an integer, any numbers to the right of the decimal point will be truncated. To convert a 2 byte compressed string representation of a number back to a numeric value, use the CVI command.

Since only strings may be stored in the buffer associated with an open random file, there exists a need to change numeric data into a string form. MKI$ provides a way to change numeric data into a string. The string formed by MKI$ will always be 2 bytes in length, regardless of the actual value to be converted.

The resultant string value obtained when performing an MKI$ command will be the compressed form of an integer, contained in a 2 byte string. After a numeric value has been changed into a 2 byte compressed string, it may then be placed into a buffer via the LSET and RSET commands. (Note: This is not the same as the STR$ command, as the STR$ command produces an ASCII string, not a compressed string representation of a number.)

Example

Suppose you have opened and fielded a random file, and wish to place an integer value into the buffer. The fielded variable you are dealing with is A2$, and the value you wish to place in the part of the buffer pointed to by A2$ is contained in the variable A2%. The following command will cause a 2 byte compressed string representation of the value stored in A2% to be written to the portion of the buffer pointed to by A2$.

```
LSET A2$=MKI$(A2%)
```

Note that the fielded length of the variable A2$ must be at least 2 bytes, and in most cases will be exactly 2 bytes.

MKS$ - Change a numeric value into a 4 byte compressed string

The MKS$ command (MaKe Single precision string) will change a numeric value into a 4 byte string which is a compressed representation of the value. This command is used primarily with the LSET and RSET commands to place numeric data into the buffer associated with an open random file. The syntax for the MKS$ command is.'

```
MKS$(numeric value)
```

where numeric value may be either a numeric constant or a numeric expression. Numeric value can represent any value which may be assigned to a single precision variable. Up to 6 significant digits will be maintained. To convert a 4 byte compressed representation of a number back to a numeric value, use the CVS command.

Since only strings may be stored in the buffer associated with an open random file, there exists a need to change numeric data into a string form. MKS$ provides a way to change numeric data into a string. The string formed by MKS$ will always be 4 bytes in length, regardless of the actual value to be converted. The resultant string value obtained when performing an MKS$ command will be the compressed form of a number, contained in a 4 byte string. After a numeric value has been changed into a 4 byte compressed string, it may then be placed into a buffer via the LSET and RSET commands. (Note: This is not the same as the STR$ command, as the STR$ command produces an ASCII string, not a compressed string representation of a number.)

Example

Suppose you have opened and fielded a random file, and wish to place a single precision value into the buffer. The fielded variable you are dealing with is A4$, and the value you wish to place in the part of the buffer pointed to by A4$ is contained in the variable A4!. The following command will cause a 4 byte compressed string representation of the value stored in A4! to be written to the portion of the buffer pointed to by A4$.

```
LSET A4$=MKS$(A4!)
```

Note that the fielded length of the variable A4$ must be at least 4 bytes, and in most cases will be exactly 4 bytes.

OPEN - Open a random/sequential disk file

The OPEN command allows you to open random/sequential data files in order that input/output may occur between the computer and the given file. The general syntax for the OPEN command is:

```
OPEN"file type",buffer number,"filespec",record length
```

"file type" is the type of file you wish to deal with (random or sequential). It may be represented as a string constant enclosed within quotes, or as a string expression.

"buffer number" is the number of the buffer you wish to use to perform the input/output from/to the disk. This may be either a numeric constant or a numeric expression, and must be an integer value within the range of 1 to the total number of active files declared when entering LBASIC, inclusive.

"filespec" is the name, extension, password and drive number of the file to be opened. Filespec must conform to all of the rules governing LDOS filespecs. It may be represented as a string constant or a string expression.

"record length" pertains to random files only, and will determine the record length used when accessing the file. It must be an integer value, and may be represented as either a numeric constant, or a numeric expression whose value must be in the range of 0 to 255, inclusive. This parameter is optional; if not used, record length will default to 256. If record length is specified as 0, it will be assumed to be 256. If the parameter BLK=OFF is specified when entering LBASIC, record length cannot be specified in an OPEN statement, and will default to 256.

Note: Devices may be substituted for files when using an OPEN"I" or OPEN"O" command. Opening devices is an advanced feature of LBASIC, and is only mentioned here as an additional implementation of the OPEN command.

In order to write information to and retrieve information from a disk file, the file must be opened using the OPEN command. The OPEN command establishes the capability of reading from and writing to a disk file by creating a file control block (FCB). This FCB contains information needed by the computer, so that the computer may interact with the disk file. In addition, the OPEN command establishes a buffer which is used by the computer as a temporary storage place for information that will pass between the computer and the disk file.

There are two types of files available to you when storing information in a disk file; sequential files and random files.

Sequential files are file types that allow for accessing data in a specified sequence. That is to say, if you wish to retrieve the 10th piece of information in a file, you must read in the nine data items preceding the item in question before it may be accessed.

Random files are file types that allow you to directly access any piece of information in a file, regardless of the physical location of the data within the file.

It is beyond the scope of this manual to discuss the techniques involved in creating and accessing information in random and sequential files. What will be provided for you here is the syntax needed to open all types of random and sequential files. For the novice, it is strongly recommended that supplementary material be obtained for the purpose of learning filing techniques.

I M P O R T A N T N O T E

It is strongly advised that no data file be in an open state at any given time using more than one buffer. LBASIC will allow you to open the same file at the same time using more than one buffer; however, this practice may lead to the destruction of data files on the diskette in question!!

Opening sequential files.

There are two basic modes available for use when dealing with sequential files; the input mode, and the output mode. The following list shows all of the different OPEN commands that may be issued when dealing with sequential files.

```
OPEN"I"    --> Open an existing sequential file for input

OPEN"O"    --> Open a sequential file for output
OPEN"OO"   --> Open an existing (old) sequential file for output
OPEN"ON"   --> Open a non-existing (new) sequential file for output

OPEN"E"    --> Open for output and extend a sequential file
OPEN"EO"   --> Open for output and extend an existing sequential file
OPEN"EN"   --> Open for output and extend a non-existing sequential file
```

The input mode of sequential files allows you to input information from an existing file. No output to the file may be done if it has been opened for input. The file to be opened for input must exist, or the OPEN"I" command will return a FILE NOT FOUND error. Once the file has been opened, information may be retrieved from it using the INPUT# and LINEINPUT# commands.

The output mode of sequential files allows you to output information to the file. No input from the file may be done if it has been opened for output. Once the file has been opened, information may be written out to it using the PRINT# command. There are six types of output modes available for use with sequential files.

The OPEN"O" output mode functions in the following manner. If the file opened does not exist, it will be created, and information will be written to the file starting at the first byte of the file. If the file opened does exist, any information previously stored in the file will be lost, as the new information to be placed in the file will be written over the existing information, starting at the first byte of the file.

The OPEN"OO" output mode functions in the following manner. If the file opened does not exist, a FILE NOT FOUND error will be generated, and the file will not be created. If the file opened does exist, OPEN"OO" will function identically to OPEN"O" in the case where the file already exists.

The OPEN"ON" output mode functions in the following manner. If the file already exists, you will not be allowed to open the file, and the error FILE ALREADY EXISTS will be generated. The existing file will not be altered in any way. If the file does not exist, it will be created, and information will be written to the file starting with the first byte of the file.

The OPEN"E" output mode functions in the following manner. If the file does not exist, OPEN"E" will function identically to OPEN"O". If the file already exists, the file will be opened, and any information that will be written to the file will be appended to the end of the existing information. The file will be extended to include both the old and the new information.

The OPEN"EO" output mode functions in the following manner. If the file does not exist, a FILE NOT FOUND error will be generated, and no file will be created. If the file already exists, the file will be opened, and any information that will be written to the file will be appended to the end of the existing information. The file will be extended to include both the old and the new information.

The OPEN"EN" mode functions identically to the OPEN"ON" output mode.

Example - Opening sequential files

Suppose that you wished to open a sequential file named MYDATA/SEQ, using buffer number 1. The statement used to open the file for input would be as follows:

```
OPEN "I",1,"MYDATA/SEQ"
```

If you wished to open the same file for output using buffer number two, one of the following commands could be used, depending on whether or not you request that the file be a new file or an old file, and whether or not you wish to extend the file:

```
OPEN"O",2,"MYDATA/SEQ"
OPEN"OO",2,"MYDATA/SEQ"
OPEN"ON",2,"MYDATA/SEQ"
OPEN"E",2,"MYDATA/SEQ"
OPEN"EO",2,"MYDATA/SEQ"
OPEN"EN",2,"MYDATA/SEQ"
```

Opening random files

Unlike sequential files, when dealing with a random file, you have the capability of reading from and writing to the file using only one OPEN command. The statements PUT and GET differentiate between writing to the file and reading from the file, respectively. There are three different types of OPEN statements that may be executed when opening a random file. They are:

```
OPEN"R"   --> Open a random file whether or not it exists.
OPEN"RN"  --> Open a random file only if it does not exist.
OPEN"RO"  --> Open a random file only if it already exists.
```

The OPEN"R" mode functions in the following manner. The file specified will be opened whether it exists or not, and will be created if it does not exist. After the file has been opened, the buffer used in the OPEN statement may be fielded using the FIELD statement, and records may then be retrieved from or placed into the file via the PUT and GET statements.

The OPEN"RN" mode functions in the following manner. If the file already exists, you will not be allowed to open it. The file will remain untouched, and the error FILE ALREADY EXISTS will occur. If the file does not exist, it will be created, and the OPEN"RN" command will function in the same manner as the OPEN"R" command.

The OPEN"RO" mode functions in the following manner. If the file does not exist, no file will be created, and the error FILE NOT FOUND will occur. If the file does exist, OPEN"RO" will function in the same manner as OPEN"R".

Example - Opening random files

Suppose you wish to open a random file named MYDATA/RND, using buffer number 3, with record lengths of 52 bytes. One of the following OPEN commands may be used to open the file, depending on the specific requirements needed by the user (i.e. open the file only if it does or does not exist).

```
OPEN"R",3,"MYDATA/RND",52
OPEN"RN",3,"MYDATA/RND",52
OPEN"RO",3,"MYDATA/RND",52
```

For more information on using both random and sequential files, refer to FIELD, GET, PUT, LSET, RSET, INPUT#, LINEINPUT#, and PRINT#.

PRINT# - Output data to a sequential file

The PRINT# command will allow you to output data to a sequential file. The syntax for the PRINT# command is:

```
PRINT#buffer number,list of constants and/or expressions
```

buffer number is the buffer used to open the file. It may be expressed as a numeric constant or a numeric expression.

list of constants and/or expressions contains the data that you wish to output to the file. Numeric constants, numeric expressions, string constants and string expressions may all be contained within this list. If more than one value is to be output to the file using a single PRINT# statement, these values must be separated by some type of delimiter. The uses of delimiters in a PRINT# command will be explained throughout this section.

The PRINT# command is used in conjunction with any type of OPEN"O" or OPEN"E" command. After a file has been opened, data may be output to the file via the PRINT# command. Once a file has been created using the OPEN"O"/OPEN"E" and PRINT# commands and then closed, the information in the file may be accessed using the OPEN11I" and INPUT#/LINEINPUT# commands.

In most cases, data written to a sequential file is stored in ASCII format. For numeric data, a sign byte will always precede the numeric information. If the value is positive, the sign byte will be represented by a space. A trailing space will always follow the ASCII representation of the value. Keeping the above in mind, the minimum amount of bytes required to store a numeric value in a sequential file is 3 (the sign byte, a digit, and the trailing space).

For string data, all characters included in the string value will be written to the file, and no preceding or trailing characters will be written to the file. Special considerations do need to be taken into account when writing string values to a sequential file, as there are some peculiarities involved with the INPUT# command when trying to access string information stored in a sequential file. These special cases will be pointed out throughout this section.

The PRINT# command resembles the PRINT command in many ways with respect to how information is physically written to the file. Some of the punctuation used in the PRINT# command will cause data to be written to the file in much the same way that this punctuation causes data to be printed to the screen using the PRINT command.

Punctuation is very important when using the PRINT# command. The following will describe the punctuation which is allowed with the PRINT# command, and the effects of using different punctuation.

Use of punctuation with the PRINT# command.

Different types of punctuation used to separate values to be output in a PRINT# statement will cause the data to be physically written to the file in different ways. The following list shows the punctuation required to separate values contained in a PRINT# statement.

```
,      - comma
;      - semicolon
","    - explicit comma
```

When separating output data contained in a PRINT# statement, you may use either a comma or a semicolon. A semicolon will cause the next piece of information to be written directly after the preceding data. A comma will cause the next piece of information to be written at the next available "tab" position in the file. Tab positions will be denoted by 16 byte blocks, starting from the last occurrence of a carriage return (0DH) in the file.

In some cases, the explicit comma is used after string information has been written to the disk, to demark the end of the string value from the beginning of the next piece of information to be written out.
The following examples will illustrate the methods used to write data to a sequential file, as well as the occurrences that will result when this data is to be retrieved.

Example 1 - Writing numeric data to a sequential file.

Suppose you wish to write two numeric values out to a sequential file, using one PRINT# command. The file you wish to write these values Out to is named DATA1/SEQ, and has been opened using buffer number 2. The variables you wish to write out to the file are A%, which has been assigned a value of 362, and B!, which has been assigned a value of -2618.7. The following PRINT# command may be used to write these values out to the file:

```
PRINT#2,A%;B!
```

The above statement will cause the values 362 and -2618.7 to be written to the file in ASCII format. The image produced on the disk by this PRINT# statement is shown below. (Note that throughout the rest of this section, the image produced by the example PRINT# statements will always follow the PRINT# statement. The image shown will be similar to the LDOS LIST (H) library command; each ASCII character will be displayed with its corresponding hex value shown below the character.)

```
     3  6  2     -  2  6  1  8  .  7
  20 33 36 32 20 2D 32 36 31 38 2E 37 20 0D
```

Note the sign byte preceding each value, and the trailing space following each value. Also note that the last byte written to the file is a carriage return (~DH). A carriage return will always be written to the file after the last item listed in a PRINT# statement.

Realize that a semicolon was used to separate the variables A% and B! in the PRINT# command. A comma could have been used instead; however, the image of the data on the disk would have changed to the following if a comma would have been used instead of a semicolon.

```
     3  6  2
20 33 36 32 20 20 20 20 20 20 20 20 20 20 20 20

 -  2  6  1  8  .  7
2D 32 36 31 38 2E 37 20 0D
```

Notice the series of spaces following the number 362. These will be written to the disk as a result of a comma being used to separate the variables A% and B!. As was noted earlier, when using a comma to separate variables in a PRINT# statement, the value following the comma will be written to the next tab position (the beginning byte of the next block of 16 bytes). As depicted in the above displays, much disk space will be wasted in writing to sequential files if the values in a PRINT# statement are separated by commas instead of semicolons.

Example 2 - Writing string data to a sequential file.

Suppose you wish to write 3 string values out to a sequential file, using one PRINT# command. The file you wish to write these values out to is named DATA2/SEQ, and has been opened using buffer number I. The variables you wish to write out to the file are A$ (which has been assigned the value "AMBER"), B$ (which has been assigned the value "BROWN"), and the string constant "GRAY". The following PRINT# command may be used to write these values out to the file:

```
PRINT#1,A$;",";B$;",";"GRAY"
```

The above statement will cause the values "AMBER", "BROWN" and "GRAY" to be written to the file. The image produced on the disk by this PRINT# statement is shown below.

```
 A  M  B  E  R  ,  B  R  O  W  N  ,  G  R  A  Y
41 4D 42 45 52 2C 42 52 4F 57 4E 2C 47 52 41 59 0D
```

There are many things to be noted in this example. The most prominent of these is the use of the explicit comma (","). You will note from the above display that along with the string values, commas were also written out to the file (since they were enclosed within quotes as part of the list of values to be written out). In most applications dealing with writing strings out to sequential files, you will need to incorporate the explicit comma within the list of values to be printed out by the PRINT#. The reason behind this stems from the way INPUT# deals with retrieving information from a sequential file.

Before continuing with more examples on the use of PRINT#, a brief discussion of using INPUT# with files created by PRINT# is in order.

How INPUT# ties together with PRINT#

As shown throughout this section, the punctuation used in the PRINT# command is very important, and determines the manner in which INPUT# will access this information. INPUT# deals with retrieving numeric data in a different fashion than it does with string data.

When INPUT# requests the input of a numeric variable, it will begin reading from the last accessed byte in the file. Any leading spaces that are encountered will be ignored. Once INPUT# finds a non-space character, it will read until it encounters either a space or a delimiter, and the value assigned to the variable will be determined by performing a VAL function on the characters read in. This is to say that any characters may be input into a numeric variable, and the inputting of string values into a numeric variable will not cause a TYPE MISMATCH error.

When INPUT# requests the input of a string variable, it will begin reading from the last accessed byte in the file, and proceed until it finds a non-space character. Once it finds a non-space character, it will read until it encounters a delimiter, and the value assigned to the variable will be all characters read in from the first non-space character to the delimiter. Note from the above description that any "leading" spaces which are present in the data file for the data element in question will be ignored by INPUT#, and the value assigned to the string will never have leading spaces.

In all cases, when INPUT# requests an input of a variable, the input will be terminated when a delimiter character is read in. For numeric inputs, delimiters can be represented by either a space, a comma, or a carriage return (0DH). In most cases, a coma should not be used as the delimiter for a numeric input.

For string inputs, a delimiter can be represented by either a comma or a carriage return. Realize that for any input of a variable, if the number of characters read in will exceed 255, the input of the variable will terminate after the 255th character has been accessed.

One point to note is that in most cases, two delimiter characters should not appear together in a sequential file. This occurrence will cause unpredictable results when trying to input information from the file.

From the above paragraphs, it can be seen that in any one physical PRINT# statement, if values are to be written Out following a string value, they must be separated from the string value by use of the explicit comma. The general format which is recommended to perform such a data write is as follows:

```
PRINT#b,...;string value;",";next value;...
```

Example 3 - Writing numeric and string data to a file.

Suppose you wish to write several string and numeric values out to a sequential file using the same PRINT# statement. The file you wish to write these values out to is named DATA3/SEQ, and has been opened using buffer number 2. The string values you wish to write Out are contained in the variables A$ (which has been assigned the value "ANN"), B$ (which has been assigned the value "BETTY") and C$ (which has been assigned the value "CAROL"). The numeric values you wish to write Out are contained in the variables A% (which has been assigned a value of 2~), B% (which has been assigned a value of 32), and C% (which has assigned a value of 23). The following will show a PRINT# statement which may be used to write these values out to the file, and the associated image that will be written to the disk as a result of performing the PRINT#.

```
PRINT#2,A%;A$;" ,";B%;B$;",";C%;C$
```

```
     2  0     A  N  N  ,     3  2     B  E  T  T
 20 32 30 20 41 4E 4E 2C 20 33 32 20 42 45 54 54
  Y  ,     2  3     C  A  R  O  L
 59 2C 20 32 33 20 43 41 52 4F 4C 0D
```

Please note from the above example that no explicit comma needs to follow numeric data. Also note that since C$ is the last variable to be written out in this PRINT# command, no explicit comma is needed after it, as a carriage return will always be written out to the file after the last variable in a PRINT# command. This carriage return will serve as the delimiter for subsequent PRINT# commands.

This concludes our discussion of the PRINT# command. It is recommended that test files be created by the user in order to explore the results of various PRINT# statements. After sequential files have been created, they may be examined by use the LDOS LIST (H) command. For further information see OPEN, INPUT#, and LINEINPUT#.

PRINT# USING - Output data to a sequential file using a specified format

The PRINT# USING command will allow you to output data to a sequential file using a specified format. The syntax for the PRINT# USING command is:

```
PRINT#buffer number,USING format string;list of values
```

buffer number is the buffer used to open the file. It may be expressed as a numeric constant or a numeric expression.

format string is the format you wish to use to write the list of values out to the file. It may be represented as either a string constant or a string expression.

list of values is the same as list of constants and/or expressions as defined in the PRINT# command.

The PRINT# USING command will allow you to output data to a sequential file in the format specified by the format string. Any format string which is allowable in the PRINT USING command will also be allowable in the PRINT# USING command, and will function in an identical manner. For more information on allowable format strings, refer to PRINT USING in the ROM Basic manual. (For more information on the specifics involved in writing information out to a sequential file, see PRINT#.)

Example

Suppose you wish to write three numeric values out to a sequential file. The name of the file is DATA/SEQ, and it has been opened using buffer number 1. The values you wish to write out are contained in the variables A% (which has been assigned a value of 25), B! (which has been assigned a value of 13.73), and C% (which has been assigned a value of -17). The format string you wish to use has been assigned to the variable A$, and has the value:

```
###   ####.###  ####
```

The following will show a PRINT# USING command that may be used to write out the above values, and the disk image created by the PRINT# USING command.

```
PRINT#1 ,USINGA$;A%,BI ,C%

    2  5                   1  3  .  7  3  0          -  1  7
20 32 35 20 20 20 20 20 20 31 33 2E 37 33 30 20 20 20 2D 31 37 0D
```

Note from the above example that the image created on disk conforms to the format string specified. Unlike the PRINT# command, the use of delimiters to separate the values to be printed out is arbitrary. That is to say, there is no difference in using a comma as a delimiter as opposed to a semicolon.

PUT - Write a record out to a random file

The PUT command is used to write information out to a random file. The information that is to be written out to the file must have been placed into the buffer that was used to open the file prior to being written out to the file. The syntax for the PUT command is:

```
PUT#,r
PUT#
```

where # is the buffer number used to open the file, and r is the record number you wish to retrieve. Both # and r may be numeric constants or numeric expressions.

If the record number (r) is not specified, the computer will first increment the current record number by one, after which it will perform a PUT of the current record number. If no current record number has been established, the computer will perform a PUT of record number one, and the current record number will be set equal to one.

Example

Suppose you wish to output data to a random file. The file you wish to perform the output to has the name FILE/RND, and has been fielded using buffer number 2. The record you wish to write out to the file is record number 23. Assume also that all of the values you wish to write out to the file have been placed into the buffer using the proper LSET and RSET commands. One of the following PUT commands may be used to write the information to the 23rd record of the file.

```
PUT2,23
N%=1:N1%=30:PUTN%+1,N1%-7
```

After executing one of the above statements, the information stored in the buffer associated with the file FILE/RND will be written out to the disk, and will be placed in the file as representing the 23rd record in the file. Once this information has been placed into the file, it may be retrieved using the GET command.

For more information on using PUT, see OPEN, FIELD, LSET and RSET.

RSET - Place data into the buffer assigned to an open file

The RSET command will allow you to place information into the buffer associated with a disk file, prior to writing this information out to the disk. It is used primarily in conjunction with random files. The syntax for the RSET command is:

```
RSET fielded string variable=value
```

fielded string variable is the variable used in the field statement that points to the location in the buffer where the data is to be placed.

value is the value that you wish to place in the buffer, and must be a string constant or string expression.

The RSET command functions identically to the LSET command, with the following exception. Rather than the information being placed into the buffer left-justified, RSET will place the information into the buffer right justified. If the length of the string to be placed into the buffer is less than the fielded length of the particular slot of the buffer, spaces will be inserted in front of the string in the buffer to make the string in the buffer the same length as specified in the field statement.

If the length of the string to be RSET into the buffer is greater than the fielded length, the right most part of the string will be placed in the buffer, and any characters to the left of the total allocated space will be truncated.

For more information on how to utilize the RSET command and the functions it performs, refer to the LSET command.

RUN - Load a Basic program from disk and execute it

The RUN command will allow you to load an LBASIC program stored on disk into the computer's memory, and immediately begin execution of that program. The syntax for the RUN command is:

```
RUN"filespec",file/variable parameter,line number
```

filespec is the name of the program that you wish to be loaded and executed, and may be represented by any valid LDOS filespec. filespec may be either a string constant or a string expression. If filespec is not included, the program currently in memory will be executed.

file/variable parameter is an optional parameter, and is used primarily when LBASIC programs are to be "chained" together. One of two different parameters are available. If the parameter R is used, any files which are currently open will remain open when the new program is loaded and executed If the parameter V is used, all open files will remain open, and all variable assignments will be maintained. This parameter, if used, must be represented as a letter (R or V), and cannot appear within quote marks, or cannot be represented by a string expression.

line number is an optional parameter, and is used to specify a line number in the program where execution is to start. If not specified, execution will begin with the first line number of the new program. It must be represented as a numeric constant.

The RUN command may be issued from the Basic READY prompt to load and execute a program, or may be used from within an LBASIC program to perform a chaining of programs. If the RUN command is given with a filespec, any program which is currently resident in memory will be overwritten, and the program specified in the RUN command will be loaded and executed.

If the RUN command is given with just a filespec (i.e. no additional parameters are specified), no variables will be retained, and any open files will be closed.

If the RUN command is given with the R parameter, all variables will be lost, but any files which were opened will remain open, and will utilize the same buffer number. Realize that if the R parameter is used, any open files must be re-fielded.

If the RUN command is given with the V parameter, any established variables will be maintained, and all open files will remain open. There are several points to be considered when using the V parameter. In addition to all files remaining open, the fielding of the buffer associated with the open file will remain intact. Hence, re-fielding is not required. Any DEFinition statements (such as DEFINT and DEFSTR) must be re-established in the program to be chained. The CLEAR command should not be encountered in the program to be chained, as execution of a CLEAR statement will close all open files and destroy any established variables.

It should be obvious to the user that if the program to be chained is longer than the calling program, or uses more variables than the calling program, an OUT OF MEMORY or OUT OF STRING SPACE error may occur. To utilize this feature to its fullest capabilities, forethought must go into the determination of variable names to be carried over from one program to another.

If the RUN command is given with the line number parameter, the program specified will be loaded, and execution will begin at the line specified. Realize that the line number specified must be an existing line number, or an UNDEFINED LINE NUMBER error will be generated.

The R/V and line number parameters may be specified individually, or they may appear together in the RUN command. If both parameters are specified, the R/V parameter must physically come before the line number parameter.

Example 1

Suppose you have an LBASIC program named MYPROG/BAS, and this program has been saved onto a disk which is currently in drive 1. One of the following commands may be given to load and execute the above program.

```
RUN"MYPROG/BAS:1"
A$="MYPROG/BAS:1":RUN A$
```

After either of the above commands are executed, any program currently in memory will be overwritten, and the program MYPROG/BAS will be loaded and executed. Any open files will be closed, and any established variables will be destroyed.

Example 2

Suppose you wish to load and execute the program MYPROG/BAS as described in the above example, except that you wish execution to begin at line ~ in the program. The following command will cause the program to be loaded) and execution will begin at line ~

```
RUN"MYPROG/BAS:1",3000
```

Example 3

This example will illustrate how to use the V parameter of the RUN command to maintain variables between chained programs. Listed below will be two programs that reference each other (PROG1/BAS and PROG2/BAS). The sequence will be started by issuing the command RUN"PROG1/BAS". Realize that both programs must have been saved on disk prior to trying to execute either.

```
5 'PROG1/BAS
10 CLEAR 2000
20 DEFINTA-Z:DEFSTRS
30 IF A=0 THEN S="PROG1/BAS"
40 CLS
50 A=A+5
60 PRINT"THIS IS ";S,"A=";A
70 IF A>100 THEN END
80 S="PROG2/BAS"
90 INPUT"PRESS <ENTER> TO RUN PROG2/BAS";S1
100 RUN"PROG2/BAS",V,20
```

```
5 'PROG2/BAS
10 CLEAR 2000
20 DEFINTA-Z:DEFSTRS
30 CLS
40 A=A+3
50 PRINT"THIS IS ";S,"A=";A
60 S="PROG1/BAS"
70 INPUT"PRESS <ENTER> TO RUN PROG1/BAS";S1
80 RUN"PROG1/BAS",V,20
```

Notice that in each of the RUN commands, the line number 20 was specified. This accomplishes two things. It causes execution to start at line 20 of each program, which will cause the CLEAR command in both programs to be bypassed. Also, line 20 must be executed, as all DEF type statements must be re-established when programs are chained using the V parameter. Although this is a very simplistic example, it should illustrate some of the steps needed to perform program chaining while retaining variable assignments.

SAVE - Save the LBASIC program resident in memory to disk

The SAVE command will allow you to save the program currently in memory to a disk file. This will allow you to store programs on disk for future use. The syntax for the SAVE command is:

```
SAVE" filespec",A
```

filespec is the file specification you wish to assign to the program file. It may be represented as either a string constant or a string expression.

the A parameter is an optional parameter. If used, the program will be saved out to the file in pure ASCII format. If not specified, the program will be saved out to the file in "compressed" format.

As LBASIC programs are being written or edited, they are contained in the computer's memory. The SAVE command provides a way to save LBASIC programs which are stored in memory out to a disk file, so that they may be referenced at some later time via the LOAD or RUN command.

When the SAVE command is given, one of two things will happen. If the filespec in the SAVE command represents a non-existing file, SAVE will create a file with the filename, extension, and password specified, and store in this file the Basic program currently in memory. If the filespec in the SAVE command represents an already existing file, SAVE will overwrite the existing file with the program in memory.

When the A parameter is not specified in a SAVE command, the program in memory will be saved to a disk file in its compressed form (i.e. compression codes will be used to represent the LBASIC commands and line numbers). If the A parameter is specified in a SAVE command, the program will be saved to the disk file in pure ASCII (e.g. the command PRINT will take up five bytes of disk storage, one byte for each character).

Note: When using the A parameter to save a program, no line in the program should exceed 240 characters in length. If a program is saved with the A parameter and a line in the program is longer than 240 characters, the program will load up to the line which is longer than 240 characters, and the rest of the program will be inaccessible. A Direct Statement in File error will also be generated.

It should be obvious that saving a program in ASCII will consume more disk space than saving the same program in compressed form, but there are certain situations where a program must be saved in ASCII. One case where you have to save a program in ASCII is if you wish to perform a MERGE of a Basic program stored on disk with a program currently in memory. The program to be merged in from disk must have been saved in ASCII, or the merge will abort with an error.

The SAVE command may be given either from the LBASIC Ready prompt, or may be incorporated as a command within a program. If used within a program, the program will SAVE itself, after which normal execution will continue.

Example

Suppose you have keyed in a Basic program, and wish to save this program out to a disk file. The drive you wish to store this file on is drive 1, and the name you wish to assign to this file is GOODPROG/BAS. One SAVE command that may be used to accomplish this might look like this:

```
SAVE"GOODPROG/BAS:1"
```

If you wish to save this program in ASCII, the following command could be used:

```
FS$="GOODPROG/BAS":SAVE FS$,A
```

Note in the above example that the filespec was represented as a string variable. Also note that the A parameter must appear as a literal constant, and cannot be expressed as a string expression.

SET EOF - Reset end of file marker to "shrink" the size of a random file

The SET EOF command may be used to "shrink" the amount of space taken up by a file, and thus free-up additional disk space. The syntax for the SET EOF command is:

```
SET EOFn
```

n represents the buffer number used to open the file in question, and can be expressed as an integer constant or an integer expression.

The SET EOF command is used primarily in conjunction with random files. In some applications, a random file may contain unwanted records at the end of the file. The SET EOF command will furnish you with a way to eliminate these unwanted records. The function it performs is to reset the end of file marker for the file in question to a value less than the current end of file marker. This will cause all records whose record numbers are greater than the new end of file marker value to be ignored, and thus make these records inaccessible. Also, the space taken up on the disk by these "eliminated" records will be added to the free space available, and thus may be reused.

To use the SET EOF command, you must open the file in question as a random file. It is highly recommended that the record length used to open the file be the same as the record length used for normal access to the file.

After the file has been opened, perform a GET of the record you wish to be the last record in the file. You may then use the SET EOF command to reset the end of file marker to the current record number, and thus eliminate all unwanted records (by doing a GET, the current record number will be changed to the value of the record which was retrieved).

Example

Suppose you have a random file named XTRA/DAT which currently contains 100 records, and you wish to eliminate the last 50 records of the file (records 5l-l00). Assume also that the file has been opened in the random mode, using buffer number 3. The following commands may be used to accomplish this "file shrinkage".

```
GET3,50:SET EOF3
```

NOTE

Be extremely careful when using the SET EOF command. Once records have been eliminated from a file using this command, they might not be recoverable!! It is beyond the scope of this manual to discuss techniques used to recover lost information in a file. The best prevention for such an occurrence is caution!

TIME$ - Return the date and time as a string

The TIME$ command will retrieve the current date and time (as kept by the real time clock) as a string. The syntax for the TIME$ command is:

TIME$

The value returned from the TIME$ command can be used in a similar manner as the value returned from the MEM command. It may be used directly (as in the statement PRINT TIME$), or may be assigned to a string variable (as in A$=TIME$). The value returned by the TIME$ command will always be a 17 character string, and will be defined by the following format:

mm/dd/yy hh:mm:ss

mm, dd, and yy represent the month, day of the month, and year respectively, as kept by the operating system. The hh, mm, and ss represent the hours (00-23), minutes (00-59) and seconds (00-59) respectively, as retrieved from the real time clock when the TIME$ command was actually executed. The slashes ("/") and colons (":") will always be present in the string, and a space will always separate the date information from the time information.

USR - Execute a user written machine language subroutine.

The USR command will allow an LBASIC program to branch to a user written machine language subroutine. The syntax for the USR command is:

variable=USRn(integer value)

variable must be a numeric variable, and in most cases should be of integer type. If a value is to be returned from the machine language subroutine, it may be contained in this variable when the machine language routine returns to LBASIC.

n is the user routine number (~-9) used to identify the routine in question (user routines are defined with the DEFUSR command). The routine number must be represented as a numeric constant.

integer value is a value which will be passed to the user machine language subroutine. It may be represented as a numeric expression or a numeric constant, and must be expressed as an integer value.

The USR command will allow you to jump to a machine language subroutine from within your Basic program. The machine language subroutine will generally be resident in high memory, and the memory used by the module must be protected, either using the MEMORY Library command, or by specifying the MEM parameter when entering LBASIC.

Prior to issuing a USR call, the starting address of the specific machine language subroutine must have been defined using the DEFUSR command.

Once the USR call is performed, execution of your Basic program will be halted, and a jump will be done to the address specified in the corresponding DEFUSR statement. Your machine language subroutine will then take over, until a return to Basic is performed in the machine language module. Once this return to Basic has been encountered, your Basic program will regain control.

Example 1 – Initiating a USR call

Suppose you have loaded and protected a machine language module. In addition, you have defined this machine language module with the following command:

```
DEFUSR5=&HF400
```

To perform a jump to this machine language module, the following command may be given:

```
XX%=USR5(1024)
```

Upon executing the above command, execution of the Basic program will be halted, and the machine language instruction at address X'F400' will be executed. The value 1024 will be passed to the machine language routine. The machine language routine will continue to be executed, until a return to Basic is encountered. If any value is to be returned from the subroutine, it will be contained in the integer variable XX% when Basic regains control.

Example 2 - Passing values to and from machine language subroutines

In the above example, the value 1024 was passed to the machine language subroutine. In order to utilize this value in the subroutine, the first statement of the machine language routine should be the following:

```
CALL 0A7FH
```

Executing the above command as the first statement in the machine language subroutine will cause the value 1024 to be placed in the HL register, with H containing the MSB, and L containing the LSB of the value.

To return a value from a machine language subroutine to Basic, you should use the following command as the last statement in your subroutine:

```
JP 0A9AH
```

After your machine language module executes the above command, control will return to Basic (the statement following the USR call), and the variable used in the USR call will be assigned the value that was in the HL register pair prior to the JP command. If no value is to be returned from your machine language module, you may use a RET command to return to Basic.

CMD"N" - LBASIC PROGRAM RENUMBERING

This LBASIC feature will renumber LBASIC program line numbers as well as all line references such as GOSUB and GOTO. The syntax is:

```
CMD"N I aaaa,bbbb,cccc,dddd"

!     optional parameter to skip the complete scan
      for errors before renumbering begins.

aaaa  line number of the current program to start the
      renumbering from.

bbbb  new line number for line aaaa.

cccc  increment between line numbers.

dddd  the last line number to be renumbered.
```

Both LBASIC/CMD and LBASIC/OV1 must be present on the disk, or a "Program Not Found" error will occur.

You cannot have a line number zero (0) if renumbering an LBASIC program.

This renumber feature will allow you to renumber all or parts of the LBASIC program currently in memory. The lines to be renumbered can be anywhere in the program. However, if the parameters you use would result in the renumbered lines being out of sequence, a BAD PARAMETERS error will occur.

If you do not specify the exclamation point (!) character, a full scan for errors will be done before the renumbering starts. If errors do exist, no lines will be changed. It is usually much easier to fix the errors before the lines are renumbered!

If you do specify the !, any error found will still abort the renumbering. However, all internal line number references will have already been changed up to the line that cause the error. Do not use the ! parameter unless you are absolutely sure that no errors exist.

The default values for the line and increment parameters are as follows:

aaaa = 1
bbbb = 20
cccc = 20
dddd = 65529

CMD"X" - LBASIC CROSS REFERENCE

This LBASIC feature will produce a cross reference of variables and line numbers for your LBASIC program currently in memory. The syntax is:

```
CMD"X devspec/filespec  parameter,<title>"

devspec/filespec is the device or file the listing will
be sent to. If not specified, it will go to the screen.

parameter specifies Variables or Lines as follows:

-V          all Variables.

=variable   only the variable specified.

-L          all Line numbers.

=number     only the line number specified.


<title>  an optional title to be printed on the top
         of each page.
```

Both LBASIC/CMD and LBASIC/OV2 must be present on a disk or a "Program Not Found" error will occur.

You cannot have a line number zero (0) if you wish to use the cross reference utility.

This cross reference feature will allow you to produce a list of the variable and line number references of an LBASIC program. This list may be sent to any device in the system, such as the *DO (video screen), *PR (line printer), etc. It may also be sent directly to a specified disk file.

Parameters are allowed to determine which variables or line numbers will be listed. If no parameter is specified, all variables and line numbers will be cross referenced.

If you wish a title to be put on the top of every page in the list, it must be specified between less-than/greater-than symbols in the command line.

L B A S I C E R R O R D I C T I O N A R Y

Incorporated in ROM Basic are various error messages and error codes. These error codes are provided for the user so that certain types of errors may be "trapped" for, and the execution of the Basic program in question will not be interrupted. As pointed out in the ROM Basic manual, the user may determine the exact nature of an error by utilizing the ERR and ERL commands.

Because many new commands are included in LBASIC which are not a part of ROM Basic, LBASIC will have in its error dictionary new error codes (disk error codes), along with the error codes found in ROM Basic. The error dictionary for LBASIC is contained in the file LBASIC/OV3. For this reason, LBASIC/OV3 must always be present on a disk in the system when programming in Basic.

This part of the manual will list the disk error codes and messages, and will include a brief description of each error. The user should realize that the descriptions given for each error are not all inclusive. That is to say, the example circumstances given for a particular error may not encompass all circumstances which could generate the error in question.

Before we begin giving these disk error codes, a few general points should be made. LBASIC's error dictionary is not as large as the error dictionary found in LDOS. For this reason, several different types of disk related errors may produce the same LBASIC error message. To pinpoint the exact nature of a disk related error, it may be beneficial to determine the LDOS interpretation of an error. After a disk related error occurs, you may determine the associated LDOS error message by performing a CMD"E". This may be useful when, for instance, you get the LBASIC error message "Disk I/O Error", as several different occurrences may cause this type of error. For more information, refer to CMD"E".

All error codes given in this manual will be the value returned by the ERR command. In your ROM Basic manual, the error codes given may be derived by the value of ERR/2 or ERR/2+1. If you wish your LBASIC program to conform to these conventions, the error codes listed here must be adjusted accordingly.

Error 100 ---------- Field Overflow

The Field Overflow error indicates that the number of bytes fielded for a random file exceeds the record length of the file (as specified in the OPEN statement).

Error 102 ---------- Internal Error

An Internal Error will occur when the error in question cannot be interpreted. One way an Internal Error may be generated is to issue a CMD"L" command, and the file to be loaded is not found.

Error 104 ---------- Bad File Number

A Bad File Number error will occur when a file is opened using an illegal buffer number (a buffer number greater than the total number of files specified when entering LBASIC), or fielding a buffer which does not correspond to an open random file.

Error 106 ---------- File Not Found

A File Not Found error indicates that the file being referenced does not exist. This error may occur after an OPEN"I', OPEN"EO", OPEN"OO", OPEN"RO", LOAD or RUN command has been issued.

Error 108 ---------- Bad File Mode

A Bad File Mode error indicates that a file is being accessed improperly. This may occur when, for instance, you try to access a file opened as a random file in a sequential manner (i.e. issue an INPUT# command after opening a file in the random mode).

Error 110 ---------- File Already Open

A File Already Open error will be generated when you try to OPEN a file using a buffer that corresponds to an already open file. Note that no error will be generated if the same file is in an open state using two different buffers at the same time (This practice is NOT advised).

Error 114 ---------- Disk I/O Error

A Disk I/O Error will occur when an input from or an output to a disk file is unsuccessful. A typical LDOS error message which is associated with the Disk I/O Error is a Parity Error.

Error 116 ---------- File Already Exists

A File Already Exists error will be generated when using an OPEN"xN" command if the file already exists.

Error 122 ---------- Disk Full

A Disk Full error will indicate that all of the free space on a disk has been consumed. In some cases) the occurrence of a disk becoming full (i.e. all of the disk space being consumed) may generate a Disk Write Protected error.

Error 124 ---------- Input Past End

The Input Past End error applies only to sequential files opened for input, and will occur when a read of the file is attempted after all data in the file has been input.

Error 126 ---------- Bad Record Number

A Bad Record Number error will be issued when record number 0 (or some other illegal record number) is accessed in a random file.

Error 128 ---------- Bad File Name

A Bad File Name error will be generated when the file specified in an OPEN, SAVE, LOAD, RUN or MERGE command does not conform to the rules governing valid LDOS filespecs.

Error 132 ---------- Direct Statement in File

A Direct Statement in File error will be generated when a LOAD is performed of a file that is not an LBASIC program (usually when a LOAD of a data file is attempted). This type of error will also be generated when an LBASIC program which was saved in ASCII is loaded, and a line in the program exceeds 240 characters in length.

Error 134 ---------- Too Many Files

The Too Many Files error will occur when an attempt is made to add another extent to a file when all directory entries have been used. This type of error will be very uncommon.

Error 136 ---------- Disk Write Protected

A Disk Write Protected error usually indicates that a write has been attempted to a write protected disk. Other types of errors may also generate a Disk Write Protected error. If the disk in question is not write protected, use CMD"E" to determine the exact error.

Error 138 ---------- File Access Denied

A File Access Denied error may be generated when a password protected file (either a data file or a program file) is referenced using an incorrect password.

Error 140 ---------- Blocked File Error

A Blocked File Error will occur if you attempt to OPEN a random file with an LRL of other than 256 after entering LBASIC and specifying the parameter BLK=OFF.

Error 142 ---------- System Command Aborted

A System Command Aborted error will occur if an LDOS command called by the CMD"command" function is manually aborted.

Error 144 ---------- Protection Has Cleared Memory

The Protection Has Cleared Memory error will be generated if an attempt is made to illegally access an EXECute only program without using the proper password. The program and variables will be cleared from memory.

LIMITED WARRANTY

The LBASIC manual is sold on an AS-IS basis. Every effort has been made to assure the correctness of the information in the LBASIC manual. Galactic Software, Ltd. Assumes no liability whatsoever, with regard to the reliability and/or fitness of this product for any application. Under no circumstances will Galactic Software, Ltd. Or its associates be held liable for the loss of TIME, DATA, PROGRAMS, or consequential damages incurred by the user as a result of information in the LBASIC manual.

FOR USER SUPPORT CALL .... 414 – 241-3080

First Edition LBASIC Model I/III