ACS-586 Computer System (Xenix Version)
MBASIC Interpreter Version 5.28
Release Notes Revision B
October 10, 1983

Installation Procedure

This version of MicroSoft BASIC (MBASIC) requires Xenix Version 2.3 or later.

To properly install MBASIC onto your system, please do the following:

1. Login as root
2. Enter: cd /tmp <cr>
3. Enter: umask 0 <cr>
4. Insert diskette into the floppy drive
5. Enter: tar xv <cr>
6. Enter: chmod 755 install <cr>

The next step is to run the installation shell script, "install". This places all MBASIC files into the appropriate directories with correct permission and owner modes. To issue "install" please do the following:

```
install <cr>
```

A test program has been supplied on your distribution diskette. To run this program please do the following:

```
mbasic test.bas <cr>
```

For further information, please refer to the README file supplied on your distribution diskette.

Notes

Memory Specifications

The MBASIC Interpreter Version 5.28 has a maxmimum 58,279 bytes of memory available for user programs. The default is 20,478 bytes.

```
mbasic <cr>

XENIX BASIC-86 V.5.28
[XENIX-86 REV. 1.00]
Copyright 1977-1983 (c) Microsoft
Created June 13, 1983
20478 Bytes free
```

PART NUMBER: 690-14628-002

To obtain the maximum available memory for user programs, the "-m" option must be specified on the command line.

```
mbasic -m 65000

XENIX BASIC-86 V.5.28
[XENIX-86 REV. 1.00]
Copyright 1977-1983 (c) Microsoft
Created June 13, 1983
58279 Bytes free
```

Known Problems

The statement OPTION BASE is not implemented at this time.

Microsoft BASIC

User's Guide
for 8086/8088
Microprocessors
and the XENIX
Operating System

LIMITED WARRANTY

Please send any comments about this documentation to:

Microsoft Corporation
Microsoft Building
10700 Northup Way
Bellevue, Washington 98004

MICROSOFT
NON-DISCLOSURE AGREEMENT AND REGISTRATION FORM

The party below agrees that it is receiving a copy of (COMPANY's trade name for licensed software) for use on a single computer only, as designated on this non-disclosure agreement. The party agrees that all copies will be strictly safeguarded against disclosure to or use by persons not authorized by (COMPANY) to use (COMPANY's trade name for licensed software), and that the location of all copies will be reported to (COMPANY) at (COMPANY's) request. The party agrees that copying or unauthorized disclosure will cause great damage to (COMPANY) and this damage is far greater than the value of the copies involved. The party agrees that this agreement shall inure to the benefit of any third party holding any right, title or interest in (COMPANY's trade name for licensed software) or any software from which it was derived.

Purchased From:

Company ____________________

Address ____________________

City, State, Zip ____________________

Phone ____________________

For Use On:

Model ____________________

Serial # ____________________

Software Product ____________________

Purchased By: (Distributor)

Name ____________________

Company ____________________

Address ____________________

City, State, Zip ____________________

Phone ____________________

Date ____________________

Purchased By: (Dealer)

Name ____________________

Company ____________________

Address ____________________

City, State, Zip ____________________

Phone ____________________

Date ____________________

Purchased By: (End-User)

Name ____________________

Company ____________________

Address ____________________

City, State, Zip ____________________

Phone ____________________

Date ____________________

NOTE: This Non-Disclosure Agreement **MUST** be signed by Party purchasing Product directly from (COMPANY). No Product will be shipped without signed agreement. It is the responsibility of Distributor and/or Dealer to transfer ownership to appropriate party.

Document No. 8108N-528-01
Part No.
Catalog No.

PACKAGE CONTENTS

Your Microsoft BASIC package contains:

One disk containing several files. One of these files is named "install.doc." The install.doc file describes the contents and use of each of the other files on the disk.

One binder containing the following documentation:

Microsoft BASIC User's Guide

The User's Guide contains all the information about Microsoft BASIC that is specific to your particular implementation or operating system. This includes a description of the features, statements, and functions affected by your implementation of Microsoft BASIC, information about disk file handling, and instructions on the use of assembly language subroutines.

Microsoft BASIC Reference Manual

The Reference Manual contains descriptions of all Microsoft BASIC features, statements, and functions. With the exceptions noted in the User's Guide, this information applies to all implementations of Microsoft BASIC.

Microsoft BASIC Quick Reference Guide

The Quick Reference Guide outlines MS-BASIC features and command syntax.

CONTENTS

INTRODUCTION

Microsoft BASIC (MS-BASIC) is the most extensive BASIC language available for microprocessors. It meets the requirements for the ANSI subset standard for BASIC, and supports many unique features rarely found in other BASICs. In addition, Microsoft BASIC (MS-BASIC) has sophisticated string handling and structured programming features that are especially suited for applications development. And, the Microsoft BASIC language is compatible with the Microsoft BASIC Compiler. Microsoft BASIC gives users what they want from a BASIC--ease of use plus the features that make a micro perform like a minicomputer or large mainframe.

In 1975, Microsoft wrote the first BASIC interpreter for the microcomputer. Today Microsoft BASIC, with over 1.5 million installations in over 20 operating environments, is recognized as the industry standard. It's the BASIC you'll find on all the largest-selling microcomputers. Many users, manufacturers, and software houses have written application programs in Microsoft BASIC.

MAJOR FEATURES

1. Four variable types: Integer (-32768 through +32767), String (up to 255 characters), Single Precision Floating Point (7 digits), Double Precision Floating Point (16 digits)

2. Extensive programming development features, including trace facilities (TRON/TROFF) for easier debugging; automatic line number generation and renumbering, including referenced line numbers (AUTO and RENUM); and program editing facilities (EDIT command and subcommands).

3. Error trapping using the ON ERROR GOTO statement

4. Device-independent I/O so that the syntax used to access disk files can also be used to access other devices

5. Lock facility (LOCK/UNLOCK), which allows you to restrict access to portions of a file or an entire file

6. PEEK and POKE statements to read and write memory locations associated with the user's program and data areas.

7. Arrays with multiple dimensions

8. Boolean operators OR, AND, NOT, XOR, EQV, IMP

9. Formatted output using the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, comma insertion

10. Assembly language subroutine calls are supported.

11. IF/THEN/ELSE and nested IF/THEN/ELSE constructs

12. MS-BASIC supports variable length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE

NOTE

Features vary from one implementation of Microsoft BASIC to the next. See Chapters 1 and 2 of this manual for an exact description of the features in your implementation of Microsoft BASIC.

SYSTEM REQUIREMENTS

Your implementation of Microsoft BASIC requires:

- 48K bytes of user memory minimum:
    - 32K for Microsoft BASIC
    - 16K for a Data Segment (BASIC will make use of up to 64K for a Data Segment, if available)
- 1 disk drive

Note: Systems running on the XENIX operating system must have a total of at least 196K.

If your system does not meet these minimum requirements, ask your computer dealer how to expand your system.

HOW TO USE THIS MANUAL

The "Introduction" to this User's Guide tells you about the major features of Microsoft BASIC, lists the system requirements, describes the syntax notation used in your MS-BASIC documentation, and provides references for learning BASIC programming.

Chapter 1 describes Microsoft BASIC as it is used with the XENIX operating system. This chapter tells you how to start-up and exit MS-BASIC, explains some conventions about filenaming and about writing programs for XENIX, and describes device-independent input/output under XENIX.

Chapter 2 describes all the statements and functions that are supported by this implementation of MS-BASIC but are not part of standard MS-BASIC. This includes statements and functions that are used only with the XENIX operating system, and statements and functions that are used differently in this implementation than in standard MS-BASIC.

Chapter 3 lists the changes that are necessary when you convert programs to Microsoft BASIC.

Chapter 4 explains disk file handling procedures.

Chapter 5, "Assembly Language Subroutines and Files," provides information about loading and calling assembly language subroutines.

SYNTAX NOTATION

The following notation is used throughout this manual in descriptions of command and statement syntax:

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
|	A vertical bar means the same thing as braces; i.e., it indicates a choice between two or more entries. At least one of the entries separated by a bar must be chosen unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

LEARNING MORE ABOUT BASIC

The manuals in this package provide complete reference information for your implementation of Microsoft BASIC. They do not, however, contain tutorial information on how to write programs in BASIC. If you are new to BASIC or need help in learning to program, we suggest you read one of the following books:

Are You Computer Literate? by Billings and Moursund, (Dilithium Press).

BASIC, by Robert L. Albrecht, LeRoy Finkel, Jerry Brown, (John Wiley and Sons, 1973).

Basic BASIC, by James Coan (Hayden Book Co.).

BASIC and the Personal Computer, by Thomas A. Dwyer and Margot Critchfield (Addison Wesley Publishing Co., 1978).

BASIC From the Ground Up, by David E. Simon (Hayden Book Co., 1978).

CHAPTER 1

MICROSOFT BASIC WITH THE XENIX OPERATING SYSTEM

This chapter presents general information you will need when you use MS-BASIC with the XENIX operating system. Included are the command lines for start-up and exit from MS-BASIC, information about writing programs for this implementation, and description of the input/output devices that are typically supported.

See Chapter 2 for a list of the language differences between this implementation and the standard MS-BASIC as described in the Microsoft BASIC Reference Manual.

1.1 HOW TO START-UP MS-BASIC

To run MS-BASIC, bring up XENIX and enter the following command:

```
mbasic [-s NN] [-f NN] [-m NN] [-l file] [-e] [file]
```

-s NN sets the maximum logical record size. The default is 128. NN must be an integer.

-f NN sets the maximum number of data files allowed. The default is 3. NN must be an integer.

-m NN sets BASIC's data space to this value. The default is 63470. NN must be an integer.

-l file causes assembly language programs to be loaded with MS-BASIC. Note that this option is the letter "el," not the number "one."

-e suppresses echo of MS-BASIC commands. This option may be useful when standard input and output are redirected.

file is the name of the MS-BASIC program to be run.

Once the start-up command line has been entered, the following sign-on banner will be displayed:

```
<product identification>
Version x.xx Copyright Microsoft Corporation 1983
xxxxxx Bytes free
```

If the <file> is specified in the command line, the sign-on banner will not be shown.

1.2 HOW TO EXIT MS-BASIC

To exit MS-BASIC and return to the XENIX operating system level, enter the command

```
SYSTEM
```

which closes all files and then returns control to XENIX.

1.3 CREATING AND EDITING MS-BASIC FILES

All files containing programs to run on MS-BASIC must be created using the MS-BASIC SAVE statement or must contain carriage returns and linefeeds (newlines) placed manually at the end of each logical line. This can be done with ed or tr (see XENIX documentation).

Filenames may be as long as the largest legal string variable. An example of a valid filename is:

```
/usr/inventory/june1980/generalstock
```

However, the operating system may impose a limit to filename length.

The line terminator for ASCII files under the XENIX operating system is the linefeed (newline) character. All carriage returns are mapped to the linefeed character. Either the linefeed or a carriage return will terminate an MS-BASIC line.

Continuing a single logical line across several physical lines is done by preceding the line terminator with the backslash (\) character. For example:

```
IF (I<0) THEN \
         PRINT "TRUE" \
ELSE \
         PRINT "FALSE"
```

The backslashes, typed before the carriage returns, cause these lines to be interpreted as a single line.

The backslash placed in any other position will retain its normal meaning as an integer divide.

The internal representation for line continuation used in binary files remains the same for all versions of MS-BASIC.

1.4 LANGUAGE DIFFERENCES

MS-BASIC for XENIX supports the following statements and functions which are not a part of standard MS-BASIC and are therefore not listed in the <u>Microsoft BASIC Reference Manual</u>:

BLOAD
BSAVE
DATE$
FILES
LOCK
LOF
SHELL
SYSTEM
TIME$
UNLOCK

In addition, the following statements and functions differ in this implementation from the descriptions in the <u>Microsoft BASIC Reference Manual</u>:

CALL
CLEAR
CLOAD/CSAVE
DEF USR
EOF
FIELD
INP
LIST
LLIST
LOC

LPOS
LPRINT
OPEN
OUT
PEEK
POKE
POS
USR
VARPTR
WIDTH

All these statements and functions are described, as they are implemented for MS-BASIC for XENIX, in Chapter 2 of this User's Guide.

1.5 DEVICE-INDEPENDENT I/O

MS-BASIC for XENIX supports device-independent input/output. This means that the syntax used to access disk files may also be used to access other devices. The devices that are supported depend upon the individual implementation of MS-BASIC. Typically, the devices that are supported are:

SCRN: Files may be opened to this device for output. All data written to a file which is opened to SCRN: is directed to the standard output device (i.e., the screen).

KYBD: Files may be opened to this device for input. All data read from a file which is opened to KYBD: comes from the standard input device (i.e., the keyboard).

LPT1: Files may be opened to this device for output. All data written to a file which is opened to LPT1: is directed to the line printer.

PIPE: Opening a file to device PIPE: opens a pipe, forks, and executes the specified child process. For example:

```
LIST "pipe:lpr"
```

would generate a listing to the lpr spooler.

```
OPEN "pipe:dir" FOR INPUT AS #1
```

would allow the directory to be read via file number 1.

```
    LIST 50,"PIPE:write davidt"
```

uses the XENIX write program to list line 50 on davidt's terminal.

For files opened to PIPE:, the LOC function returns 1 if any characters are ready to be read from the pipe. Otherwise, it returns 0.

For files opened to PIPE:, the LOF function always returns 0.

For files opened to PIPE:, the EOF function returns -1 (true) if no processes have the pipe opened for output and no data is available to be read from the pipe. Otherwise, it returns 0 (false).

CHAPTER 2

LANGUAGE DIFFERENCES FOR MS-BASIC WITH XENIX

Some of the statements and functions listed below do not appear in the *Microsoft BASIC Reference Manual*, because they are not part of all implementations of MS-BASIC. Others are listed in the *Reference Manual* but are used differently under this implementation. In all cases except the FIELD statement, where only a minor difference exists, a complete description is given here so that you do not have to go to the *Reference Manual* for additional information.

2.1 BLOAD STATEMENT

Because the BLOAD statement is not included in all versions of MS-BASIC, it is not listed in the Reference Manual.

The BLOAD statement allows a program or data that has been saved as a memory image file to be loaded in memory. A memory image file is a byte-for-byte copy of what was originally in memory. (See BSAVE for information about saving memory image files.)

A CLEAR statement must be performed before a BLOAD statement can be executed. Then the memory image file can be loaded into the portion of the Data Segment between the areas specified in <expression1> and <expression3> of the CLEAR statement.

BLOAD is often used for, but is not restricted to, loading assembly language programs. It may also be used to load compiled Microsoft (r) Pascal or Microsoft (r) FORTRAN routines, for example.

Format: BLOAD <filename> [,<offset>]

where <filename> is a string expression containing the device designation and filename. (The device designation is optional).

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535.

Purpose: To load the specified memory image file into memory.

Remarks: BLOAD observes the following rules:

1. If the device is omitted, the current drive is assumed.

2. If the offset is omitted, the offset contained in the file (i.e., the address specified by the BSAVE statement when the file was created) is used. Therefore, the file is loaded into the same location from which it was saved.

Examples

```
10 'Load subroutine at F000
30 BLOAD"PROG1",&HF000 'Load PROG1
```

This example loads PROG1 at address F000.

2.2 BSAVE STATEMENT

Because the BSAVE statement is not included with all versions of MS-BASIC, it is not listed in the Reference Manual.

The BSAVE statement allows data or programs to be saved as memory image files on disk. A memory image file is a byte-for-byte copy of what is in memory.

A CLEAR statement must be performed before a BSAVE can be executed. Then the memory image file can be saved in the portion of the Data Segment between <expression1> and <expression3> as specified in the CLEAR statement.

Format: BSAVE <filename>,<offset>,<length>

where <filename> is a string expression containing the device designation and filename. (The device designation is optional).

<offset> is a numeric expression returning an unsigned integer in the range 0 to 65535.

<length> is a numeric expression returning an unsigned integer in the range 1 to 65535. This is the length in bytes of the memory image file to be saved.

Purpose: To save the contents of the specified area of memory as a disk file.

Remarks: The <filename>, <offset>, and <length> are required in the syntax.

Example:

```
10 'Save PROG1
30 BSAVE"PROG1",&HF000,256
```

This example saves 256 bytes starting at F000 in the file PROG1.

2.3 CALL STATEMENT

The CALL statement is used to interface machine language programs with MS-BASIC. This explanation supplements the one in the Reference Manual.

Format: CALL <variable name># [(<argument list>)]

<variable name> is the name of the subroutine being called. On the first call, MS-BASIC obtains the address of the routine from the XENIX namelist of the executing version of MS-BASIC. MS-BASIC will place the segment and offset of the routine in the double precision variable <variable name>. On subsequent calls, the address will be obtained from this double precision variable.

Note that the double precision variable must not exist before the first call to the routine.

<variable name> may not be an array variable name.

<argument list> contains the variables or constants, separated by commas, that are to be passed to the routine.

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is the recommended way of calling 8086 machine language programs with Microsoft BASIC. See Chapter 5 for a complete description of how assembly language routines are called.

When a CALL statement is executed, control is transferred to the user's routine. Values are returned to MS-BASIC by including the variable name which will receive the result in the <argument list>.

Example:

```
100 CALL FOO#(A,B$,C)
```

Line 100 calls routine FOO. FOO was linked with MS-BASIC using the -l option. MS-BASIC will find the address of routine FOO by examining the appropriate namelist when FOO is first called. This address will then be stored in the double precision variable FOO#. On subsequent calls to FOO, MS-BASIC will retrieve the address of FOO from double precision variable FOO#.

2.4 CLEAR STATEMENT

For MS-BASIC with XENIX, a third expression has been added to the syntax and the meanings of expressions 1 and 2 have been changed:

Format:

```
CLEAR [,[<expression1>][,<expression2>]]
    [,<expression3>]
```

<expression1> indicates how many bytes are to be allocated for MS-BASIC variables, user program variables, user variables, and BLOAD/BSAVE space. This number may be as large as 64K. The default is the current allocated space.

<expression2> is the number of bytes to reserve for stack space. This number may be as large as 1024. The default is the currently allocated space. Upon entry into MS-BASIC, 1024 bytes are allocated.

<expression3> is the number of bytes to be used by MS-BASIC for program and variable space. <expression3> must be less than or equal to <expression1>. The default is <expression1>. Users may access the space between the data segment addresses given in <expression1> and <expression2> by using PEEK, POKE, BLOAD, or BSAVE.

Examples:

```
CLEAR
CLEAR ,32768
CLEAR ,,516
CLEAR ,32768,1024,32768
```

The first example accepts defaults for all three parameters. The second example sets the number of bytes to be allocated at 32768 and accepts the defaults for the other two parameters. The third example accepts the defaults for expressions 1 and 3, setting expression 2 at 516 bytes. The final example sets all three parameters.

2.5 CLOAD/CSAVE STATEMENTS

The CLOAD and CSAVE statements are not included in this implementation of MS-BASIC.

2.6 DATE$ FUNCTION

The DATE$ function retrieves the current date. This function is not included in the Reference Manual.

Format:	DATE$
Purpose	To retrieve the current date. (To set the date, use the DATE$ statement.)
Remarks:	The DATE$ function returns a ten-character string in the form mm-dd-yyyy, where mm is the month (01 through 12), dd is the day (01 through 31), and yyyy is the year (1980 through 2099).
Example:	10 PRINT DATE$ The DATE$ function prints the date, calculated from the date set with the DATE$ statement.

2.7 DATE$ STATEMENT

The DATE$ statement sets the current date. This statement is not included in the <u>Reference</u> <u>Manual</u>.

Format: DATE$=<string expression>

where <string expression> returns a string in one of the following forms:

```
mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy
```

Purpose: To set the date for the DATE$ function.

Example:

```
10 DATE$="01-01-1981"
```

The current date is set at January 1, 1981.

2.8 DEF USR STATEMENT

The DEF USR statement is not included with this implementation of MS-BASIC.

2.9 EOF FUNCTION

This description of the EOF function contains more detail than the one in the Reference Manual; otherwise, no changes have been made for this implementation.

Format: EOF(<file number>)

Purpose: Returns -1 (true if the end of a sequential file has been reached.

For random access files, EOF returns -1 (true) if the last GET attempted to read beyond the end of file.

Remarks: Use EOF to test for end of file while inputting, to avoid "Input past end" errors.

This function is valid for any file which is opened for input. A file opened to KYBD: is at its end when <Control-D> is pressed.

Example:

```
130 IF EOF(2) THEN 50
```

2.10 FIELD STATEMENT

The FIELD statement is described in detail in the <u>Microsoft BASIC Reference Manual</u>. The only difference for this implementation is that field definitions are ignored after a file is closed.

2.11 FILES COMMAND

The FILES command lists the names of the files on a specified disk. This command is not included in the Reference Manual.

Format: FILES [<filename>]

where <filename> includes a filename and optional device designation.

Purpose: To print the names of files residing on the specified disk.

Remarks: If <filename> is omitted, all the files in the currently selected directory will be listed. <filename> is a string formula which may contain wildcard characters as defined by XENIX.

Note: The FILES statement is identical to the statement:

```
SHELL "ls-l
```

FILES statement parameters are concatenated to this SHELL command. For example:

```
FILES "*.BAS
```

is identical to SHELL "ls-l *.BAS

Examples:

```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "B:"  (equivalent to "B:*.*")
FILES "TEST?.BAS"
```

2.12 INP FUNCTION

The INP function is not supported by this implementation of MS-BASIC.

2.13 LIST STATEMENT

A <filename> parameter has been added to the LIST statement for this implementation of MS-BASIC.

Format:

```
LIST [[<line number>[-[<line number>]]]
   [,<filename>]]
```

<line number> is in the range 0 to 65529.

<filename> is the name of the file where the listing will be placed.

Remarks: If the optional filename specification is omitted, the specified lines are listed to the screen. If the line range is omitted, the entire program is listed.

When the dash (-) is used in the line range, three options are available:

1. If only the first number is given, that line and all higher numbered lines are listed.
2. If only the second number is given, all lines from the beginning of the program through the given line are listed.
3. If both numbers are given, the inclusive range is listed.

Examples:

```
LIST ,"LPT1:"
```

lists the program to the line printer.

```
LIST 10-20
```

lists lines 10 through 20 to the screen.

```
LIST 10- ,"SCRN:"
```

lists lines 10 through last to the screen.

2.14 LLIST STATEMENT

With most implementations of MS-BASIC, LLIST sends output to the lineprinter. Under XENIX, LLIST sends output to the lineprinter's spooler, and the output is not actually printed until an END, SYSTEM, or CLEAR statement with parameters is executed.

Format: LLIST [<line number>[-[<line number>]]]

Purpose: To send all or part of the program currently in memory to the lineprinter's spooler.

Remarks: Output will not be printed until an END, SYSTEM, or CLEAR with parameters is executed. (CLEAR with no parameters does not release spooled output.)

If two files are opened to LPT1:, their output is not intermixed, but is spooled separately. Their output is not intermixed with LPRINT output.

LLIST assumes a 132-character printer.

The options for LLIST are the same as for LIST.

Examples: See LIST.

2.15 LOC FUNCTION

With MS-BASIC for XENIX, the LOC function performs differently than described in the <u>Reference Manual</u>.

Format: LOC(<file number>)

where <file number> is the number under which the file was opened.

Purpose: With random access files, LOC returns the number of the last record read or written to the file.

With sequential files, LOC returns the number of records read from or written to the file since it was opened.

For files opened to KYBD:, LOC returns 1 if any characters are ready to be read from the standard input device. Otherwise, it returns 0.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

If the file is sequential, this statement ends program execution if more than 50 sectors have been written to or read from the file since it was opened.

If the file is random access, this statement ends program execution if the current record number is higher than 50.

2.16 LOCK STATEMENT

The LOCK statement is not included in some versions of MS-BASIC, and therefore is not listed in the Microsoft BASIC Reference Manual.

Format:

```
LOCK [#]n [,READ][,WAIT][,[<record number>]
   [ TO <record number>]]
```

See "Remarks" for discussion of parameters.

Purpose: To restrict access to specified portions of a file.

Remarks: LOCK will restrict access by other programs to file number n in the <record number> range or for the entire file. If file number n has been opened for sequential input or output, the entire file is locked and any range is ignored.

If the file is opened in random mode, the range specifies the records to be locked. If a starting record number is not specified, record number 1 is used. A terminating record number must be specified when a range is used.

Locks can be partial or total. A total lock will prevent any access by another program to the locked portion of the file. A partial lock will allow reading by another program but will prevent any modification of the locked region of the file. A partial lock is applied to the file by specifying the READ option. A total lock is the default; it is specified by not specifying READ.

If another program has locked a portion of the requested file, two options are available:

1. The first option is to return control to the program immediately with an accompanying error message. All the standard MS-BASIC error handling facilities can be used to trap and examine this error. If error trapping is not active, the error message will be "Permission denied."

   This is the default option. It is chosen if the WAIT option is not specified in the LOCK statement.

2. The second option is to wait until the program that issued the original lock unlocks the requested region of the file. The presence of the WAIT option in the LOCK statement will force this action. It is possible to interrupt this wait by pressing <Control-C> and then resume waiting by typing CONT.

   It is possible to get into a deadlock situation when waiting for a lock request. For example, assume that a program has opened file A and has locked and executed a LOCK request with the WAIT option on file B. A deadlock will occur if another program has already locked file B and has executed a LOCK statement with the WAIT option against file A.

   XENIX will attempt to detect any deadlock situations. If one is detected, a "Deadlock" error message is returned by MS-BASIC.

   Multiple LOCK statements have a cumulative effect. Locking records 1 through 3 and then locking 10 through 100 will leave 1 through 3 and 10 through 100 locked.

   Locking a record that is already locked will have no effect; the record will remain locked, and no error will be generated.

   If a record is locked more than once with different locking options, the options specified in the last LOCK statement will apply.

Note: We recommend that you do not open a single file on multiple channels simultaneously. If it becomes necessary to open a single file more than once, be aware that: Locks against the same file using different channels will act the same as multiple LOCK statements issued against a single file. For example:

```
10 OPEN "r",1,"foobar"
20 OPEN "r",2,"foobar"
30 LOCK #1,1-3
40 LOCK #2,READ,2-5
```

will leave record 1 through 5 of foobar locked. In addition, records 2 through 5 will be locked in READ mode.

Locking is always done on a per file basis, not on a channel number basis. Therefore, the first channel that is closed will release all locks against the file, including locks made using another channel number. If file #2 is closed in the above example, all locks are released, including records 1 through 3 that were locked for file #1.

Records are locked based upon their positions and sizes in the file. The unit of measure is the byte. If a file is opened more than once with different record sizes and if locks are then applied due to cumulative locking, portions of a record can inadvertently become locked.

Example:

```
10 LOCK #2,WAIT,3-50
```

Lines 3 through 50 of file #2 are locked; if another program has already locked this area of the file, MS-BASIC will wait until the original lock has been removed.

2.17 LOF FUNCTION

Since this function is not included with all versions of MS-BASIC, it is not listed in the Reference Manual.

Format: LOF(<file number>)

<file number> is the number of the file to be searched.

Purpose: To return the number of bytes in the file.

Remarks: LOF is valid for any file. Files opened to LPT1:, KYBD:, SCRN:, or PIPE: always return 0.

Example: 20 PRINT LOF(3)

Returns the number of bytes in file number 3.

2.18 LPOS FUNCTION

The action performed by this function differs from that described in the Microsoft BASIC Reference Manual.

Format: LPOS(X)

X is a dummy argument.

Purpose: To return the column position for device LPT1:.

Example: `100 IF LPOS(X)>60 THEN PRINT CHR$(13)`

If the column position is higher than 60, the character represented by CHR$(13) will be printed.

2.19 LPRINT STATEMENT

Under MS-BASIC for XENIX, output sent to the printer with the LPRINT statement is spooled, rather than printed immediately.

Format: LPRINT [<list of expressions>]

<list of expressions> is a string of numeric or string expressions that are to be printed, separated by semicolons.

Purpose: To print data at the line printer.

Remarks: LPRINT is the same as PRINT except that the output goes to the line printer.

Output will not actually be printed until an END, SYSTEM, or CLEAR statement with parameters is executed (CLEAR with no parameters does not release spooled output).

Example:

```
30 PRINT X "SQUARED IS" X^2
```

where X=9, prints "9 SQUARED IS 81".

2.20 OPEN STATEMENT

Although the OPEN statement described in the <u>Reference Manual</u> remains valid, a new format makes the statement more readable and more comprehensive. In the new format, the <filename> is moved forward, the LEN keyword is added, and the APPEND option is added.

Format:

```
OPEN <filename> [FOR <mode>] AS
  [#]<file number>[LEN=<record length>]
```

<filename> and <file number> identify the file that is to be opened. The <file number> must be in the range 1 to 15. This number is associated with the file as long as it is open and refers other disk I/O statements to the file.

<mode> can be one of the following:

INPUT	Specifies sequential input mode.
OUTPUT	Specifies sequential output mode.
APPEND	Specifies sequential output mode and sets the file pointer at the end of the file and the record number as the last record of the file. A PRINT# or WRITE# statement will then extend the file.

The <mode> can be abbreviated by the first letter: I, O, or A.

If <mode> is omitted, the default random access mode is assumed.

<record length> sets the record length for random access files. Do not use this option with sequential files.

The record length cannot exceed the maximum set with /S: at start-up (the default for /S: is 128 bytes). If the <record length> is omitted, the default length of 128 bytes is assumed.

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be opened before any disk I/O operation can be performed on it. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

A file can be opened for sequential input or random access on more than one file number at a time. A file may be opened for output, however, on only one file number at a time.

Example:

```
10 OPEN "MAIL.DAT" FOR APPEND AS #1
```

2.21 OUT STATEMENT

The OUT statement is not supported by this implementation of MS-BASIC.

2.22 PEEK FUNCTION

The following explanation of the PEEK function is more detailed than the one in the Reference Manual.

Format: PEEK(I)

I specifies the memory location to be read. For the interpretation of a negative value of I, see Section 2.32, "VARPTR."

You can PEEK into the portion of the Data Segment from the beginning of the user's program to the last byte managed by MS-BASIC. You are not allowed to PEEK into the area reserved for the Code Segment or MS-BASIC control variables. An attempt to do so will result in a "Permission denied" error.

Purpose: To return the byte (decimal integer in the range 0 to 255) read from memory location I.

Remarks: PEEK is the complementary function of the POKE statement.

Example:

```
10 A = PEEK(&H5A00)
```

Returns the byte stored in location &H5A00 and assigns it to variable A.

2.23 POKE STATEMENT

This description of the POKE statement is more detailed than the one in the Reference Manual.

Format: POKE I,J

I is the memory location where the byte is to be placed. For information about the interpretation of negatives values, see VARPTR.

J is the byte that is to be stored. It must be a decimal integer in the range 0 to 255.

Purpose: To store byte J in memory location I.

Remarks: MS-BASIC generates a "Permission denied" error if the memory location is outside the user-writable data space (which includes the user variables, strings, BLOAD/BSAVE area, and File Data Blocks). I.e., you are not allowed to POKE into the area reserved for the code segment or MS-BASIC control variables.

Example:

```
10 POKE &H5A00,&HFF
```

Data byte &HFF is placed in memory location &H5A00.

2.24 POS FUNCTION

The explanation of POS in the Reference Manual is valid except that in this implementation the device is always the screen.

Format: POS(I)

I is a dummy argument.

Purpose: To return the current column position of the cursor on the screen. The leftmost position is 1.

Example:

```
20 IF POS(I)>60 THEN PRINT CHR$(13)
```

If the cursor is at a column number higher than 60, the character represented by CHR$(13) will be printed.

2.25 SHELL FUNCTION

The SHELL function is not included in all versions of MS-BASIC and therefore is not listed in the Reference Manual.

Format: SHELL(<string expression>)

<string expression> is a command that is to be executed by sh.

Purpose: To start an asynchronous (child) process and return the process id.

Remarks: The child process executes sh, which in turn executes the command passed in <string expression>. MS-BASIC resumes execution immediately, without waiting for the child process to terminate.

2.26 SHELL STATEMENT

The SHELL statement is not included with all versions of MS-BASIC and therefore is not listed in the Reference Manual.

Format:	SHELL[<string expression>]
	<string expression> is an instruction to be executed by sh.
Purpose:	To start a child process which executes sh.
Remarks:	MS-BASIC does not resume execution until the child process terminates. If an argument is included, sh then executes the argument. If no argument is included, sh waits for commands from standard input.

2.27 SYSTEM COMMAND

The SYSTEM command is not used with all implementations of MS-BASIC, and therefore is not described in the Reference Manual.

Format:	SYSTEM
Purpose:	To close all files and reload the operating system into memory.
Remarks:	SYSTEM closes all files and returns control to XENIX.

2.28 TIME$ FUNCTION

Because this function is not included in all implementations, it is not included in the Reference Manual.

Syntax: TIME$

Purpose: To retrieve the current time. (To set the time, use the TIME$ statement.)

Remarks: The TIME$ function returns an eight-character string in the form hh:mm:ss, where hh is the hour (00 through 23), mm is minutes (00 through 59), and ss is seconds (00 through 59). A 24-hour clock is used; 8:00 p.m., therefore, would be shown as 20:00.

Example:

```
10 PRINT TIME$
```

Prints the time, calculated from the time set with the TIME$ statement.

2.29 TIME$ STATEMENT

The TIME$ statement sets the current time. This statement is not included in the Reference Manual.

Format: TIME$=<string expression>

where <string expression> returns a string in one of the following forms:

hh	(sets the hour; minutes and seconds default to 00)
hh:mm	(sets the hour and minutes; seconds default to 00)
hh:mm:ss	(sets the hour, minutes, and seconds)

Purpose: To set the time for the TIME$ function.

Remarks: A 24-hour clock is used; 8:00 p.m., therefore, would be entered as 20:00:00.

Example:

```
10 TIME$="08:00:00"
```

The current time is set at 8:00 a.m.

2.30 UNLOCK STATEMENT

Because the UNLOCK statement is not included in all implementations of MS-BASIC, it is not described in the <u>Reference Manual</u>.

Format: UNLOCK [#]<file number> [<record number>] [TO <record number>]

<file number> is the number of the file to be unlocked.

The <record number> options set the range of records to be unlocked.

Purpose: To release access restrictions placed on a file by the LOCK statement.

Remarks: If a record number or range is specified and the file is opened in random mode, only those records in the range are unlocked.

Example:

```
30 UNLOCK #1,3 TO 50
```

Releases a previous LOCK on records 3 through 50 in file number 1.

2.31 USR FUNCTION

The USR function is not supported under this implementation of MS-BASIC.

2.32 VARPTR FUNCTION

The only difference between the desription given here for VARPTR and the one in the Reference Manual is the addition of the "Note" in the portion on Format 2.

Format 1: VARPTR(<variable name>)

Format 2: VARPTR(#<file number>)

Purpose: Format 1

Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise, an "Illegal function call" error results. Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned. (See Chapter 5, "Assembly Language Subroutines," for discussion of the string descriptor.) The address returned will be an integer in the range -32768 to 32767. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Note: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2

For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

Note: If this function is used within an MS-BASIC program, it should always be used immediately before its value is used. This is necessary because closing another file may cause the file data buffer for this file to be moved in memory.

Example:

```
20 X = USR(VARPTR(Y))
```

Returns the address of the first byte of data identified with variable Y.

2.33 WIDTH STATEMENT

The format and comments in the _Reference Manual_ remain valid. Note that Format 2 below is equivalent to the format

```
     WIDTH[LPRINT]<integer expression>
```

(the format given in the _Reference Manual_) when the <device> is the lineprinter.

Format 1: WIDTH <file number>,<size>

Format 2: WIDTH <device>,<size>

<file number> is the number of the MS-BASIC file that is being output.

<size> is the desired line width, in characters. The <size> must be a valid numeric expression returning an integer in the range 1 to 255.

<device> identifies the device where the output is sent.

Purpose: Format 1

Changes the line width of file <file number> to <size>. This means that if more than <size> bytes are output on one line to the file, a newline will be forced.

Format 2

Has no effect on files currently opened to that device. A subsequent OPEN <device> FOR OUTPUT AS <file number> will use this value for width while the file is open.

Remarks: Valid width for all devices is 1 to 255. Any value outside these ranges will result in an "Illegal function call" error, and the previous value will be retained.

Specifying WIDTH 255 disables line folding, thus giving the effect of infinite width.

Examples:
```
10 WIDTH 3,132
10 WIDTH "LPT1:",132
```

CHAPTER 3

CONVERTING PROGRAMS TO MICROSOFT BASIC

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before running them with MS-BASIC. This chapter lists some specific things to look for.

3.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Microsoft BASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Microsoft BASIC string concatenation.

In Microsoft BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	Microsoft BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	Microsoft BASIC
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

3.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
     10 LET B=C=0
```

to set B and C equal to zero. Microsoft BASIC would interpret the second equals sign as a logical operator and set B equal to -1 if C equalled 0. Instead, convert this statement to two assignment statements:

```
     10 C=0:B=0
```

3.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With Microsoft BASIC, be sure all statements on a line are separated by a colon (:).

3.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

CHAPTER 4

MICROSOFT BASIC DISK I/O

Disk I/O procedures for the beginning MS-BASIC user are examined in this chapter. If you are new to MS-BASIC or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames.

4.1 PROGRAM FILE COMMANDS

The following list reviews the commands and statements used in program file manipulation.

SAVE <filename>[,A]	Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, MS-BASIC uses a compressed binary format.) (See Section 4.2 below for discussion of how to save protected files.)
LOAD <filename>[,R]	Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files. (LOAD <filename>,R and RUN <filename>,R are equivalent.)

RUN <filename>[,R]	RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <filename>,R and LOAD <filename>,R are equivalent.)
MERGE <filename>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and MS-BASIC returns to command level.
KILL <filename>	Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.
NAME <oldfilename> AS <new filename>	To change the name of a disk file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

4.2 PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
     SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

4.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

4.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data written to a sequential file are ASCII characters. They are stored, one item after another (sequentially), in the order they are sent. They are read back in the same way.

The statements and functions that are used with sequential files are:

CLOSE
EOF
INPUT#
LINE INPUT#
LOC
LOCK
OPEN
PRINT#
PRINT# USING
UNLOCK
WRITE#

See the Microsoft BASIC Reference Manual for detailed descriptions of these statements and functions.

Creating a Sequential File

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.	`OPEN "DATA" FOR "O" AS #1`
2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.)	`PRINT#1,A$;B$;C$`

3. To access the data in the file, you must CLOSE the file and reopen it in "I" mode.	`CLOSE #1`
	`OPEN "DATA" FOR "I" AS #1`

4. Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,X$,Y$,Z$`

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
     PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was opened. For example,

```
100 IF LOC(1)>50 THEN STOP
```

would end program execution if more than 50 sectors had been written to or read from file #1 since it was opened.

Program 1 is a short program that creates a sequential file, "DATA", from information you input at the terminal:

Program 1 - Create a Sequential File

```
10 OPEN "DATA" FOR "O" AS #1
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

Program 2 accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978:

Program 2 - Access a Sequential File

```
10 OPEN "DATA" FOR "I" AS #1
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

Adding Data to a Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. Instead, use the OPEN statement with the APPEND mode, as described in Section 2.20.

4.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because MS-BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

CLOSE
CVD/CVI/CVS
FIELD
GET
LOC
LOCK
LSET/RSET
MKD$/MKI$/MKS$
OPEN
PUT
UNLOCK

See the <u>Microsoft BASIC Reference Manual</u> for detailed discussion of these statements and functions.

<u>Creating a Random File</u>

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.	`OPEN "FILE" FOR "R" AS #1 LEN=3`
2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.	`FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$`
3. Use LSET to move the data	`LSET N$=X$`

into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.	LSET A$=MKS$(AMT) LSET P$=TEL$
4. Write the data from the buffer to the disk using the PUT statement.	PUT #1,CODE%

The LOC function with random files returns the current record number. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

Program 3 writes information that is input at the terminal to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a fielded string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Program 3 - Create a Random File

```
10 OPEN "FILE" FOR "R" AS #1 LEN=32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Accessing a Random File

The following program steps are required to access a random file:

1. OPEN the file in "R" mode. `OPEN "FILE" FOR "R" AS #1 LEN=32`

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file. `FIELD #1 20 AS N$, 4 AS A$, 8 AS P$`

NOTE

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.	GET #1,CODE%
4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.	PRINT N$ PRINT CVS(A$)

Program 4 accesses the random file "FILE" that was created in Program 3. When the three-digit code is input at the terminal, the information associated with that code is read from the file and displayed:

Program 4 - Access a Random File

```
10 OPEN "FILE" FOR "R" AS #1 LEN=32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

Program 5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

Program 5 - Inventory

```
120 OPEN "INVEN.DAT" FOR "R" AS #1 LEN=39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
         "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
         IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
```

```
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
         " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
          CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
         GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

CHAPTER 5

MS-BASIC ASSEMBLY LANGUAGE SUBROUTINES

This chapter explains how to load assembly language subroutines and how to call them with the CALL or CALLS statements.

5.1 LOADING ASSEMBLY LANGUAGE SUBROUTINES

Under the XENIX operating system, memory space for MS-BASIC is separated into instruction and data space. Any assembly language subroutine will reside in the instruction space and will use the data space. If you use self-modifying code, note that this separation may prevent self-modification.

To link and load files that contain assembly language subroutines with MS-BASIC, start by using the -l ("el") option with the MS-BASIC command line. The assembly language file must be a relocatable file.

Only one assembly language file can be loaded per -l option. If more than one file is needed, use additional -l options. For example, the following syntax would assemble and run MS-BASIC with the assembly language files asml.s and asm2.s:

```
     as -o asml.o asml.s
     as -o asm2.o asm2.s
     mbasic -l asml.o -l asm2.o
```

When the -l option is used, the relocatable MS-BASIC library is linked and loaded with all the specified assembly language files and the "C" library; and a local file named "basic" is produced and executed. This means that a linked version of the assembly language file plus MS-BASIC resides in the user's current directory.

A new copy of this linked file is created in the current directory each time MS-BASIC is executed with the -l option. However, if you do not need to re-link the file when you want to run it again, you can use the already-existing file (the one created by the most recent link) by invoking MS-BASIC using the name of the local file rather than "mbasic." In this case, do not include -l in the command line. The local filename will be /usr/current directory/basic. Often the user profile is set to search for local copies of a file first. If this is the case, specifying basic will suffice.

If no -l option appears in the command line, the assembly language file will not be linked with MS-BASIC, and the copy of MS-BASIC that resides in /usr/bin will be executed.

In summary, MS-BASIC with XENIX creates a local version of MS-BASIC when assembly language subroutines are to be interfaced with MS-BASIC. If these routines are to be called during MS-BASIC program execution, the local version of MS-BASIC must be executing. It will execute as a result of the -l option being included with the command line, or if a local version of MS-BASIC is executed after having originally been created by the command line.

5.2 CALLING ASSEMBLY LANGUAGE SUBROUTINES

Assembly language subroutines are called from within a program by using the CALL or CALLS statement. Most calls are made with the CALL statement; CALLS works much the same way but is used to access MS-FORTRAN routines.

5.2.1 CALL STATEMENT

Format: CALL <variable name># [(<argument list>)]

<variable name> is a double-precision variable that names the subroutine being called.

Note that certain language processors create names beginning with the underline character (_). This is not a legal MS-BASIC variable name. Therefore, MS-BASIC searches for two entry points. The preferred entry point is the same as the <variable name>. If this entry point is not found, MS-BASIC will search for the entry point constructed by using the underline character.

MS-BASIC does not preserve case. Therefore, all entry point names must be capitalized.

On the first invocation of the CALL <variable name># statement, MS-BASIC will obtain the 32-bit entry point address from the name list of the executing MS-BASIC (see XENIX documentation for "nlist(3)"). MS-BASIC will then store the 32-bit address in <variable name>. Since 32-bits of precision requires a double precision variable, "Type mismatch error" will be displayed for any other variable type.

Note that the double-precision variable <variable name> must not be defined before the first call to the routine.

<argument list> contains the variables that are passed to the external subroutine.

No previously unreferenced scalar variable may follow an array element in the <argument list>. If this is attempted, an "Illegal function call" error will result.

Invoking the CALL statement causes the following to occur:

1. For each argument in the argument list, the 2 byte offset of the argument's location within the data segment (DS) is pushed onto the stack.

2. MS-BASIC's return address code segment (CS), and offset (IP) are pushed onto the stack.

3. Control is transferred to the user's routine.

These actions are illustrated by the two following diagrams, which illustrate first, the state of the stack at the time of the CALL statement, and second, the condition of the stack during execution of the called subroutine.

```
                ------------------------
  high         |                        |
addresses      |  argument 0            |
     |         |  argument 1            |
     |         |      .                 |     Each argument is a 2-byte
     |         |      .                 |       POINTER into memory
     | C       |      .                 |
   S | o       |   argument  n          |
   t | u       |                        |
   a | n       +------------------------+
   c | t       | return segment address |
   k | e       +------------------------+
     | r       | return offset          |
     |         +------------------------+ <--Stack pointer
     V         |                        |       (SP reg. contents)
    low        |                        |
addresses      |                        |

=========================================================
        Stack layout when CALL statement
                  is activated
```

The subroutine now has control. Arguments may be referenced by moving the Stack pointer (SP) to the Base Pointer (BP) and adding a positive offset to (BP).

```
                    ------------------------
  high             |                        |
addresses          |   argument  0          |\
    |              |   argument  1          | \
    |              |       .                |  }<--Absent if any argumen
    |              |       .                | /     referenced within a
    |C             |       .                |/      procedure
   S|o             |   argument  n          |
   t|u             |                        |
   a|n             +------------------------+
   c|t             | return segment address |}<--Absent in local procedu
   k|e             +------------------------+
    |r             | return offset          |
    |              +------------------------+ <--Stack pointer
    |              |    old stack marker    |      (SP reg. contents)
    |              +------------------------+ <--New stack marker
    |              |   local variables      |\
    |              |      .                 | \
    |              |      .                 |  }<--Only in reentrant pro
    |              |      .                 | /
    |              |                        |/
    |              +------------------------+
    |              |   This space may be    |
    |              |   used during pro-     |   Stack pointer may change
    |              |   cedure execution     |     during procedure execu
    |              |      .                 |
    |              |      .                 |
    |              |      .                 |
    V              |                        |
   low             |                        |
addresses          |                        |

=================================================
        Stack layout during execution of
              of a CALL statement
```

You must observe the following rules when coding a subroutine:

1. The called subroutine may destroy the AX, BX, CX, and DX registers. All other registers must be preserved.

2. The called subroutine must know the number and length of the arguments passed. References to arguments are positive offsets added to (BP) (assuming the called routine moved the current stack pointer into BP; i.e., MOV BP,SP). That is,

the location of p1 is at 8(BP), p2 is at 6(BP), p3 is at 4(BP), ...etc.

3. The called subroutine must perform an intersegment return. On exit from the subroutine, arguments may be left on the stack. If arguments are left on the stack, MS-BASIC will detect and remove them.

4. Values are returned to MS-BASIC by including in the argument list the variable name which will receive the result.

5. If the argument is a string, its offset points to 3 bytes called the "String Descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

IMPORTANT

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
20 A$ = "BASIC"+""
```

This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

6. Strings may be altered by user routines, but the length <u>MUST NOT</u> be changed. MS-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

The following sequence of 8086 assembly language demonstrates access of the arguments passed and storing a return result in the variable 'C'.

```
MOV     BP,SP    ;Get current Stack posn in BP.
MOV     BX,6[BP];Get address of B$ dope.
MOV     CL,[BX] ;Get length of B$ in CL.
MOV     DX,1[BX];Get addr of B$ text in DX.
        .
        .
        .
```

```
MOV     SI,8[BP];Get address of 'A' in SI.
MOV     DI,4[BP];Get pointer to 'C' in DI.
MOVS    WORD    ;Store variable 'A' in 'C'.
RET     6       ;Restore Stack, return.
```

IMPORTANT

The called subroutine must know the variable type for numeric arguments passed. In the above example, the instruction

```
MOVS WORD
```

will copy only 2 bytes. This is fine if variables A and C are integers. We would have to copy 4 bytes if they were Single Precision and copy 8 bytes if they were Double Precision.

When the call is made, register [AL] contains a value which specifies the type of argument that was given. The value in [AL] may be one of the following:

2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [BX] register pair points to the Floating Point Accumulator (FAC) where the argument is stored:

FAC	is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.
FAC-1	contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

If the argument is an integer:

FAC-2	contains the upper 8 bits of the argument.
FAC-3	contains the lower 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-2	contains the middle 8 bits of mantissa.
FAC-3	contains the lowest 8 bits of mantissa.

If the argument is a double precision floating point number:

FAC-7 - FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string:

the [DX] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in BASIC's Data Segment.

IMPORTANT

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. See the CALL statement above.

5.2.2 CALLS Statement

The CALLS statement should be used to access FORTRAN subroutines. CALLS works just like CALL except that with CALLS, each of the arguments on the stack is a 4-byte pointer into memory, rather than a 2-byte pointer.

The CALLS statement is not available with all implementations of MS-BASIC and therefore is not discussed in the Reference Manual.

INDEX