M T S

The Michigan Terminal System

Volume 10: BASIC in MTS

by

Edward J. Fronczak

and

Clark E. Lubbers

December 1980

The University of Michigan Computing Center
Ann Arbor, Michigan

```
**************************************************
*                                                *
*   This obsoletes the September 1974 edition.   *
*                                                *
**************************************************
```

DISCLAIMER

The MTS Manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the _Computing Center Newsletter_, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

December 1980

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed processor computer can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center and the physical facilities provided are described in a separate publication entitled _Introduction to the Computing Center_.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are updated by means of CCMemos, users should check the Memo List, copy the files *CCMEMOS or *CCPUBLICATIONS, or watch for announcements in the _Computing Center Newsletter_, to ensure that their MTS volumes are fully up to date.

Volume 1: _The Michigan Terminal System_, December 1979
Volume 2: _Public File Descriptions_, December 1978
Volume 3: _System Subroutine Descriptions_, October 1976
Volume 4: _Terminals and Tapes_, November 1980
Volume 5: _System Services_, April 1980
Volume 6: _FORTRAN in MTS_, December 1978
Volume 7: _PL/I in MTS_, July 1977
Volume 8: _LISP and SLIP in MTS_, June 1976
Volume 9: _SNOBOL4 in MTS_, September 1975
Volume 10: _BASIC in MTS_, December 1980
Volume 11: _Plot Description System_, August 1978
Volume 12: _PIL/2 in MTS_, December 1974
Volume 14: _360/370 Assemblers in MTS_, August 1978
Volume 15: _FORMAT and TEXT360_, April 1977
Volume 16: _ALGOL W in MTS_, September 1980
Volume 17: _Integrated Graphics System_, December 1980

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury

General Editor

December 1980

PREFACE TO REVISED VOLUME 10

"When you wish to produce a result by means of an instrument, do not allow yourself to complicate it."

Leonardo da Vinci

Revised Volume 10 of the MTS manuals represents a complete documentation of the University of Michigan BASIC System as of this date. It documents all aspects of the system as a text and reference, and attempts to be as self-sufficient as possible since BASIC is meant to be a self-sufficient subsystem of MTS. Since BASIC is terminal-oriented, the user is referred to MTS Volume 1, MTS and the Computing Center, for coverage of conversational use of terminals above and beyond the simple teletype usage presented in Appendix F in Volume 10. To assist the user, both a structural overview and an extensive table of contents have been provided, along with a large alphabetical subject index. Various charts, tables, sample programs, etc. are indexed in the List of Examples. The appendices at the back of the manual contain useful summaries, complete sample programs run at a terminal, and such information as a list of frequently used constants.

Only a small portion of this volume is needed to compose and run simple BASIC programs. Sections I, II, III, III.1, IV.A-IV.B, V.A-V.C, V.I, IX, Appendix F (teletype usage), and the sorting and plotting examples in Appendix O provide a good base.

The changes to the September 1974 Volume 10 for this revised volume are summarized on the next page.

Special thanks to Dr. R. C. F. Bartels, Gail Lift, and Kathy Young for helping to proofread this document.

Edward J. Fronczak
Clark E. Lubbers

CHANGES FOR THE REVISED EDITION

The following changes have been made since the September 1974 edition.

The BASIC command prefix character, "%", has been changed to a slash, "/", so as not to conflict with "%" as a device command prefix.

A new parameter, SIGDIGITS, has been added to the /SET command to allow the user to control the number of significant digits printed for numeric output.

A new command, /MTSCMD, has been add to allow the user to issue MTS commands from BASIC. Also MTS commands with the $ prefix are supported.

Table of Contents

List of Examples

I WHAT IS BASIC?

BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code. The language was developed at Dartmouth College under the direction of Professors John G. Kemeny and Thomas E. Kurtz to enable persons with little or no computer science background to write programs in a way that resembles standard mathematical notation. It is a simple and flexible language that practically anyone can use.

BASIC at the University of Michigan evolved from the computer project design course CCS 673 in the Department of Computer and Communication Sciences during the Winter term of 1969. It was felt that a need existed for an inexpensive, user-oriented, interactive language which could be used not only for computer education but for research as well. The BASIC language was chosen for its suitability for meeting these goals. An existent system (see reference 2, Section XIV) was chosen which contained a rich mixture of general computational facilities, matrix operations, character string handling, and file input/output. The result was to be a file-oriented system which allowed temporary and permanent storage of programs or data. Also desirable was a simple but comprehensive facility for assisting the user in debugging program errors. The overall efficiency of the U of M BASIC System results in part from the design of the "virtual file", a file which resides in the computer's fast memory while being used so that it is capable of being more readily accessed than files stored on slower, auxiliary storage devices. The virtual file may be saved on an auxiliary device when it is no longer needed. Hence, all file-dependent operations such as program and data generation, program translation, and file text-processing could be optimized. At the termination of the design course a system nucleus realizing a part of the original design was achieved. With experience gained from the course, the system was over half redesigned and largely extended to reach the current state. Paramount here was the decision to always translate a BASIC program into machine code and design a debugging facility that would allow a program to operate normally at essentially machine speed.

The final result is an inexpensive, flexible, efficient, user-oriented, highly interactive, file-based system with a simple programming language to perform standard and matrix computation using double-precision arithmetic, extensive string manipulation and file operations, and a powerful command language to perform general resource manipulation, program debugging, and text-editing.

The authors wish to acknowledge the original contributions of the members of the design course: Arundhati Badrinath, Jerome Berkman, Mark Bowles, Thomas Boynton, Jeanne Byers, James Conklin, Thomas Dingwall, John Foy, Edward Fronczak, William Honig, Ralph Koch, Sam Listiak, Clark Lubbers, Gordon Lyon, Steven Muchnick, Isao Ohnishi, Dick Peterson, Dorothy Robling, Dick Salisbury, Newton Siever, Fred Swartz, Esther Tepper, and Steven Weiss.

Also appreciated are the suggestions of Professors B. A. Galler and L. K. Flanigan and the guidance of the course mentor, Professor B. W. Arden who brought forth the notion of the virtual file.

II INTRODUCTION TO U OF M BASIC

This section explains the requirements for using BASIC and describes the relationship of the user to the BASIC system and the computer. It defines the file organization which is fundamental to the creation and storing of programs and data. Also, through a series of examples, the user is introduced to the system resources. Finally, means of identifying various parts of the system are described.

II.A REQUIREMENTS FOR USING BASIC

To use BASIC, the user must first obtain authorization to use the computer by applying at the Computing Center Business Office (764-2121). He should request a Computing Center identification number, an initial password, and terminal privileges. If he intends to save permanent copies of programs and/or data in the computer, he should also request a number of pages of disk space. A general rule of thumb is to allow 6 pages for each program to be saved, although large programs might require 3 or 4 more pages. For other details on the University of Michigan facility, see reference 1.

II.B THE COMPUTER SYSTEM ORGANIZATION

The computer at the University of Michigan Computing Center is a complex arrangement of computing machinery (hardware). To relieve the user of the burden of learning the details of this computer, a series of programs (software) were developed that handle all the complexities and provide the user with an easier means of communication with the machine. These programs are collectively referred to as MTS (the Michigan Terminal System). This system is oriented toward terminal and batch operation with emphasis on the former. A terminal is a typewriter-like device that is connected to the computer usually over the telephone system. This allows the user to interact with the computer. Batch operation involves submitting a program deck (called a job) to the computer to be scheduled for non-interactive processing. One advantage of batch operation is the availability of printers for large amounts of computer output. Even with the powerful facilities available in MTS, however, to acquire a facility for composing, running, debugging programs, and doing text-editing still requires extensive study on the part of the user. On the other hand, the BASIC System has an easy-to-learn programming language and provides the above features in a simpler form to learn at a lower cost. BASIC uses low overhead features of MTS along with its own facilities to interact with the user. The user calls MTS over the telephone, signs on MTS, and requests the BASIC subsystem. Once he is operating within BASIC, details of MTS need not concern him. However, the user is referred to MTS Volume 1 (reference 1) which gives an orientation to the University Computing Center and its facilities, especially the conversational mode in MTS, i.e., terminal usage.

II.C <u>BASIC RESOURCES</u>

BASIC is a file-oriented system in that files are used to store programs and data in permanent or temporary form. Some related general definitions are needed to understand this facility.

<u>File (Definition)</u>
A BASIC file is a sequence of variable length lines of characters, each line being associated with a number. Files may be used to store programs or data; therefore, there are different file types.

<u>Line Number (Definition)</u>
A file line number is an <u>integer</u> in the range 0 to 99999 inclusive.

<u>Line Length</u>
Lines are restricted to being 1 to 254 characters in length.

<u>File Name</u>
A file name is a sequence of 1 to 7 letters or decimal digits, the first one being a letter, e.g., PROG1, DIV, LONGFIL. A line range may be placed immediately after the name to restrict processing in some cases to a part of a file. For example, PROG1(10,30) refers to lines 10 through 30 of file PROG1, and PROG1(10) refers to lines 10 through the end of file PROG1.

<u>Permanent File</u>
A permanent file is one that has been explicitly saved by the user on an auxiliary storage device such as magnetic disk.

<u>Temporary File Copy</u>
<u>Working File Copy</u>
<u>Core File</u>
<u>Virtual File</u>
All the above names apply to a file which is used as a scratch pad or "temporary copy" to manipulate a program or data. This is the "virtual file" mentioned in Section I. When a user refers to a file, one of two things will occur:

(1) If a permanent file under that name exists on permanent storage, a temporary copy of that permanent file is created. The user may manipulate the temporary copy without altering the permanent one. To update (replace) the permanent copy with the working copy, the user need only save the latter.

(2) If no permanent file exists, an empty working copy is produced. After manipulating it the user can save it to produce a permanent copy.

In any event, if the temporary file is not explicitly saved by the user, it ceases to exist after the user signs off BASIC. All file manipulation in BASIC is done with "temporary" files.

File Types (S, D, O)

There are three file types in BASIC: S, D, and O. There may be three different types associated with a single file name.

S-Type (Source Program) File

A file which contains the source language statements of a BASIC program.

D-Type (Data) File

A file which contains data. It may be a data file associated with a particular program or just an ordinary data file. See Section V.F for details.

O-Type (Object) File

A file which contains the machine language translation of the source program in the file having the same name.

File Type References

A file type may be referred to by following the name of the file by an at-sign (@) and an S, D, or O. For example, PROG1@S (assumed if just PROG1 is given), PROG1@D, and PROG1@O refer to the S, D, and O files of program PROG1. Note that the @S is optional when referring to a source file. In some contexts, the file type is obvious and the user need not specify it. The file type must follow the line range if given, e.g., P1(20,50)@D.

Command Language

A language in which the user controls the manipulation of the BASIC system resources by issuing commands to the system. It allows him to create, destroy, put information into, list, edit, empty, and save program and data files, etc. Moreover, there are facilities for creating, running, and debugging programs and sharing programs and data with other users. The BASIC command language facility is detailed in Section IV.

Programming Language

A language in which the user writes a set of statements called a program describing the steps in the solution of a problem. BASIC translates these programs into the language the computer understands, namely, the machine language. The BASIC programming language facility is described in Section V.

II.D USING BASIC (INTRODUCTORY EXAMPLES)

The five examples on the following pages are devised to illustrate the most frequently used facilities of BASIC. They are serially dependent so they must be read in sequence. It is assumed that the user has already signed on MTS and has invoked the BASIC System as described in Section III. Knowledge of the details of the MTS signon and BASIC invocation procedures is not necessary to the understanding of this section. These examples were prepared on a terminal which has uppercase and lowercase capabilities.

```
:/open div
& "DIV" has been created.
:10 input a,b
:20 let y=a/b
:30 print a,b,"ans =";y
:40 gito 10
:/run
& No such statement, missing parenthesis or =.
  40 GITO 10
& Error(s) in program. - Correct and try again.
:40 goto 10
:/list
  10 INPUT A,B
  20 LET Y=A/B
  30 PRINT A,B,"ANS =";Y
  40 GOTO 10
&End-Of-File
:/run
?2,4
  2             4             ANS = 0.5
?2,8
  2             8             ANS = 0.25
?/endfile
+ At Line "10" in Program "DIV"
+ Program Ends
:
```

Example 1. Creating and Running a Program

In the above example, the colon (:) on the first line is a command mode input prefix character which signals to the user that BASIC is waiting for him to issue a command. All lines typed with a slash (/) prefix are BASIC command lines. All messages from BASIC in command mode are prefixed with an ampersand (&). In this session, the user is going to create and run a program to carry out a division. To do this, he must first have a file in which to store the program. He types /OPEN DIV and a "temporary copy" of the source file DIV is created since a permanent copy did not exist. DIV is now the "active file", that is, anytime a line is typed prefixed with a legal line number, it will go into this file at that position. There can only be one active file at a time.

He types in four statements numbering by 10s since he wants to allow for making insertions later on. Statement 10 will read two numbers into A and B from the terminal . Statement 20 will divide the first by the second and store the result in the cell Y. Statement 30 will print the value of A, followed by the value of B, followed by a character string, followed by the value of Y. Statement 40 should be a GOTO to line 10, but the user mistyped it. Note that the statements could be typed in any order since the line numbers define the order within the file. Being optimistic, the user requests via /RUN that the program be run. He need not explicitly mention the program name since /RUN assumes the "active file" if none is specified. BASIC detects the error in statement 40 and complains. The ampersand prefixes again tell the user that BASIC is still in command mode as does the colon on the following line. The user retypes the statement correctly and lists the source file to re-examine the program. (Instead of retyping statement 40, use could have been made of the edit command /ALTER 'I'O' 40 to change the I to an O. Had the user needed to delete a statement from the program, this could have been done by just typing the line number of the statement.) The End-of-File message indicates that BASIC listed to the end of the source file, and the user is again presented with a colon. As indicated, the program is rerun. This time it is successful and the program begins to run in "<u>debug mode</u>", that is, a mode where if logical or data errors are discovered, the user in addition to being notified of the errors will be given an opportunity to interact with the program to find the errors. We will not pursue this here since Example 3 covers it in detail. The question mark (?) prefix is caused by statement 10's asking for values for A and B. The user types in order the values, separating each by a comma (blanks would be o.k. too). The 2 is read into A and the 4 into B. Statement 20 takes the value of A and divides it by B, thus obtaining 0.5 and storing it into Y. Next the PRINT statement will print A, B, a character string, and Y. The output paper is divided into fields 15 columns wide. Each data item is printed at the beginning of a field unless it is preceded by a semi-colon, in which case it immediately follows the previous item on the line. Moreover, numbers are always printed with one preceding blank. So A and B and the character string are printed starting in columns 1, 16, and 31, respectively. The value of Y immediately follows the string. The blank prefix at the beginning of the line indicates that the user's program printed the message. Statement 40 returns control to statement 10 and the process continues once more. The third time the program requests input, the user indicates the end of data by typing /ENDFILE. Alternatively, an end-of-file could have been issued via the appropriate keys (control-C on a Model 35 Teletype - see Appendix F). BASIC acknowledges the end of the run and returns the user to command mode where the user again sees the colon prefix. Note that the plus signs (+) on the messages preceding the colon indicated that these messages were issued in "debug mode". At this point the user can leave BASIC (in which case the temporary copy of DIV disappears unless it is saved) or continue to develop and run the program.

```
********************
* IMPORTANT NOTE!  *
********************
```

If the user is typing a long program or a large set of data into a BASIC file, a save (see the /SAVE command) of the partially completed file should be made at fairly frequent intervals so that in the event of a machine malfunction, which causes the loss of the user's core files, only that which has been typed since the last "save" need be retyped.

```
 :/open div                    |  :/open fil2@d
 & "DIV" has been created.     |  & "FIL2(D)" has been created.
 :/number                      |  :/number 5,10
   10_input a,b                |    5_1,2,3,4
   20_let y=a/b                |    15_5,6,7
   30_print a,b,"ans =";y      |    25_1.2,3.4,-5.3
   40_gito 10                  |    35_/unn
   50_/unnumber                |  :
 :                             |
```

Example 2. Using Automatic Line-Numbering

Referring to Example 1, consider an alternative to the user's supplying the line numbers on the lines as they are typed into the "active file". On the left side of Example 2 above, the file is open and made active just as before. Now the user's typing /NUMBER (or /NUM) causes BASIC to go into "numbering mode" in which BASIC supplies the line numbers (by default, starting with 10 in increments of 10). BASIC types the prefix "10_" and the user types the remainder of the line. This continues until the program has been typed, i.e., the "50_" is issued. The user then types /UNNUMBER to leave "numbering mode". Note that the slash (/) prefix here is necessary to differentiate between a command and a data line. The / prefix, however, was not necessary on the lines with colon (:) input prefixes. The remainder of the session may now proceed just as in Example 1.

Referring to the right side of the above example, note that this procedure is not restricted to program files. The user may use it to generate data files. It is necessary only to open the data file (e.g., /OPEN FIL2@D) and enter numbering mode to type in the data. Notice that this time the user specifies the starting line number (5) and the increment (10). Alternatively, one could simply supply one's own line numbers outside of numbering mode. Next, we will discuss program debugging.

```
:run div
?1,2
  1              2              ANS = 0.5
?2,0
+ Attempt to divide by zero.
+ At Line "20" in Program "DIV"
+Ready!
>list 20
  20 LET Y=A/B
>mod b 4
>dis b
+ B = 4
>goto 20
  2              4              ANS = 0.5
?2,0
+ Attempt to divide by zero.
+ At Line "20" in Program "DIV"
+Ready!
>goto 10
?5,6
  5              6              ANS = 0.8333333
?/endfile
+ At Line "10" in Program "DIV"
+ Program Ends
:
```

Example 3. Debugging a Program

Again referring to the program resulting from Example 1, note that it does not check for a zero divisor for statement 20; hence, a run-time error could occur as we shall see. The user runs the program explicitly referring to DIV. If it is the active file (e.g., if we are continuing Example 1), the name could be left off. If this is really a different terminal session and the user saved a permanent copy of DIV in Example 1 before signing off, then this time a "working copy" of DIV is made from the permanent one. In any event, the program is now running. When the program asks the user for data the second time, a value of zero is entered for B. When BASIC attempts to execute statement 20, it discovers the division error and prints two messages (with + "debug mode" prefixes) indicating the error type and where it occurred. The user is then notified that BASIC is ready for interactive debugging. The > debug input prefix indicates that the user may issue debug commands as well as most normal commands (e.g., LIST). Not remembering what line 20 is, he lists it. Noticing that the divisor is B, the user modifies B to 4 and displays it. Then statement 20, which uses the same value for A with the new value of B, is reexecuted. Note that if changes in the program statements had been necessary, it would have been necessary to rerun the program after making the changes. To do this, one would have had to leave "debug mode" and go back to "command mode" by typing /STOP. Since this change is a data value change, the user can continue the program. To illustrate another alternative the user had when this error arose, we will repeat the error by again giving a value of zero for B. This time when going into debug mode, the user simply transfers control to the input statement 10 to reread values for both A and B. After one more good set of data, the program is terminated by a user-supplied end-of-file just as in Example 1.

```
:/open div
:15 if b=0 then 50
:50 print "you are trying to divide by zero"
:60 print "enter replacement divisor"
:70 input b
:80 goto 15
:list
  10 INPUT A,B
  15 IF B=0 THEN 50
  20 LET Y=A/B
  30 PRINT A,B,"ANS =";Y
  40 GOTO 10
  50 PRINT "YOU ARE TRYING TO DIVIDE BY ZERO"
  60 PRINT "ENTER REPLACEMENT DIVISOR"
  70 INPUT B
  80 GOTO 15
&End-Of-File
:run
?1,2
  1              2             ANS = 0.5
?2,0
 YOU ARE TRYING TO DIVIDE BY ZERO
 ENTER REPLACEMENT DIVISOR
?4
  2              4             ANS = 0.5
?/endfile
+ At Line "10" in Program "DIV"
+ Program Ends
:save@all div
&Done
:
```

Example 4. Building Error-Checking into a Program

Continuing with DIV program from Example 1, the programmer decides that the program, in order to be used by someone else not knowing the programming details, should be modified to error check the input data and interact with the person using it. The file is open (if it is not already open) and, because the user left space between line numbers, a new statement can be inserted after statement 10 to check B for being zero. The statements 50 through

80 are added at the end of the program to query the user for a replacement divisor and return to statement 15 to error check it again (users repeat errors occasionally). The user then lists the program to see its final form and makes a test run. IT WORKS!! It is then saved (source plus object - there is no data file) for future use. BASIC responds by typing "Done", indicating that permanent copies have been saved. See Section IX on saving programs. In addition, the programmer could permit his or her files for other users to access. The programmer could also permit the object file DIV@O so that the other users could just execute it rather than run it. This process is discussed in the next example. See Section X for more information about sharing files.

```
:/execute div

?2,4
  2              4             ANS = 0.5
?6,9
  6              9             ANS = 0.6666667
?/endfile
& At Line "10" in Program "DIV"
& Program Ends
:
```

Example 5. Executing a Program (Production Runs)

Now that the DIV program has been perfected (?), it is ready to be used as a production program. Since running a program via /RUN (or equivalent) involves a translation process, which while not being very costly could be avoided to save some money, BASIC has the facility to execute the object file of a program that has already been translated into object form. The user types /EXECUTE (or just EX) followed by the file name if it is not the name corresponding to the active file. The program begins normal processing. The executing program does not run in "debug mode"; hence, if the program developed some error, it would just stop after printing of the error message and return to command mode. To invoke "debug mode" for executing a program, the user need only add PAR=DEBUG to his /EXECUTE command. In this case, before execution begins, the user's program will enter "debug mode" and +Ready! will be printed followed by a > prefix prompting the user for debug commands. Then, if the command /START is typed, the program will go into execution.

II.E PREFIX CONVENTIONS (OR WHO IS TALKING TO YOU?)

The following describes the prefix conventions for output messages or for requesting input from the user. All prefixes refer to BASIC except where noted otherwise.

#	MTS requesting input or printing output in MTS command mode.
&	Command mode output
:	Command mode input
+	Debug mode output
>	Debug mode input
blank	BASIC program output
?	BASIC program input request or prompting in command mode
=	BASIC program input request via the INP function
*	Continuation line expected for input prompting or current line being printed is a continuation of the previous printed line.
nn_	Numbering mode prefix where nn is the line number of the line to be typed
$	BASIC program non-quoted string input request via the LINPUT statement or prompting of multi-line input for the COMPLAIN command.

Example 6. Prefix Conventions in BASIC

II.F GETTING BASIC'S ATTENTION (THE PANIC BUTTON)

If during the course of any operation in the system (program running, file copying, etc.) you wish to stop, suspend, or terminate the operation, there is a "PANIC BUTTON" that may be pushed. On most terminals, there is some way of issuing an attention by pressing an attention key or a combination of keys (see Appendix F on Teletype operation). BASIC will acknowledge the attention and enter a mode appropriate for the operation. For example, if you think that your program is running in a loop, issuing an attention puts you in debug mode where you could possibly trace the logic via the /TRACE command.

CONGRATULATIONS! If you have made it this far, you are well on your way to becoming a skilled BASIC user.

III ENTERING AND LEAVING BASIC

The example on the following page was prepared on an IBM 2741 terminal which has uppercase and lowercase capability. For complete details on using various terminals, the user is referred to the appropriate sections in MTS Volume 1 (reference 1). For simplified instructions on using a Teletype, see Appendix F of this text.

```
         MTS : ANN ARBOR (DC05-0030)
  (1)  #$signon usid
       #ENTER USER PASSWORD.
  (2)  ?▪▪▪▪▪▪▪▪▪▪▪▪
       #**LAST SIGNON WAS: 20:54.09   07-27-71
       #  USER "USID" SIGNED ON AT  00:31.58 ON 07-28-71
  (3)  #$run *basic
       #EXECUTION BEGINS
       :/signoff
       & Off at 00:33.23 on 07-28-71


       #EXECUTION TERMINATED
  (4)  #sig $
       #OFF AT 00:34.01    07-28-71
       #   $1.16
       #$2443.13
```

Example 7. Entering and Leaving BASIC

Signing On MTS

The user telephones MTS and is acknowledged via the standard MTS signon message. MTS then prompts the user (point 1) to issue a $SIGNON command (abbreviated SIG) by giving the # prefix. After doing so, the user is prompted for a password (point 2) via the ? prefix. If it is correct (it was in this example), two lines of signon information are printed and the user is again prompted (at point 3), this time, for any MTS command. At this point or at any point hereafter where a # prompting character is presented, the user may type the MTS command $RUN *BASIC to enter the BASIC System.

Signing On BASIC

The MTS command at (3) invokes the BASIC System and BASIC prompts the user for input by printing a colon (:). The system is now in "command mode" and at anytime thereafter the user is prompted with a colon. He then proceeds with his programming (in this example, he signs off BASIC).

Signing Off BASIC

To leave BASIC, the user may type any of the following BASIC commands: /BYE, /GOODBYE, /QUIT, or /SIGNOFF. If any of these is followed on the same line by PAR=STAT, then statistics for using BASIC are printed (e.g., elapsed time since the $RUN *BASIC was issued, etc.); otherwise, just the time and date of signoff are printed. In any case, BASIC returns to MTS and the user is prompted for MTS commands (point 4).

Signing Off MTS

To sign off MTS, the user may type any of the following MTS commands when presented with a # prompting character (as at point 4): $SIGNOFF ($SIG is the abbreviation), $SIG SHORT, or $SIG $, where the commands give less MTS signoff statistics as you go from left to right.

To sign off MTS directly from BASIC (: prefix), the user may issue the BASIC command /LOGOFF with an optional PAR=STAT. To control the amount of MTS signoff statistics, either SHORT or $ may be placed after the /LOGOFF (e.g., /LOG $ PAR=STAT or /LOG $).

The specifications on the following page are provided for extended usage of the $RUN command in invoking the BASIC System.

*BASIC

Contents: The object module of the University of Michigan BASIC System.

Usage: The BASIC system is invoked by the $RUN command.

Logical I/O Units Referenced:

None

Example: $RUN *BASIC PAR=BS,NC,LC,NOSIG

Description: This program is intended for use as a self-contained system for debugging, modifying, and running programs written in the BASIC language. Commands, source program lines, and data are read from the MTS pseudo-device *SOURCE*. Output from BASIC is written to the MTS pseudo-device *SINK*.

Parameters: The user may specify the following options, separated by commas in the "PAR=" field of the "$RUN *BASIC" command. The entries may appear in any order. A standard default will be assumed for any missing parameters. Following each parameter in the list below is an abbreviated form for the option. The default form of any option is dependent on the "device type" of *SINK*.

BACKSPACE	(BS)	If the output device is recognized as a "terminal", underlining and overstriking through the use of the "backspace character" will be attempted (e.g., underlining text, such as <u>BASIC</u>).
NOBACKSPACE	(NB)	No backspacing will be attempted.
BATCH	(BAT)	Command or program errors will cause BASIC to terminate operation.
NOBATCH	(NBT)	Errors will not cause BASIC to terminate operation.
ECHO	(E)	All command lines will be echoed to the output device.

LC		Upper- and lowercase output for system messages and user-generated text will be attempted.
UC		Upper- and lowercase output will be disabled. <u>All</u> messages will be entirely in uppercase.
NC		The user will not be queried before the destroying, emptying, or freeing of a BASIC file.
NOSIG	(NS)	The "signon message" after the $RUN command will not be printed.
SIG	(S)	The "signon message" will be printed.

All of the above parameters except NOSIG cause the setting of "user-defined" switches which are interrogated by BASIC to effect the specified behavior. These switches may be modified <u>later in the session</u> through the use of the appropriate BASIC commands. The various defaults for these parameters are detailed below according to the "device type". The "PAR=" field is used to override these defaults. The NOSIG option is never defaulted and the BATCH, NC, and ECHO options are defaulted in batch mode regardless of device type.

Batch	<u>The device type is not checked</u>. The defaults are NOBACKSPACE, BATCH, ECHO, NC, and UC. The default output line length is 131.
ARU	The defaults are NOBACKSPACE, NOBATCH, and UC. The default output line length is 255.
2741 and 1050	The defaults are NOBACKSPACE, NOBATCH, and LC. The default output line length is 255.
Data Concentrator (2741 and TTY Model 37 only)	The defaults are NOBACKSPACE, NOBATCH, and LC. The default output line length is 255.

CRT Devices	Same as for the Data Concentrator. Information (usually in the form of a footer) will be produced if possible. The information is 1) whether the user is currently in either BASIC or MTS, and, if in the former, 2) the current mode (command, debug, or running), 3) the name of the currently active file, and 4) the name of the program currently being run (excluding any programs referenced in a series of calls).
All other devices	The defaults are NOBACKSPACE, NOBATCH, and LC. The default output line length depends on the maximum output line length for the device.

III.1 BASIC TUTORIAL SYSTEM (LEARNING BASIC AT A TERMINAL)

Tutorial System

The BASIC Tutorial System is a collection of lectures, programming exercises, and sample training programs to assist the user in learning BASIC while using it. The user is interactively guided through the tutorial series to learn fundamental concepts of the command language (file manipulation, program invocation, etc.) and programming fundamentals while using the BASIC programming language (program building blocks, program physical and logical structure, repetitive processes, etc.).

The tutorial system is meant to be a fundamental introduction and not a replacement for this text; however, the guided usage of BASIC that it provides should impart confidence to a beginning student.

To invoke the tutorial system, one simply calls MTS from a terminal, signs on MTS, invokes the BASIC system, and types /TUTOR as illustrated.

```
1) Call MTS from a terminal (see Appendix F on Teletype usage).
2) Type $SIGNON USID  (your private user id)
3) Type your password.
4) Type $RUN *BASIC
5) Type /TUTOR
```

Example 7.1 Invoking the BASIC Tutorial System

After completing 5), the user may interactively select all or parts of the tutorial series.

IV BASIC COMMAND LANGUAGE

"Your wish is my command."

IV.A <u>THE COMMAND LANGUAGE FACILITY</u>

The command language facility is the nucleus of the BASIC System. It is through this facility that the user manipulates files and runs and debugs programs. In particular, the user can create, destroy, put information into, list, edit, empty, copy, and obtain statistics on his files (e.g., permit status, number of lines, etc.). Permanent copies may be saved and possibly shared by other users. During the course of a terminal session, the user may query the system for statistics such as, time of day, the elapsed time in BASIC, etc. By means of a HELP facility, information can be obtained about the system while using it. There are well over 50 commands in the command language with just a few synonyms for the convenience of users of other BASIC systems (e.g., /BYE, /SIGNOFF). With all this facility it still was designed to be <u>basic</u> by default and complex (comprehensive) by extension. There are facilities to satisfy both the novice and the expert. Next, we will discuss when commands may be entered and then present a small subset of commands common to most situations. A complete command summary is given in Appendix J and detailed specifications are given in Section IV.G.

IV.B WHEN MAY COMMANDS OR DATA LINES BE GIVEN?

Some background is needed to discuss this issue. First, BASIC commands are prefixed with a slash (/). They may be abbreviated and in most cases the / need not be given. Second, BASIC is always in one of the following two modes.

Command Mode
This is the normal mode of the system. In this mode, BASIC reads either a command to process or a data line to put into the "active file". Input requests are prefixed with a colon (:) and all output messages are prefixed with an ampersand (&).

Debug Mode
This is the default system mode when a program is running. Program errors detected by BASIC cause it to enter interactive program debugging with the user. A request for input is prefixed by a greater than (>) sign and all output messages are prefixed with a plus (+) sign. See Section VII for details.

Legality of commands depends on the current mode. For example, one cannot issue a debugging command in command mode since there is no program running to debug. Also, a command to run a program cannot be given in debug mode since there is already a program running. Most commands are legal in both modes (e.g., list, open, line editing, etc.). In either of these modes data lines may be given to put into the "active file".

Active File
This is a file into which input lines prefixed with line numbers are placed at those line positions. A file is declared to be the "active file" (there can be only one) by the user's issuing a /OPEN command or by explicitly referring to the file in either the /EDIT or /SCAN text-editing commands. The active file may be either an S- or D-type file.

Numbering Mode
An additional mode in which BASIC supplies the line numbers for the line going into the active file. Commands may be given in this mode but they must be prefixed with a / sign; otherwise, they are interpreted as data. Also being in numbering mode still implies that you are in either command mode or debug mode.

<u>When To Enter Commands or Data Lines (RULE)</u>

If either a : (command mode) or > (debug mode) prefix is typed by BASIC, the user may type in commands or data. It is also possible in "numbering mode" when the user is presented with a number prefix (e.g., 50_) for the line contents to be entered. <u>At no other time</u> may commands or data lines be typed into the "active file". However, the CMD function may be used to issue commands or data lines from a program while it is running.

<u>/ENDFILE (The Exception to the Rule)</u>

/ENDFILE is used when a BASIC program is reading input data from the terminal, or possibly from a file to generate an "end-of-file" or "end-of-input" condition. It may be typed when a BASIC program issues either a question (?), equals (=), or dollar ($) prefix to demand input.

<u>Deleting Lines from the Active File</u>

If the user types a line number followed by a carriage return, the corresponding line will be deleted from the active file.

IV.C A MINI-SUBSET OF COMMANDS

The following reduced list of commands will satisfy most users' needs. The minimum number of characters is underlined, with the / being necessary only in numbering mode.

/OPEN FNAME	Opens a "temporary copy" of FNAME and makes it the "active file", e.g., /OP P1@D, OP PROG1. The user may then type data lines into the "active file".
/NUMBER S,I	Enters "numbering mode" and BASIC provides line numbers starting with S in increments of I for the user's data lines going into the "active file". If S and I are left out, the default is 10 in increments of 10. For example, /NUM 5,10 defines a numbering sequence 5, 15, 25, etc., where S is 5 and the increment I is 10.
/UNNUMBER	Leaves "numbering mode".
/RELEASE	Deactivates the "active file" so that data lines typed in will not be entered. This is a good precaution against typing errors accidentally being entered into the "active file".
/LIST FNAME	Lists the lines in FNAME. A line range may be given to list part of a file, e.g., /LIST PROG(10,70)@D. If FNAME is left out, the entire "active file" is listed, e.g., /LIST.
/LIST S,L	Lists the "active file" from line S to line L. If L is not given, only line S is listed, e.g., /LIST 10,30 or /LIST 10.
/SAVE FNAME	Saves the "temporary copy" FNAME on permanent storage. FNAME could be an O, D, or S file. If the @ALL modifier is used with the command, all file types of name FNAME are saved, e.g., /SA P1, /SA@ALL P1, /SA P1@D.

/RUN FNAME	Runs the program whose description is contained in the source file FNAME. FNAME may have a line range; hence, it is possible to run a section of a program within that range (if it makes sense), e.g., /RUN PROG1 or /RUN PROG1(30,70). Note that /RUN produces an object (machine code) file which may be executed via /EXECUTE.
/STOP	If the user is placed in debug mode and he does not wish to debug the program using the debugging facilities (probably a bad decision unless the user is sure of the errors), this command will terminate program execution. See Section VII for debugging details.
/EXECUTE FNAME	The object file associated with FNAME is executed with debugging off. FNAME may specify a line range like for /RUN. For example, /EX PROG1, /EX PROG1(30,70).

The following command is useful if the user understands MTS files. Otherwise, he may wish to proceed to the next section (IV.D).

/INCLUDE MTSFNAME — Inputs lines to BASIC from an MTS file. MTSFNAME is any legal MTS fdname. The lines are only read by the BASIC command processor not by a BASIC program. The lines may be commands or data lines. For example, if the MTS file BPROG has in lines 1 to 4, the following in order:

```
10 INPUT A,B
20 Y=A+B
30 PRINT A,B,Y
40 GOTO 10
```

then the user can have the program put into a BASIC file by issuing:

```
:/OPEN PROG1
:/INCLUDE BPROG
:
```

Note the difference between MTS line numbers and BASIC line numbers. If the user wishes to use numbering mode, the MTS file must <u>not</u> contain the BASIC line numbers. For example, in line 1 of BPROG one would have:

```
          INPUT A,B
```

In this case, the following would generate exactly the same file contents as in the previous example.

```
          :/OPEN PROG1
          :/NUM
            10_/INCLUDE BPROG
          :
```

IV.D BASIC COMMAND MODE

Introduction

BASIC command mode is the default mode for the system. In BASIC command mode, the BASIC Command Language Interpreter (CLI) is looking for the next command to process or the next data line to enter into the current active file. It is in BASIC command mode that the user initially begins interaction with BASIC and issues the various commands for running programs. It is to this mode that the user returns after the programs have terminated execution.

In BASIC command mode, input lines are read from the MTS pseudo-device named *SOURCE*. *SOURCE* refers initially to the user's terminal or to the card deck for batch jobs submitted (unless it was changed in MTS by using the MTS command $SOURCE). This initial assignment is changed to the MTS pseudo-device named *MSOURCE* (which has the same initial assignment as *SOURCE* but which cannot be changed) when the user issues an attention. The user may change this initial assignment from BASIC command mode by using the /INCLUDE command.

Command Lines and Data Lines
Data Lines and Command Lines

All lines read in BASIC command mode are either:

(1) interpreted as commands, or
(2) written into the current active file.

The current active file is established with the OPEN command. Another OPEN command will establish a different active file; the RELEASE, FREE, or DESTROY commands will remove it.

A slash (/), occurring as the first character in the input line, is used to indicate that the line is a command line. If an input line does not begin with a slash, it *may* be interpreted by BASIC as a data line. Data lines are put into the current active file. Every data line must have a line number associated with it. This is accomplished in BASIC in two ways: either BASIC supplies the line number (automatic line numbering on), or the user supplies the line number himself (automatic line numbering off).

In the first method (automatic line numbering on), BASIC automatically assigns a line number to each input line. At a terminal this line number will be printed as the input prefix. Automatic numbering is enabled by the NUMBER command. When numbering is enabled, BASIC is in "numbering mode". Any line which is entered will be interpreted as a data line unless the first character is a slash (/) in which case BASIC will interpret the line as a command. The following example illustrates the differences between command lines and data lines:

```
:/NUMBER
  10_/COMMENT THIS IS A COMMAND LINE
  10_REMARK THIS IS A DATA LINE
  20_/UNNUMBER  /* TURN IT OFF
:COMMENT THIS IS A COMMAND LINE
:30 I AM A DATA LINE
```

Example 8. Command Lines and Data Lines

The first line in this example turns automatic line numbering on. The second line is interpreted as a command since it begins with a slash. The third line is placed in the active file. Notice that the line number 10 was repeated from the second line. This happened because that line was interpreted as a command. Numbering is turned off by the fourth line. Again take note of the fact that the slash was necessary for the line to be interpreted as a command. Line 5 illustrates the fact that commands need not be begun with a slash when BASIC is <u>not</u> in "numbering mode". The last line in this example illustrates the second method of associating a line number with a data line.

In the second method (automatic line numbering off), the user must supply a line number with each line entered. This line number is specified by placing a legal BASIC line number starting as the first non-blank character of the line. A legal BASIC line number is an integer number between 0 and 99999 inclusive. BASIC scans the line for a line number and if present, places the line into the active file at the indicated line position. Using the second method, the user may place lines beginning with a slash in the active file. Lines beginning with numbers must have a blank separating them from the line number itself unless the first method (automatic line numbering on) is being used (the blank is supplied).

If there is no current active file, data lines entered by either method will generate the comment:

"There is no active file to put that in."

The data lines will otherwise be ignored.

Continuation Lines

If the last character of an input line to BASIC is a minus sign (-), then the next input line is assumed to be a continuation of the current line. Continuation begins with the first character of the next line, which will replace the minus continuation character in the previous line. BASIC will change the input prefix to an asterisk (*) to indicate that a continuation line is expected. Only one continuation line is allowed. If a minus is typed as the last character of a continuation line, it will be left "as is" and ignored. The continuation character may be changed by issuing the SET command for the CONTCHAR parameter. The following example illustrates the use of continuation lines:

```
:COMMENT@ECHO THIS LINE WILL BE -
*CONTINUED
&COMMENT@ECHO THIS LINE WILL BE CONTINUED
:SET CONTCHAR=+
:COMM THIS LINE NOT CONTINUED -
:COMM@E THIS ONE +
*IS+
&COMM@E THIS ONE IS+
```

Example 9. Continuation Lines

IV.E GENERAL SYNTAX OF A COMMAND

Command Syntax

In the BASIC command language, each command is recognized by its minimum distinguishable abbreviation, usually one or two letters. A few commands, the documentation commands, begin with a dollar-sign ($) but most begin with a slash (/). In any situation but "numbering mode", the slash may omitted. The dollar-sign, however, cannot be. A command line beginning with an asterisk (*) is recognized as a comment line.

Commands may always be abbreviated. Except in "numbering mode", as many blanks or commas as desired may separate the command name from the beginning of the line. One or more blanks or commas must separate the command name from its subsequent parameters, if specified. Command parameters also must be separated from each other by one or more blanks or commas. The BASIC CLI will recognize the string " /*" as the start of a command comment field, the rest of which is ignored. The following example illustrates correct syntax for commands:

```
:/COPY A TO B       /* FULL BLOWN
:COP A B            /* MINIMUM ABBREVIATION
:  COP A,B          /* LEADING BLANKS
:COP,A,B            /* COMMAS EVERYWHERE
:COPAB              /* ILLEGAL
& Invalid command.
```

Example 10. Correct Syntax of Commands

IV.F COMMAND MODIFIERS

Modifiers (Command)

Command modifiers are composed of an at-sign (@) followed by one or more characters. Each modifier must be appended to the command which it modifies without intervening blanks. More than one such modifier may be appended to a command name. For example:

```
EMPTY@NC@ECHO FILE(1,43)
```

Two classes of modifiers are defined:

(1) Modifiers which override permanent switch assignments for the duration of the command to which they are appended. See the SET command writeup for a detailed explanation of these switch settings.

CONFIRM	Turn the confirm switch on for this command. Ask the user for confirmation before carrying out this command.
NOCONFIRM	Turn the confirm switch off for this command. Don't ask the user for confirmation.
ECHO	Turn the echo switch on for this command. Echo this command line back to the user.
NOECHO	Turn the echo switch off for this command. Don't echo this command line.
ERROR	Set the error switch to complete. Print the error message if this command generates one.
NOERROR	Set the error switch to terse. Do not print error message if one is generated by this command.
TERSE	Turn the terse switch on for this command. Enter "terse mode".
NOTERSE	Turn the terse switch off for this command. Leave "terse mode".
VERIFY	Turn the verify switch on for this command. Verify the effect of this command.
NOVERIFY	Turn the verify switch off for this command. Do not verify the command's effect.

(2) Modifiers which control the action performed by the command.

ALL	Apply the given command throughout the given line range, rather than only once. Or, for some commands, apply the command to all file types under the given name.
EVERY	Apply the given command repetitively for every occurrence of a successful condition within a line. For the /ALTER text-editing command, this means to alter every occurrence of a successful string match in the line.
AE	An abbreviation for ALL and EVERY (i.e., @AE is equivalent to @ALL@EVERY).

All of the above modifiers in (1) except ERROR may be abbreviated to only the first letter unless they begin with "NO", in which case they are abbreviated by "N" or "¬" and the third letter, e.g., NV or ¬V. The ERROR modifier may be abbreviated to ER and, NOERROR to NR. Most of the modifiers are applicable only to certain commands. It is not an error to specify them for any command but in in some cases they are non-functional. For a table showing which command modifiers are applicable to each command, the user is referred to Appendix D. For an explanation of how these modifiers effect the command, refer to the individual writeups which follow this section.

IV.G COMMAND SPECIFICATIONS

Command Prototype Conventions

The following notation conventions are used in the prototypes of the commands:

lowercase	- represents a generic type which is to be replaced by an item supplied by the user.
uppercase	- indicates material to be repeated verbatim in the command.
brackets []	- indicates that material within the brackets is optional.
braces {}	- indicates the material within the braces is a generic type which is to be replaced by an item supplied by the user.
bullets •••	- indicates that the preceding syntactic unit(s) may be repeated.
underlining	- indicates the minimum abbreviated form of the command or parameter. Longer abbreviations will be accepted.

The following pages give complete specifications for the commands in the BASIC command language.

ALTER

Purpose: To change lines in a file without completely retyping them.

Prototype: /ALTER ['{strng1}'{strng2}'] [{line#}] [{file}]

Parameters:

'	is a "edit" pattern delimiter character which may be any character that is neither alphanumeric nor a asterisk (library file prefix).
strng1	is the string to be replaced in the line(s) specified. It may be null.
strng2	is the string with which "strng1" is to be replaced. It also may be null. Together "strng1" and "strng2" are referred to as the "edit pattern". If they are left out, they default to the previous "edit pattern".
line#	is the line number at which "editing" is to be done or, if the ALL command modifier is given, is to begin. If left out, it defaults to the current value of the "line pointer", if it is defined. If the "line pointer" is undefined, it defaults to zero (0). If this parameter is specified explicitly, it becomes the new "line pointer".
file	is the name of the file in which lines are to be edited. If given, it becomes the new "active file" and the line range, if specified, applies. If omitted, it defaults to the "active file", which is the last file explicitly referenced in a SCAN, EDIT, or OPEN command.

Usage: This command is a synonym for the EDIT command. See its description.

BLIST

Purpose: To list all "breakpoints" in the currently running program.

Prototype: /BLIST

Usage: This command is a synonym for the BREAKLIST command. See the following description.

BREAKLIST

Purpose: To list all "breakpoints" in the currently running program.

Prototype: /BREAKLIST

Usage: This command causes a listing of the "breakpoints" in the currently running program. If there are no "breakpoints" set, BASIC will type the comment:

••• You have no breakpoints set. •••

For a description of a "breakpoint" and its usage, see the section on the BREAKPOINT command.

Restriction:

The BREAKPOINT command may be given only in "debug mode".

Effect: The current "breakpoints" are listed.

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there are no "breakpoints" to list.

Examples:

```
/BREAKLIST
BREAKL@¬ECHO
```

BREAKPOINT

Purpose: To set a "breakpoint" in the currently running program.

Prototype: /BREAKPOINT {line#} [, {line#}] •••

Parameters:

line# is a line number in the currently running program at which you wish to stop, i.e., halt execution and check the program's status. It should not be the line number of a non-executable statement such as DIM, DATA, or FILE. Such line numbers will be undefined.

Usage: This command places the line numbers given as parameters on a list of "breakpoints" maintained by the CLI. A "breakpoint" is an executable statement in the currently running program which has been specified by the user as a place to suspend execution and await commands. Execution is suspended at a "breakpoint" *before* the statement at the given line number.

When the program being run attempts to execute the statement at the "breakpoint", control is returned to the CLI which types out the comment:

```
      ••• "¤¤¤" Breakpoint *** •••
```

followed, unless in "terse mode" (See SET command), by:

```
      ••• Ready •••
```

followed by the prompting character for "debug mode", indicating that BASIC is awaiting a command. The status of the program being executed has been preserved and may be examined and modified with the appropriate commands. The program may be restarted with the CONTINUE command which executes the statement at the "breakpoint" and resumes normal running of the program. To restart at some other point, the GOTO command may be used. "Breakpoints" are removed with either the RESTORE or the CLEAN command. A list of current "breakpoints" may be obtained with the BREAKLIST command. For a more complete description of the action and interaction of a program running with "breakpoints", see Section VIII on debugging a BASIC program.

Illegal or undefined line numbers are ignored but complained about. All legal ones are processed.

<u>Restriction</u>:
The BREAKPOINT command may be given only in "debug mode".

Effect: The line numbers given are added to the list of "breakpoints" maintained by the CLI.

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there are no statements at which to put "breakpoints".

••• Missing or illegal parameters. Command Ignored. •••

means that no line numbers were given to set as "breakpoints".

••• "¤¤¤" is an illegal line number. •••

means that ¤¤¤ (something intended to be a line number) is not a valid line number, which is ONE TO FIVE NUMERIC DIGITS.

••• "¤¤¤" is an undefined line number. •••

means that ¤¤¤ is a legal line number but is non-executable or does not exist in the currently running program.

Examples:
```
/BREAKPOINT 10 20 30
/B 10,5678,99
B 39 /* SET A BREAKPOINT
```

BYE

Purpose: To terminate operation of the BASIC subsystem.

Prototype: /BYE [PAR=STAT]

Usage: This command is a synonym for the QUIT command. See its description.

CATALOG

Purpose: To list all "temporary" or "core" files.

Prototype: /CATALOG

Usage: This command lists all "temporary" or "core" files. Normally only those with a "file type" of Source are listed. If the ALL command modifier is given, all "file types" will be listed. Any file which has been OPENed and not FREEd or DESTROYed is a "core" or "temporary" file. Initially there are no "temporary" files. If there are none, BASIC will type the comment:

••• You are not using any files. •••

Effect: All "temporary" files of the types specified are listed.

Modifiers: ALL,ECHO,ERROR

ALL — If this modifier is specified, all "file types" will be listed.

Examples:

```
/CATALOG
CA@A
```

CLEAN

Purpose: To remove all "breakpoints" which have been set by the BREAKPOINT command.

Prototype: /CLEAN

Usage: This command removes all line numbers in the list of "breakpoints" maintained by the CLI. For a description of a "breakpoint" and its usage, see the section on the BREAKPOINT command.

Restriction:
The CLEAN command may be given only in "debug mode".

Effect: All "breakpoints" in the program are removed.

Modifiers: ECHO,ERROR

Error Messages:

```
••• Not in Debug Mode. - No program currently running. •••
```

means that the user is not in "debug mode". This command is therefore illegal since there can be no "breakpoints" to remove.

Examples:

```
/CLEAN
CL
```

<u>COMMENT</u>

Purpose: To allow insertion of comments in the command stream to BASIC.

Prototype: /<u>COMM</u>ENT [{text}]

Parameters:

<u>text</u>	is anything at all which the user wishes to type.

Usage: This command is completely ignored by BASIC except for ECHOing. If the ECHO function is enabled, the <u>entire</u> line will be echoed.

Effect: This command does nothing at all.

Modifiers: ECHO

Examples:

```
/COMMENT THIS IS A COMMENT
COMM SO IS THIS
COMM
```

COMPILE

Purpose: To compile (translate to "machine code") a BASIC program.

Prototype: /COMPILE [{file}] [PAR={parm} [,{parm}] ···]

Parameters:

file is the name of the file to be compiled. If left out, the entire "active" file is assumed. A line range may be specified as part of the file name at the time this command is issued to limit the range of execution if the EXECUTE option is on.

parm is one of the several "keyword parameters" which may be specified to the BASIC compiler. Following is the complete list of these optional parameters. Each parameter listed actually represents a pair of parameters which are an option and its negation. The negation of an option is specified by prefixing it with the word "NO". Thus, the negation of the EXECUTE option is NOEXECUTE. For most of the options, the negation is the default case. Where this is *not* true, the parameter *will be* flagged with a superscript plus ($^+$). The part of the parameter which is underlined represents the letters which must be typed. Next to each parameter is its meaning.

Keyword	Option referred to
EXECUTE$^+$	The program should be immediately executed if there were no errors in the source statements.
SAVE	A permanent copy of the object should be made on permanent storage if there were no errors.
LIST	A listing of the source should be made.

<u>M</u>AP+	A storage map of variables should be generated.
<u>D</u>EBUG	Debug mode should be entered after successful compilation.
<u>C</u>ODE	A mnemonic listing of the object code should be made.
<u>P</u>OSTFIX	A listing of the modified Polish postfix produced by the lexical scan should be made.
<u>SY</u>MTAB	A compile-time symbol table listing should be produced.

Usage: This command causes the BASIC source statements in the given file to be translated to machine code. These instructions are then stored in a BASIC file for later use. This file is called the object file (referenced with the @<u>O</u>BJECT file name modifier) and may be saved permanently. If either the EXECUTE (default) or the DEBUG option is specified, the translated program will be loaded for execution. In the case of the DEBUG option, the user will be prompted for "debugging" commands before execution is started. To start execution of a program compiled with the DEBUG option, type /START. The CODE, SYMTAB, and POSTFIX options are intended for use mainly by system programmers when checking out problems with BASIC itself. The output generated might be interesting but would certainly be voluminous. The MAP option must be used if variables are to be DISPLAYed or MODIFIed when debugging the program.

<u>Restriction</u>:
The COMPILE command may not be given when in "debug mode" or by a program through the CMD built-in function.

Effect: The BASIC source statements in the given file are translated into machine code. If there are no syntactic errors, the machine code is stored in the "object" file. The program may be sent into execution immediately or "debug mode" may be entered depending upon the specification of the EXECUTE or DEBUG keyword parameters.

Modifiers: ECHO,ERROR

Error Messages:

••• Command illegal in debug mode. •••

means that the user is in "debug mode" and therefore may not give this command.

••• Illegal file name or there is no active file. •••

means that the file name specified was an illegal name or, that no name was specified and there was no active file to default to.

••• Illegal keyword and/or parameter. - Command ignored. •••

means that one of the parameters specified in the "par field" for the BASIC compiler was not recognized as a legal parameter.

••• File does not exist. - Command ignored. •••

means that the file name given was a legal file name but did not exist.

••• Error(s) in program. - Correct and try again. •••

means that one or more syntactic errors were detected in the source statements. The translation, in this case, is aborted and BASIC resumes "normal" command mode.

Examples:

```
/COMPILE PROG
/COM PAR=LIST,POSTFIX,CODE,NOEXECUTE
COM PAR=S NOE
```

COMPLAIN

Purpose: To allow the user to give an evaluation of BASIC (complaints, compliments, suggestions, etc.) to the BASIC system staff.

Prototype: /COMPLAIN [{text}]

Parameters:

text	is an optional one-line user evaluation. If not given, a multi-line input may be entered.

Usage: Either one-line or multi-line text entries may be given. In the latter case, when text is not specified on the command line, BASIC will prompt the user (with $ prefix) for multi-line input up to an end-of-file.

Effect: The user's signon id, the current time, date, and the text entered will be saved in a BASIC system log which is checked periodically by the BASIC system staff for user feedback.

Modifiers: ECHO (for echoing only the command line)

Examples:

```
/COMPLAIN I LIKE BASIC.

/COMPL
I LIKE THE NEW "EVERY" MODIFIER.
IT SAVED ME EDITING TIME.
/ENDFILE
```

CONTINUE

Purpose: To restart execution of a "loaded" BASIC program at the current line number.

Prototype: /CONTINUE

Usage: This command will cause the user's program to restart execution at the current line. If it is typed at a breakpoint, the current line is the statement at that breakpoint. If it is typed after a program interruption (i.e., overflow or divide-by-zero etc.), the current line is the statement at which the interrupt occurred. If the program has not yet started running, the current line is the first executable line of the program. Thus, if a CONTINUE command is issued before a program has started running, it is equivalent to a START command.

Restriction:
The CONTINUE command may be given only in "debug mode".

Effect: The "loaded" program is restarted at the current line.

Modifiers: ECHO,ERROR

Error Messages:

```
••• Not in Debug Mode. - No program currently
running •••
```

means that the user is not in "debug mode". This command is therefore illegal since there can be no current line to restart at.

Examples:

```
/CONTINUE
C
```

COPY

Purpose: To copy lines from one BASIC file to another.

Prototype: /COPY [{frmfile} [TO] [{tofile}]]

Parameters:

frmfile	is the name of the file from which lines are to be copied. If this parameter is left out, the tofile must be also. In this case, the active file is copied to the terminal or printer.
tofile	is the name of the file to which lines are to be copied. If this parameter is left out, the specified lines will be copied to the terminal .

Usage: This command may be used to copy the contents (or portion thereof specified) of either a BASIC source file or a BASIC data file to another BASIC file of either type. The portion of the "from" file copied is specified by the line number range (if none is given, the entire file) appended to the file name. Note that the lines copied from the "from" file replace the corresponding lines of the "to" file. The other lines in the "to" file are not altered in any way; therefore, one may want to empty the "to" file before copying to it. If the second parameter is left out, the COPY command becomes equivalent to the LIST command.

Effect: The portion of the "from" file specified is edited into the "to" file, if one is given. If the "to" file is left out, the portion of the "from" file is listed on the terminal .

Modifiers: ECHO,ERROR

Error Messages:

```
••• Missing or illegal parameters.   Command
Ignored. •••
```

means that no "from" file was specified and there was no "active" file to default to.

```
••• "¤¤¤" does not exist. •••
••• Command ignored! •••
```

means that the "from" file specified ("¤¤¤") does not exist. The command is aborted.

```
••• Illegal file type attribute. Command
ignored •••
```

means that the file type attribute specified was undefined or was OBJECT which is illegal for this command. The command is aborted.

Examples:

```
/COPY FILEA FILEB
COP QUAD TO QUADRAT
COP Q23(1,457) AXO67@DATA
COP WERT(4,67)@DATA WERTS
```

DEBUG

Purpose: To compile and load a BASIC program with entry to "debug mode" prior to execution, allowing the specification of debugging commands.

Prototype: /DEBUG [{file}] [PAR={parm} [,{parm}] ···]

Parameters:

file	is the name of the file to be compiled and loaded for execution. If left out, the entire "active" file is assumed. A line range may be specified as part of the file name at the time this command is issued to limit the range of execution of the program.
parm	is one of the several "keyword parameters" which may be specified to the BASIC compiler. For a description and list of these parameters, see the COMPILE command description.

Usage: This command causes the BASIC source statements in the given file to be translated into machine code (compiled). The translated program is loaded. The user is then prompted by BASIC with a "Ready" except in "terse mode" (See TERSE under the SET command). In any case, the prompting prefix is changed to a greater-than sign (>), indicating that debugging commands such as DISPLAY, MODIFY, BREAKPOINT, RESTORE, CONTINUE, etc. may be entered. To start execution of the loaded program, a CONTINUE, GOTO, or START command must be issued. The program terminates execution itself and BASIC returns to "normal" command mode when control passes to a statement outside the specified line number range (the entire program if no explicit line range was given). For a more complete description of the action and interaction of a program in "debug mode", see Section VIII on debugging a BASIC program.

Restriction:
The DEBUG command may not be given when in "debug mode" or by a program through the CMD built-in function.

Effect: The BASIC source statements in the given file are translated into machine code. The translated program is loaded and "debug mode" entered. The user is then prompted for more commands.

Modifiers: ECHO,ERROR

Error Messages:

••• Command illegal in debug mode. •••

means that the user is in "debug mode" and therefore may not give this command.

••• Illegal file name or there is no active file. •••

means that the file name specified was an illegal name or that no name was specified and there was no active file to default to.

••• Illegal keyword and/or parameter. - Command ignored. •••

means that one of the parameters specified in the "par field" for the BASIC compiler was not recognized as a legal parameter.

••• File does not exist. - Command ignored. •••

means that the file name given was a legal file name but did not exist.

••• Error(s) in program. - Correct and try again. •••

means that one or more syntactic errors were detected in the source statements. The translation, in this case, is aborted and BASIC resumes "normal" command mode.

Examples:

```
/DEBUG QUAD PAR=S
DE QUAD
DE
```

DESTROY

Purpose: To destroy both the permanent and "temporary" or "core" copies of a file.

Prototype: /DESTROY {file}

Parameters:

file	is the name of the file to be destroyed. This parameter must be specified explicitly. The name must not be a BASIC library file name (e.g., *SORT).

Usage: This command causes all copies of the mentioned file to be destroyed. This means that both the "temporary" (sometimes referred to as the "working" copy) and the permanent copy will be destroyed. Unless the ALL command modifier is given, only the one file named will be destroyed. If the "confirm switch" has been turned on (with the SET command or by the @CONFIRM modifier), BASIC will ask the user for confirmation before carrying out this command. First, the line:

```
    ••• "¤¤¤" is to be destroyed. Please
    confirm. •••
```

where ¤¤¤ is the name of the file to be destroyed, will be printed. The user will then be prompted with a question mark prefix (?) for a reply. For the command to be carried out, the reply must be either "OK" or "O.K." or "YES".

Effect: The permanent and "temporary" copies of the file(s) given are destroyed.

Modifiers: ALL,CONFIRM,ECHO,ERROR,TERSE

ALL	If this modifier is specified, all "file types" of the given file will be destroyed.
CONFIRM	Specifying this modifier corresponds to the default setting for the "confirm switch" in conversational use. This default is "on" except in "batch". Most frequently, the negation of this modifier would be used (NOCONFIRM or the abbreviated form NC), specifying that the "confirmation sequence" not be performed.

TERSE   If this modifier is given, BASIC will <u>not</u> acknowledge that the file(s) were destroyed, by printing the comment 'Done'.

Error Messages:

```
••• No active file allowed in this context •••
```

means that the file to be destroyed was not specified, which is not permissible.

Examples:

```
/DESTROY PACKRAT
DES SLUG@D
DES@ALL FILE2B
```

DISPLAY

Purpose: To display a variable in the currently running program.

Prototype: /DISPLAY {var} [, {var}] •••

Parameters:

var	is a variable in the currently running program which is to be displayed. It may be either a string or numeric variable. The variable may be subscripted, but the subscript(s) must be integer constants.

Usage: This command will display the current contents of the specified variables. An entire matrix or array cannot be displayed, but subscripted variables may be displayed individually. Variables cannot be DISPLAYed if the MAP option to the compiler (the option is default) is not specified.

Restriction:
The DISPLAY command may be given only in "debug mode".

Effect: The contents of the variables specified are displayed, one per line.

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there can be no variables to display.

••• Missing or illegal parameters. Command Ignored. •••

means that no variables were given to be displayed.

••• "¤¤¤" has no map - Sorry. •••

means that ¤¤¤ (the currently running program) was compiled without the MAP option. Variables cannot be displayed without a "map".

```
••• Illegally specified name and/or
subscript. •••
••• At or near "¤¤¤". •••
```

means that the variable at or near ¤¤¤ was illegally specified. One way to do this is to specify a subscript which is not an integer constant.

```
••• Subscript(s) out of range. •••
••• At or near "¤¤¤". •••
```

means that the subscript(s) specified for the variable at or near ¤¤¤ were either too large or negative.

```
••• "¤¤¤" not in program as described. •••
```

means that the variable ¤¤¤ was not in the program as specified. If specified as an array or matrix (subscripted), there was no matrix or array by that name. If specified as a "simple" variable (unsubscripted), there was no such "simple" variable. Possibly, the variable was undefined in any sense.

Examples:

```
/DISPLAY A B(12,3) CC DD1(8)
D A(0) A(1) A(2) A(3) A(4)
D X,Y,Z(2,3)
```

EDIT

Purpose: To change lines in a file without completely retyping them.

Prototype: /EDIT ['{strng1}'{strng2}'] [{line#}] [{file}]

Parameters:

'	is a "edit" pattern delimiter character which may be any character that is neither alphanumeric nor a asterisk (library file prefix).
strng1	is the string to be replaced in the line(s) specified. It may be null.
strng2	is the string with which "strng1" is to be replaced. It also may be null. Together "strng1" and "strng2" are referred to as the "edit pattern". If they are left out, they default to the previous "edit pattern".
line#	is the line number at which "editing" is is to be done or, if the @ALL command modifier is given, is to begin. It defines the line pointer. If omitted, the current value of the line pointer is used (if it is defined) or if the file is explicitly specified, then the first real line within the file's line range defines the line pointer.
file	is the name of the file in which lines are to be "edited". If given, it becomes the new "active" file and the given line range applies. If omitted, it defaults to the "active file", which is the last file explicitly referenced in a SCAN, EDIT, or OPEN command.

Usage: This command is used to change a line in a file without completely retyping it. The *first* (left-most) and *only* the first occurrence of "strng1" is replaced by "strng2". If the ALL command modifier is specified, this process is performed for each line in the file, from the given line number onward, in which "strng1" occurs. If no occurrence of the string can be found, BASIC will print the comment:

••• String not found. •••

Note that even if the ALL command modifier is specified, _only_ the first occurrence of "strng1" is replaced for any line (unless the @EVERY modifier is used to alter every occurence within a line). If the verify switch is "on", each line altered will be printed for each alteration within the line. (The "verify switch" may be turned off by either the SET command or by use of the NOVERIFY modifier.) The line number, if specified explicitly, becomes the new "line pointer".

Effect: In the lines specified in the given file, the first occurrence of "strng1" is replaced by "strng2". Lines which are so altered are printed if the "verify switch" is on. The line pointer points to the last line edited. If a pattern was correctly specified, it is retained. If an error occurs as a result of the other parameters, they are ignored unless the error message indicates the contrary.

Modifiers: AE,ALL,ECHO,ERROR,EVERY,VERIFY

ALL	If this modifier is specified, all lines in the given file which contain "strng1", from the line number specified onward, will be altered. This means that editing will _not_ stop until all such lines are altered (unless an attention is given).
EVERY	If specified, every occurrence in the line will be altered.
VERIFY	If this modifier is specified, lines which are altered will be printed.
AE	A combination of ALL and EVERY. That is, @AE is equivalent to the combination @ALL@EVERY.

Error Messages:

••• Illegal Pattern. Command ignored. •••

means that the "edit pattern" was illegally formed, probably with a missing delimiter.

••• Illegal file name or there is no active file. •••

means that the file name specified was illegal, or that no file was specified and there was no active file.

••• There is no previous pattern. •••

means that no "edit pattern" was specified and the previous one is undefined, possibly due to a RESET command or to just "signing on".

••• Non-existent line specified - Parameters retained. •••

means that the line number specified does not exist in the given file but the specified parameters have been retained for further use in editing.

••• Line no. outside range - Parameters retained. •••

means that line# is outside the range specified for the file.

Examples:

```
/EDIT 'PRNT'PRINT' 30 MYFILE
E #PRINT#P'T# 30
E@A 'print'Print' 10 MYFILE
E@A
E@EV :OUT:IN:  30
ED@ALL@EVERY 'FYLE'FILE' F1(30,70)
E@AE "DATUM"DATA" 50 F3(25,120)
```

EMPTY

Purpose: To empty all or part of a "temporary" or "core" file.

Prototype: /EMPTY {file}

Parameters:

file is the name of the file to be emptied. Unless the file is an "object" file, only the part of the file specified by the line number range will be emptied. This parameter must be specified explicitly.

Usage: This command will empty the "temporary" or "core" copy (sometimes called the "working" copy) of a file or a portion thereof. Note that this command *does not effect the permanent copy* of the file mentioned - only the "temporary" or "core" copy. If the user wishes to empty part of the permanent copy of the file, the part of the working copy should first be emptied and then saved with the SAVE command. However, saving a completely empty file will cause the permanent copy to be destroyed thus reducing permanent file tenancy costs. If a line number range is specified, only the lines in that range will be removed from the file. This does not apply for "object" files which will, in any case, be entirely emptied.

Unless the "confirm switch" is "off", BASIC will prompt the user for confirmation before carrying out the command. First, the line:

••• "¤¤¤" is to be emptied. - Please confirm. •••

where ¤¤¤ is the name of the file, will be typed, followed by the request for confirmation. This request is denoted by a question mark (?) input prefix. If the command should be carried out, the user must reply with either "OK" or "O.K." or "YES".

Effect: The portion of the given file specified is emptied.

Modifiers: CONFIRM,ECHO,ERROR,TERSE

CONFIRM Specifying this modifier corresponds to the default setting for the "confirm switch" in conversational use. This default is "on" except in "batch". Most frequently, the negation of this modifier would be used (NOCONFIRM or the abbreviated form NC), specifying that the "confirmation sequence" should not be performed.

TERSE	If this modifier is given, BASIC will <u>not</u> acknowledge that the file was emptied, by printing the comment 'Done'.

Error Messages:

```
••• No active file allowed in this context •••
```

means that the file to be emptied was not explicitly given. This is not permissible.

Examples:

```
/EMPTY MYFILE
EMPTY MYFILE(10,45)@D
EM HISFILE@O
EM@NC FILEFIL(1,78)
```

ENDFILE

Purpose:	To signal a logical end-of-file to BASIC.
Prototype:	/ENDFILE
Usage:	This command is used to signal an end-of-file to BASIC. In addition to using it in "command mode" to terminate numbering or leave "include mode", this command may also be placed anywhere in a "data" file to generate an end-of-file. It is also recognized when placed in an input line to the INPUT statement.
Effect:	A logical end-of-file is generated.
Modifiers:	None
Examples:	/ENDFILE

EXECUTE

Purpose: To load and start a BASIC program which has been previously compiled.

Prototype: /EXECUTE [{file}] [PAR=DEBUG]

Parameters:

file is the name of the file to be loaded and started. If left out, it defaults to the entire "active" file. A line range may be specified as part of the file name at the time this command is issued to limit the range of program execution. The file must have been previously compiled. The compilation must have taken place on the current "sign on", unless an "object" file was previously saved.

Usage: This command will effect the loading of the "object file" of the program with the given name, and, if the optional "PAR=DEBUG" (may be abbreviated to "PAR=D") was specified, control will pass to the user to enter debugging commands. If the optional parameter was not given, then execution will commence immediately. The program terminates execution itself and BASIC returns to "normal" command mode when control passes to a statement outside the specified line number range (the entire program if no explicit line range was given). To use this command, an object file must exist. One is generated by using either the COMPILE, the DEBUG, or the RUN command. This command is most useful for running programs which are already completely checked out.

Restriction: The EXECUTE command may not be given when in "debug mode" or by a program through the CMD built-in function.

Effect: The file specified is loaded and, unless the optional "PAR=DEBUG" is given, started at the first line in the given line range. If the optional parameter is specified, "debug mode" is entered and the user is prompted for debugging commands. In this case, if the user wishes to start program execution, the command /START should be issued.

Modifiers: ECHO,ERROR

Error Messages:

••• Command illegal in debug mode. •••

means that the user is in "debug mode" and therefore may not give this command.

••• Illegal file name or there is no active file. •••

means that the file name specified was an illegal name or, that no name was specified and there was no active file to default to.

Examples:

```
/EXECUTE FIL(20,400)
EX FILE34(90,680) PAR=DEBUG
EX PAR=D
```

FILESNIFF

Purpose: To print certain information such as size, number of lines, average line length, and permit status about a BASIC file.

Prototype: /FILESNIFF [{file}]

Parameters:

file is the name of the file about which information is desired. If left out, it defaults to the "active" file.

Usage: This command will print the following information about the given file:

File Name	The name of the file in BASIC.
Type	The file type S,D, or O.
No. Lines	The number of lines in the file.
Pages	The file size in pages[1].
Avg. Line Length	The average line length in bytes.
Line Blk Usage	The percentage of allocated pages (the file size) which is actual information.
Permanency	Whether or not a permanent copy of the file exists.
Permit Status	If a permanent copy exists, whether or not it is permitted.

Effect: The information about the given file which is detailed above is printed.

Modifiers: ECHO,ERROR

Error Messages:

••• Illegal file name or there is no active file. •••

means that no file was specified and there was no "active" file to default to.

----

[1]A page of storage on the computer is equal to 4,096 8-bit bytes.

```
Examples:   /FILESNIFF MINR
            FI
            FIL PROG1@D
            FIL P2@O
```

FREE

Purpose: To destroy only the "temporary" or "core" copy of a file.

Prototype: /FREE {file}

Parameters:

file	is the name of the file to be freed. This parameter must be specified.

Usage: This command causes the "temporary" copy (sometimes called the "working" copy), not the permanent copy, of the specified file to be destroyed. If, for example, the user has extensively altered his file and would like to begin again with a fresh original copy, the "working" copy can be destroyed by use of this command and then the OPEN command can be issued for the same file. This will reset the "working" copy to be identical to the permanent copy of this file. Unless the ALL command modifier is given, only the one file named will be freed.

Unless the "confirm switch" is "off", BASIC will prompt the user for confirmation before carrying out the command. First, the line:

••• "¤¤¤" is to be freed. - Please confirm. •••

where ¤¤¤ is the name of the file, will be typed, followed by the request for confirmation. This request is denoted by a question mark (?) input prefix. If the command should be carried out, the user must reply with either "OK" or "O.K." or "YES".

Effect: The files specified are freed.

Modifiers: ALL,CONFIRM,ECHO,ERROR,TERSE

ALL	If this modifier is specified, all "file types" of the given file will be freed.
CONFIRM	Specifying this modifier corresponds to the default setting for the "confirm switch" in conversational use. This default is "on" except in "batch". Most frequently, the negation of this modifier would be used (NOCONFIRM or the abbreviated form NC), specifying that

the "confirmation sequence" should not be performed.

TERSE If this modifier is given, BASIC will <u>not</u> acknowledge that the files were freed, by printing the comment 'Done'.

Error Messages:

••• No active file allowed in this context •••

means that the file to be freed was not specified, which is not permissible.

Examples:

```
/FREE PAKRAT
F@NC SLUG@D
F@A FILE2B
```

GET

Purpose: To create and/or get a BASIC file and make it active for editing.

Prototype: /GET {file}

Parameters:

file is the name of the file to be gotten. If a file is being created, the user must specify a legal BASIC file name: ONE TO SEVEN ALPHANUMERIC CHARACTERS, THE FIRST OF WHICH IS ALPHABETIC. This parameter must be given explicitly. A line range is optional. A BASIC library file name may be specified if one wishes to obtain an "active" file copy for his use.

Usage: This command is a synonym for the OPEN command. See its description.

GOODBYE

Purpose: To terminate operation of the BASIC subsystem.

Prototype: /GOODBYE [PAR=STAT]

Usage: This command is a synonym for the QUIT command. See its description.

GOTO

Purpose: To restart execution of a "loaded" BASIC program at a given line number.

Prototype: /GOTO {line#}

Parameters:

line# is a line number in the currently running program at which execution is to start or continue. It should not be the line number of a non-executable statement such as DIM, DATA, or FILE. Such line numbers will be undefined.

Usage: This command will cause the program to start (or continue) execution with the statement having the given line number. It is especially useful for restarting a statement that caused an error after the user has made modifications to the data in the program.

Restriction:
The GOTO command may be given only in "debug mode".

Effect: The "loaded" program is restarted at the given line number.

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there can be no program to restart.

••• "¤¤¤" is an illegal line number. •••

means that ¤¤¤ (something intended to be a line number) is not a valid line number, which is ONE TO FIVE NUMERIC DIGITS.

••• "¤¤¤" is an undefined line number. •••

means that ¤¤¤ is a legal line number but is non-executable or does not exist in the currently running program.

```
••• Where the #$?;!*%> do I go? •••
```

means that the line number to restart at was left out. The command is therefore cancelled.

Examples:

```
/GOTO 345
G 1090 /* REENTER THE DATA
```

HELP

Purpose: To obtain information about statements, commands, functions, concepts, etc., in BASIC.

Prototype: /HELP [{name}]

Parameters:

name	is the name of the item about which information is to be obtained. If left out, this parameter defaults to "/HELP".

Usage: This command prints out an explanation for the item with the given name. In specifying the name of a command, the slash (/) must be specified in order to differentiate between commands and other items with the same name (e.g., the /STOP command and the STOP statement in the BASIC programming language). To obtain a list of items or categories about which information can be given, type /HELP HELPLIST.

Effect: Information about the given item is printed out. If the TERSE modifier is used, an abbreviated form of information is given if such exists for the item in question.

Modifiers: ECHO,TERSE

Error Messages:

```
••• Information not available. •••
```

means that no information at all could be found about a command or statement with the given name. Possibly, the name was improperly spelled.

Examples:

```
/HELP /LIST
H /FILESNIFF
H
H FOR
H NEXT
H =<
H #OUT
H VARIABLE
H STOP      /* THE STOP STATEMENT
H /STOP     /* THE /STOP COMMAND
```

INCLUDE

Purpose: To include input to the BASIC CLI from sources other than the terminal (conversational) or the card deck (batch), such as MTS files.

Prototype: /INCLUDE [{fdname}]

Parameters:

fdname is any legal MTS file or device name, with or without line number range or implicit or explicit concatenation. If left out, this parameter defaults to the last "fdname" mentioned in an INCLUDE command. In this case, BASIC continues reading "at the point where it last left off". If there was no previous "fdname", the command has no effect.

Usage: This command causes BASIC to read "command" and "data" lines from the "fdname" specified, usually an MTS file. When doing this, BASIC is said to be in "include mode". Note that in "include mode" commands as well as "data" lines may be entered. Also, BASIC may be simultaneously in "include mode" and in "numbering mode" as in the case where data lines do not have BASIC line numbers as prefixes. Input for the INPUT or MAT INPUT statements is *not* affected by the INCLUDE command. This means that even if the user has included a file with commands to run a program, input for these statements in that program will still be from the terminal. Input from the included "fdname" is terminated and BASIC leaves "include mode" when an end-of-file is encountered, an attention is given, the RESET command is processed, or when a "data line" is encountered and there is no "active" file. To get an echo of the commands from the "fdname", the SET command may be used to enable ECHOing. If a /INCLUDE command is given in an included file, it *permanently* overrides the previous one.

Effect: BASIC enters "include mode"; future commands and "data lines" are read from the specified "fdname".

Modifiers: ECHO

```
Examples:   /INCLUDE >RDR3
            I W070:BASICTEST(1,90)+FILE5
```

(1) A sample terminal session using MTS files. BASIC line numbers *are not* stored in the MTS file.

```
#$CRE X
# FILE "X" HAS BEEN CREATED.
$ED X
:INS
?INPUT A,B
?PRINT A+B
?       (null line)
:MTS
#$RUN *BASIC
:/OPEN PROG1
& "PROG1" HAS BEEN CREATED.
:/NUM
    10_/INCLUDE X
:/LIST
 10 INPUT A,B
 20 PRINT A+B
&END-OF-FILE
```

(2) A sample terminal session using MTS files. BASIC line numbers *are* stored in the MTS file.

```
#$CRE X
# FILE "X" HAS BEEN CREATED.
$ED X
:INS
?10 INPUT A,B
?20 PRINT A+B
?       (null line)
:MTS
#$RUN *BASIC
:/OPEN PROG1
& "PROG1" HAS BEEN CREATED.
:/INCLUDE X
:/LIST
 10 INPUT A,B
 20 PRINT A+B
&END-OF-FILE
```

LIBRARY

Purpose:	To obtain information about programs in the BASIC public library (BASIC *-files).
Prototype:	/<u>LIB</u>RARY
Parameters:	None
Effect:	The user will be notified as to the contents of the BASIC library according to library file names and/or subject categories, if necessary.
Modifiers:	ECHO
Note:	BASIC *-files are <u>not</u> MTS *-files; hence, they should be accessed only through BASIC.
Example:	/LIB

LINERANGE

Purpose: To list the numbers of the first and last lines in a BASIC file.

Prototype: /LINERANGE [{file}]

Parameters:

file	is the name of the file for which the first and last line number are to be listed. If left out, it defaults to the "active" file.

Effect: This command will print the first and last line numbers in the specified file. If the file is empty, BASIC will print the comment:

```
••• No Lines. •••
```

indicating there are no first and last lines.

Modifiers: ECHO,ERROR

Error Messages:

```
••• Illegal file name or there is no active
file. •••
```

means that the file name specified was an illegal name or, that no name was specified and there was no "active" file to default to.

Examples:

```
/LINERANGE REPAD@O
LINER FIL3E
LIN
```

LINE#

Purpose: To position the "line pointer" and "scan pointer" used by the EDIT and SCAN commands, respectively.

Prototype: /LINE# [{line#}]

Parameters:

line# is the line number to which the "line pointer" is to be positioned. If left out, it defaults to the current value of the "line pointer". In addition to line numbers, the symbols *, *F, and *L are recognized as the current value of the "line pointer", the first line number in the file, and the last line number in the file, respectively. The file in question is the last file mentioned in either a SCAN, EDIT, or OPEN command (i.e., the "active" file). If a line range was specified via these commands, then *F and *L will refer to the actual first and last lines, respectively, within that range.

Usage: This command will position the "line pointer", which is referenced by the EDIT command (by default), to the line number specified. The "scan pointer", which is referenced by the SCAN command, is positioned to one plus the given line number. The given line number must exist in the file. If the "verify switch" is on (default - may be changed via /SET or the @NOVERIFY modifier), the line having that line number will be printed.

Effect: The "line pointer" is positioned at the given line number. The "scan pointer" is positioned at one plus the given line number. If the "verify switch" is on, the line with the given line number is printed.

Modifiers: ECHO,ERROR,VERIFY

VERIFY If this modifier is specified, the line at the given line number will be printed.

Error Messages:

••• There is no active file. •••

means that there is no active file to use.

••• "¤¤¤" is an illegal line number. •••

means that ¤¤¤ (something intended to be a line number) is not a valid line number, which is ONE TO FIVE NUMERIC DIGITS.

••• That line number does not exist. •••

means that the line number specified does not exist in the file (or line range, if previously specified).

Examples:

```
/LINE# 4561
LINE# *F
LINE#@V
```

LIST

Purpose: To list all or part of a BASIC file.

Prototype:
```
1) /LIST [{file}[({begin}[,{end}])]]
2) /LIST {begin}[,{end}]
```

Parameters:

file	is the name of the file to be listed with optional line range given. If form 2) is given, the "active" file is assumed.
begin	is a line number in the file at which listing is to begin. It need not exist. If it doesn't, listing will begin with the first actual line in the file with a higher line number.
end	is a line number in the file at which listing is to end. It need not exist. If it doesn't, listing will end with the next lowest actual line in the file. If omitted in form 2), it defaults to "begin" and, if "begin" *does* exist, the comment "End-Of-File" will be suppressed. If omitted in form 1), it defaults to the last line of the file.

Usage: This command will list all or part of a BASIC file. Either "Data" or "Source" files may be listed but not "Object" files. A listing in progress may be aborted by issuing an attention.

Effect: The portion of the file specified is listed.

Modifiers: ECHO,ERROR

Error Messages:

••• Illegal file name or there is no active file. •••

means that the file name specified was an illegal name or, that no name was specified and there was no "active" file to default to.

••• Illegal file type attribute. Command ignored •••

means that the file type attribute specified was undefined or was OBJECT which is illegal for this command. The command is aborted.

••• "¤¤¤" is an illegal line number. •••

means that ¤¤¤ (something intended to be a line number) is not a valid line number, which is ONE TO FIVE NUMERIC DIGITS.

Examples:

```
/LIST MYFILE
LIST HISFILE(1,45)@D
L 10,20
L 10
L
```

LOGOFF

Purpose: To terminate operation of the BASIC subsystem and return to MTS for an immediate MTS "signoff".

Prototype: /LOGOFF [{parm}] [PAR=STAT]

Parameters:

parm — is a parameter to be passed on to the "master system" (MTS), indicating what kind of "signoff" statistics should be printed. The parameter is appended "as is" to the "signoff command" which is sent to MTS. The options, in increasing order of information obtained from MTS at MTS signoff, are $, SHORT, and LONG. If omitted , it defaults to "SHORT".

Usage: This command causes BASIC to terminate operation and return to the master system in which it is operating. If the optional "PAR=STAT" is typed, statistics for using BASIC will be printed before return to the master system. In addition, this command also causes the user to be immediately "signed off" from the master system, upon return to it.

Effect: BASIC terminates operation. The user is immediately thereafter "signed off" the master system (MTS).

Modifiers: ECHO

Examples:

```
/LOGOFF
LO@ECHO PAR=STAT
LOG $
```

L#

Purpose: To position the "line pointer" and "scan pointer" used by the EDIT and SCAN commands, respectively.

Prototype: /L# [{line#}]

Parameters:

line# is the line number to which the "line pointer" is to be positioned. If left out, it defaults to the current value of the "line pointer". In addition to line numbers, the symbols *, *F, and *L are recognized as the current value of the "line pointer", the first line number in the file, and the last line number in the file, respectively. The file in question is the last file mentioned in either a SCAN, EDIT, or OPEN command (i.e., the "active" file). If a line range was specified via these commands, then *F and *L will refer to the actual first and last lines, respectively, within that range.

Usage: This command is a synonym for the LINE# command. See its description.

MODIFY

Purpose: To change the contents of a variable in the currently running program.

Prototype: /MODIFY {var} {value} [{var} {value}] •••

Parameters:

var	is a variable in the currently running program which is to be modified. It may be either a string or numeric variable. The variable may be subscripted, but the subscript(s) must be integer constants.
value	is the value to which the variable is to be changed. For numeric variables, it may be any legal BASIC numeric constant. For string variables, the rules for string constants in BASIC apply. If uppercase conversion is off, string constants will not be converted to uppercase before modification.

Usage: This command will modify the contents of the specified variables to the given values. An entire matrix or array cannot be modified, but subscripted variables may be modified individually. Variables cannot be MODIFied if the MAP option to the compiler (the option is default) is not specified. If an error condition occurs within a list, all variables to the left of that point have been modified.

Restriction:
The MODIFY command may be given only in "debug mode".

Effect: The contents of the variables specified are modified to the given values.

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there can be no variables to modify.

••• Missing or illegal parameters. Command Ignored. •••

means that no variables were given to be modified.

••• "¤¤¤" has no map - Sorry. •••

means that ¤¤¤ (the currently running program) was compiled without the MAP option. Variables cannot be modified without a "map".

••• Illegally specified name and/or subscript. •••
••• At or near "¤¤¤". •••

means that the variable at or near ¤¤¤ was illegally specified. One way to do this is to specify a subscript which is not an integer constant.

••• Subscript(s) out of range. •••
••• At or near "¤¤¤". •••

means that the subscript(s) specified for the variable at or near ¤¤¤ were either too large or negative.

••• "¤¤¤" not in program as described. •••

means that the variable ¤¤¤ was not in the program as specified. If specified as an array or matrix (subscripted), there was no matrix or array by that name. If specified as a "simple" variable (unsubscripted), there was no such "simple" variable. Possibly, the variable was undefined in any sense.

••• Illegal numeric constant for "¤¤¤" •••
••• Modification stops. •••

means that the value specified for the variable ¤¤¤ was illegal. The command is aborted at this point.

••• Missing enclosing quote(s). •••
••• Modification stops. •••

means that the string constant specified for a string variable was illegally formed. The command is aborted at this point.

••• Missing value. •••

means that the value for the final variable was left out.

Examples:

```
/MODIFY X 3 AA(5) "STRNG"
MOD Y(1) 7 Y(2) 33 Y(3) 69 Y(4) 1E-6
M XX,"Hi there ED old boy."
```

MTS

Purpose: To return the user to the "master system" (MTS) for a possible return to BASIC later.

Prototype: /MTS [{text}]

Parameters:

text	is optional text (e.g., an MTS command) to be passed to MTS for processing.

Usage: This command returns the user to the "master system" (MTS). This return is made in such a way that the user may return to BASIC by issuing the appropriate system command ($RESTART). CAUTION: If the user issues any of the following MTS commands: $RUN, $LOAD, $UNLOAD, or $SIGNOFF, the BASIC System will cease to be available until it is rerun via $RUN *BASIC.

Effect: The user is returned to the "master system" (MTS) and the text, if given, is passed on to MTS.

Modifiers: ECHO

Examples:

```
/MTS
MT
MTS $EMP MFILE OK
```

MTSCMD
$MTScommand

Purpose: To issue an MTS command from BASIC command mode or from a BASIC program.

Prototype:
```
/MTSCMD [{[$]MTScmd}]
$MTScmd
```

Parameters:

MTScmd is an MTS command. In the second prototype form, the MTS command prefix ($) is required.

Usage: The command is used to issue MTS commands from either BASIC command mode or from a BASIC program.

Effect: The command is passed to MTS for processing as an MTS command. After command execution, control returns to the part of BASIC that issued the command. It may be issued by a BASIC program using the CMD function or by the terminal user in BASIC command mode.

Modifiers: ECHO

Examples:
```
I=CMD("MTSC@NOECHO $SET TERSE=ON")
I=CMD("MTSC@NOE $CREATE X")
$SET TERSE=ON
```

NUMBER

Purpose: To start automatic numbering of input lines from the user to the "active" file.

Prototype: /NUMBER [{start} [, {incr}]]

Parameters:

start is the number the automatic line numbering is to start with. If omitted , it defaults to 10.

incr is the number that is to be added to a line number to get the next one. If omitted , it defaults to 10.

Usage: This command causes BASIC to go into "numbering mode" so that the user can type statements after the line numbers supplied by BASIC. It should be used when the user wishes to enter information into a file but does not want to have to type line numbers before every line. The lines will be numbered starting with "start" in steps of "incr". If the parameters are not entered by the user, they both default to ten (10). BASIC leaves "numbering mode" when an end-of-file is encountered, an attention is given, a RESET or UNNUMBER or RELEASE command is processed, or there is no "active" file.

Note: Commands entered in "numbering" mode *must* be prefixed by a slash (/) to be recognized as commands.

Effect: BASIC enters "numbering mode". Line numbers starting with "start" in steps of "incr" are supplied to the user as input prefixes. In "batch" or "include mode", the line numbers will still be supplied but will not be printed.

Modifiers: ECHO,ERROR

Error Messages:

••• "¤¤¤" is an illegal line number. •••

means that ¤¤¤ (something intended to be a line number) is not a valid line number, which is ONE TO FIVE NUMERIC DIGITS.

```
Examples:   /NUMBER 240,2
            NUM 50
            N
```

<u>OBJSCAN</u>

Purpose: To dump the contents of a BASIC "object" file.

Prototype: /<u>OB</u>JSCAN [{file}]

Parameters:

<u>file</u> is the name of the file to be dumped. If left out, it defaults to the "active" file.

Usage: This command will cause a hexadecimal dump of the object code, constant pool, line directory, and relocation dictionary contained in the "object" file of the name specified. It will also dump the map, if present. Intended for use by "system programmers", it should be of little use to the average user. The output it generates might be interesting but will certainly be voluminous.

Effect: The object code, constant pool, line directory, relocation dictionary, and map (if present) contained in the "object" file of the name specified are dumped in hexadecimal.

Modifiers: ECHO,ERROR

Error Messages:

••• Illegal file name or there is no active file. •••

means that the file name specified was an illegal name or, that no name was specified and there was no "active" file to default to.

Examples:

```
/OBJSCAN FERDNOK
OBJ FDER@O
OB
```

OPEN

Purpose: To create and/or get a BASIC file and make it active for editing.

Prototype: /OPEN {file}[{begin}[,{end}]]

Parameters:

file	is the name of the file to be opened. If a file is being created, the user must specify a legal BASIC file name: ONE TO SEVEN ALPHANUMERIC CHARACTERS, THE FIRST OF WHICH IS ALPHABETIC. This parameter must be given explicitly. An optional line range may be given for later use with the SCAN, EDIT, and LINE# commands. A BASIC library file name may be specified if one wishes to obtain an "active" file copy for editing, etc.
begin	is the first line in the optional line range. If omitted (in which case *end* must also be omitted), the default is zero.
end	is the last line in the optional line line range. If omitted, it defaults to 99999.

Usage: This command will get the BASIC file with the given name, and make it active for editing. If the user already has a file with the given name, it is made active. If no current copy exists, BASIC will check permanent storage to see if the user has previously saved a file with this name. In this case, a "temporary" or "working" copy is made, and it becomes active. This "current copy" will initially contain the same lines as the permanent copy at the time it was last saved. In the case that no current copy exists and no permanent copy exists, a file with the given name is created, and the user is notified of this. Note that, in this case, only a "temporary" file is created. To make a permanent copy of a file, it *must* be saved with the SAVE command, which saves *only* the "current contents" (the contents at the time the SAVE command is issued). If a line range is specified, it will be saved and used for the SCAN and

EDIT commands on that file (provided that these commands do not respecify the line range). The line pointer will be defined to be either the line number of either the first actual line of the file or the first actual line number in the given line range, whichever is larger.

Files in BASIC are said to be "temporary" or "working" because they will exist only for as long as the user is "signed on" the system. These files are sometimes called "core" files because they exist only in the "short term" memory of the computer (this memory is referred to as "core"). Permanent copies of these "working" files can be made, but the user must request this explicitly by issuing a SAVE command. Thus OPENing a file does not affect the permanent copy. Any changes to a "temporary" or "working" copy will not affect the permanent copy <u>until</u> a SAVE command is issued.

Effect: The specified file is gotten (a "core" or "working" copy is created) and made active for editing. If a "working" copy already exists, it is only made active. If no such copy does exist, one is fetched from permanent storage or created in "core" as necessary. The file becomes the "active" file. The entire file is opened even if a line range has been specified for use with the text-editing commands.

Modifiers: ECHO,ERROR

Error Messages:

```
••• Illegal file name! •••
••• Command ignored! •••
```

means that the file name specified was not a legal BASIC file name which is ONE TO SEVEN ALPHANUMERIC CHARACTERS THE FIRST OF WHICH IS ALPHABETIC.

```
••• Missing or illegal parameters. Command
Ignored. •••
```

means that no file name was specified.

```
••• Illegal file type attribute. Command
ignored •••
```

means that the file type attribute specified was undefined or was OBJECT which is illegal for this command. The command is aborted.

```
Examples:   /OPEN RECURS
            OP MYFILE@D
            O XX123
            OPEN X(50,150)
            OPEN X(100)      /* FROM LINE 100 TO END OF THE FILE
```

PERMCATALOG

Purpose: To obtain a list of all permanent BASIC files and their permit status.

Prototype: /PERMCATALOG

Effect: A list of file entries (three per line) is printed. Each entry consists of a unique file name followed in parentheses by the file types (S, D, and/or O) that exist on permanent storage for that entry. If the file of that type and name is permitted (via the PERMIT command), then a "P" will follow the type attribute. For example, the entry FILE1(DP,OP,S) corresponds to three files (D-, O-, and S-type) of name FILE1 with one (the S-file) not permitted.

Modifiers: ECHO

Messages:

```
••• Total = NNN files using MMMMM pages. •••
```

gives at the end of a non-null listing a total BASIC permanent file count and total number of permanent storage used.

```
••• You have no BASIC permanent files. •••
```

means just that.

```
••• No catalog table space available! •••
••• Free some temporary files and try again! •••
```

means that no space can be obtained in the computer fast memory for the permanent file list. This is due to the user's having too many temporary files open (by RUNning, OPENing, EDITing, etc.). So, some space (approximately two files) must be FREEd by him to obtain the catalog list.

Example: PERMC

PERMIT

Purpose: To permit other users access to the permanent copy of a BASIC file.

Prototype: /PERMIT {file}

Parameters:

file — is the name of the BASIC file to be permitted. It may be of any type. This parameter must be specified explicitly.

Usage: This command will permit all other MTS users access to the permanent copy of the specified file. This access will be gained with the use of the SHARING command. Other users will only be able to obtain "working" copies of the permanent file. They will not be able to change it. The file will remain permitted until DESTROYed or explicitly "unpermitted" with the UNPERMIT command. Unless the ALL command modifier is given, only the one file named will be permitted. It is not necessary to UNPERMIT a file prior to saving or destroying it.

Effect: The permanent copies of the files specified are permitted for access by other users. It is of MTS permit class "READ to OTHERS". For knowledgeable MTS users, this will augment the permit status given to the file by them via the MTS file-permitting facility.

Modifiers: ALL,ECHO,ERROR,TERSE

ALL — If this modifier is specified, all "file types" of the given file will be permitted.

TERSE — If this modifier is given, BASIC will not acknowledge that the files were permitted, by printing the comment 'Done'.

Error Messages:

```
••• That is not a permanent file. •••
```

means that there was no permanent copy of the specified file to permit.

Examples:

```
/PERMIT FILE23
PERMIT@A@T QUARD@D
```

PREFIX

Purpose: To change the prefix for comments from BASIC to the user.

Prototype: /PREFIX {prefix}

Parameters:

prefix is the prefix to be used by BASIC when printing comments to the user. It will be truncated if over ten characters.

Usage: This command changes the prefix used by BASIC when directing messages to the user. If the command is given in "debug mode", the new prefix will be used only until BASIC returns to "normal mode".

NOTE: This command does not effect input prefixes.

Effect: The prefix for messages from BASIC to the user is changed to the given prefix.

Modifiers: ECHO,ERROR

Error Messages:

```
••• Illegal keyword and/or parameter.  - Command
ignored. •••
```

means that no prefix was given or the prefix given was a break character.

Examples:

```
/PREFIX NORMAL:
PREF@¬ECHO DEBUG:
PRE &
```

PRINT

Purpose: To list such factors as cost, CPU time, and elapsed time having to do with the current run of BASIC.

Prototype: /PRINT {keyword} [, {keyword}] ···

Parameters:

keyword is any of the following mnemonics. Only the letters underlined need be typed.

Keyword	Factor referred to
VMSIZE	The number of pages[1] of storage being used.
TOD	The date and time of day.
ELTIME	The number of minutes since signing on BASIC.
CPUTIME	The amount of CPU time the user has used since signing on BASIC.
COST	Approximate cost of using BASIC since signing on BASIC.
VMINTEGRAL	Storage used across time (page-seconds).

Usage: This command causes a listing of the factors given. The values listed reflect the status of the particular run at the time the command is given. Thus, PRINTing the ELapsed TIME will print the elapsed time from when the user signed on BASIC to the time the command was given, etc.

Effect: The values of the factors given are listed on the current output device.

----

[1]A page of storage on the computer is equal to 4,096 bytes (8-bit).

Modifiers: ECHO,ERROR

```
Examples:   /PRINT COST
            /PRINT@¬ECHO VMSIZE TOD ELTIME CPUTIME
            P@ERR V C T E CO VMI
```

QUIT

Purpose: To terminate operation of the BASIC subsystem.

Prototype: /QUIT [PAR=STAT]

Usage: This command causes BASIC to terminate operation and return to the master system in which it is operating. If the optional string "PAR=STAT" is typed, statistics for using BASIC will be printed before return to the master system.

Effect: BASIC terminates operation.

Modifiers: ECHO

Examples:

```
/QUIT
Q@ECHO PAR=STAT
```

<u>RELEASE</u>

Purpose: To make the current "active" file inactive.

Prototype: /<u>REL</u>EASE

Usage: This command causes BASIC to deactivate the current "active" file. A file in BASIC is said to be "active" when any data line typed in is expected to go into THAT file. The current "active" file is always the last file referenced by an OPEN command. Thus, files are always explicitly made "active" or "inactive" by the user. After a RELEASE command, there is, of course, NO "active" file, and any data lines typed in will cause BASIC to type the comment:

••• There is no active file to put that in. •••

Thus, this command is useful for protecting the user from inadvertently modifying his file. As a convenience, numbering is also turned off since there will be no "active" file to edit.

Effect: The active file is made inactive. Any data lines after this command and before the next OPEN command will be flagged. Numbering is turned off.

Modifiers: ECHO,ERROR

Examples:

```
/RELEASE
REL
```

RENAME

Purpose: To change the name of a "temporary" file.

Prototype: /<u>REN</u>AME {file} [TO] {filenm}

Parameters:

<u>file</u> is the name of the file to be renamed. It may be a user's copy of a BASIC library file (the purpose may be to save it on the user's own permanent storage after renaming and possibly editing it).

<u>filenm</u> is any legal BASIC file name: ONE TO SEVEN ALPHANUMERIC CHARACTERS, THE FIRST OF WHICH IS ALPHABETIC.

Usage: This command changes the name of a file. File type attributes (S or D but <u>not</u> O) may be specified on either file parameter <u>file</u> or <u>filenm</u>. The name change does not effect any permanent copy of the given file if one should exist. The new name cannot be the same as another open[1] file. If the file is active under the old name, it is deactivated. Accessing a file under the old name refers to the permanent copy under that name or, if that does not exist, generates an error comment.

Effect: The given file is renamed and a "Done" message is produced (if TERSE is not on). If active, the file is deactivated. References by the old name access a different file.

Modifiers: ECHO,ERROR,TERSE

Error Messages:

••• Missing command parameter. - Command ignored. •••

means that one or both of the parameters to this command were left out.

----

[1]An open file is any file which has been accessed in any manner during the current run and has <u>not</u> been FREEd or DESTROYed.

```
      ••• Illegal file name •••
      ••• Command ignored •••
```

means that the new name for the file is an illegal file name.

```
      ••• New name is already in use for given
     type. •••
```

means that a file already is open under the new name.

```
      ••• A permanent file by that name already
     exists. •••
```

is a warning message issued after renaming <u>has</u> been done. The permanent copy has <u>not</u> been altered.

```
      ••• Object files not allowed for renaming! •••
```

means that an "O" attribute was given on one of the file name parameters.

Examples:

```
/RENAME OLDNAME TO NEWNAME
REN@¬EC OLD NEW
REN PROG@S TO PROG2@D
```

<u>RENUMBER</u>

Purpose: To renumber a BASIC source or data file.

Prototype: 1) /<u>RENU</u>MBER [{file}[({s}[,{i}])]]
2) /<u>RENU</u>MBER [{s}[,{i}]]

Parameters:

<u>file</u>	is the name of the file to renumber. If form 2) is specified, the "active" file is assumed. The file may be either of type S or D.
<u>s</u>	is the starting line number to be assigned to the first line of the file. If not specified, it defaults to 10.
<u>l</u>	is a <u>positive</u> line number increment to be used to compute the remaining new line numbers of the file. If omitted, it defaults to 10.

Usage: The primary raison d'etre for the command is to provide space for line insertions when no space exists. The command is to be applied only to non-empty S- or D-files. The process may be aborted by issuing an attention. Incomplete programs (S-files with unresolved GOTOs) are allowed to be renumbered. In this case, undefined line number references will be replaced by a "#" sign. Programs (incomplete or not) should be <u>syntactically</u> correct; however, renumbering may be attempted to determine the syntactic errors since a syntactic scan of statements is done during the renumbering process and errors will be reported to the user. Values of <u>s</u> and <u>i</u> should be such that all new line numbers are valid (in the range 0 through 99999). Lines may grow in length during renumbering due to increased line number size and may possibly exceed the maximum line size. Line truncation may occur. The user will be notified after an attempt to delete a sufficient number of unnecessary blanks from the expanded, potentially truncatable lines has failed. If a source file is renumbered and an object file exists, then the user will be warned that the line numbers for the object do not necessarily correspond to those of the source. This is a reminder to rerun the source file to produce new object and update the permanent copy of the object if there is one.

Effect: Empty files are not renumbered (the user is notified). For syntactically <u>in</u>correct S-files, the errors will be printed and renumbering will be cancelled. D-files or syntactically correct S-files with valid new line numbers will be renumbered. If lines were truncated, a list of the new numbers of those lines will be printed for user reference. If an S-file has been renumbered and undefined line number references exist, a list of the new numbers of those statements which contain the undefined references (replaced by "#" signs) will be printed.

Modifiers: ECHO,ERROR,TERSE

Messages: Those not documented below are self-explanatory.

```
      ••• Syntax errors - Renumbering cancelled.  •••
```

means that the syntactically incorrect statements that have been printed caused renumbering to be aborted.

```
      •••  Renumbering  will  result  in illegal line#
     (>99999).  •••
```

means that the combination of <u>s</u> and <u>i</u> is invalid for the number of lines in the file and would produce invalid (too large) line numbers if used; hence, renumbering is aborted.

```
      ••• Renumbering storage not available - too many
     files in use - free some and try again!  •••
```

means that no space in the computer fast memory is available to contain line number correspondence tables. This is due to the user's having too many temporary files open (by running, opening, editing, etc.). So, some space must be /FREEd to allow renumbering to occur.

Examples:

```
/RENUM          /* DEFAULT OF 10,10 ON ACTIVE FILE.
RENUM X         /* DEFAULT OF 10,10 ON FILE X.
RENUM X@D       /* DEFAULT OF 10,10 ON DATA FILE X.
RENUM X(300)    /* DEFAULT INCREMENT OF 10.
RENU X(300,20)  /* NO DEFAULTS USED HERE.
RENU 300,20     /* USE ACTIVE FILE.
RENU 300        /* DEFAULT INCR.=10 FOR ACTIVE FILE.
```

RESET

Purpose: To reset various "user defined" switches and variables to their initial (default) values.

Prototype: /RESET

Usage: This command resets all those switches and variables which may be altered by the user through the BASIC Command Language Interpreter to their initial values. These initial values are assigned at the time the user enters BASIC. Thereafter, they are subject to change via such commands as SET and PREFIX. These switches/variables and their default values are listed below:

Switch	Default Value
ATLINE	ON
COLWIDTH	15
CONFIRM	ON (terminal) OFF (batch)
DATAECHO	OFF (terminal) ON (batch)
DEFDIM	10
DFILCRE	ON
DFILEMP	ON
ECHO	OFF (terminal) ON (batch)
ERROR	COMPLETE
JUSTIFY	LEFT
RESTORE	GLOBAL
STRCON	"" (the null string)
TERSE	OFF
VARCON	0
VERIFY	ON

The above switches and variables are those which may be changed directly by means of the SET command. For an explanation of their meaning, see the SET command description. In addition, the RESET command will: turn numbering off if it is on (same as /UNNUMBER), release the active file if there is one (same as /RELEASE), change the CLI output prefix to & (same as /PREFIX &), set the "scan pattern" to the null string, discard the current "edit" pattern if there is one, and leave "include mode" if in it (see the INCLUDE command description). In short, the RESET command returns the user to essentially the same state as when BASIC was first entered.

<u>Restriction</u>:
The RESET command may not be given when in "debug mode" or by a program through the CMD built-in function.

Effect: The switches and variables in the list above are set to the given default values. Numbering is turned off, the active file is released, the CLI output prefix is set to ampersand (&), the scan and edit patterns are set to null, and future command input is expected to be from the user ("include mode" is terminated if necessary).

Modifiers: ECHO,ERROR

Error Messages:

```
        ••• Command illegal in debug mode. •••
```

means that the user is in "debug mode" and therefore may not give this command.

Examples:

```
/RESET
RESE@ERROR
```

RESTORE

Purpose: To remove "breakpoints" which have been set by the BREAKPOINT command.

Prototype: RESTORE {line#} [, {line#}] •••

Parameters:

line# is a line number in the currently running program which has been set by the BREAKPOINT command as a "breakpoint" and which is to be removed. Attempting to restore a line number which is not actually a "breakpoint", will not be flagged.

Usage: This command removes the line numbers given as parameters from the list of "breakpoints" maintained by the CLI. For a description of a "breakpoint" and its usage, see the BREAKPOINT command description. Illegal or undefined line numbers are ignored by BASIC and complained about.

Restriction:
The RESTORE command may be given only in "debug mode".

Effect: The line numbers given are no longer "breakpoints".

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there can be no "breakpoints" to remove.

••• Missing or illegal parameters. Command Ignored. •••

means that no line numbers were given to be removed.

••• "¤¤¤" is an illegal line number. •••

means that ¤¤¤ (something intended to be a line number) is not a valid line number, which is ONE TO FIVE NUMERIC DIGITS.

••• "¤¤¤" is an undefined line number. •••

means that ¤¤¤ is a legal line number but does not exist in the currently running program.

Examples:

```
/RESTORE 10 20 30
/RE 40,50,67
```

RUN

Purpose: To compile and execute a BASIC program.

Prototype: /RUN [{file}] [PAR={parm} [,{parm}] ···]

Parameters:

file is the name of the file to be translated into machine code and executed. If left out, the entire "active" file is assumed. A line range may be specified as part of the file name at the time this command is issued to limit the range of execution of the program.

parm is one of the several "keyword parameters" which may be specified to the BASIC compiler. For a description and list of these parameters, see the section on the COMPILE command.

Usage: This command causes the BASIC source statements in the given file to be translated into machine code. The translated program is then loaded and execution begun at the first line number in the specified line range (the first executable statement in the program if no explicit line range was given). The program is executed in "debug mode", so that errors only suspend execution and do not terminate it. If an error occurs, the user is prompted with a "Ready" except in "terse mode" (See SET command). In any case, the prompting prefix is changed to a plus (+), indicating that debugging commands such as DISPLAY, MODIFY, BREAKPOINT, RESTORE, CONTINUE, etc., may be entered. To terminate execution and return to "normal" command mode, enter the STOP command. The program terminates execution itself and BASIC returns "normal" command mode when control passes to a statement outside the specified line number range (the entire program if no explicit line range was given). For a more complete description of the action and interaction of a program in "debug mode", see the section on debugging a BASIC program.

Restriction:
The RUN command may not be given when in "debug mode" or by a program through the CMD built-in function.

Effect: The BASIC source statements in the given file are translated into machine code. The translated program is loaded and execution begun, in "debug mode", at the given line number.

Modifiers: ECHO,ERROR

Error Messages:

``` 
••• Command illegal in debug mode. •••
```

means that the user is in "debug mode" and therefore may not give this command.

```
••• Illegal file name or there is no active file. •••
```

means that the file name specified was an illegal name or, that no name was specified and there was no active file to default to.

```
••• Illegal keyword and/or parameter. - Command ignored. •••
```

means that one of the parameters specified in the "par field" for the BASIC compiler was not recognized as a legal parameter.

```
••• File does not exist. - Command ignored. •••
```

means that the file name given was a legal file name but did not exist.

```
••• Error(s) in program. - Correct and try again. •••
```

means that one or more syntactic errors were detected in the source statements. The translation, in this case, is aborted and BASIC resumes "normal" command mode.

Examples:

```
/RUN PROG1
/RUN PAR=LIST,SAVE
R
```

SAVE

Purpose: To make a permanent copy of a given "temporary" file.

Prototype: /SAVE [{file}]

Parameters:

file is the name of the file to be saved on permanent storage. The name must *not* be a BASIC library file name (e.g., *SORT). A file-type modifier (e.g., @O, @D) may be specified.

Usage: This command makes a permanent copy of the file(s) specified on permanent storage. Unless the ALL modifier is given, only the *one* file specified will be saved.

Effect: The given files are saved on permanent storage if they are not empty. *Empty files are not saved.* If a permanent copy corresponding in name and type to the empty file exists, it is destroyed for permanent file tenancy cost reduction.

Modifiers: ALL,ECHO,ERROR,TERSE

ALL If this modifier is specified, *all* "file types" of the given file will be saved.

TERSE If this modifier is given, BASIC will *not* acknowledge that the files were saved, by printing the comment 'Done'.

Error Messages:

••• Illegal file name or there is no active file. •••

means that the file name specified was an illegal name or, that no name was specified and there was no active file to default to.

Examples:

```
/SAVE@¬TERSE FILE1
SA@A FILE1@D
SA FILE1@O
```

SCAN

Purpose: To find the occurrence(s) of a certain string in a file.

Prototype: /<u>SC</u>AN ['{string}'] [{file}]

Parameters:

<u>'</u> is an "edit" pattern delimiter character which may be any character that is neither alphabetic nor a asterisk (library file prefix).

<u>string</u> is the string to be found in the given file. If left out, this parameter defaults to the last pattern explicitly specified in a SCAN command. In any case, this parameter becomes the new "scan pattern".

<u>file</u> is the name of the file which is to be scanned for the given string. If explicitly specified, it becomes the "active file". If omitted, it defaults to the last file explicitly specified in a SCAN, EDIT, or OPEN command, i.e., the "active file". Regardless of where the file was explicitly specified (SCAN, EDIT, or OPEN), a line range may be given at the time of specification to restrict the editing or scanning.

Usage: This command causes BASIC to search the given file, starting at the current value of the "scan pointer", for the first occurrence (or, if the @ALL modifier appears, all occurrences) of the given "scan pattern". If the "verify switch" is on (default - may be turned off via /SET or by use of the @NOVERIFY modifier), each line in which the string is found is printed. The "scan pointer" is set to one plus the line number of the last line at which an occurrence was found. The "line pointer" is set to the line number itself, on success. If no occurrence of the string can be found, BASIC will print the comment:

```
      ••• String not found. •••
```

There are several ways in which the scan and line pointers may be changed. Explicitly giving a file name will set the scan and line pointers to the first line number in the file as specified (e.g., if a line range is given). The LINE# command may also be used to change them. Using the RESET command will cause them to be undefined. If the file name is defaulted, the scan and line pointer values are determined from the setting resulting from the last text-editing command issued (i.e., /SCAN, /EDIT, or /OPEN).

Effect: The given file is scanned from the "scan pointer" onward for occurrence(s) of the given string. If at least one occurrence is found, the "scan pointer" is set to the line number plus one of the line at the last successful occurrence. The "line pointer" is set to the line number itself. If no occurrence is found, both pointers are unchanged. If an edit pattern was successfully specified, it is retained. If errors are detected, the remaining parameters are ignored.

Modifiers: ALL,ECHO,ERROR,VERIFY

ALL If this modifier is specified, the entire file from the "scan pointer" onward will be scanned. This means that scanning will not stop until all occurrences have been found.

VERIFY If this modifier is specified, lines in which occurrences are discovered will be printed. The negation of this modifier, NOVERIFY, inhibits printing of these lines.

Error Messages:

••• Illegal or missing pattern. Command ignored. •••

means that the "scan pattern" was illegally formed, probably with a missing final delimiter.

••• No scan pattern defined. Command ignored. •••

means that no scan pattern is currently defined. One was never defined or a RESET command was issued.

••• Illegal file name or there is no active file. •••

means that the file was specified and was illegal or that no file was specified and there was no active file.

```
Examples:   /SCAN 'PRINT' F(10)
            SC F(67,999)
            SC@A@V 'PRINT' FILE
            SC
            SC $PRINT$ F(10,40)
            SC 1AB1CD1
```

SET

Purpose: To change the value of various "user-defined" system parameters.

Prototype: /SET {keyword}={value} [{keyword}={value}] ...

Parameters:

keyword represents one of the various "user-defined" system parameters the user may modify.

value is a permissible value for the given system parameter.

Usage: This command may be used to change the value of certain system parameters used by BASIC. These parameters are interrogated by BASIC to determine such things as what the default dimension of arrays should be. Below is the list of permissible keywords, their possible values, their default value, and a description of their meaning. The keywords may be abbreviated to any "initial substring" containing at least the letters underlined. A vertical bar symbol (|) will be used to separate alternative values for a keyword. These values may not be abbreviated. Again lowercase letters surrounded by braces ({}) will be used to indicate a generic type to be replaced with an item supplied by the user.

ATLINE=[ON|OFF|+|¬] Default: ON(+)

The value of this switch is used to determine whether or not a message indicating the current line in a running program should be printed on errors, after the STOP command, etc. The symbol plus (+) is recognized, as shown above, to mean "ON". The symbol not-sign (¬) is recognized to mean "OFF".

BACKSPACE=[ON|OFF|+|¬] Default: depends

If the current output device is recognized by BASIC as a "terminal", the value of this switch is used to determine whether or not "backspacing" may be done. That is, if the switch is "ON", BASIC assumes that "backspace characters" will be recognized. This switch is initially set according to the "device type" of the current "MTS source" (*SOURCE*). This variable is not reset by the RESET command.

COLWIDTH={integer} Default: 15
CWD={integer}

The value of this parameter is used to determine the width of an output field for the PRINT and MAT PRINT statements. When items in the "i/o list" for either of these statements are separated by commas, BASIC will place them in the output line at multiples of the value of this parameter. This value may be set to any unsigned integer between two (2) and fifteen (15), inclusive. CWD is an alternative name for COLWIDTH.

CONFIRM=[ON|OFF|+|¬] Default: ON(+)

BASIC interrogates this switch to determine if the user should be queried before DESTROYing, EMPTYing, or FREEing a file. The value of this switch may be temporarily overridden, for the duration of a single command, by the use of the @CONFIRM modifier (@CONFIRM implies ON, @NOCONFIRM (NC) implies OFF). The symbols plus (+) and not-sign (¬) are recognized to mean "ON" and "OFF", respectively. NOTE: This switch defaults to "OFF" in batch.

CONTCHAR={character} Default: depends

The value of this parameter is used to determine the single character to be used as an indicator for continuing an input line (anywhere in BASIC) on another input line. The continuation character is recognized as such only when it is the last character of a given line. Only one continuation line will be recognized. This character defaults to the current "MTS continuation character", initially. Changing it in BASIC will not change the corresponding parameter in MTS. It may be set to any character but blank.

DATAECHO=[ON|OFF|+|¬] Default: depends

The value of this parameter is used to control the printing of data lines being edited into the "active" file. The parameter being ON causes the data lines to be echoed; otherwise, they are not. The value is defaulted ON in batch mode and OFF at a terminal.

<u>D</u>EFDIM={integer}   Default: 10

The value of this system parameter is used to determine the default dimension of arrays which are not explicitly dimensioned by the DIM statement. It may be set to any unsigned integer between between 1 and 32767 inclusive.

<u>DEV</u>ICE=[TNPTR|PNPTR|2741|35]   Default: depends

The default is set to the device the user is using, e.g., a Model 35 Teletype. Setting this system parameter will actually affect several internal BASIC switches. These switches are interrogated in the formatting and generation of carriage control for messages originating with BASIC itself. It should be set to the "device type" of the current output device for BASIC. Besides the ones listed above, the device types "37" and "33" are also recognized. If none of these apply, the "closest approximation" should be used.

<u>DF</u>ILCRE=[ON|OFF|+|¬]   Default: ON(+)

The setting of this switch is used to determine if "data files" referenced by BASIC statements, while a program is running, should be automatically created if non-existent. When this switch is "ON", any reference by a WRITE FILE, READ FILE, BACKSPACE FILE, etc., to a non-existent "data file" will cause a "temporary" file with the given name (specified in the FILE statement) to be created. If this switch is "OFF", the program will suspend execution with an appropriate error message.

<u>DFILE</u>MP=[ON|OFF,+|¬]   Default: ON(+)

The setting of this switch is used to determine whether or not the "data file" for the program to be either "run", "debugged", or "compiled" should be emptied when the first data statement in the program is encountered (during translation). If this switch is "ON", the program's data file will be emptied <u>only if</u> the program itself contains data statements. If this switch is "OFF", then the lines in the text of the data statements are simply inserted into the data file at the corresponding line numbers. Since the user may create the data file and insert lines into it independently of the program (e.g., /OPEN, /NUMBER, etc.), the latter mode is desirable.

ECHO=[ON|OFF|+|¬]   Default: OFF(+)

This parameter setting determines whether or not command lines, including comments, should be echoed. If this switch is "ON", all command lines will be printed to the user as they were entered, except that the command itself will be converted to uppercase. The symbols plus (+) and not-sign (¬) are recognized to mean "ON" and "OFF", respectively. The setting of this switch may be temporarily overridden, for the duration of a single command, by the use of the @ECHO modifier (@ECHO implies ON, @NOECHO (NE) implies OFF). NOTE: This switch defaults to "ON" in batch.

ERROR=[COMPLETE|TERSE]   Default: COMPLETE

If this variable is set to "TERSE", error messages generated by command errors will not be printed. The error message will instead be saved and the input prefix for the next line will be set to the string "ERROR:", indicating that the previous command line was in error. If at this point the command WHAT? is entered, the error message will be printed. If anything else is entered, it will be processed as a "normal" command line. This behavior would be true even if in "numbering" mode or "include" mode, which would be suspended for only the one line prefixed by "ERROR:". If this parameter is set to "COMPLETE" (default), error messages will be processed normally. The setting of this switch may be temporarily overridden, for the duration of a single command, by the use of the @ERROR modifier (@ERROR implies COMPLETE, @NOERROR (NR) implies TERSE).

JUSTIFY=[LEFT|RIGHT|L|R]   Default: LEFT(L)

The value of this parameter is used by BASIC to determine whether items in the "i/o list" for the PRINT or MAT PRINT statements should be right- or left-justified within their fields. This justification is applied only for items separated by commas whose length is less than the current value of COLWIDTH. The strings "LEFT" and "RIGHT" may be abbreviated to "L" and "R".

LINELEN={integer}   Default: depends

The value of this parameter is used to determine the maximum length of an output line for the PRINT and MAT PRINT statements. If small enough, it may also cause some comments generated by BASIC to be truncated. It is initially set when the user "signs on" to the maximum length of a line which can be written to the device on which he is "signed on". For batch, this would be the width of a printer line (131). This variable is not reset by the RESET command.

RESTORE=[GLOBAL|LOCAL]   Default: GLOBAL

The setting of this system parameter is used to determine whether or not "data files" should be restored, i.e., rewound. When the user issues a command to run his program, his data files are initially restored. If he is in GLOBAL mode, every time a program is CALLed or CHAINed to, the data files referred to in the called program are restored. In LOCAL mode, the called program must give explicit RESTORE statements to restore files. Default operation is in GLOBAL mode. NOTE: In GLOBAL mode, files will not be restored until they are "read" or "written" for the first time.

SIGDIGITS={number}|OFF   Default: 7

The setting of this system parameter determines the number of significant digits to be printed on output. The {number} is the number of significant digits desired by the user. It must be an integer in the range 1 through 18. If it is specified out of range or OFF is specified, BASIC will use the default value.

STRCON={string} Default: "" (null)

The value of this parameter is used to initialize all string variables and string arrays when loading a program. It may be set to any legal BASIC string constant.

TERSE=[OFF|ON|¬|+] Default: OFF(¬)

The setting of this parameter is used to determine whether or not certain comments such as 'Done' or 'Ready' should be printed. If it is "ON", BASIC will not acknowledge that such commands as FREE, EMPTY, SAVE, DESTROY, and PERMIT have been successfully carried out by printing the comment 'Done'. If "OFF", the comment 'Ready' will also be suppressed when BASIC first enters "debug mode" after an error in a running program. The value of this switch may be temporarily overridden, for the duration of a single command, by the use of the @TERSE modifier (@TERSE implies ON, @NOTERSE (NT) implies OFF).

UNDFLOW=[OFF|ON|¬|+] Default: OFF(¬)

The setting of this system parameter is used to determine if the user should be notified of "floating-point underflows", i.e., exponent underflow. This parameter defaults to a value passed on by the master system, which is currently "OFF". This parameter is not reset by the RESET command.

VARCON={number} Default: 0.0

The value of this system parameter is used to initialize all numeric variables when loading a program. It may be set to any legal BASIC numeric constant.

VERIFY=[ON|OFF|+|¬] Default: ON(+)

The value of this parameter is interrogated by the SCAN command to determine if lines in which occurrences of the "scan pattern" are found should be printed. The behavior for the EDIT command is similar. The value of this switch may be temporarily overridden, for the duration of a single command, by the use of the @VERIFY modifier (@VERIFY implies ON, @NOVERIFY (NV) implies OFF).

Effect: The various system parameters specified are set to the values given.

Modifiers: ECHO,ERROR

Error Messages:

```
••• Illegal keyword and/or parameter. Command
ignored. •••
```

means that no keyword-parameter was specified. The command was therefore unnecessary.

```
••• "¤¤¤" is an illegal keyword. •••
```

means that ¤¤¤ was not recognized as a legal keyword.

```
••• "¤¤¤" is an illegal value for "###". •••
```

means that ¤¤¤ was not recognized as a legal value for the system parameter ###. Possibly, the user typed a substring of a legal value. Only the abbreviations detailed above will be recognized.

Examples:

```
/SET LINELEN=120
S U=¬
s strcon="Now is the time for all GOOD members"
S D=3,DF=OFF,ECHO=ON
```

SHARING

Purpose: To gain access to a BASIC file belonging to another user.

Prototype: /SHARING {file} {usid}

Parameters:

file is the BASIC name of the file which is to be gotten from another user. This parameter must be specified explicitly.

usid is the MTS "signon id" of the user from which the file is to be gotten. This parameter must be specified explicitly.

Usage: This command will "open" the file with the given name, and will initialize it from the permanent file of the same name belonging to the given user. A file with the same name and attribute must not already be in use (as a temporary copy obtained via OPEN, RUN, etc.) under the id of the user issuing the SHARING command. Unless the ALL command modifier is given, only the *one* file named will be so shared. To share another user's file, the file owner must have permitted it with the PERMIT command.

Effect: A file with the given name is "opened", being initialized from the permanent BASIC file of the same name belonging to the given user. The file is *not* made "active". It must be "opened" with the OPEN command to make it "active".

Modifiers: ALL,ECHO,ERROR,TERSE

ALL If this modifier is specified, *all* "file types" of the given file will be shared.

TERSE If this modifier is given, BASIC will *not* acknowledge that the file(s) were shared, by printing the comment 'Done'.

Error Messages:

••• Missing command parameter. - Command ignored. •••

means that one or both of the parameters to this command were omitted.

••• You already have opened "TXXXXXXX" - invalid sharing. •••

means that you are using a file (temporary copy) by name XXXXXXX and type T already.

••• USID’s "TXXXXXXX" file non-existent •••

means that the user with id "USID" has no such file with that name (XXXXXXX) and attribute (T).

••• USID’s "TXXXXXXX" file cannot be shared. •••

means that USID’s file XXXXXXX of type T is not permitted.

Examples:

```
/SHARING RETRRD W070
SH@A DINGO,MTA
```

SIGNOFF

Purpose:	To terminate operation of the BASIC subsystem.
Prototype:	/SIGNOFF [PAR=STAT]
Usage:	This command is a synonym for the QUIT command. See its description.

SINGLESTEP

Purpose: To enter or leave "singlestep mode" for a running program, in which program statements are executed _one at a time_ instead of continuously.

Prototype: /_SI_NGLESTEP [{switch}]

Parameters:

switch is either "ON" or "OFF" to cause BASIC to enter or leave, respectively, "singlestep mode". If missing, BASIC assumes "ON".

Usage: This command may be used to execute a BASIC program one line at a time. In "singlestep mode", before each statement is executed, the user is told the statement number and is given control to type in debugging commands. To cause the statement to be executed, a CONTINUE command should be issued. This mechanism gives the user a chance to look at a program in "slow motion" and to make corrections.

Restriction:
The SINGLESTEP command may be given only in "debug mode".

Effect: If "switch" is "ON" ("OFF"), BASIC enters (leaves) "singlestep "mode".

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there can be no program to "singlestep" through.

Examples:

```
/SINGLESTEP ON
SINGLE OFF
SI
```

START

Purpose: To start execution of a "loaded" BASIC program at the first "executable" statement.

Prototype: /<u>STA</u>RT

Usage: This command will cause the user's program to start execution at the first "executable" statement. Note that this command does not affect in any way the contents of any variables in the program. That is, variables will <u>not</u> be reset to the default values they had upon loading.

<u>Restriction</u>:
The START command may be given only in "debug mode".

Effect: The "loaded" program is started at its first "executable" statement.

Modifiers: ECHO,ERROR

Error Messages:

```
••• Not in Debug Mode. - No program currently
running. •••
```

means that the user is not in "debug mode". This command is therefore illegal since there can be no program to "start".

Examples:

```
/START
STA
```

STATUS

Purpose: To list those statistics which would be printed, if requested, when the user "signs off" BASIC.

Prototype: /<u>STAT</u>US

Usage: This command will print the cost, CPU time, elapsed time, and storage used up to the point the command is given. These factors are calculated from the time the user "signed on" to BASIC. These statistics change, of course, throughout a particular run. The STATUS command gives their current values.

Effect: The cost, CPU time, elapsed time, and storage used on the current run of BASIC up to the time the command is issued are printed.

Modifiers: ECHO

Examples:

```
/STATUS
STAT
```

STOP

Purpose: To terminate execution of the currently running program and return to "normal" BASIC command mode.

Prototype: /STOP

Usage: This command stops execution (running) of a "loaded" program and BASIC leaves "debug mode" and enters "normal" command mode. The program is unloaded. Note that this command does not stop BASIC itself. It only terminates "debug mode".

Restriction: The STOP command may be given only in "debug mode".

Effect: BASIC leaves "debug mode" and enters "normal" command mode.

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore unnecessary.

Examples:

```
/STOP
ST
```

SYSTEM

Purpose: To return the user to the "master system" (MTS) for a possible return to BASIC later.

Prototype: /SYSTEM [{text}]

Parameters:

text	is optional text (e.g., an MTS command) to be passed to MTS for processing.

Usage: This command returns the user to the "master system" (MTS). This return is made in such a way that the user may return to BASIC by issuing the appropriate system command ($RESTART).

Effect: The user is returned to the "master system" (MTS) and the text, if specified, is passed on to MTS.

Modifiers: ECHO

Examples:

```
/SYSTEM
SY
SYS $EMP MTSFILE OK
```

TRACE

Purpose: To enable the tracing of various actions the currently running program might perform, such as calling another program, returning to another program, or chaining to another program.

Prototype: /TRACE {keyword} [, {keyword}] ...

Parameters:

keyword is any of the following mnemonics, representing the indicated program action. The keywords may be abbreviated to any "initial substring" containing at least the letters underlined.

Keyword	Program action traced
CALLS	Every time a CALL is issued, the statement at which the CALL is made is printed as well as the program being CALLed. The user is then prompted for debugging commands for the CALLed program.
CHAINS	Whenever a CHAIN is issued, the statement at which the CHAIN was made is printed along with the program being CHAINed to. The user is then prompted for debugging commands for the CHAINed to program.
LINES	Before each statement is executed, its line number is printed.
OFF	All tracing is turned off.
PROGRAM	CALLs and CHAINs from the given program and RETURNs to the given program are traced. The program's name must be given immediately after the keyword, separated by at least one blank. Only one program at a time can

be traced in this manner. It will be the last one specified.

RETURNS Every time a RETURN is made to another program, the program to which the RETURN is being made is printed. The user is then prompted for debugging commands for the program RETURNed to.

Usage: This command turns on tracing of the specified program actions. It can enable the tracing of such actions as the CALLing of a program, CHAINing to a program, RETURNing to a program, or even the logical flow from statement to statement within the program. Tracing will not interfere with the actions caused by other debugging commands such as BREAKPOINT and SINGLESTEP.

Restriction:
The TRACE command may be given only in "debug mode".

Effect: Tracing of the specified program actions is enabled, or all tracing is disabled.

Modifiers: ECHO,ERROR

Error Messages:

••• Not in Debug Mode. - No program currently running. •••

means that the user is not in "debug mode". This command is therefore illegal since there is no program whose actions can be traced.

••• Illegal keyword and/or parameter. - Command ignored. •••

means that no keywords were specified or that the program name for the "PROGRAM" keyword was missing or that one of the keywords specified was not recognizable.

Examples:

```
/TRACE CALLS
TRA P MYPROG
T C CH L R
T O
```

TUTOR

Purpose: To invoke the BASIC tutorial system in order to learn about BASIC while using it.

Prototype: /TUTOR

Usage: Upon entering the command, the user will be able to select various parts of the BASIC interactive tutorial series to learn about BASIC's facilities such as command language, use of files, and programming language (variables, constants, computation, testing, loops, etc.). The series consists of lectures, programming exercises, and sample training programs.

Modifiers: ECHO

Examples: TU

UNNUMBER

Purpose: To turn off automatic numbering of input lines from the user to the "active" file.

Prototype: /UNNUMBER

Usage: These command causes BASIC to leave "numbering mode", in which line numbers are supplied to the user as input prefixes.

Note: The /-sign is necessary to indicate to BASIC that a command is being given. Typing "UNNUMBER" without the /-sign will enter those eight characters into the current line in the "active" file. An alternate way of leaving "numbering" mode is the issuing of an attention.

Modifiers: ECHO

Examples:

```
/UNNUMBER
/U
```

UNPERMIT

Purpose: To revoke the permission granted to other users for accessing the permanent copy of a BASIC file.

Prototype: /UNPERMIT {file}

Parameters:

file is the name of the BASIC file for which permission to access is to be revoked. This parameter must be specified explicitly.

Usage: This command will revoke the permission previously granted other users to access the given BASIC file. Unless the ALL command modifier is given, only the *one* "file type" specified will be unpermitted. No special actions will be noticed if the file is already unpermitted.

Effect: Permission to access the permanent copy of the specified file(s) is revoked.

Modifiers: ALL,ECHO,ERROR,TERSE

ALL If this modifier is specified, *all* "file types" of the given file will be unpermitted.

TERSE If this modifier is given, BASIC will *not* acknowledge that the file(s) were unpermitted; i.e., "Done" will not be printed.

Error Messages:

```
••• That is not a permanent file. •••
```

means that there was no permanent copy of the given file to "de-permit".

Examples:

```
/UNPERMIT FILE23
UNPER@T@A QUARD
UNP TISH@D
```

<u>WHAT?</u>

Purpose: To retrieve an error message which has been suppressed because the system parameter "ERROR" has been set to "TERSE".

Prototype: /<u>W</u>HAT?

Usage: If the system parameter "ERROR" is set to "TERSE" (with the SET command), error messages generated by command errors will <u>not</u> be printed. The error message will instead be saved and the input prefix for the next line will be set to the string "ERROR:", indicating that the previous command line was in error. This command will retrieve that saved error message. It must be given as input for the line prefixed by "ERROR:", or the error message will be lost. If given in any other context, BASIC will print the comment: ••• Nothing! ••• , indicating that there is no error message.

Effect: The saved error message, if any, is printed.

Modifiers: ECHO

Examples:

```
/WHAT?
W
```

WHERE?

Purpose: To find out the current line in a running program.

Prototype: /<u>WHE</u>RE?

Usage: This command will print the current line in the currently running program. The program name and "level", if greater than zero, will also be printed. The message will not be suppressed even if the "ATLINE" system parameter has be set to "OFF" (with the SET command).

<u>Restriction</u>:
The WHERE? command may be given only in "debug mode".

Effect: The current line, program name, and "level" (if greater than zero) of the currently running program are printed.

Error Messages:

```
••• Not in Debug Mode. - No program currently
running. •••
```

means that the user is not in "debug mode". The command is therefore illegal since there can be no program for which the information could be printed.

Examples:

```
/WHERE?
WHE
```

Documentation Commands

This section will describe a set of commands which were added to U of M BASIC for the express purpose of aiding in its documentation. These commands, which are available to any user of BASIC, are presented here separately because of their specialized function. All of these commands may be given at any time in "normal" or "debug" command mode. They are all prefixed with the character dollar-sign ($), and only one command modifier is applicable to them: the ECHO modifier. These commands perform such functions as spacing, page-skipping, and enabling and disabling of full uppercase and lowercase output for system messages. The commands and their functions are detailed below. The command name may be abbreviated to any "initial substring" containing at least the letters underlined.

$BATCH

causes BASIC to operate as if it were in "batch mode", wherein any command error or program error will cause the system to terminate operation.

$EJECT

causes a page-skip on a printer and 10 line skips on a terminal. Whether on not the current output device is a printer or a terminal, is determined by a switch internal to BASIC. This switch is initialized when the user first signs on, and may be changed only by SETting the user-defined system parameter DEVICE. See the SET command description for details.

$LC

enables full uppercase and lowercase output for system messages. The operation of this command overlaps with the action caused by SETting the "user defined" system parameter DEVICE. Depending upon the value assigned this parameter, uppercase and lowercase output will be either enabled or disabled.

$<u>NOB</u>ATCH

causes BASIC to operate as if it were in "conversational" mode. The user is discouraged from using this command in batch.

$<u>SP</u>ACE {n}

causes a skip of n lines, where n should be an unsigned integer. If it is missing or is not an integer, a skip of one line will be generated.

$<u>TI</u>TLE {string}

causes the string given to be processed by the BASIC "formatted printout" routine. The string should be delineated by quotes (") if it contains blanks or commas. The quotes will <u>not</u> be processed. For a description of what this routine does, set Section XII on the FORPRT processor.

$<u>UC</u>

disables full uppercase and lowercase output for system messages. <u>All</u> messages will be entirely in uppercase after execution of this command. Also, see the description of the $LC command.

V BASIC PROGRAMMING LANGUAGE

Described herein is the BASIC Programming Language as implemented at the University of Michigan. It differs in minor ways with extant versions of BASIC but still, for the most part, retains the essence of BASIC.

The U of M BASIC Programming Language has facilities for general computation and testing, input and printing of data, character string manipulation, matrix and vector operations, file input/output, built-in as well as user-defined functions, and a user-defined subroutine facility which admits fully-recursive procedures. Also noteworthy is the powerful CMD built-in function which allows the user to issue BASIC commands from a running program.

V.A <u>WHAT IS A BASIC PROGRAM?</u>

The following was aptly stated in the 1965 vintage BASIC manual by John G. Kemeny and Thomas E. Kurtz of Dartmouth College.

"A program is a set of directions, a recipe, that is used to provide an answer to some problem. It usually consists of a set of instructions to be performed or carried out in a certain order. It starts with the given data and parameters as the ingredients, and ends up with a set of answers as the cake. As with ordinary cakes, if you make a mistake in your program, you will end up with something else -- perhaps hash!"

A program is a set of directions to be given to the computer. It must be completely and precisely stated since the computer is not human and thus cannot infer what you mean by the directions. Moreover, the directions must be in a language that <u>both</u> you and the computer can understand and yet be simple enough for you to use. The BASIC language is user-oriented with the machine being taught to translate any BASIC program into its own machine language, thereby relieving the user of the necessity of learning the machine language. It's like being in a foreign country with an interpreter to assist you. Now for the details.

<u>Program (Definition of BASIC)</u>
A BASIC <u>program</u> is a sequence of statements in the BASIC Programming Language, each of which has a <u>unique statement number</u> preceding it. These statements are typed by the user into a BASIC source (S) file by means of the BASIC file-editing facility. The <u>name</u> of the program is the same name as that of the file which contains it. For example, the following sequence of input lines to BASIC (the first being a command line) will create a program PROG1 (consisting of four statements) and store it in the source or program file PROG1.

```
/OPEN PROG1
10 INPUT A,B
20 LET Y=A/B
30 PRINT A,B,Y
40 GOTO 10
```

The statement numbers also represent line numbers in the program file; hence, the statements may be typed in any order. This statement ordering by line number is a <u>physical ordering</u>. Note that the <u>logical ordering</u> (the order in which the statements are performed) is determined by the way that the program is written to be <u>run</u> or <u>executed</u>, that is, the physical order is used until a <u>control statement</u> such as statement 40 is encountered. In the above example, the logical ordering is 10, 20, 30, 40, 10, 20, etc.; that is, a loop consisting of four statements. A program consists of executable and/or non-executable statements. Non-executable statements simply define certain conditions or quantities to assist BASIC in running the program. They are not effective (i.e.; performed) while a BASIC program is being run, therefore, the user can never give control to them. The only non-executable statements in BASIC are DATA, DIMENSION, FILE, and REMARK (or equivalent). For example, the DIMENSION statement is used to declare the amount of storage space that the user will need for data. Once this statement has been interpreted by BASIC, it is no longer needed while the program is running since the space is allocated only once just prior to the running of the program. Executable statements, however, are performed while the program is running. The following concepts and definitions are necessary to the understanding of a BASIC program.

Program Name

The name consists of 1 to 7 letters or decimal digits, the first character being a letter, e.g., A, PROG1, LONGPGM.

Program Size

There may be no more than 500 statements in a BASIC program.

Statement Numbers

The statement numbers must be integers in the range 0 to 99999; each statement has a unique number. E.g., there cannot be two statements numbered 10. It is good idea not to number the statements with consecutive integers but rather by tens, for example, to provide space for inserting new statements between old ones. If one ever runs out of space for insertions (i.e., between consecutive line numbers), the /RENUMBER command may be used to obtain the required space.

Statement Syntax

There can be only one statement per line. It usually begins with a keyword (e.g., READ, PRINT) indicating the function to be performed. The remainder of the statement is determined by the specific keyword. For example, a PRINT statement contains a list of things to print, whereas a GOTO statement has a line number indicating where to transfer control. The maximum statement length allowed is 254 characters.

Keyword (Statement)

A keyword begins each BASIC statement to indicate the function that the statement is to perform. Almost all keywords have abbreviations (e.g., R’D for READ, W’E# for WRITE FILE, etc.). Some are optional (e.g., LET) since statement context is sufficient in these cases to determine the function to be performed.

Statement Continuation

If, while typing in a statement, the very last character of the physical line is a minus sign (-), then upon issuing a carriage return at the terminal, the user may continue typing the statement on the next input line. Note that the minus sign is not part of the statement. Also, the continuation character, which is initially the same as the MTS continuation character, may be changed via the /SET command. The prompting character for the continuation of the line is an asterisk (*).

Output Line Length

The maximum length of a line written by BASIC is 255 characters. Output lines, whether printed by the BASIC system or by a running BASIC program are printed as a sequence of carriage length segments if long lines are directed to a terminal. For a printer, the lines are truncated to the printer page width. For other output devices (e.g., magnetic tape, paper tape) the user should consult the appropriate MTS manuals (Volumes 1, 4, or 13).

Comments (in the Programming Language)

Any statement beginning with an asterisk (*) or the keyword REMARK is taken to be a comment statement, where the remainder of the line contains the comment. Also, comments may be placed on the same line after a BASIC statement, separated from the statement by the two adjacent characters /*. For example:

```
10 INPUT A,B   /* GET TWO NEW VALUES.
20 * THIS IS A COMMENT.
30 REMARK THIS IS TOO.
```

{} Convention
Statement Prototype Conventions

In all statement writeups, the braces {} surrounding a quantity in a statement means that that part of the statement is to be supplied by the user. All parts of a statement form not in braces must be typed exactly as specified by the prototype. For example, the GOTO statement prototype is GOTO {ln} where ln is a BASIC line number. The user may make substitutions for ln when composing a GOTO statement but must type the GOTO as is, e.g., GOTO 120.

A BASIC program is constructed by using primitive or fundamental building blocks to perform computation and other data manipulation processes. The next section provides precise definitions of the BASIC primitives.

V.B BUILDING BLOCKS OF THE LANGUAGE

The following constructions are intended to provide means of defining, performing computation on, and storing data.

Data Types

BASIC has two data types, double-precision numbers and character strings.

Variable

A quantity whose value may be changed.

Constant

A quantity which absolutely cannot be changed, e.g., a fixed number or string.

Number (Double-Precision)

A number in BASIC may be either positive, negative, or zero. It may be written with or without a decimal point, e.g., -10, +13, 69.8, -1.414, etc. It must be, in magnitude, in the range $.539760534693402789 \times 10^{-78}$ to $.723700557733226211 \times 10^{76}$ inclusive, and have at most 18 decimal digits or be exactly zero. For example, 3.14159265358979324. A number may be followed by the form Esnn to indicate multiplication by a power of 10, where nn is a one- or two-digit exponent and s is its sign. For example, 1.0E3 represents 1000, -1.2E-3 is equivalent to -0.0012, and 35E60 is 35×10^{60}.

Numeric Constant
Literal Data (Number)

A constant written according to the rules for representing a number in BASIC.

Variable (Simple)

Variable representing a single storage cell into which a number may be stored. Its name is either a single letter, or a letter followed by one or two decimal digits, e.g., A, B1, C19. There are 2,886 possible names.

Array

A one-dimensional list of cells (like a column of mailboxes in a post office) or a two-dimensional rectangular arrangement of cells (like a series of mailbox columns joined together appearing like a checkerboard with rows and columns). An array name is composed according to the rules for forming a simple variable name. An array, however, is distinct from the simple variable having the same name. For example, one may have a simple variable

A and an array A in the program with BASIC distinguishing them via context.

Subscript

A numerical result (actually an arithmetic expression, which is defined later) whose value when truncated to a whole number is used to refer to a specific cell in an array. Two subscripts are used for two-dimensional arrays to number the rows and columns while only one is necessary for one-dimensional arrays. The subscript must not exceed the maximum row (or column) number defined for the array being referenced.

Vector

A one-dimensional array whose cells are integer numbered 0, 1, 2, etc.; up to a maximum index m, called the dimension of the vector. A vector of dimension m has m+1 cells. The cells are referred to by following the vector name by a parenthesized subscript. For example, A(2) and A(I) are legal references where the latter reference depends on the value of I.

Matrix

A two-dimensional array whose rows and columns are integer numbered from 0 upward to the highest row and column numbers defined for the array, say m and n, respectively. The matrix is called a m by n (alternately m x n) matrix and it consists of (m+1)*(n+1) cells. A reference to a cell in the matrix is of the form a(i,j) where a is the matrix name and i and j are the row and column subscripts of the cell in question, e.g., A(1,0), B(0,6), A(2,3), and B(3*K,L), where the third reference is to the cell in row two and column three of the matrix A. The last reference depends on the current values of K and L.

String

A string is a sequence of 0 to 127 characters. It may contain letters, decimal digits, special punctuation such as commas and blanks, etc. See Appendix A for the list of permissible characters.

String Length

The number of characters in the string.

Delimiter

Character that is used on the left and right sides of a quantity to fix its bounds.

String Delimiter (")

This is the delimiter for string constants (see below).

String Constant
Literal Data (String)

A constant character string whose content is delimited by quotation marks. The delimiting quotes are not a part of the string. For example, "HELLO" and "STRING 33" are string constants of length 5 and 9, respectively. If the string is to contain a quote, it must be represented by two adjacent quotes. For example, "THE "" IS A QUOTE" has length 16 (the pair "" counts as one character).

Null String

This is the string of length zero. It is represented by the constant "".

String Variable

A string variable is a vector of cells, each of which may contain a BASIC string. The name must consist of two identical letters, or two identical letters followed by a single decimal digit; e.g., AA, BB9. The cells are referenced in the same manner as with numeric vectors (e.g., AA(3), BB(3*I+2)). The string name may be used to refer to the zeroeth cell (e.g., AA refers to AA(0)). There are 286 possible string variable names. But remember that since string variables are really vectors, each cell may act as a separate variable (e.g., AA(1), AA(7), etc.).

Dimensioning

The user must, in some way, allocate space for arrays (numeric or string). By default, vectors are of dimension 10 (11 cells) and matrices are of dimension 10 by 10 (121 cells). However, the dimension may be explicitly defined by means of the DIMENSION statement (see Section V.D).

Initialization of Variables Before Program Execution

Just before a BASIC program begins running, all of its numeric variables (including array elements) are set to zero. Likewise, all string variable cells are set to null (""). These preset values may be changed via the /SET command. The user can preset certain variables to possibly different values by generating a file (say the program's data file) and read the values into the variables at the beginning of the program. This may be accomplished by DATA and READ statements.

Built-In Functions

There are a number of built-in procedures in BASIC to perform computation frequently done by past computer users. For example, the function SQR(X) computes the positive square root of its argument X. See Section X.G or Appendix B.

<u>Files for Programs or Data</u>
A file is an ordered set of lines each having a unique integer line number in the range 0 to 99999. The file name must contain 1 to 7 letters or decimal digits, the first being a letter, e.g., PROG1.

<u>Data Files</u>
The files used for the BASIC file statements are data files (type D, e.g., PROG1@D). They may be generated via the command language, the BASIC file statements, or the DATA statement. See Section V.F.

<u>Program File (Source)</u>
<u>Source File (Program)</u>
A program file contains the BASIC source language statements which define the program having the same name as the file. It is of type S.

<u>Operator (General)</u>
An operator is a symbol (such as /, +, or *) which indicates an action to be performed on some data.

<u>Operand</u>
A data value (such as B, 1, C(1,2)) which is the subject of an operator.

<u>Binary Operator</u>
<u>Operator (Binary)</u>
An operator which requires <u>two</u> operands (e.g., the * in A*B).

<u>Unary Operator</u>
<u>Operator (Unary)</u>
An operator which requires only <u>one</u> operand (e.g., the - in -B).

<u>Operator Precedence</u>
<u>Precedence (of Operators)</u>
The precedence that one operator has over another. For example, multiplication is performed before addition in evaluating A+B*C; hence * has <u>higher</u> precedence than + (see Numeric Operators below).

<u>Equal Precedence Operators</u>
<u>Operators (Equal Precedence)</u>
Consecutive operators of equal precedence are performed from left to right. For example, the operators + and - are such that A+B-C is evaluated from left to right just as if (A+B)-C had been written.

Numeric Operators
Operators (Numeric)

Listed below are the BASIC numeric operators in _decreasing_ order of precedence with equal precedence on the same line. All are binary except as noted.

- (unary)	_unary_ minus (e.g., -B)
**	exponentiation (e.g., A**3 for A^3)
*,/	multiplication, division
+,-	addition, subtraction (e.g., A-B)

Arithmetic Expression
Expression (Arithmetic)

An arithmetic expression is used for computation with numbers. It will be defined recursively, that is, in terms of itself. As a basis, an arithmetic expression is either a numeric constant or variable (subscripted or simple); e.g., -15, A, B(3), C(I,J). Two arithmetic expressions when used as operands for a binary operator produce an arithmetic expression; e.g., 3/I, A+B, C(I,J)/4. An arithmetic expression when either preceded by a unary minus or surrounded by matching left and right parentheses is also an arithmetic expression; e.g., -A, (A+B), (1.5). Furthermore, a numeric built-in function value may be used anywhere a simple variable may be used in an expression; e.g., 3*SQR(X), A(COS(Q)*2,J), SIN(Z)**2+COS(Z)**2. As one can see, very complicated expressions can result. Examples follow.

```
-((A-B(I+3,J))/(SQR(X)+Z**2))
(B(K)+C(I,J))/2.0
(A+B)*((C+D)-(P/Q)**3)
A**B**C
(1)
```

Parentheses in Arithmetic Expressions
Arithmetic Expressions (Parentheses in)

Left and right matching parentheses may be used to alter the order of computation to suit the user when normal precedence does not provide the proper order. For example, if one wants to take the negative of the quantity A raised to the power B, -A**B will not work since unary minus has higher precedence than **. The expression so written is equivalent to (-A)**B, but the programmer may simply write -(A**B) to achieve the desired result.

Evaluation of Arithmetic Expressions
Arithmetic Expressions (Evaluation of)

An arithmetic expression is evaluated from left to right with consecutive operators of equal precedence being processed from left to right; otherwise, the higher precedence operators are performed first (again left to right). Listed below are arithmetic expressions with their order of evaluation indicated by parentheses.

Expression	Evaluation Order
-A**B	(-A)**B
A+B/2	A+(B/2) not (A+B)/2
A+B-1-D	((A+B)-1)-D
A+-B	A+(-B)
A**B**C	(A**B)**C
A*B-C*D	(A*B)-(C*D)
SQR(B*B-4*A*C)	SQR((B*B)-((4*A)*C))
A*B+C	(A*B)+C not A*(B+C)
A(I,2*J-1)/B**C	A(I,(2*J)-1)/(B**C)

Underflows in Computation

If the result of a computation is so small that it no longer can be represented in BASIC as a non-zero number (e.g., 10^{-100}), then the result is set to zero. This action is transparent to the user. If the user wishes to be notified of underflows, the command /SET UNDFLOW=ON should be used.

Overflows in Computation

If a computation results in a number too large for the computer to handle (e.g., a number larger in magnitude than 10^{75}), an error condition results during program execution and BASIC informs the user.

Errors (Computational, Number Representation)

Not all numbers can be exactly represented internally (in hexadecimal form) in the computer. Moreover, arithmetic computation may not produce exact results (e.g., 1/3 is not exactly one-third) due to fixed-size memory cells and arithmetic registers of the computer. The user is cautioned to keep these points in mind when doing general computation, comparing numbers (possibly a tolerance should be used), or computing indices (possibly a rounding procedure should be used). For example, 3/10*10 will yield a result slightly less than 3 since 3/10 cannot be exactly represented in the machine, but 3*10/10 will produce 3 exactly since 3 and 30 (3*10) are exactly represented in the computer. Hence, the order of computation can also affect the result. A general rule is that integers whose magnitudes are in the range zero through 72,057,594,037,927,935 (16^{14}-1) inclusive are exactly representable. If the result of an operation on integers in this range is also an integer in this range, it too will be exactly representable. Non-integer numbers (written with a decimal point and possible exponent, e.g., 0.1, 1.3E10) are not generally exactly representable, since numbers of this type are stored internally in the form: 16 to the integer power (-64 through 63) times $[n(1)/16^1+n(2)/16^2+\cdots+n(14)/16^{14}]$ where the n(i) range from

0 to 15 inclusive. Moreover, operations on these numbers may result in rounding and truncation as well as loss of significant low-order digits (e.g., adding a large number to a very small one). The user should be aware of this computational environment and its limitations.

Carriage Control Concept (Printing)
A character at the beginning of an output line which is used to control positioning of the paper on printing. It is interpreted only by the PRINT statement. For example, a blank for single space, 0 (zero) for double space, 1 for skip to the top of a page (6 spaces at a terminal), and & to suppress the carriage return after printing the line. See Appendix E for a complete list.

With the above constructs, the user is now ready to attempt some general computation as described in the next section.

V.C GENERAL COMPUTATION

This section deals with the statements that are common to all phases of the programming language, that is, numeric computation, terminal input and output, numeric testing, iteration, and program termination. Sections V.D through V.G deal with the more specialized aspects of the language such as matrices, strings, file operations, and procedures.

The self-explanatory program on the following page is designed to illustrate usage of the statements defined in this section.

Sample Gas Mileage Computation Program

```
 10  * THIS PROGRAM COMPUTES GAS MILEAGE.
 20  * IT INPUTS INITIAL AND FINAL ODOMETER READINGS
 30  * (ASSUMING A FULL TANK AT BOTH TIMES) AND
 40  * THE NUMBER OF TIMES GAS WAS PURCHASED IN
 50  * THE MILEAGE INTERVAL.
 60  * IT THEN READS GALLONS AND COST FOR EACH
 70  * PURCHASE AND COMPUTES AND PRINTS THE TOTAL
 80  * DISTANCE TRAVELED, TOTAL GAS USED, TOTAL COST,
 90  * AND OF COURSE THE GAS MILEAGE.
 100 * IT ALSO ERROR CHECKS THE DATA.
 105 PRINT "1*** GAS MILEAGE PROGRAM ***"
 106 PRINT "ENTER INITIAL AND FINAL ODOMETER READINGS"
 107 PRINT "&AND NUMBER OF GAS PURCHASES"
 110 INPUT O1,O2,P
 120 LET T=O2-O1    /* TOTAL DISTANCE
 130 IF O2 > O1 THEN 190   /* WAS IT DRIVEN FORWARD?
 140 IF O2 = O1 THEN 170   /* DID IT MOVE?
 150 PRINT "YOU WENT BACKWARDS!"
 160 STOP    /* FOR SUCH A USER, TERMINATE THE PROGRAM
 170 PRINT "YOU WENT NOWHERE!"
 180 STOP
 190 IF P <> 0 THEN 220
 200 PRINT "YOU MUST HAVE PUSHED THE CAR!"
 210 STOP
 220 G=0     /* ACCUMULATIVE GALLONS
 230 C=0     /* ACCUMULATIVE COST
 235 PRINT "ENTER THE";P;" GALLON-COST PAIRS."
 240 FOR I=1 TO P   /* READ ALL PURCHASES
 250 INPUT G1,C1    /* GALLONS, COST
 260 IF G1 < 0 THEN 360   /* NEGATIVE GAS?
 270 IF G1=0 THEN: IF C1>0 THEN: PRINT "HIGHWAY ROBBER!"
 280 IF C1<0 THEN: PRINT "SUCH A DEAL - WHERE'S THE STATION?"
 290 G=G+G1    /* ACCUMULATE THE GALLONS
 300 C=C+C1    /* LIKEWISE FOR COST
 310 NEXT I     /* THE END OF THE READ LOOP
 320 PRINT "YOU TRAVELED ";T;" MILES AND USED"
 330 PRINT G;" GALLONS AT A COST OF ";C;" DOLLARS."
 340 PRINT "YOUR GAS MILEAGE IS "; T/G ;" MPG."
 350 STOP     /* NORMAL END.
 360 PRINT "YOU DRAINED THE TANK? - TRY AGAIN!"
 370 GOTO 250
```

Example 11. Sample Gas Mileage Computation Program

REMARK
*
/* Convention
Comments (Program Documentation)

Purpose: To insert remarks into the BASIC source program for documentation purposes.

Prototype:

```
REMARK {com}
* {com}
```

where:

com is any string of characters to be used as a comment.

Abbreviations:	REM for REMARK
	none for *

Effect: The statement is *non-executable* and should not be transferred to.

Note: It is also possible to generate comments in BASIC by placing /* {com} at the end of *any* BASIC statement.

Examples:

```
10 REMARK HI
20 REM  MULT. BY A
30 *    READ SOME NUMBERS
40 INPUT X,Y  /* GET THEM FROM THE TTY
```

LET (Numeric Assignment)
Numeric Assignment (LET)
Assignment (Numeric LET)

Purpose: To assign a numerically computed value to a numeric variable.

Prototype: LET {v} = {exp}

where:

- exp is an arithmetic expression.
- v is a numeric variable (possibly subscripted) into which the value of exp is to be stored. A, B(1,2), and C(J) are examples of legal numeric variables.

Abbreviation: The LET is optional.

Effect: The expression exp is evaluated and the result is stored in v. This = is an assignment = as contrasted with the = used with the IF statement to make comparisons. Context tells BASIC what kind of = it is dealing with.

Note: The user may be tempted to write A=B where A and B are both arrays in order to transfer the contents of the cells of B to A. This is incorrect. The MAT LET (assignment) statement should be used.

Moreover, if A and B were matrices, A=B would not even refer to them but instead to the simple variables A and B since one may have in BASIC a simple variable with the same name as an array.

Examples:

```
10 LET A=B(1,2)+4.1
20 B(3,J)=CMD("/OPEN X")
30 C(I,J)=A(I,K)*BB(K,J) /* MATRIX MULTIPLICATION
40 I=I+1      /* INCREMENTING A COUNTER
50 C(SQR(Z+Q)*I+ATN(P),J)=(RND(X)+COS(W))/3
60 S=S+A(I)    /* ACCUMULATING A SUM OF NUMBERS
70 P=P*B(K)    /* FORMING A PRODUCT
```

<u>INPUT (Terminal Input)</u>
<u>PRINT (Terminal Output)</u>
<u>Input (Terminal - General)</u>
<u>Terminal Input (General)</u>
<u>Output (Terminal - General)</u>
<u>Terminal Output (General)</u>

Purpose:	To read data from or print output on a terminal.

Prototypes:

```
INPUT {ilist}
PRINT {olist}
```

where:

<u>ilist</u> is a list of BASIC <u>variables</u> and/or SKP references separated by commas. The variables may be numeric or string. They may be subscripted (referring to matrix or string array elements). For example, the statement INPUT A,B(3),C(4,J),AA,BB1(3) will read three numeric values followed by two character strings from the terminal. INPUT A,SKP(2),B will read one number, skip the next two, and read the fourth number into B. See the SKP writeup for details (the SKP argument range is 0 to 1000, inclusive).

<u>olist</u> is a list of BASIC variables (subscripted or not), constants, complex arithmetic expressions, and/or TAB references separated by format characters. The format characters allowed are commas (,), semi-colons (;), and colons (:). See Section V.I on Special Input/Output Controls for details of their effect. String variables and constants are allowed, but string expressions are <u>not</u>. If the list is null, a blank line is printed.

Abbreviations:	I'T for INPUT P'T for PRINT

Effect: <u>INPUT</u> is used to read user-supplied data values at a terminal. BASIC prompts the user to type in data by issuing a question mark (?) prefix character. The data items are to be typed in separated by either blanks or commas. If a string being read in has blanks or commas within it, it should be enclosed in quotation marks ("). For example, 3,AB,"AB CD ,3" is a list of a number, the string "AB", and the third string which has commas and blanks in it. The first data item typed in is stored in the first variable on ilist, the second item into the second variable, etc., until <u>all</u> variables on the list have been read into. Each time an INPUT is issued a new line is requested, so it is impossible, for example, to use two INPUT statements to read a single input line. If the user ends an input line before typing all the data items to be read, BASIC will prompt him for more data.

To terminate (STOP) the currently running program when an INPUT is given, the user need only type /ENDFILE or give an end-of-file. BASIC will then return to command mode.

<u>PRINT</u> is used to print out data values contained in the program issuing the PRINT statement. The output line is divided (by default) into 15 column-wide fields (up to the maximum line length for the terminal device, e.g., 5 fields for a 75-column teletype). Each data item in olist is printed in order as given in the list left-adjusted at the next available field boundary. For example, PRINT A,B will print A starting in column 1 and B in column 16. If all the output does not fit on one line, successive items will go on new output lines as necessary. The adjustment within the field and the field width may be changed via the /SET command parameters JUSTIFY and COLWIDTH, respectively. Also the user may tab to a column before printing or print packed output. See Section V.I on Special Input/Output Controls. If the first list item is a string, its first character will be interpreted as carriage control. If it is a legal carriage control, it is used and removed from the line; otherwise, single spacing (blank) is assumed with the line being unaltered. See Appendix E for a complete list of available carriage controls.

Note: The input and output lists are completely evaluated prior to reading or printing; hence for this version of BASIC, one cannot read two numbers (in a single INPUT statement) where the first is a subscript for the second variable (e.g., I, A(I)) since the subscripts are currently evaluated prior to processing the list. Thus, if I=2 and one issues INPUT I,A(I), then when the user types the line 4,5, a 5 is read into A(2). One can, however, issue two INPUT statements, one to read the subscripts and then one to read into the array elements corresponding to those subscripts. Another alternative is to use the INP function as a subscript for the first reference to a subscript in an input list. For example, if the statement INPUT A(INP(I)),B(I) is given and the user types in response a 4 (when prompted by an = sign) followed by 5,6 (when prompted by a question mark), then I is assigned 4 and A(4) and B(4) are assigned 5 and 6, respectively.

For output lists, being able to write expressions as a list item creates an interesting situation. First of all, note that each element (going from left to right) on the output list is evaluated. If it is a variable (subscripted or not), its name is recorded. If it is a constant or complex expression, the value is recorded. This list of variable names and data values is then processed from left to right for printing, with the variable names being used to access the variables' values. Note that due to this second pass over the list, a variable occurring on the list before an expression which could modify it on the first pass may contain a value that one might not expect. For example, if one writes PRINT A,INP(A) then the INP function will read into A a new value. Then the PRINT statement, instead of printing the old value followed by the new, will print the new value twice.

Examples:

```
10 * READING FROM THE TERMINAL
20 INPUT A,B
30 INPUT A,SKP(3),B
40 INPUT A, BB(1), C(1), C(2)
50 * PRINTING AT A TERMINAL
60 PRINT A,B
70 PRINT "ANS =";W
75 PRINT "X",TAB(30),"COST"
80 PRINT X,TAB(30),C
```

INP (Terminal Input)
OUT (Terminal Output)
Terminal Input (INP Numeric)
Terminal Output (OUT Numeric)

Purpose: To allow simple numeric terminal input and output in arithmetic expressions.

Prototypes:

```
{r}=INP({v})
{r}=OUT({exp})
```

(INP and OUT are built-in functions.)

where:

r	represents the numeric value of either INP or OUT.
v	is a simple numeric variable to be read into from the terminal.
exp	is an arithmetic expression whose value is to be printed at the terminal.

Effect: INP({v}) is equivalent to the statement INPUT {v} except that the prompting character is an equal (=) rather than a question mark (?). Furthermore, INP returns as its value r which is whatever was read into v.

OUT({exp}) is equivalent to the statement PRINT {exp}. Furthermore, OUT returns as its value r, the value of exp.

Note: These functions allow complex statements such as this one-line program to read two numbers and print their sum.

```
10 GOTO (10,20,30), I+OUT(INP(X)+INP(X))
```

More practical examples follow.

Examples:

```
5 * READ IN AT THE TERMINAL TWO
6 * SUBSCRIPTS FOLLOWED BY THE
7 * CORRESPONDING ARRAY ELEMENTS.
```

```
10 DIM A(10)
20 INPUT A(INP(I)),A(INP(J))

22 * DEFINE A FUNCTION WHICH PRINTS OUT
23 * ITS VALUE (AX+B) AFTER READING IN ITS
24 * COMPONENTS A AND B.
30 DEF FNP(X)=OUT(INP(A)*X+INP(B))
40 Z=FNP(5)
41 * WOWIE TWO INPUTS, A PRINT, AND A FCN EVAL
```

<u>GOTO (Simple)</u>
<u>Branching (Simple)</u>

Purpose: To transfer control from one part of a BASIC program to another.

Prototype: GOTO {ln}

where:

<u>ln</u> is the line number of an executable statement in the program issuing the GOTO.

Abbreviation: GO

Effect: Control is given to the statement having that line number.

Note: Control may not be transferred to non-executable statements such as REM, DIM, DATA, etc.

Examples:

```
25 GOTO 10
30 GO 75
```

GOTO (Computed)
Branching (Many way - Computed)

Purpose: To transfer control from one part of a BASIC program to another chosen from a list of possibilities.

Prototype: GOTO ({lnlist}), {exp}

where:

<u>lnlist</u> is an ordered list (first, second, etc.) of line numbers of executable BASIC statements in the program issuing the GOTO. The line numbers must be separated by commas in the list. For example, 10,100,5 is a three-element list.

<u>exp</u> is an arithmetic expression whose numeric value (rounded down to the nearest integer) is used to select a line number from the list. The comma separating the expression from the line number list is optional.

Abbreviations: GO
The comma before the expression is optional.

Effect: The value of the expression is computed and error checked for being between one and the number of elements of the list inclusive. If the value is illegal (e.g., 4 or -1 for a three-element list), an error comment is given and program execution is terminated (or suspended); otherwise, the value is used as an index into the line number list to select the statement to transfer to. For example, if A=4 and B=0.5, then GOTO (30,70,10,6), A*B will transfer to statement 70, which has the second line number.

Note: Control may not be transferred to non-executable statements such as REM, DIM, DATA, etc.

```
Examples:       30 GOTO (40,1,999),3*A
                70 GO (40,100),B
                10 GO(10,10,5,30) 6*SQR(X)
```

IF (Algebraic)
Comparisons (Numeric)
Numeric Comparisons

Purpose: To test relationships between numeric data or expressions and conditionally either transfer control to another part of a BASIC program or execute another BASIC statement.

Prototypes:

```
IF {exp1}{reln}{exp2}, THEN {ln}
IF {exp1}{reln}{exp2}, THEN ({lnlist}), {exp3}
IF {exp1}{reln}{exp2}, THEN:  {Bstmnt}
```

where:

ln	is a BASIC line number just as for a simple GOTO.
lnlist	is an ordered list of line numbers of executable BASIC statements just as for a computed GOTO; e.g., 10, 35, 5.
exp3	is an algebraic expression whose value is used to select a line number from lnlist just as for a computed GOTO; e.g., 3*A+B.
Bstmnt	is an executable BASIC statement (without a line number preceding it); e.g., PRINT "HELLO" or Y=A*B+SQR(Z).
exp1	is an arithmetic expression.
exp2	is an arithmetic expression.
reln	is a relational operator to be used to compare the values of exp1 and exp2. The following operators are allowed:

=	is equal to
>	is greater than
<	is less than
<>	is not equal to
<=	is less than or equal to
>=	is greater than or equal to

As alternates for the last three operators one may write:

¬=	is not equal to
¬>	is not greater than
¬<	is not less than

No other relational operators are allowed.

Abbreviations: The comma before the THEN is optional.

Effect: The expressions exp1 and exp2 are evaluated and compared for satisfying the relation specified by reln (e.g., A>B). If the relationship does not hold, then control passes to the next executable statement after the IF. If the relationship holds (e.g., A is greater than B), then control is given to the part specified after the THEN. In the case of the first two prototypes, the control is handled like the simple and computed GOTOs, respectively. For the third prototype (note the colon after the THEN), control is given to the executable BASIC statement Bstmnt. If Bstmnt is not a GOTO or RETURN or a statement that terminates the program, then after it has been completed, control is passed on to the statement after the IF. In the case of Bstmnt being either a CALL or a GOSUB, it is completed when a return has been made to it.

Examples:

```
10 IF X=Y THEN 100
20 IF I<=N THEN (30,70,20),I
30 IF B(I,J)>A(K,J) THEN:  P=3*Q
40 IF P>Q THEN:  IF Q>R THEN 301
50 *
60 * FOR LOOP EXAMPLE
70 I=1
80 IF I>N THEN 110
90 A(I)=1-B(I)
100 I=I+1
105 GOTO 80
110 MAT PRINT A
```

<u>FOR (Looping)</u>
<u>NEXT (Looping)</u>
<u>Looping</u>
<u>Repetitive Processes</u>

Purpose: To repeatedly execute a series of BASIC statements.

Prototypes:

```
FOR {v}={i} TO {l}
FOR {v}={i} TO {l} STEP {s}
NEXT {v}
```

where:

<u>v</u> is a numeric non-subscripted loop <u>v</u>ariable; e.g., I, A12, B7.

<u>i</u> is an arithmetic expression whose numeric value is used to <u>i</u>nitialize the loop variable v; e.g., A*B, 3*SQR(X), 1.

<u>s</u> is an <u>optional</u> arithmetic expression whose numeric value is used as an increment or <u>s</u>tep to be added to the current value of the loop variable v to produce a <u>new</u> value for v. If s is not given, a value of 1 is assumed; e.g., 1, 3*COS(X), T+W.

<u>l</u> is an arithmetic expression of the <u>l</u>imit on the value of the loop variable v.

Abbreviations: N'T for NEXT
none for FOR

Effect: The series of statements to be repeated should be begun with a FOR statement for a loop variable v and terminated by a NEXT statement for that <u>same</u> variable. Execution proceeds as follows: Upon entering the FOR-NEXT loop the expression i is calculated and stored in the loop variable v. The limiting value l is then computed and compared with v. The type of comparison is controlled by the algebraic sign of s. For non-negative s values, if v is greater than l, the loop terminates and control is passed to the statement following the NEXT v; otherwise, the loop is entered. When s is negative, the comparison with l is a "less than" comparison. That is, the loop terminates if v is less than l; otherwise, the loop is entered. The series of statements within the loop are executed until the NEXT v corresponding to the FOR is encountered. The step s and limit l are computed with s being added to v to produce a new value of v. For non-negative s, if the new v value is greater than l, the loop terminates as before; otherwise, it continues. For negative s, the less-than comparison is made as explained before.

FOR-NEXT loops may contain inner FOR-NEXT loops as long as the inner loops are completely contained (_nested_) in the outer loops. Loops on the _same_ variable may be nested but the user should exercise caution since non-terminating (infinite) loops may result. Also, a NEXT statement always occurs later (has a higher statement number) than the corresponding FOR statement. One may transfer in and out of these loops freely.

Examples:

```
10 REM ADD THE ODD INTEGERS FROM 1 TO 17
20 S=0.0
30 FOR I=1 TO 17 STEP 2
40 S=S+I
50 NEXT I
55 *
60 REM NOW ADD TWO MATRICES THE HARD WAY
70 DIM A(3,5),B(3,5),C(3,5)
80 FOR I=1 TO 3
90 FOR J=1 TO 5
100 C(I,J)=A(I,J)+B(I,J)
110 NEXT J
120 NEXT I
130 *
140 * A COMPLEX EXAMPLE.
150 FOR P=A*B TO 3/SQR(Z) STEP B*Z-A
160 S=S*P
170 Z=Z+1.0      /* THIS ALTERS THE LOOP LIMIT.
180 NEXT P
185 *
190 REM ADD THE ODD INTEGERS FROM 1 TO 17
200 REM ONLY DO IT BACKWARDS.
210 S=0.0
220 FOR I=17 TO 1 STEP -1
230 S=S+I
240 NEXT I
```

Terminating or Suspending Program Execution
Stopping a Running Program (Methods of)

There are numerous ways to stop the execution of a program.

(1) Executing the STOP statement.

(2) Running off the end of the program (i.e., trying to go past the physically last statement in the program).

(3) Giving an end-of-file via the terminal control (e.g., control-C at most teletypes) or via the /ENDFILE command when your program is trying to read input from the terminal.

(4) Hitting the attention key (e.g., BREAK on most teletypes). This is useful when one realizes after starting up a program to run that there are errors (perhaps a loop). BASIC will enter interactive debugging mode (see (7)).

(5) Generating an error in your program that BASIC detects (e.g., dividing by zero). BASIC notifies you of the error and enters interactive debugging mode (see (7)).

(6) Executing a PAUSE or WAIT statement. BASIC normally goes into interactive debugging mode (see (7)).

(7) No matter how you arrived there, if you are in interactive debugging mode (> prefix), you may either type /STOP or issue debugging commands. The only case where this mode is skipped is when one executes via /EXECUTE a program with the debug option off (the default).

(8) Calling on the CMD function from the program to issue a stop.

PAUSE (Suspend Execution)
WAIT (Suspend Execution)

Purpose: To suspend execution of a BASIC program until the user wishes to continue.

Prototypes:

```
PAUSE {com}
WAIT {com}
```

where:

com is an *optional* comment that may follow the PAUSE or WAIT statements. This comment is *not* printed during execution.

Abbreviations: P’E for PAUSE
none for WAIT

Effect: WAIT or PAUSE cause the program to type a question mark and suspend execution until the user gives a carriage return. If the user types STOP and gives a carriage return, all execution ceases and control is returned to BASIC command mode. Any other comment typed between the question mark and the carriage return is allowed and ignored.

Note: If the program is running in debug mode (via the /DEBUG or /RUN commands), the normal debug mode prefix character > is used instead of the question mark. The user may then use any debugging commands to modify or display variables, set breakpoints, etc. To continue the program in this case, one simply types /CONTINUE (or C).

These statements are useful for pausing to allow the user a chance to display some temporary results before the program continues.

Examples:

```
20 PAUSE
60 P’E  THE PAUSE THAT REFRESHES
70 WAIT
```

STOP (Terminate Execution)

Purpose: To terminate execution of a BASIC program (or programs).

Prototype: STOP {com}

where:

com is an *optional* comment that may follow the STOP statement. This comment is *not* printed during execution.

Abbreviation: S'P

Effect: If the program is *not* in debug mode, it and all other programs which may have called on it (e.g., PROG1 calls PROG2 which calls PROG3 which issues a STOP) are terminated and control is returned to command mode. In debug mode, however, the user is notified that the STOP has occurred and he is given control to issue BASIC debugging commands.

Note: STOP stops everything while END just stops the program issuing the END and RETURN does the latter conditionally.

Examples:

```
30 STOP ASAP
500 S'P THE RACER'S EDGE
10 STOP
180 IF A=B THEN: STOP
```

V.D <u>MATRIX AND VECTOR OPERATIONS</u>

<u>Array Storage and Accessing Concepts</u>
<u>Matrix (Main Definitions)</u>
<u>Vector (Main Definitions)</u>

A <u>matrix</u> is a two-dimensional rectangular <u>array</u> having rows and columns (like the horizontal rows and vertical columns of a checkerboard). If the number of rows equals the number of columns, the matrix is <u>square</u>. The rows and columns are numbered from 0 to m and from 0 to n, respectively, in integer values. Hence, there are m+1 rows and n+1 columns totaling (m+1)*(n+1) cells for storing numbers. The values m and n are called the <u>row and column dimensions</u> of the m by n matrix. The <u>matrix dimension</u> is m by n. A <u>row-vector dimension</u> is m by 0 while a <u>column-vector dimension</u> is 0 by n. The user may refer to elements of a matrix (say A) by giving the row and column indices (subscripts). For example, A(2,3) refers to the element of A in the second row and third column. Column vectors (one-dimensional arrays) may be referenced via a single subscript (e.g., A(2), A(I)); however, they are treated in BASIC as two-dimensional arrays with BASIC automatically generating a zero second subscript if it is not supplied. For example, A(3) is the same as A(3,0). This allows redimensioning of one-dimensional arrays as two-dimensional and vice versa. Row vectors are referenced via two subscripts, the first being zero. E.g., B(0,I) or B(0,5). Note that in BASIC, numeric arrays may be either one- or two-dimensional, while string arrays are strictly one-dimensional.

<u>Redimensioning (Matrix-Vector)</u>

The user assigns space for these arrays via DIMENSION statements. This fixes the maximum size of each array. He may, however, redimension the arrays within these fixed bounds (e.g., make a 3 by 3 matrix into a 2 by 4 or a 9 by 0 matrix). This may be done while reading the matrices (from the terminal or BASIC data file) or by the matrix RDM built-in function.

<u>Matrix Facility</u>

BASIC provides statements to read and write matrices at a terminal or to BASIC data files. Moreover, there are statements to add, subtract, multiply, divide (solve simultaneous linear equations), scalar multiply, unary negate, transpose, invert, and provide space for matrices. Also, there are statements to generate some standard matrices such as the identity. One may also assign one matrix to another. The specifications follow.

DIMENSION (Reserve Array [Vector or Matrix] Space)
Matrix (Space for)
Vector (Space for)

Purpose: To allocate space for arrays, that is, vectors or matrices.

Prototype: DIMENSION {dlist}

where:

dlist is a list of names of singly or doubly dimensioned numeric arrays and/or singly dimensioned string arrays. Each numeric array name is followed by either the form (m,n) or (m), the latter being equivalent to (m,0). String names are followed by the form (m) which is strictly a single-dimension form (i.e., not equivalent to (m,0)). The components m and n are non-negative constants. For example, A(15), B(2,3), and SS(5).

Abbreviation: DIM

Effect: A fixed number of cells are permanently reserved for the course of the running of the program (e.g., each time a /RUN is given).

For doubly dimensioned arrays, (m+1)*(n+1) cells are reserved for storage of data. This allows for zero-indexed rows and columns. For example, DIM A(2,2) sets aside space for cells A(0,0), A(0,1), A(0,2), A(1,0), A(1,1), A(1,2), A(2,0), A(2,1), and A(2,2) in that order. For singly dimensioned arrays, m+1 cells are reserved. For example, DIMENSION A(3) sets aside space for the numeric cells A(0), A(1), A(2), and A(3), in that order. This is equivalent to DIM A(3,0). Likewise, DIM AA(2) provides space for the strings AA(0), AA(1), and AA(2).

Stated simply, one receives one more row and one more column than requested for two-dimensional arrays (namely, a zero row and zero column). For one-dimensional arrays, one receives one extra cell, namely, the zeroeth.

Column vectors are of dimension m by 0 and row vectors 0 by n; e.g., DIM X(3,0) Y(0,3).

RULE: The dimensioning of a variable must precede its usage (i.e., the DIM statement must have a lower line number than any statement which uses whatever is dimensioned in the DIM statement).

Note: If no dimension statements are given by the user, default dimensioning is used, that is, 10 by 10 for doubly subscripted arrays and 10 for singly subscripted arrays (numeric or string). This default may be changed via the /SET command on the keyword DEFDIM (e.g., /SET DEFDIM=5) prior to making a run for which the default dimensioning is to be changed.

It is possible to refer to a two-dimensional numeric array (matrix) in a singly subscripted manner. The result is a reference to the 0 column of the matrix (e.g., A(2) refers to A(2,0)). Moreover, a singly dimensioned numeric array may be referred in a doubly subscripted manner (e.g., A(2,0) refers to A(2)). Also, one can redimension a singly dimensioned numeric array to be two-dimensional or vice-versa. (See the matrix RDM function and redimensioning on matrix input from a terminal or matrix read from a file.) The only constraint is that the redimensioning does not require more cells than reserved via the original dimensioning.

The only way to change the total number of cells reserved for each array is to retype the dimension statement and rerun the program.

Examples:

```
10 DIM A(5),B(2,3)
20 DIM BB2(15),C(0,3),D(3,0)
```

Now an example with redimensioning.

```
10 DIM A(8) /* NOW HAVE 9 CELLS
15 A(2)=5 /* A VALID STATEMENT
20 MAT A=RDM(2,2)
```

```
30 * A IS NOW 2 BY 2, I.E., (2+1)*(2+1)CELLS
35 A(2,1)=69 /* A VALID STATEMENT
40 MAT A=RDM(7,0)
50 * A IS NOW ONE-DIMENSIONAL
55 A(6)=13 /* A VALID STATEMENT
60 A(8)=12 /* ILLEGAL
```

Statement 60 is illegal since 8 exceeds the new dimension (i.e., 7) of A. Note also that DIMensioning fixes the number of cells for an array and gives it an initial structure, while ReDiMensioning is a dynamic process during the running of a program which allows size changes within the confines of the original dimensioning as well as structural modifications.

MAT INPUT (Terminal Matrix Input)
MAT PRINT (Terminal Matrix Output)
Input (Terminal - Matrix)
Terminal Input (Matrix)
Output (Terminal - Matrix)
Terminal Output (Matrix)

Purpose: To read entire matrices from or print them out on a terminal.

Prototypes:

```
MAT INPUT {milist}
MAT PRINT {molist}
```

where:

<u>milist</u> is a list of BASIC numeric matrix names separated by commas. Each name may be followed by (m,n) which represents new row and column dimensions for the particular matrix. The dimensions m and n should be nonnegative arithmetic expressions subject to the restriction that (m+1)*(n+1) must be less than or equal to (M+1)*(N+1) where M and N are the original row and column dimensions specified by a dimension statement or defaulted. If the form (m,n) follows a matrix name, the matrix is redimensioned (as if RDM was used) prior to issuing the INPUT statement. For example, if A was originally dimensioned 3 by 3, then MAT INPUT A(4,2) reads 8 numbers into the redimensioned 4 by 2 matrix, while MAT INPUT A would read 9 numbers into the 3 by 3 matrix.

<u>molist</u> is a list of BASIC matrix names separated by commas or semi-colons. Normally, the output is printed left-adjusted in the 15-column wide fields just as for PRINT. However, if a semi-colon follows a matrix name, that matrix is printed in the packed form just as for PRINT; for example, MAT PRINT A,B;C,D; etc.

Abbreviations: M'P for MAT PRINT
M'I for MAT INPUT

Effect: MAT INPUT reads entire matrices row-wise (e.g., A(1,1),A(1,2),A(2,1),A(2,2),A(3,1),A(3,2) for a 3 by 2 matrix A) from a terminal in the same manner as the normal INPUT statement. Only the elements of positive index are read (except for the case of row and column vectors). There is continuous prompting with a question mark prefix (?) until the entire input list has been read in. The program may be terminated via a /ENDFILE or an end-of-file control on the terminal.

MAT PRINT prints matrices row-wise (see above) at a terminal with each row beginning a new line of output. If a row is too long to be printed on one line, it is printed on sucessive new lines. Only the elements of positive index are printed (except for row and column vectors). The output is printed in the 15-column fields unless packing is specified, in which case a blank separates one data item from the next. For efficiency, column vectors are printed out horizontally just like rows of a matrix. Note that the field width and justification may be changed by issuing the /SET command on the keywords COLWIDTH and JUSTIFY. This may be done in command mode prior to running the program or done during running via the CMD built-in function. See Section V.I on Special Input/Output Controls.

Note: It is not possible to redimension matrices on output via MAT PRINT.

Examples:

```
10 * ADDING TWO MATRICES
20 DIMENSION A(3,3),B(3,3),C(4,4)
30 MAT INPUT B,C(3,3)
40 MAT A=B+C
45 * PACKED PRINTING
50 MAT PRINT B;C;A;
60 * NOW RIGHT-ADJUSTED UNPACKED PRINTING
70 I=CMD("/SET JUSTIFY=RIGHT")
80 MAT PRINT B,C,A
```

MAT + (Addition)
MAT - (Subtraction)
MAT * (Multiplication)
MAT / (Simultaneous Linear Equations)
MAT * (Scalar Multiply)
MAT - (Unary Negation)
MAT = (Assignment)

Purpose: To perform matrix assignment, addition, subtraction, multiplication, scalar multiplication, unary negation, or to solve a system of simultaneous linear equations in double precision.

Prototypes:

```
MAT LET {m1} = {m2} + {m3}      (1)
MAT LET {m1} = {m2} - {m3}      (2)
MAT LET {m1} = {m2} * {m3}      (3)
MAT LET {m1} = {m2} / {m3}      (4)
MAT LET {m1} = ({exp}) * {m2}   (5)
MAT LET {m1} = -{m2}            (6)
MAT LET {m1} = {m2}             (7)
```

where:

_m1_,_m2_,_m3_ are matrices.

exp is an arithmetic expression.

Abbreviation: The LET is optional.

Effect: For _all_ operations, only the elements with _positive_ indices are manipulated (i.e., the zeroeth rows and columns are ignored). The exceptions are row and column vectors which have 0 row dimension and 0 column dimension, respectively (e.g., 0 by 2 or 2 by 0). Moreover, the dimensions of the matrices for the operations must be conformable (appropriate for the operations). The same matrix can be used for m1, m2, or m3 unless it is explicitly disallowed by the operation. For convention, a matrix of m rows and n columns is a m by n matrix.

(1) and (2) All matrices must have the same number of rows (m) and the same number of columns (n). Matrix m3 is _added_ to (or _subtracted_ from) m2 component-wise and the result is placed in the corresponding component cells of m1. For example, A(1,1)=B(1,1)+C(1,1), etc.

(3) Let m2 be of dimension m by n. Then m3 must have n rows; say it is n by p. The resulting matrix m1 must then be of dimension m by p. Moreover, m1 must not be the same matrix as m2 or m3. Under these conditions the usual matrix multiplication of m2 by m3 is performed with the result being placed in m1. Vectors are not allowed for this operation, but, the matrices could be m by 1 or 1 by n.

(4) Performing C=B/A is equivalent to solving the system A C = B for C where A is square and non-singular. C and B are different n by p matrices where p can be 1 or greater. However, the unknown matrix C, typically, is n by 1 and so is B. Vectors are not allowed for this operation, but, the matrices could be m by 1 or 1 by n. The numerical technique used is double-precision Gaussian elimination with partial pivoting.

For example, to solve:

$$\begin{aligned} 3x + 2y &= 5 \\ 2x + y &= 4 \end{aligned}$$

one has the matrix equation:

$$\begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \end{bmatrix}$$

or A C = B.

(5) The matrices m1 and m2 must have the same dimensions (both m by n). The expression exp is evaluated and used to multiply each element of m2 to produce the corresponding element in m1. For example, if exp is 5, A(1,1)=5*B(1,1), etc.

(6) This has the same requirements as (5). Each element in m2 is stored into the corresponding cell of m1 with its sign changed. This is equivalent to (5) with exp of -1.

(7) The matrices m1 and m2 must have the same dimensions (both m by n). Each element of m2 is moved to the corresponding cell in m1, e.g,. A(1,1)=B(1,1), etc.

Examples:

```
10 DIM A(3,2),B(3,2),C(3,2)
20 DIM F(2,3),D(3,3),X(3,1),Z(3,1),W(3,1)
30 MAT INPUT A,B,F,Z
40 MAT C=A+B
50 MAT D=C*F
60 MAT X=Z/D
70 MAT W=(10)*X
80 MAT PRINT X;
```

<u>RDM (Just Redimensioning)</u>
<u>CON (Matrix of Ones)</u>
<u>IDN (Identity Matrix)</u>
<u>ZER (Matrix of Zeros)</u>
<u>Redimensioning (Procedure, Matrix-Vector)</u>
<u>Matrix (of Ones or Zeros)</u>
<u>Matrix (Identity)</u>
<u>Identity Matrix</u>

Purpose: To redimension a matrix <u>or</u> to fill a matrix with all ones, all zeros, or the identity matrix and <u>optionally</u> redimension the matrix.

Prototypes:

```
MAT LET {m1} = RDM({r},{c})
MAT LET {m1} = CON({r},{c})
MAT LET {m1} = IDN({r},{c})
MAT LET {m1} = ZER({r},{c})
```

where:

<u>m1</u> is a matrix to be filled. Only the elements of positive indices are filled.

<u>r</u> is an <u>arithmetic expression</u> whose value represents the new row dimension of the matrix, e.g., M*10+4, 3.6, 5, etc.

<u>c</u> is an <u>arithmetic expression</u> whose value represents the new column dimension of the matrix.

Abbreviation: The LET is optional.

Effect: If r and c are given for CON, IDN, or ZER, then they are used to redimension the matrix m1. For RDM, they <u>must</u> be given. The values of r and c are converted to whole numbers r′ and c′ which are then tested for legality. Assuming m1 to be an m by n matrix, it has (m+1)*(n+1) cells (add 1 to m and n for zero row and column). This is a fixed number according to the original dimensioning (via the DIM statement or using the default), so (r′+1)*(c′+1) must not exceed (m+1)*(n+1). For example, one can redimension a 3 by 3 matrix (of 16 cells) to a 2 by 4 (of 15 cells plus one being unused). After redimensioning, the matrix operations (including subscription) will be confined to the new index ranges of 1 to r' and 1 to c'. However, the restriction on new dimensions (for another redimension operation) is based on the original dimensions of the given matrix.

If r and c are omitted (e.g., {m1}=CON) for CON, IDN, or ZER, then the current dimensions are used.

RDM just redimensions m1.

CON fills the matrix m1 with ones.

ZER fills the matrix m1 with zeros.

IDN places zeros in m1 except in the diagonal elements (having equal row and column index, i.e., A(1,1), A(2,2), etc.) where it places ones. The matrix m1 must be square in this case.

Note: Redimensioning, for example, a 4 by 4 matrix to a 3 by 3 does not give one the 3 by 3 submatrix of the original matrix.

Examples:

```
100 MAT A=CON(6,3)
200 MAT B=CON
301 MAT C=IDN(3*I,3*I)
402 MAT D=ZER(ABS(P*Q-R),7)
500 MAT E=RDM(5,3)
```

TRN (Matrix Transpose)
INV (Matrix Inverse)
Matrix Transpose
Transpose (Matrix)
Matrix Inverse (see also MAT /)
Inverse (Matrix - see also MAT /)

Purpose: To find the inverse or transpose of a matrix.

Prototypes:

```
MAT LET {m1} = TRN({m2})      (1)
MAT LET {m1} = INV({m2})      (2)
```

where:

m1,*m2* are *square* matrices.

Abbreviation: The LET is optional.

Effect:

(1) The transpose of the matrix m2 is stored into m1.

(2) The inverse of the matrix m2 is stored in m1.

Note: For either operation only the elements of positive index are manipulated (i.e., the zeroeth rows and columns are ignored). Moreover, *m1 must not be the same matrix as m2*. For INV, if m2 has no inverse, the user will be notified.

Examples:

```
10 DIM A(3,3),B(3,3),C(3,3)
20 MAT INPUT A
30 MAT B=TRN(A)
40 MAT C=INV(A)
50 MAT PRINT B,C
```

V.E STRING OPERATIONS

String Facility

A BASIC string is a sequence of 0 to 127 characters. There are _string constants_, which are delimited by quotes (for example, "STRING CONST"), where a quote within a constant is represented by two adjacent quotes (e.g., "THIS "" IS A QUOTE"). The string of length _zero_ is called the _null string_ and is represented by "" (a constant with no characters). _String variables_ are defined to be one-dimensional arrays of default dimension 10 (hence allowing 11 string cells indexed from 0 to 10) although explicit dimension information may be given (see the DIMENSION statement description in the matrix and vector operations section V.D). A sequence of characters within a string is called a _substring_. If the substring starts at the beginning of the string, it is an _initial substring_. If it terminates at the end of a string, it is a _terminal substring_.

BASIC provides operations for string I/O (file or terminal), string assignment, left and right substring extraction, substring scanning and replacement, string concatenation, length determination, string to number and number to string conversion, numeric string testing, hexadecimal string conversions, and certain built-in functions. Strings may also be compared. Specifications follow.

LET (String Assignment)
String Assignment (LET)

Purpose: To transfer the contents of one string to another.

Prototype: LET {s1}={s2}

where:

_s1_ is a string _variable_ (possibly subscripted) into which the string s2 is to be stored. It _cannot_ be a constant. CC and BB(3) are examples of legal string variables.

_s2_ is a string constant or a string variable (possibly subscripted). For example, "STRING CONSTANT", AA, BB1(I).

Abbreviation: The LET is optional.

Effect: The content of s2 is stored into s1. The content of s2 is unaltered.

Note: Since a string variable is a vector of string cells (e.g., AA(0), AA(1), ··· , AA(10)), one would be tempted to write AA=BB to transfer all elements of BB to AA (assuming AA has as many cells as BB). This is _wrong_, since AA=BB means AA(0)=BB(0). To effect the desired transfer of an entire string array, a FOR-NEXT loop is needed.

Note also that _multiple assignments_ are _not_ allowed (e.g., AA=BB=CC="ST" is _illegal_).

Examples:

```
100 LET AA(3)=BB(2)
200 AA="HELLO THERE"
300 CC3(I)=DD3
```

LINPUT (Non-Quoted, One-String-per-Line Input)
Input (Terminal - Non-Quoted String)
Terminal Input (Non-Quoted String)

Purpose: To input from the terminal, non-quoted strings, one per line.

Prototype: LINPUT {ilist}

where:

ilist is a list of *string variables* sub-scripted or not) and/or SKP references separated by commas. For example, LINPUT AA,SKP(2),BB(4). The SKP argument may be 0 to 1000, inclusive.

Abbreviations: L'T

Effect: The effect is the same as that of the INPUT statement but the operation is specific to non-quoted string data input and the prompting character is a $-sign rather than a question mark. The strings are read in "as is"; hence, embedded quotes in the input need not be doubled as in the case of string input via the INPUT statement. For each variable specified in ilist, BASIC prompts the user to type in a data *line* to be stored in that variable. The list is processed from left-to-right just as for the INPUT statement. All characters typed on one line are read. There are no restrictions on the line content but the length may not be greater than *127* or *truncation* will occur. The SKP function may be used to skip input lines. For example, LINPUT AA,SKP(2),BB(4) will read the first line into AA, skip the next two lines, and read the fourth line into BB(4). Issuing an end-of-file causes program termination just as with INPUT.

Examples:

```
10 LINPUT AA,BB(3)
20 L'T SKP(1),BB(2)
```

See the concordance program in Appendix O for an in-context example.

Concatenation (String +)
String Concatenation (+)

Purpose: To adjoin or concatenate two strings together to form one composite string.

Prototype: LET {s1}={s2}+{s3}

where:

_s2_,_s3_ are either string constants or string variables (possibly subscripted), e.g., "HELLO", AA, CC3(5).

_s1_ is a string _variable_ into which the concatenated result is to be stored.

Abbreviation: The LET is optional.

Effect: The content of s3 is adjoined to the right of the content of s2 and the result is stored in s1. The contents of s2 and s3 are unaltered. For example, if AA contains "HELLO" and BB(2) contains " THERE", then after XX(2)=AA+BB(2), XX(2) will contain "HELLO THERE". If the result is longer than 127 characters, it is truncated to 127, i.e., the initial substring of length 127.

Note: The statement form is limited to two string operands for concatenation; hence, AA(1)=BB(3)+CC+DD is _illegal_.

Examples:

```
100 LET AA(1)=BB(3)+CC
201 AA = BB(2) + "THERE"
300 XX="Lau" + "ghs"
```

Extraction (String *)
String Extraction (*)

Purpose: To extract a given number of characters from the left or right end of a string.

Prototypes:

```
LET {s1} = ({exp}) * {s2}     (left extract)
LET {s1} = {s2} * ({exp})     (right extract)
```

where:

s2 is the string variable or constant from which the characters are to be extracted.

exp is an arithmetic expression whose value is the number of characters to be extracted from s2. The absolute value of exp is truncated to an integer before it is used.

s1 is the string variable (possibly subscripted) into which the extracted characters are to be stored.

Abbreviation: The LET is optional. If exp contains no multiplication or addition operations, the parentheses may be omitted.

Effect: The expression is evaluated and converted to a non-negative integer and used as a character count (call it c). If c is 0, then the null string (string of length 0) is stored into s1. If c is equal to or greater than the length of s2, all of s2 is stored into s1. Otherwise, c characters are extracted from either the left or right end of s2 (depending on the statement form) and stored into s1. In all cases, s2 is not altered (unless s1 and s2 are identical). For example, AA=4*"JELLO" places "JELL" into AA. If X=0.5 and BB="BASIC" then CC=BB*(6*X) or CC=BB*(2.5+X) places "SIC" into CC. Note that the * in the expression 6*X has different meaning than the * after the BB.

Examples:

```
100 AA(2) = 3 * BB
372 BB1= CC*(COS(X)+SQR(Y))
```

```
10 CC = N*"ABCDEF"
9999 DD="0123456789"*(A+B)
```

NTS (Number To String Conversion)
Number to String Conversion

Purpose: To produce a string form of a numeric value.

Prototype: LET {s1} = NTS({exp})

where:

- exp is an arithmetic expression.
- s1 is a string variable into which the result of the conversion is to be placed.

Abbreviation: The LET is optional.

Effect: The expression exp is evaluated and the numeric value is converted into character form and stored into s1.

The character form is as follows. A short form with at most seven digits is attempted; i.e., "SX.XXXXXX" to "SXXXXXXX.", where S is a sign. If none of these forms is possible, then the form "SX.XXXXXXESXX" is used where ESXX indicates 10 to the power SXX, S being the sign of the exponent XX. In either representation, extraneous leading and trailing zeros as well as a trailing period are removed. The signs are produced only if they are minus.

Examples:

```
100 AA(1) = NTS(A*SQR(X))
20 BB3(4) = NTS(I)
```

<u>STN (String to Number Conversion)</u>
<u>ISN (Is the String Numeric)</u>
<u>String to Number Conversion</u>
<u>Numeric String Test</u>
<u>String (Test for Purely Numeric)</u>

Purpose: To convert the external string form of a number to its numeric value. To find out if a string is a number.

Prototypes:

```
{r}=STN({s})
{r}=ISN({s})
(STN and ISN are built-in functions)
```

where:

- <u>s</u> is a string variable or constant to be converted or tested.
- <u>r</u> represents the <u>numeric</u> value of either STN or ISN.

Effect: ISN tests the string s for being numeric (e.g., "-3.0", "1.5E9"). It returns an r of 0 for YES and a 1 for NO.

STN converts the string s, if it is numeric, into a number and returns it as the result r. If s is non-numeric, a 0 is returned. Note that ISN must be used to differentiate between a string whose numeric value is 0 and a non-numeric string.

Note: STN and ISN are built-in functions which may be invoked wherever an arithmetic expression is allowed.

Examples:

```
100 S1 = ISN(AA(3))
200 X = STN(AA(3))
350 IF ISN(CC)<>0 THEN 710
400 IF ISN(DD1(3))=0 THEN:  X=STN(DD1(3))
```

HEX (Hex Equivalent of a String)
XTS (Hex to String Conversion)
Hex to String Conversion
String to Hex Conversion

Purpose: To convert a character string consisting of hex digits to a string of characters that the digit pairs represent. Also, to obtain, as a character string, the hex digits equivalent to a given character string.

Prototype:

```
LET {s1} = HEX({s2})
LET {s3} = XTS({s4})
```

(HEX and XTS are built-in functions.)

where:

s2	is a string variable or constant of even length whose characters represent hex digits; e.g., "C1C2F1" represents the characters "AB1".
s4	is a normal BASIC character string (variable or constant).
s1	is a string *variable* into which the result of HEX is to be stored.
s3	is a string *variable* into which the result of XTS is to be stored.

Abbreviation: The LET is optional.

Effect: When HEX is invoked, the string s2 is checked for *pairs* of *hex* digits (e.g., "004040C1C4"). If s2 is null, s1 is set to the null string; otherwise, the hex pairs are converted to their corresponding characters and the resultant string stored in s1. In any case, s2 is unaltered; e.g., "C24BF3" results in "B.3".

When XTS is invoked, the length of s4 is checked. If it is longer than 63 characters, only the first 63 are used in the conversion. If s4 is null (has a length of 0), the null string is stored in s3; otherwise, the characters in s4 are used to form a string whose characters represent the hex equivalent of s4 and the result is stored in s3. In any case, s4 is unaltered. For example, "AZ8 +." results in "C1E9F8404E4B".

Note: HEX and XTS are string-valued functions which are restricted to the above statement forms.

Examples:

```
100 XX=HEX("C8C9")
200 YY(3)=XTS("HELLO")
```

CLS (String Time of Day)
DAT (String Date)
UID (String User ID)
Time of Day (String)
Date (String)
User ID (String)

Purpose: To obtain the time of day, date, or user id as a string.

Prototypes:

```
LET {s} = CLS()
LET {s} = DAT()
LET {s} = UID()
```

(CLS, DAT, and UID are string-valued built-in functions.)

where:

s is the string *variable* into which the string produced by one of the functions is to be stored.

Abbreviation: The LET is optional.

Effect: CLS returns as its value the time of day in the form "HH:MM.SS" where HH is hours, MM is minutes, and SS is seconds of the time relative to midnight; e.g., "13:31.55".

DAT returns as its value the date as "MM-DD-YY" where MM is the month, DD is the day, and YY is the last two digits of the year; e.g., "09-27-38".

UID returns as its value the user identification (MTS signon id).

Note: These functions being string-valued are restricted to the above statement forms.

Examples:

```
100 LET TT(1)=CLS()
110 DD=DAT()
120 UU=UID()
130 PRINT "HELLO TO ";UU;" ON ";DD
140 PRINT "AT THE TONE, THE TIME WILL BE: ";TT(1)
```

BFR (Scan String Definition)
RPB (Replacement String Definition)
String Scan with Replacement (Definitions)

Purpose: To define the scan string for use with the EDT or SCN built-in functions or to define the replacement string for the EDT function.

Prototype:

```
{r}=BFR({s})
{r}=RPB({s})
```

(BFR and RPB are built-in functions.)

where:

- _s_ is the string (constant or variable) to be defined as either the "scan string" or the "replacement string".
- _r_ represents the return value of either BFR or RPB.

Effect: The string s is defined to be the "scan string" if BFR is called, and the "replacement string" if RPB is called. In either case, the function value returned is 0. This allows a call on either to be _add_ed to an arithmetic expression without altering the expression's value. For example,

```
Y=3*SQR(X+BFR("AB")) or
GOTO(10,30,70),I+RPB("CD").
```

Note: BFR and RPB are numeric-valued built-in functions which may be used anywhere an arithmetic expression is allowed. See the writeups on SCN and EDT for in-context examples.

Examples:

```
10 I=BFR(AA)
20 I=RPB("HELLO")
30 *
40 C=BFR("AB")+SCN(AA)
50 *
60 SS=EDT(AA(3+BFR(BB)+RPB(CC)))
```

SCN (Scan for Substring Position)
String Scan (Substring Position Determination)

Purpose: To locate the beginning of a particular substring within a string.

Prototype: {r}=SCN({s1})
(SCN is a built-in function)

where:

_s1_ is the string scanned for the "scan string" (which is defined via the BFR built-in function).

_r_ is the numeric value returned as the number of the character in s1 which begins the "scan string".

Effect: Invoking the function BFR on s2 defines s2 as the scan string (the value of BFR is always 0). The SCN function searches the argument string s1 from left to right for an occurrence of s2. If none can be found, r is 0; otherwise, r is the number (numbering from the left) of the left-most character in s1 which begins s2. If s2 is null, r is also 0. For example, if AA="HOW NOW BROWN COW" then I=BFR("OW") followed by C=SCN(AA) would set C to 2.

Note: SCN is a normal built-in function and may be invoked anywhere an arithmetic expression is allowed.

Examples:

```
100 C=SCN("ABADABADOO")
20 GOTO(30,40,50),SCN(QQ)
50 *
60 * COMPLEX EXAMPLE
70 AA="HOW BASIC"
80 I=BFR("BAS")
90 C=SCN(AA)
100 * ALTERNATELY
110 C=BFR("BAS")+SCN(AA)
120 * IN ANY CASE, C IS 5.
```

<u>EDT (String Search and Replacement)</u>
<u>String Scan with Replacement (Usage)</u>

Purpose: To search a string for a given substring and produce a new string with the given substring replaced by a "replacement" string.

Prototype: LET {s1}=EDT({s2})

where:

<u>s2</u> is the string variable or constant to be searched for the "scan string" (defined via the BFR built-in function).

<u>s1</u> is the string <u>variable</u> into which the edited result is to be stored.

Abbreviation: The LET is optional.

Effect: The string s2 is searched (left to right) for the first occurrence of the "scan string". If the search fails or if the "scan string" is null, the content of s2 is stored into s1. Upon success, a new string consisting of s2 with the left-most occurrence of the "scan string" being replaced by the "replacement string" (defined via the RPB built-in function) is stored into s1. In all cases s2 is not altered. For example, if the statements I=BFR("ART") and J=RPB("URKS") are executed, then AA=EDT("GET SMART") will place "GET SMURKS" into AA. If the result is longer than 127 characters, it is truncated on the right to the initial substring of length 127.

Note: The "replacement string" may be null if one wishes to delete substrings. Since EDT is string-valued, it is restricted to the above statement form.

Examples:

```
100 AA="BONJOUR MADAME"
110 I=BFR("JOUR")
120 J=RPB("SOIR")
130 BB=EDT(AA)    /* "BONSOIR MADAME".
140 *
150 * NOW ALL IN ONE STATEMENT
```

```
160 BB=EDT(AA(BFR("JOUR")+RPB("SOIR")))
199 * ANOTHER GEM
200 XX(BFR("O")+RPB("A"))="HO HO HO"
210 IF SCN(XX)=0 THEN 240
220 XX=EDT(XX)
230 GOTO 210
240 PRINT XX;" IS A LAUGH."
```

IF (String)
Comparisons (String)
String Comparisons

Purpose: To compare strings and conditionally either transfer control to another part of a BASIC program or execute another BASIC statement.

Protoypes:

```
IF {s1}{reln}{s2}, THEN {ln}
IF {s1}{reln}{s2}, THEN ({lnlist}), {exp}
IF {s1}{reln}{s2}, THEN:  {Bstmnt}
```

where:

ln	is a BASIC line number just as for a simple GOTO.
lnlist	is an ordered list of line numbers of executable BASIC statements just as for a computed GOTO; e.g., 10, 35, 5.
exp	is an algebraic expression whose value is used to select a line number from lnlist just as for a computed GOTO; e.g., 3*A+B.
Bstmnt	is an executable BASIC statement (without a line number preceding it); e.g., PRINT "HELLO" OR Y=A*B+SQR(Z).
s1	is a single BASIC string.
s2	is a single BASIC string.
reln	is a relational operator to be used to compare the strings s1 and s2. The following operators are allowed:

=	is equal to
>	is greater than
<	is less than
<>	is not equal to
<=	is less than or equal to
>=	is greater than or equal to

As alternates for the last three operators one may write:

¬=	is not equal to
¬>	is not greater than
¬<	is not less than

<u>No</u> other relational operators are allowed.

Abbreviations: The comma before the THEN is optional.

Effect: The strings s1 and s2 are compared for satisfying the relation specified by reln (for example, AA(3)="GOOD"). The ordering of characters for comparisons is given in Appendix A. For strings of <u>equal length</u>, this ordering is used exclusively (e.g., "AC" is greater than "AB" and "B3" is greater than "BZ"). If one is shorter (say, of length n) than the other, then it is compared with the first n characters of the longer just as for strings of equal length. If no difference is found (i.e., the shorter is an <u>initial substring</u> of the longer), then the longest is the greatest. For example, "AC" is greater then "ABC" while "ABC" is greater than "AB".

If the relationship reln does not hold, then control passes to the next executable statement after the IF. If the relationship reln holds (e.g., AA <u>is</u> greater than BB), then control is given to the part specified after the THEN. In the case of the first two prototypes, the control is handled like the simple and computed GOTOs, respectively. For the third prototype (note the colon after the THEN) control is given to the <u>executable</u> BASIC statement Bstmnt. If Bstmnt is not a GOTO or RETURN or a statement which terminates the program, then after it has been <u>completed</u>, control is passed on to the statement after the IF. In the case of Bstmnt being either a CALL or a GOSUB, it is completed when a return has been made to it.

Examples:

```
10 IF "AB"¬=XX THEN 100
20 IF AA(3)>BB(I) THEN (30,40), Q*3
30 IF AA3(J)<=QQ5(2), THEN:  AA3(J)="VOID"
40 IF PP(1)>PP(2) THEN:  IF PP(2)>PP(3) THEN 300
50 *
60 * SEARCH EXAMPLE (LOOKING FOR JOHN)
70 FOR I=1 TO N
```

```
80 IF QQ(I)="JOHN" THEN 100
90 NEXT I
99 PRINT "JOHN IS NUMBER";I;" IN LINE."
```

V.F FILE OPERATIONS

File Operation Facility

A BASIC file is an ordered set of lines with integer numbers in the range 0 to 99999. Each line in a data file is a sequence of data items separated by either blanks or commas. The BASIC file operations are on files of type D, i.e., data files. Moreover, a data file is treated as a continuous stream of data as if the lines were adjoined in order. Because of its line nature, it may however, be generated via line-editing facilities of the command language or via special line-oriented DATA statements within a BASIC program. The file operation statements in a BASIC program will still treat the data as a stream. The data items whether entered by DATA statements or via the command language must be separated by blanks or commas with strings containing blanks or commas being quoted. Quotes within quoted strings must be doubled. All these details are automatically observed by the file write statements when they are used to generate a file.

There are statements to read and write all types of data, including entire matrices, into data files. Moreover, there are operations for rewinding, backspacing, and end-of-file testing.

The Program's Data File
Data File (Program's)

A program (say PROG1) may preset data in its data file (PROG1@D) by DATA statements which generate a file whose corresponding lines contain the data defined in the statements. The data may be read via the READ statement when the program is running. In fact, this mechanism may be used to initialize variables in a program. The statements which apply to the data file are: DATA, READ, WRITE, BACKSPACE, RESTORE, MAT READ, MAT WRITE, and IF ENDFILE.

General Data Files
Data Files (General)

In addition to referring to its own data file, a program may refer to others by placing their names on a file list via the FILE statement (e.g.,FILE F1,FILJOE,FTHREE). There are statements corresponding to those used to refer to the program's data file. These simply contain a reference to the position of the file in question on the program's file list (e.g., READ FILE 2,A,B reads two numbers from the data file FILJOE). The statements which apply are: DATA, FILE, READ FILE, WRITE FILE, BACKSPACE FILE, RESTORE FILE, MAT READ FILE, MAT WRITE FILE, and IF ENDFILE. The file of index zero is, by definition, the program's data file, so READ is equivalent to READ FILE 0.

File Manipulation (Extended)

The user may copy, line-edit, empty, destroy, etc., BASIC files (S, D, or O whenever legal) via the CMD function. Note that although the BASIC file input/output is sequential in nature, by using the line-editing facility, one can perform indexed write operations (i.e., open a file and issue lines with line-number prefixes). See the CMD writeup for details and examples (Section V.H).

Statement specifications follow.

DATA (Data File Initialization)
Data Initialization
Initialization of Data
Presetting of Data

Purpose: To store data into the program’s data file. To define data for the presetting of program variables.

Prototype: DATA {dlist}

where:

<u>dlist</u> is a list of <u>constant</u> data items to be stored in the program’s data file. The items which may be strings or numbers are separated by commas or blanks. Strings which contain commas or blanks should be enclosed in quotes ("); e.g., DATA 1,2,AB,"AB, CD"

Abbreviation: None

Effect: The data list of the DATA statement is written by BASIC into the program’s data file as the line having the same line number as the DATA statement itself. For example, 30 DATA 1,2,3 places 1,2,3 as line 30 in the program’s data file.

Note: This statement in conjunction with the READ and/or MAT READ statements is useful for presetting variables in a program. See the examples below.

Note also that normally each time a BASIC program is RUN (or DEBUGged), its data file is emptied before the data lines are inserted. However, if the command /SET DFILEMP=OFF is issued, this will not be the case. Therefore, it is possible to generate part of the data file via the BASIC command language (/OPEN, /NUMBER, etc.) and generate the rest via the DATA statements in the program (some of which could even replace existing lines in the file).

It is possible to read the data in a data file more than once. If one wishes to reuse (reread) the data in the file, the RESTORE statement can be issued beforehand.

Examples:

```
1 * SAMPLE PRESETTING OF VARIABLES
5 * FIVE NUMBERS
10 DATA 1,2,3,4,5
15 * TWO STRINGS
20 DATA HELLO,"HELLO THERE"
30 DIM A(5),BB(2)
40 MAT READ A
50 READ BB(1),BB(2)
```

READ (Data File Input)
WRITE (Data File Output)
File Input (Data File)
Data File Input
Input (Data File)
File Output (Data File)
Data File Output
Output (Data File)

Purpose: To read data from or to write data into the program's data file.

Prototypes:

```
READ {ilist}
WRITE {olist}
```

where:

ilist is a list of numeric or string *variables* (subscripted or not) and/or SKP references separated by commas just as for the INPUT statement. For example, READ A,B or READ A,SKP(2),B or READ A,C(3),D(3,J),AA,BB1(3).

olist is a list of numeric or string variables (subscripted or not), numeric or string constants, or complex arithmetic expressions separated by commas. Since all file output is packed, for efficiency, no special output controls are allowed.

Abbreviations: R'D for READ
W'E for WRITE

Effect: *READ* reads data from the program's data file in essentially the same way that INPUT reads from a terminal. The data file may have been generated by DATA statements in the program, or by line-editing facilities of the command language (/OPEN, /NUMBER, etc.), or by a program writing into the file. In any case, the data file is treated as a continuous stream of data even though the data is stored in the file as a sequence of lines. For example, if the first line of the data file contains three numbers, then READ A will read the first one into A and READ B,C will read the next two into B and C. This differs from INPUT in that each time a READ is issued, a new line is _not_ necessarily demanded. If there is no more data to be read, an end-of-file condition (which may be tested afterward via the IF ENDFILE statement) is recorded by BASIC. Another READ issued without a preceding RESTORE or BACKSPACE is considered an error, in this case.

WRITE writes data into the program's data file as a stream of data. For example, if the data file is empty, then WRITE A,B followed by WRITE C,D,E generates a data file consisting of the 5 data items in order. The second WRITE started where the first one left off. _All_ strings written into the data file are enclosed in quotation marks (") so as to resolve any possible ambiguities in READing them later. This includes internal quotes being doubled.

Note: If one is positioned into a file (by means of a READ or SKP) and one then _WRITE_s into that file, _any data in the file beyond that just written will be removed by BASIC_. This is similar to writing a stream of data on a magnetic tape. The logical end-of-tape is defined to be the end of whatever was just written.

Examples: Presetting variables in the program PGM.

```
10 DIMENSION C(3),D(2,2)
20 DATA 1,2,3,4
30 READ A
40 READ B,C(1),D(1,2)
```

Now rewrite the file with new data (after emptying it).

```
70 I=CMD("/EMPTY@NC@T PGM@D")
80 WRITE "HELLO",AA(1),3,6*SQR(Z),W
```

The file now has two strings and three numbers in it.

MAT READ (Data File Matrix Input)
MAT WRITE (Data File Matrix Output)
File Input (Data File - Matrix)
Data File Input (Matrix)
Input (Data File - Matrix)
File Output (Data File - Matrix)
Data File Output (Matrix)
Output (Data File - Matrix)

Purpose: To read entire matrices from or write them into the program's data file.

Prototypes: MAT READ {milist}
MAT WRITE {molist}

where:

milist is a list of matrix names separated by commas just as for MAT INPUT. Optional redimensioning is also allowed with this statement as well; e.g., MAT READ A,B(3,2),C.

molist is a list of matrix names separated by commas.

Abbreviations: M'D for MAT READ
M'E for MAT WRITE

Effect: MAT READ reads data from the program's data file row-wise into matrices just as MAT INPUT does at a terminal. An end-of-file condition (which may be tested via the IF ENDFILE statement) is generated when no more data is encountered while reading. Another MAT READ (or READ) issued without a prior RESTORE or BACKSPACE is considered an error.

MAT WRITE writes data row-wise from matrices into the program's data file just like MAT PRINT, except that all file output is packed as with the normal WRITE statement.

Examples:

```
10 * PROGRAM PROG1 ADDS MATRICES
20 DIMENSION A(2,2),B(2,2),C(3,3)
30 DATA 1,2,2,1,3,2,2,3
40 MAT READ B,C(2,2)
```

```
50 MAT A=B+C
60 I=CMD("/EMPTY@NC@T PROG1@D")
70 MAT WRITE A
```

If statement 60 was omitted, matrix A would be written into PROG1's data file just after matrix C.

FILE (File List Definition)

Purpose: To define a file list for the program to use with the READFILE, WRITEFILE, BACKSPACEFILE, and RESTOREFILE file statements.

Prototype: FILE {flist}

where:

flist is a list of BASIC file names separated by commas. A BASIC file name consists of one to seven letters or decimal digits with the first character being a letter, e.g., PROG1, F2, etc. The files listed here are data files (of type D) not program files (of type S, containing the source program lines) or object files (of type O, containing the object or translation of a source file).

Abbreviation: F’E

Effect: All the FILE statements in the program are collected and placed in ascending order of line number. The file lists are processed from left to right from the lowest numbered statement to the highest. The file names encountered are assigned integer indices 1,2,3 etc., with the index 0 being reserved for the program’s data file. The result is a file list for reference by the various BASIC file manipulation statements.

Note: A file name may occur more than once in the file list; moreover, even the program’s data file name may be present. For example, FILE A1,B,A1,PGM,F2 is an allowable statement in the program PGM.

Examples:

```
10 FILE F1,F2   /* FIRST, SECOND
20 FILE F3,F1   /* THIRD, FOURTH FILES.
30 READ FILE 1,A,B
40 BACKSPACE FILE 1
50 READ FILE 1,C /* REREAD THE SECOND ITEM.
```

```
60 RESTORE FILE 4   /* REWIND F1
70 READ FILE 4,X,Y  /* FIRST TWO ITEMS.
```

READ FILE (General File Input)
WRITE FILE (General File Output)
File Input (General)
Input (General File)
File Output (General)
Output (General File)

Purpose:	To read data from or to write data into a BASIC data file.

Prototypes:

```
READ FILE {exp},{ilist}
WRITE FILE {exp},{olist}
```

where:

exp	is an arithmetic expression whose value, truncated to an integer, is used as an index into the program's file list to select the file to be read or written. If exp is 0, the program's data file is used.
ilist	is a list of numeric or string _variables_ (subscripted or not) and/or SKP references separated by commas just as for an INPUT statement; e.g., READFILE A,B or READFILE A,SKP(2),B or READFILE A,C(3),D(3,J),AA,BB1(3).
olist	is a list of numeric or string variables (subscripted or not), numeric or string constants, or complex arithmetic expressions separated by commas. Since all file output is packed, for efficiency, no special output controls are allowed.

Abbreviations: READ# or R'D# for READ FILE
WRITE# or W'E# for WRITE FILE

Effect: READ FILE and WRITE FILE perform in the same manner as READ and WRITE except that the file referred to is either a file in the program's file list (defined via the FILE statement) or the program's data file (when exp is 0). For example, if FILE FONE,FTWO is the first FILE statement in the program PROG1, then READ FILE 1,A refers to the data file FONE while READ FILE 0,A reads the program's data file (PROG1@D).

Examples:

```
5  DIM B(2)
10 FILE ALPHA,BETA
20 FILE GAM,SAM,JOE
30 READ FILE 3,A,B(1)
40 READ FILE 3*A+B(1),P,Q
50 WRITE FILE 1,P,Q
55 * EMPTY THE THIRD FILE
60 I=CMD("/EMPTY@NC GAM@D")
70 WRITE FILE 3,"HELLO",P,Q
```

MAT READ FILE (General Matrix File Input)
MAT WRITE FILE (General Matrix File Output)
File Input (General - Matrix)
Input (General File - Matrix)
File Output (General - Matrix)
Output (General File - Matrix)

Purpose: To read entire matrices from or write them into a BASIC data file.

Prototypes:

```
MAT READ FILE {exp},{milist}
MAT WRITE FILE {exp},{molist}
```

where:

exp	is an arithmetic expression whose value truncated to an integer is used as an index into the program's file list to select the file to be read or written.
milist	is a list of matrix names separated by commas just as for MAT INPUT. Optional redimensioning is also allowed with this statement as well; e.g., MAT READFILE A,B(3,2),C.
molist	is a list of matrix names separated by commas.

Abbreviations: MATREAD# or M'D# for MAT READ FILE
MATWRITE# or M'E# for MAT WRITE FILE

Effect: MAT READ FILE and MAT WRITE FILE perform in the same manner as MAT READ and MAT WRITE except that the file referred to is either a file in the program's file list (defined via the FILE statement) or the program's data file (when exp is 0). For example, if FILE FONE,FTWO is the first FILE statement in the program PROG1, then MAT READ FILE 1,A refers to the data file FONE while the statement MAT READ FILE 0,A reads the program's data file (PROG1@D).

Examples:

```
10 * READ A MATRIX FROM FILE DF2 AND INVERT IT
20 * AND WRITE IT INTO DF1.
```

```
30 DIM A(2,2),B(2,2)
40 FILE DF1,DF2
50 MAT READ FILE 1,A
60 MAT B=INV(A)
70 MAT WRITE FILE 2,B
```

Note that statements 50 and 70 could be M’D# 1,A and M’E# 2,B respectively.

IF (End-of-File Test)
Data File End Test
File End Test

Purpose: To test for an end-of-file condition arising from the reading of a data file and conditionally transfer control to another part of a BASIC program or execute another BASIC statement.

Prototypes:

```
IF ENDFILE [{exp1}], THEN {ln}
IF ENDFILE [{exp1}], THEN ({lnlist}), {exp2}
IF ENDFILE [{exp1}], THEN:  {Bstmnt}
```

where:

ln	is a BASIC line number just as for a simple GOTO.
lnlist	is an ordered list of line numbers of executable BASIC statements just as for a computed GOTO, e.g., 10, 35, 5.
exp1	is an algebraic expression whose value is used to select a file from the list that was defined via the FILE statement (c.f., FILE). This expression is optional. If it is omitted, a value of zero is assumed. In this case, the program's data file end-of-file condition will be tested. For example, "IF ENDFILE THEN 20" in program PROG2 will check for an end-of-file condition on PROG2@D.
exp2	is an algebraic expression whose value is used to select a line number from lnlist just as for a computed GOTO, e.g., 3*A+B.
Bstmnt	is an executable BASIC statement (without a line number preceding it), e.g., PRINT "HELLO" OR Y=A*B+SQR(Z).

Abbreviation: The comma before the THEN is optional.

Effect: The expression exp1 is evaluated and truncated to an integer value, call it i. The resultant value must be between 0 and the number of file references in the FILE definition statements in the program containing the given IF statement. If so, it is used to select the ith file from the ordered list defined by the FILE statements in that program. If the last operation on the file was a READ and it encountered the end of the file, then the result of the test is true; otherwise, it is false. If false, control is passed to the next executable statement after the IF. If true, then control is given to the part specified after the THEN. In the case of the first two prototypes, the control is handled like the simple and computed GOTOs, respectively. For the third prototype (note the colon after the THEN) control is given to the <u>executable</u> BASIC statement Bstmnt. If Bstmnt is not a GOTO or RETURN or a statement that terminates the program, then after it has been <u>completed</u>, control is passed on to the statement after the IF. In the case of Bstmnt being either a CALL or a GOSUB, it is completed when a return has been made to it.

Note: Since BASIC automatically restores all files referred to by a program being called by another (except if the /SET RESTORE=LOCAL command has been given), the implicit restore is considered a file operation. Hence, if PROG1 encounters an end-of-file while reading a file and it CALLs on PROG2 which tests the end-of-file condition, the test would fail.

Examples:

```
10 * THIS PROGRAM’S NAME IS PROG.
20 FILE PAT, CLARK, ED
30 FILE M,RCFB,JUNE
40 READ FILE 4, X,Y
50 IF ENDFILE 4, THEN 200
60 * READ FROM PROG’S DATA FILE
70 READ Z,W
80 IF ENDFILE, THEN 200
       .
       .
       .
200 PRINT "I RAN OUT OF DATA."
210 STOP
```

RESTORE (Data File Rewind)
RESTORE FILE (General File Rewind)
BACKSPACE (Data File Backspace)
BACKSPACE FILE (General File Backspace)
Data File Rewind - Backspace
File Rewind - Backspace (General)
Rewinding of Data Files

Prototypes:

```
RESTORE
BACKSPACE
RESTORE FILE {exp}
BACKSPACE FILE {exp}
```

where:

exp is an arithmetic expression whose value, truncated to an integer, is used as an index into the program's file list to select the file to be restored or back-spaced. If exp is 0, the program's data file is used.

Abbreviations:

R'E for RESTORE
RESTORE# or R'E# for RESTORE FILE
B'E for BACKSPACE
BACKSPACE# or B'E# for BACKSPACE FILE

Effect:

RESTORE FILE restores (rewinds) the file referred to so that the next read or write operation starts at the beginning of the file. This is useful for rereading the same data. Note that this operation does not empty the file.

RESTORE restores the program's data file (is equivalent to RESTORE FILE 0).

BACKSPACE FILE backspaces the file referred to by one data item so that the next read or write operation will be on the previous data item. For example, if a file contains four data items and one reads the first three, issues a backspace, and reads a number, this last number read is the third item in the file. Note that it is illegal to backspace off the beginning of a file.

BACKSPACE backspaces the program's data file (is equivalent to BACKSPACE FILE 0).

Note: Each time a BASIC program is invoked by another BASIC program (via CALL or CHAIN), by default, the data files referred to by the called program are automatically restored by BASIC just prior to their first use in the called program, for this call. This is GLOBAL restoration. If the command /SET RESTORE=LOCAL is issued (in command mode or via the CMD function), then this automatic restoration is turned off; hence, the user is responsible for issuing his own restores.

Examples: Initialize two matrices to the same matrix using RESTORE.

```
10 DIM A(2,2),B(2,2)
20 DATA 1,2,3,4
30 MAT READ A
40 RESTORE
50 MAT READ B
```

Some miscellaneous unrelated statements.

```
10 R'E# 1
20 BACKSPACE
30 B'E# 3*Q
40 BACKSPACE FILE 3
```

V.G PROCEDURES: SUBROUTINES AND FUNCTIONS

Procedures (General Considerations)

When a procedure is widely used (e.g., computing a square root or cosine), it is wise to define a standard method of invoking that procedure and place it in a library which everyone has access to. Hence, time is saved for all the users since they need not program the procedure when they require it. Moreover, if specialized computation is to be performed in many places within a program, the user may wish to define his own function to perform the computation (e.g., 3*X+2 where the value of X may vary). If more than just simple functional computation is involved, the user may wish to define a sequence of statements within the program to carry out the procedure and then invoke it when needed. A user may also wish to define entire programs as procedures and invoke them by any programs that need them. These procedures result in a reduction in programming overhead, an increase in programming convenience, and greater compactness of programs.

Functions (Introduction to Built-In)

There is a set of functions built into BASIC to perform mostly numerical computation such as sine, cosine, square root, tangent, etc. There are some string- and matrix-valued functions which are defined in their respective sections of this text. The rest are defined later in this section.

Functions (Introduction to User-Defined Internal One-Line)

By means of the DEF statement, the user may define a one-line function (e.g., DEF FNA(X)=3*X+2) which may be invoked anywhere in the same program (e.g., Y=FNA(4) resulting in 14). Any procedure so defined, has access to any data item in the program where it is defined. Hence, DEF FNQ(X)=A*X+B, where A and B are program variables is legal.

Subroutine (Introduction to User-Defined, Internal)

The user may define a set of statements called a subprogram or subroutine representing a procedure and invoke the subroutine by issuing a GOSUB to its first line. When it finishes the required work, the subroutine returns control to the mainstream in the program via a RETURN. Note that since this subroutine is internal to the program defining it, it has access to all data items in the program.

Subroutine (Introduction to User-Defined, External)

An external subroutine is an entire BASIC program (contained in a source (S) file) which is to be invoked by CALL or CHAIN statements in BASIC programs. It can call on itself. Moreover, it is callable only by BASIC programs. The name of the subroutine is the name of the file which contains it (e.g., PROG2). The subroutine returns to the program that invoked it (i.e., the calling program) via the RETURN or END statements. Since this subroutine does not have access to the data values in the calling program, the BASIC file operations (read, write, etc.) may be used to pass information between a calling program and the subroutine. With these mechanisms, fully recursive subroutines (see the factorial example for CALL) may be written. Note that it is required that the subroutine be translated into an object form prior to its use. See the CALL, CHAIN writeup for details.

Specifications of the procedure facility follow.

Functions (Predefined Numeric)
ABS (Absolute Value)
ATN (Arctangent)
CLK (Numeric Time of Day)
Time of Day (Numeric)
COS (Cosine)
CRP (Can Reply - Conversation Mode Determination)
EXP (Power of e)
INT (Integer Part)
LEN (String Length)
LOG (Natural Logarithm)
RND (Random Number)
SGN (Sign)
SIN (Sine)
SQR (Square Root)
TAN (Tangent)
TIM (Elapsed Time in BASIC)

Purpose: Numeric double-precision function evaluation.

Prototype: All the above functions are invoked via the form FUN({exp}) where FUN is the function name and the argument exp is an arithmetic expression. Two exceptions are LEN({s}) where s is a string variable or constant and RND({v}) where v is a simple numeric variable. For all functions, the return value is numeric.

Effects:

ABS computes the absolute value of exp.

ATN computes the arctangent (in radians) of exp.

CLK returns the time of day in seconds relative to midnight. The argument is ignored, so CLK(0) suffices.

COS computes the cosine of the angle exp (which must be in radians).

CRP has value zero if the user is truly conversational (at a terminal); otherwise, the value is one. The argument is not used but must be supplied due to function conventions (a zero argument will suffice).

EXP computes e (the natural logarithm base) to the power exp. The argument exp must not exceed 174.673.

INT computes the integer part of exp, e.g., 2.0 for 2.9 and -2.0 for -2.9.

LEN returns as its value, the length (possibly 0) of the string argument s.

LOG computes the natural logarithm (to the base e) of exp. The argument exp must be greater than 0. To obtain the base 10 logarithm, multiply the result by 0.434294481903251828.

RND is a pseudo-random number generator which uses the argument variable v for storage of a random number base that it uses for generation. If the variable v is zero, RND supplies a starting value for the base. If v is non-zero, it is used to produce a starting base (initially). This is useful for generating separate random number sequences. Each time RND is invoked, it produces a normalized (0.0 to 1.0 inclusive) pseudo-random number using Lehmer's method (Communications of the ACM, February 1969, pp. 85-86). Using this method with an optimum base, a cycle length of over two billion will be obtained. The user should be cautioned that even with this large discrete set of available pseudo-random numbers, his transformation of this set into another discrete set (e.g., a finite set of integers) may not produce a uniform distribution. This will be the case particularly if the set that the user is transforming the random numbers to is too large or if an insufficient sample of random numbers is used. A skewness will occur. Moreover, when using pseudo-random numbers for multi-dimensional problems, the situation becomes exponentially worse with increasing dimension. The user is referred to the literature on the subject: 1) Marsaglia, G., "Random Numbers Fall Mainly in the Planes," Proc. Nat. Acad. Sci., 60, 5, September 1968, pp. 25-28. and 2) Whittlesey, John R. B., "On the Multidimensional Uniformity of Pseudorandom Generators," Communications of the ACM, May 1969, p. 247. Examples follow.

```
10 * LET RND SUPPLY THE BASE
20 B=0
30 R1=RND(B) /* FIRST RAND.  NO.
40 R2=RND(B) /* SECOND ONE
50 *
60 * USER SUPPLIES THE BASE
70 B2=CLK(0) /* HOW RANDOM IT IS.
80 R3=RND(B2)
```

SGN returns an indicator of the algebraic sign of its argument: +1 for positive, -1 for negative, and 0 if the argument is 0.

SIN computes the sine of the angle exp (which must be in radians).

SQR computes the positive square root of its argument exp provided that it is nonnegative; otherwise, it complains.

TAN computes the tangent of the angle (expressed as radians) provided that the angle is less than pi (3.141592653...) times 2**50 (approximately 3.5E15) in magnitude. If the angle is near one of the singularities of the function, a signed number approximating infinity (1.0E75) is returned.

TIM returns as its value the elapsed time in seconds since the BASIC system was last entered from MTS (via $RUN *BASIC) by the user.

Note: The following constants may prove useful:

```
e=2.71828182845904524
pi=3.14159265358979324
log10e=0.434294481903251828
```

The following formula may also prove useful:

```
log10(X)=log10e*LOG(X)
```

Examples:

```
10 Y=ABS(3*X)
20 T(1)=ATN(A+B*Z)
30 T(I)=INT(CLK(0)+0.5)
40 C1=0.5*(EXP(X)+EXP(-X))
50 L=LEN(AA(5))
55 L1=.434294481903251828
60 L10=L1*LOG(X)  /* LOG10 VIA LOG BASE E
70 GOTO(100,200,300),2+SGN(SIN(P))
80 R=(-B+SQR(B*B-4*A*C))/(2.0*A)
90 C3=TAN(3.1515926536*Q/180.0)
100 IF TIM(0)>120 THEN: PRINT "YOUR TIME IS UP."
110 IF CRP(0)=0 THEN: P'T "YOU ARE AT A TERMINAL."
```

DEF (User-Defined Function)
Function (Specifications of User-Defined)

Purpose: To allow the user to define his own numeric function of one variable.

Prototype: DEF FN{l}({d})={exp}

where:

- l is a letter of the alphabet to be used to name the function. Hence, there are 26 possible function names (e.g., FNA, FNB, etc.).
- d is a dummy variable which represents the argument to be given to the function. This variable is unique to the function definition and is not confused by BASIC with a real variable of the same name elsewhere in the program. For example, DEF FNA(X)=3*X+2 defines a function FNA which multiplies whatever it is given by 3, adds 2 and returns the result as its value. Here, X is the dummy variable.
- exp is an arithmetic expression which is to be evaluated by the function. It may or may not involve the dummy variable d. If it does, every occurrence of d in exp is replaced by the argument value given when the function is invoked. The result of the evaluation is the function value.

Effect: The DEF statement simply defines the function. In order to effect the computation, the function must be invoked in an arithmetic expression. For example, in the above case of FNA, one could write Z=FNA(5) and 17 (the result of FNA(5)) would be stored in Z.

Note: The expression may contain references to BASIC built-in functions or even user-defined functions. In the latter case, the restriction is that the function not be directly or indirectly defined in terms of itself. So DEF FNA(X)=1+FNA(X+3) is _illegal_. So is the combination of:

```
DEF FNA(X)=1+FNB(X+3),
DEF FNB(Z)=3*FNC(Z), and
DEF FNC(Q)=SQR(6/FNA(Q+1)).
```

Examples: Define three functions and use them.

```
10 DEF FNA(X)=3*X+5
20 DEF FNB(Z)=COS(FNA(Z)+1)
30 DEF FNP(X)=INP(A)*X+INP(B)
     .
     .
     .
100 F=FNA(D+E)
110 G=FNB(FNA(2)*FNB(F))
120 PRINT A,3*B,FNP(6)
```

Note that since the output list is computed before printing is done, FNP(6), which modifies (reads into) A and B, is invoked before A is printed; hence, A contains the new value read in by the INP function reference. However, since the output list is processed from left to right, 3*B involves the old value of B because it precedes the evaluation of FNP(6). The variable A contains a new value only because as a variable it does not involve computation in the output list like complex expressions do; hence, when A is encountered for printing, its current value (the new one) is used.

GOSUB (Invoke Internal Procedure)
Subroutine (Invocation of Internal)

Purpose: To invoke a procedure contained in the BASIC program.

Prototype: GOSUB {ln}

where:

ln is the line number of the first executable statement of the procedure to which control is to be given.

Abbreviation: G'B

Effect: A program is initially at zero GOSUB level. Each time a GOSUB is issued, that place in the program is marked, the GOSUB level is increased by one, and control is given to the executable statement having the line number ln in that program. When a RETURN statement is issued within the same program, the GOSUB level is decreased by one, and control is returned to the next statement after the last GOSUB. At most 21 GOSUBs can be issued without matching returns; otherwise, an error condition results. Also, if more RETURNs are issued than GOSUBs, the program returns to whatever invoked it (usually /RUN in command mode).

Note: This mechanism may be used to invoke procedures needed at various points within a program.

Examples: Define and call a procedure to read two values, add them, store the result, and print it out. Excerpts from a complete program follow.

```
100 GOSUB 500
110 P=COS(Z)
       .
       .
       .
200 GOSUB 500
210 Q=SIN(Z)
```

```
      .
      .
      .
500 INPUT X,Y
510 Z=X+Y
520 PRINT "SUM =";Z
530 RETURN
```

The internal procedure consists of lines 500 through 530. Note that it can refer to any variable in the program since it is a procedure internal to the program. After being invoked by statement 100, the procedure returns to statement 110. Similarly, the RETURN is made to statement 210 after the invocation at statement 200.

<u>CALL (Invoking Another Program)</u>
<u>CHAIN (Relinquish Control to Another Program)</u>
<u>Subroutine (Invocation of External, Suspend Execution)</u>
<u>Subroutine (Relinquish Control to External)</u>

Purpose: To invoke another BASIC program and possibly eliminate the invoking program (and any program which invoked it) from the run.

Prototype:

```
CALL {sub}
CHAIN {sub}, {ln}
```

where:

<u>sub</u> is the name of the BASIC file containing the program to be CALLed or CHAINed to. This file is assumed to be the O (object) file containing the translated (machine language) version of the S (source) file sub; hence, <u>the user should have already translated this program prior to CALLing or CHAINing to it</u> (e.g., /COMPILE {sub} PAR=NOEXECUTE).

<u>ln</u> is the line number in the program to be CHAINed to.

Abbreviation: C'L for CALL
C'N for CHAIN

Effect: <u>CHAIN</u> is used to invoke another program starting at a given line number and to terminate all other programs currently running. For example, if A calls B which calls C, then A, B, and C are considered as running. If, however, C then chains to D, then A, B, and C are terminated and D begins running just as if it had been invoked directly from command mode (via /RUN, /EXECUTE, etc.).

<u>CALL</u> is used to invoke (call) a program (starting with the first executable statement of that program) to accomplish some task. Once the called program has finished its task, it may return to the calling program via the END or RETURN (at zero GOSUB level). The return is to the next executable statement after the CALL. A program may be called by itself, directly or indirectly (via a

sequence of calls). For example, A calls B which calls C which calls D which calls B. The first level of invocation of any program is level one. Each progressive instance of a program in a sequence of calls is at one higher level than the preceding instance of the same program. Hence, the first occurrence of B in the above sequence is at level one and the second occurrence of B is at level two. A, C, and D are all at level one.

Note: Each time a program is called or chained to, it is given a new set of variables (arrays, matrices, strings, and simple variables). The numeric variables are set to zero as well as all array or matrix elements. String variables (arrays) are nulled. In the case of CALLing, these variables in one instance of a program are different from the variables in another instance of the same program. For example, if P1 calls P2 which calls P1, the variable X in the level two instance of P1 is distinct from X in the level one instance of P1. When the level one instance is returned to, its variables have the same values as they did prior to the CALL on P2.

The READ FILE and WRITE FILE statements may be used to communicate information between calling and called programs. The calling program can write into a file. The called program can then read the information and use it to compute results which it can write back into the file before returning to the calling program. The calling program can then read the results. See an example of this below.

Examples: (1) Program which calls another program to compute the roots of the quadratic equation $ax^2+bx+c=0$. Assume that the following has been performed.

<u>Subroutine Example (Quadratic Equation)</u>
<u>Function Example (Quadratic Equation)</u>

```
/OPEN ROOTS
10 FILE ROOTCOM
20 READ FILE 1,A,B,C
30 R=SQR(B*B-4*A*C)
40 X1=(-B+R)/(A+A)
50 X2=(-B-R)/(A+A)
```

```
60 I=CMD("/EMPTY@NC@T ROOTCOM@D")
70 WRITE FILE 1,X1,X2
80 RETURN
90 RESTORE FILE 1
/COMPILE PAR=NOEX
```

Then the following sequence of code within a running BASIC program will compute the roots.

```
      .
      .
      .
10 FILE ROOTCOM
20 INPUT A1,B1,C1
25 RESTORE FILE 1   /* OPENS ROOTCOM (FIRST REF.)
30 I=CMD("/EMPTY@NC@T ROOTCOM@D")
40 WRITE FILE 1,A1,B1,C1
50 CALL ROOTS
55 RESTORE FILE 1
60 READ FILE 1,R1,R2
70 PRINT "ROOTS ARE:  ";R1;" AND ";R2
```

(2) Now define a program which calls on itself to produce factorials. Assume that the following has been performed.

<u>Recursive Programming</u>
<u>Function Example (Recursive Factorials)</u>
<u>Subroutine Example (Recursive Factorials)</u>

```
/OPEN FACT
10 READ N    /* READ FROM THE DATA FILE.
20 IF N=1 THEN:  RETURN
30 RESTORE
40 WRITE N-1
50 CALL FACT
60 RESTORE
70 READ S
80 RESTORE
90 WRITE S*N   /* N*((N-1)!)
100 RETURN
/COMPILE PAR=NOEX
```

Then the following program will compute factorials using the program FACT while writing and reading FACT's data file.

```
10 FILE FACT
20 INPUT M
30 RESTORE FILE 1
```

```
40 WRITE FILE 1,M
50 CALL FACT
60 RESTORE FILE 1
70 READ FILE 1,F
80 PRINT M;" FACTORIAL IS ";F
90 GOTO 20
```

<u>RETURN (Return to GOSUB or Calling Program)</u>
<u>Subroutine (Internal or Cond. External Return)</u>
<u>Function (Conditional Return from External)</u>

Purpose: To return to that part of a BASIC program that issued a CALL or GOSUB statement.

Prototype: RETURN {com}

where:

<u>com</u> is an <u>optional</u> comment that may follow the RETURN statement. This comment is <u>not</u> printed during execution.

Abbreviation: R'N

Effect: If no GOSUBs have been given in the program issuing the RETURN, the effect is that of an END statement. That is, if the program was invoked in BASIC command mode (via /RUN, /EXECUTE, or /DEBUG), control is returned to command mode. If another program issued a CHAIN to it, the RETURN also returns to command mode. Otherwise, it was invoked by another BASIC program via a CALL statement, so control is given to the statement after the CALL in the calling program.

In the case where a GOSUB has been issued for which no RETURN has been made, control is passed to the statement immediately after the GOSUB. This return is always made to the <u>last</u> GOSUB issued.

Note: RETURN conditionally terminates the program issuing the return as compared with END, which does this unconditionally, and STOP, which stops <u>all</u> programs executing (e.g., in a linked sequence of CALLs).

Examples:

```
1000 RETURN
2569 R'N TO CALLING PROGRAM
```

END (Return to Calling Program)
Subroutine (Unconditional Return from External)
Function (Unconditional Return from External)

Purpose: To terminate the BASIC program currently executing and return to whoever invoked it.

Prototype: END {com}

where:

com is an _optional_ comment that may follow the END statement. This comment is _not_ printed during execution.

Effect: If the BASIC program was directly invoked via a command (/RUN, /EXECUTE, or /DEBUG), control is returned to BASIC command mode. If the program was invoked (CALLed) by another BASIC program, control is returned to the statement after the CALL in the calling program. If it was invoked via a CHAIN statement in another BASIC program, control is returned to command mode.

Note: END unconditionally terminates the program issuing the END as compared with RETURN which does this conditionally. STOP stops _all_ programs executing (e.g., in a linked sequence of CALLs).

Examples:

```
30 END
50 IF I>J, THEN:  END THIS PROGRAM
```

V.H THE CMD (COMMAND) FUNCTION

CMD (Command Function)
Commands from a BASIC Program

Purpose: To issue command lines and/or data lines to BASIC from a program that is running.

Prototype: {r}=CMD({s})

where:

_s_ is a string (variable or constant) containing a _single_ BASIC _command (or data line)_ to be given to BASIC for processing. It may be any command except /RESET, /RUN, /COMPILE, /DEBUG, or /EXECUTE. _s_ may be a data line to be edited into the "active" file.

_r_ is the value returned by CMD indicating the level of success in processing the command. The return r=0 if the command was successfully carried out, r=1 if the command is valid but illegal in this context (see above), and r=2 if the command is just plain illegal in any context (e.g., /SCREAM or /EMPTY@JUNK X). For s as a data line, if there is no "active" file, the return will be r=1; otherwise, r=0.

Effect: The command in s is given to BASIC for processing. If legal, it is executed. The CMD function then returns the indicator r indicating success or failure. For example, I=CMD("/EMPTY@NC PROG1@D") empties PROG1's data file while bypassing user confirmation by using the NC (no confirm) modifier. If a data line is given, it is edited into the "active" file (if it exists).

Usage: CMD is an extremely useful function. It can be used to copy files, empty or destroy files, edit files (via /OPEN and issuing data lines with line numbers via CMD, or /ALTER, /SCAN, etc.), or modify or display variables (via /DISPLAY or /MODIFY). In fact, there is very little that one cannot do with the CMD function and a little imagination. Examples follow.

Examples: <u>Sample Lines</u>

```
10 I=CMD(AA(4))
20 J=CMD("/SET JUSTIFY=RIGHT")
30 GOTO(70,80,90),1+CMD(BB)
```

<u>Complex Example One (Self-Annihilation)</u>

If PROG1 issues CMD("/DES@ALL@NC@T PROG1"), then after PROG1 has terminated its execution, it will no longer exist (i.e., permanent and working copies of its data and source files as well as the translated program (its object file)). The NC modifier defeats user confirmation of the /DESTROY (i.e., confirmation is not necessary) and the @T suppresses the "Done" message.

<u>Complex Example Two (Line-Editing of a Data File)</u>

```
10 I=CMD("/OPEN X@D")
20 I=CMD("10 1,2,3")
30 I=CMD("20 4,5,6")+CMD("30 7,8")
40 I=CMD("LIST")+CMD("REL")
```

or equivalently

```
10 I=CMD("OP X@D")+CMD("NUM")
20 I=CMD("1,2,3")+CMD("4,5,6")+CMD("7,8")
30 I=CMD("/UNN")+CMD("LIST")+CMD("REL")
```

Note that X@D being open makes it the active file. Hence, the previously active file (if there was one) is no longer active.

<u>Complex Example Three (Interacting with MTS)</u>

If a program issues CMD("/SYSTEM") or CMD("/MTS"), then the BASIC System, while still remaining available, turns control over to MTS; hence, all lines typed by the user go to MTS until the <u>MTS</u>

command $RESTART is typed to return control to BASIC. The program which issued the CMD will then resume. If before typing $RESTART the user issues a $RUN of some program in MTS (or $LOAD, $UNLOAD, or $SIGNOFF), BASIC will cease to be available until it is rerun (via $RUN). Otherwise, the user can, for example, create MTS files, edit them, possibly destroy some (when file space is tight), and then return to BASIC via $RESTART.

Complex Example Four (Creating a BASIC Program)

```
10 I=CMD("/OPEN NEWPGM") /* SOURCE FILE
20 I=CMD("10 INPUT A,B")+CMD("20 Y=A+B")
30 I=CMD("30 PRINT A,B,Y")+CMD("40 GOTO 10")
40 I=CMD("/REL")
```

Later, the user can issue /RUN NEWPGM after the current program terminates.

V.I SPECIAL INPUT/OUTPUT CONTROLS

TAB (Output Tab Function)
SKP (Input Data-Skipping)
; (Packed Output)
, (Normal I/O)
: (Output Carriage Return)
COLWIDTH (Output Column Width)
JUSTIFY (Output Field Justification)
Format (Input/Output Controls)
Input/Output Controls (Formating)

Purpose: To control input and/or output processing.

Prototypes:

```
TAB({exp1})
SKP({exp2})
```

where:

exp1 is an arithmetic expression whose value denotes the column to tab to for printing of the next output item. If exp1 exceeds the line length for the device (e.g., teletype), the remainder upon division by the line length will be used.

exp2 is an arithmetic expression whose absolute value (truncated to an integer) denotes the number of data items to be skipped over on input. It must not exceed 1000.

Effect: All of these are controls used in either input or output lists (restrictions are stated below).

TAB is used to tab only for terminal output (or for the printer in MTS batch). The TAB reference is separated from the other list items by commas or semi-colons. For example, the statement PRINT A,TAB(20);B will print the value of B starting in column 20.

SKP is used to skip over data items on input (terminal or BASIC file reading). The SKP reference is separated from other input items by commas. It is restricted to non-matrix reading, i.e., INPUT, LINPUT, or READFILE. For example, INPUT A,SKP(2),B reads A, skips past the next two input data items, and reads B. The range of the SKP argument is 0 to 1000 inclusive.

, is used as a normal separator of data items in an input or output list. It may be replaced by ; or : in certain contexts, e.g., INPUT A,B or MAT READFILE 1,A,B.

; is used to indicate packed output for regular or matrix terminal output (or printed output in MTS batch). In the matrix case, it must follow the matrix name to which it applies. The matrix elements are printed separated by a single blank. For example, MAT PRINT A;B,C; will print A packed, B normally spaced, and then print C packed. In the normal terminal output, any two list items separated by a semi-colon will be printed with the second immediately following the first, e.g., PRINT "X =";P*Q. If the second is a number, it will be separated from the first by a blank since numbers are printed with one leading blank. If JUSTIFY=RIGHT (see below) then a semi-colon must follow the second list item in order to override the right adjustment.

: is used to give a carriage return (start a new line for terminal output). It may only be used with the PRINT statement (unnecessary for MAT PRINT). It may occur at the end of an output list or separate any two list items. For example, PRINT A,B:C:D will print three lines with A and B on the first, C on the second, and D on the third.

COLWIDTH The normal terminal output field width is 15 columns. The user may issue via the CMD built-in function a /SET command to change this parameter, e.g., I=CMD("/SET COLWIDTH=10"). See the /SET command description for details.

JUSTIFY With the PRINT or MAT PRINT terminal output statements, the data items are left-justified (by default) in the output fields. The user may change this by issuing a /SET command via the CMD built-in function while running a program.

Consult the /SET command description. For example, I=CMD("/SET JUSTIFY=RIGHT"). Also, note that JUSTIFY=RIGHT affects the semi-colon packing as described above.

VI RUNNING A BASIC PROGRAM

Running a Program

There are four ways to run a BASIC program; namely, via /RUN (the most common), /EXECUTE (to avoid program retranslation), /DEBUG (to enter debugging commands prior to execution), and /COMPILE (to translate programs to be CALLed on or CHAINed to by other programs). When any of the above except for /EXECUTE is given (e.g., /RUN PROG1), the source file (S) containing the program is translated into a machine language version of the program which is stored in the object file (O). If during translation DATA statements are encountered, the data file (D) is first emptied (or created if it does not exist) and the DATA lines are edited into it. The action of emptying the file may be inhibited by issuing the command /SET DFILEMP=OFF. The object file is then executed in "debug mode" so that program errors will cause BASIC to interact with the user to debug the program. For details on "debug mode", see the next section on debugging. If /EXECUTE is specified with PAR=DEBUG, the object file is executed with the same provisions for debugging as above. Note that it is economical to use /EXECUTE for running checked-out programs.

In any case, just prior to execution, all program string variables are nulled (set to "") and all numeric variables and array elements are set to zero. Note that these initialization constants may be changed via the /SET command before executing the program (the STRCON and VARCON parameters). All files referred to in FILE statements within the given program are restored (rewound or set to their beginning) if they already exist. If they do not exist, they will be automatically created whenever they are manipulated (READ, WRITE, RESTORE, BACKSPACE) during execution. This may be defeated via the command /SET DFILECRE=OFF. If the program calls (or chains) to another program, _each_ time the called program is invoked, the files it manipulates are restored just prior to their _first_ use in the called program. The positioning of the non-manipulated files is not altered. The command /SET RESTORE=LOCAL will defeat this global restoration making the user responsible for explicitly issuing RESTORE statements in the programs for _all_ files.

In case of an error in the program during execution, an error comment indicating what the error was is typed and the user is notified of the program name and the line in which the error occurred. The user is then prompted for debugging the program.

For example:

```
     +Attempt to divide by zero.
     +At Line "20" in Program "PROG1"
     Ready!
     >
```

where the greater-than prefix means that the user may enter debugging commands as well as "normal" commands. If the user wishes a shorter form of notification, the command /SET ATLINE=OFF may be issued to eliminate the "At Line" comment and/or the command /SET TERSE=ON to eliminate the "Ready!". The "At Line" comment may be retrieved at the user's discretion by issuing the command /WHERE?.

The user may terminate a program that is running by giving a /ENDFILE (or the appropriate end-of-file character) when prompted for input at the terminal (via INPUT or MAT INPUT). To interrupt a program (which possibly is in error - e.g., an infinite loop), the user should give an attention via the attention key at the terminal (e.g., BREAK at a teletype). This will cause BASIC to enter interactive "debug mode". In this mode, one simply types /STOP to terminate the program or gives whatever debug commands that seem appropriate to discover the skulking error(s).

VII DEBUGGING A BASIC PROGRAM

Debugging a Program

Debugging programs can be compared with detective work in that it involves searching for clues for evidence as to what went wrong with the program. If there is not enough evidence, you must produce some clues to help you. Some fairly standard ways of debugging are: computing the results by hand while following the program logic (drudgery), choosing special, simple test data which *may* illuminate errors, or inserting print statements in the program to print out intermediate results or indicate the logical paths that the program is taking.

To assist the user in debugging his program, BASIC has a powerful debugging facility. This facility allows the user to trace program flow, pause in the program when it reaches certain points (possibly to examine results), display or modify variables, or alter the logic by transferring control to other parts of the program (possibly redoing some computation after changing some variable values), etc.

In order to use the facility, the user must be in interactive "debug mode" which is entered either by running a BASIC program (see Section VI) which results in some fatal error (e.g., divide by zero) or which is interrupted by the user giving an attention, or by explicitly issuing the command /DEBUG PROG where PROG is the name of the program to be debugged. /DEBUG, the second way of entering interactive debugging, readies the program for execution but allows the user to give debugging commands before starting the program via the /START command.

In the first (more common) case, BASIC prints out an error message (prefixed by a plus (+) sign to indicate debugging messages) indicating the error, and followed by a comment indicating where and in what program the error occurred. If the user gave an attention, BASIC types ATTN!, instead. In any case, it then types "Ready!" and prompts the user with the debugging prefix (>). At this point the user may issue debugging commands. For example, assume the following program DIV:

```
     10 INPUT A,B
     20 Y=A/B
     30 PRINT A,B,"ANS =";Y
     40 GO TO 10
```

If while running DIV the user typed 2,0 as input to statement 10, BASIC would respond as follows:

```
+ Attempt to divide by zero.
+ At Line "20" in Program "DIV"
+ Ready!
>
```

At this point the user could display B and see that it is zero (via /DISPLAY B), modify B (say to 4 via /MOD B 4), and then retry the computation (via /GOTO 20). Another option is to /GOTO 10 and read in a new set of values. If the user just wants to stop the program, /STOP should be typed.

For more complex programs, other facilities are useful such as tracing program logic by obtaining a list of line numbers of statements as they are executed via /TRACE LINES. Breakpoints, statements in the program where BASIC should pause and allow interactive debugging, may be set with the /BREAKPOINT command (e.g., /BREAKPOINT 20 30). This is useful for pausing to modify or display variables before executing the statements at the breakpoints. The user may also obtain a list of all the breakpoints that have been set via the /BREAKLIST command. To remove all breakpoints, the user types /CLEAN. To remove only a specified few, /RESTORE should be typed followed by the line numbers of the statements having the breakpoints to be removed. To continue after a breakpoint, the user should type /CONTINUE; in fact, /CONTINUE may be used to continue a program whenever it pauses in debug mode. The same is true with /STOP. The user can also "slow down the program" by essentially stepping through the program statement by statement (via /SINGLESTEP) as if a breakpoint were set at every statement. The user continues in "singlestep mode" until he decides to resume "normal speed" via /SINGLESTEP OFF.

For programs which CALL or CHAIN to others, CALLs, CHAINs, and RETURNs may be traced with BASIC, pausing in the program that the statement being traced is going to. BASIC will indicate the name, line number, and program level (if it is greater than 1 - see the CALL writeup).

The user can only debug a program when it is the current program being executed; hence, if A calls B and an error in B occurs, the debug commands entered apply only to B, not to A. If, however, the user wishes to debug a program which will eventually be called (or returned to), CALLS and RETURNS may be traced. When notified that, for example, B has returned to A, debugging commands for A may be issued. Note that breakpoints set for a program are not remembered when that program returns to another program. For example, if B has breakpoints and it returns to A, then its breakpoints are lost. These restrictions for multiple programs were part of a design decision to minimize storage overhead in program debugging. This decision was based on the expectation that most BASIC programs would not be that complex. In any case, the user who does need to extensively debug multiple programs still can use the current facilities with a little more work on his part. Moreover, since subroutines are BASIC programs which may take calling parameters in files, they can be debugged individually by the user placing data in a file and running the subroutines as single programs.

After the errors have been found (hopefully all of them), the user may make corrections in the program file via the BASIC line- or text-editing facilities. However, in order to run the program with these corrections, it is necessary to return to normal command mode (via /STOP) and then /RUN the corrected program(s) (possibly also recompiling any erroneous CALLed programs).

The user may issue _any_ BASIC command while debugging _except_ for /RESET, /RUN, /COMPILE, /DEBUG, and /EXECUTE. All others are legal. The commands which can be given _only_ when a program is running are: /BREAKLIST, /BLIST, /BREAKPOINT, /CLEAN, /CONTINUE, /DISPLAY, /GOTO, /MODIFY, /RESTORE, /SINGLESTEP, /START, /STOP, /TRACE, and /WHERE? The /WHERE is used when ATLINE has been turned off via the /SET command and the user wishes to know where an error occurred that invoked interactive debugging. Please refer to the respective command writeups for details on their usage.

VIII TEXT-EDITING OF PROGRAM OR DATA FILES

Text-Editing of Files

To make alterations of lines in a BASIC source or data file, the user may simply open the file and type in the new line. If as a result of programming or data composition errors the user has numerous changes to make, the BASIC text-editing commands may be used to search for text and alter it. There are four commands that apply: /OPEN, /SCAN, /EDIT, and /LINE#. For the first three, an explicit file reference (necessary for /OPEN) will make the file the current active file. In addition, two internal pointers are used for scanning and editing text; namely, the "scan pointer" and the "line pointer". Definitions will be given here informally via examples. The user should consult the respective command writeups for details.

The /OPEN command allows one to make a file active for editing. One can type data lines directly into the active file or delete lines from the active file by just typing their line numbers. For contextual scanning and editing, the /SCAN and /EDIT commands may be used.

The /SCAN command allows one to search a file for occurrences of a sequence of characters (commonly called a pattern or string) and print out the line number(s) and contents of the line(s) where the pattern occurs. A general rule is that the delimiting character for the pattern may be any character that is neither alphanumeric nor an asterisk. The character chosen by the user must not be part of the pattern (and in the case of the EDIT command, a part of the replacement) due to its delimiting function. Examples will clarify this. For example, if one wants to find the first line in the source file PROG1 that contains the misspelled statement:

```
          IMPUT A,B
```

one simply issues the command:

```
          /SCAN 'IMPUT' PROG1
```

and BASIC will make PROG1 the "active file," set the "scan pointer" to the beginning of PROG1, and search the file line by line until it finds the desired line. Assuming that line 30 is the first line containing the given pattern, BASIC would print:

```
            30 IMPUT A,B
```

At this point, the "line pointer" is at line 30, and the "scan pointer" is set to 31 so that if the user wishes to scan further into the file, he may just issue another SCAN. To correct line 30, the user could either retype the entire line or issue the command:

```
          /EDIT :M:N:  30 PROG1   or
          /EDIT #M#N#
```

since /EDIT uses the "line pointer" if no line number is given and the "active file" if one is not explicitly given. Either command line would replace the "M" by "N" in line 30. In any event, to look for more misspelled "IMPUT", the user types:

```
          /SCAN
```

and the previous scan pattern 'IMPUT' is used on the "active file" starting at the "scan pointer". Note that if a new pattern is being searched for (e.g., "PRONT" - a misspelling of "PRINT") then:

```
          /SCAN "PRONT"
```

will continue from the "scan pointer" line onward in the search. If the user wishes to start the search from the beginning of the file, he just rementions the name PROG1 to reset the "scan pointer" to the beginning of the file. That is:

```
          /SCAN 'PRONT' PROG1
```

To scan within a line range of a file one could write, for example:

```
          /SCAN 'PRONT' PROG1(30,120)
```

To search for <u>all</u> lines which have occurrences, one uses the @ALL modifier on the command. In this case, the "line pointer" points to the last line found in the search and the "scan pointer" to one past that, unless the search is unsuccessful, in which case no pointer is changed. In all cases of /EDIT or /SCAN, the lines are printed unless the @¬VERIFY modifier (or @NV) is used. Note also that to change all occurrences of a pattern in a file, @ALL may be used with /EDIT. This changes only the <u>first</u> occurrence (left-most) of the pattern in each line that it is found. To change <u>every</u> occurrence within a line, one uses the @EVERY modifier (or @EV) as in:

```
          ED@EV 'CLARK'PAT' 50
```

To change <u>all</u> lines and <u>every</u> occurrence within each line the @ALL and @EVERY modifiers may be used in the combination @ALL@EVERY (or @AE) as in:

```
          EDIT@ALL@EV 'CLARK'PAT' FILE2@D or
          ED@AE 'CLARK'PAT' FILE2@D
```

/EDIT will edit only a single line unless the @ALL modifier is specified, in which case it will edit all lines starting with the current line onward. For example:

```
          /EDIT 'PRINT'WRITE' 10 PROG
```

will edit the left-most occurrence of "PRINT" to "WRITE" in line 10 of file PROG.

To edit all lines in PROG one could write:

```
          /EDIT@ALL 'PRINT'WRITE' PROG
```

To restrict the line range one could issue:

```
          /ED@ALL 'PRINT'WRITE' PROG(30,120)
```

or alternately:

```
          /OPEN PROG(30,120)
          /ED@A 'PRINT'WRITE'
```

To start editing all lines from a given point in a previously defined file, one could write, for example:

```
          /OPEN PROG(30,120)
          /ED@A 'PRINT'WRITE' 60
```

to edit from line 60 to 120. One could then specify a new pattern to be used starting, for example, at line 80 as in:

```
          /ED@A 'INPUT'READ' 80
```

The /LINE# (or /L#) command is used to position the "line pointer" and print (if verification is on) the contents of the line. For example,

```
          /LINE# 10
```

sets the "line pointer" to 10 and prints the line number followed by the contents of the line. The special symbols:

```
          *, *F, and *L
```

are used with /LINE# to refer to the current line (the one the "line pointer" points to), the first and last lines of the file (or line range, if previously specified), respectively. For example:

```
          /LINE#@NV  *F
```

sets the "line pointer" to the first line of the file without printing the line. An abbreviated form is:

```
          L#@NV  *F
```

In summary, to scan or edit a file (S- or D-type), the user can either make it active via /OPEN and then refer to it, <u>or</u> explicitly refer to it the first time in using /EDIT or /SCAN to establish it as the "active file". Afterwards, he may explicitly mention line numbers when editing or use /SCAN to search for lines and then refer to the "current line" (via the "line pointer"). Moreover, the "line pointer" may be repositioned via L# to a specific line number or generically (*F or *L) to the first or last line of the file. Operations on multiple lines may be specified by using the @ALL modifier. To reduce the amount of printing (of line contents) the @NV modifier may be used for local effect on a single command or

```
          /SET VERIFY=OFF
```

for a global effect on all commands. Note also that since the file name may contain a line range, for example,

```
          PROG1(20,70)
```

the editing and scanning may be limited to part of the file. In this case *F and *L refer to the actual first and last lines within this range. The patterns for /EDIT or /SCAN may be left out of the commands if one is referring to the <u>same</u> patterns used previously with them. Hence, if line 30 contains "HAHA", to change it to "HOHO" one issues

```
          /EDIT 'A'O' 30   followed by
          /EDIT
```

which repeats the edit on line 30 to change the second "A" to "O". Alternately, one could use the @EVERY modifier with only one EDIT command:

```
          EDIT@EV 'A'O' 30
```

Also, remember that the patterns must be defined with sufficient context to indicate the proper character being searched for or in need of alteration. For example, if line 30 has

```
          PRINT I,K
```

and one wishes to change it to

```
          PRINT J,K
```

then

```
          /EDIT 'I'J' 30
```

will not work. The result is

```
          PRJNT I,K
```

that is, the first I was changed as requested. Taking in some context, e.g.,

```
          /EDIT 'I,'J,'
```

will work.

One point regarding the null string (string of length zero): If it is the left part of a pattern for /EDIT, e.g.,

```
          /EDIT ''A'
```

then the right part is inserted at the beginning of the line. If it is the right part of an edit pattern, whatever the left part matches is replaced by null (i.e., deleted).

Examples: The following commands progressively scan for two lines in PROG1 containing "PRINT" and substitute "WRITE".

```
/SCAN 'PRINT' PROG1
/EDIT 'PRINT'WRITE'
/SC
/ED
```

To edit all such lines:

```
/ED@A 'PRINT'WRITE' PROG1
```

Other miscellaneous unrelated examples

Print the current line (the line pointer points to).

```
     L# *
```

Alter all occurrences without printing, starting at line 20 in PROG2.

```
     E@A@NV 'PRINT'WRITE' 20 PROG2
```

Reposition "line pointer" to file beginning.

```
     L# *F
```

List all lines containing "INPIT" in the range 30 to 70 in PROG1.

```
     SC@A 'INPIT' PROG1(30,70)
```

Change all primes (') to quotes (") in the data file F1.

```
          ED@AE :':": F1@D
```

Change all primes (') to colons (:) in lines 120 through 170, inclusive, in data file F2.

```
          ED@AE "'":" F2(120,170)@D
```

Delete the first occurrence of the letter "A" in line 10 of the active file.

```
          ED 'A'' 10
```

Scan for all lines having the character 2 in the data file F3.

```
          SC@A 121 F3@D
```

Here the delimiting character may be numeric (e.g., 1) since the SCAN command does not use line numbers in isolation. Hence, 121 will be interpreted as a pattern (of a single character 2 delimited by ones) rather than as a line number. This would not be the case for the EDIT command.

IX SAVING PROGRAMS AND DATA ON PERMANENT STORAGE

Permanent Files (Saving)
Permanent Storage (Saving Files on)
Files (Saving on Permanent Storage)

In order to save programs or data for later use (e.g., for a later terminal session), the user must have MTS disk space assigned to the user id. There should be a minimum of 2 pages for each BASIC file that the user wishes to save. For example, if the S, O, and D files of PROG1 are to be saved, the user must have at least 6 disk pages available. Typically, one saves the source (S) and data (D) files and issues /RUN to obtain a new object file the next time the program is needed. In some cases, object (O) files are saved when those programs are used frequently as subroutines and the user wishes to reduce expense by eliminating the cost of translating them each time they are needed. This must be compared with the increased cost of permanent storage and the convenience of having the programs immediately accessible when needed.

When the user requests that a file be saved, BASIC tries to obtain permanent storage space to contain it. If none exists, BASIC notifies the user; otherwise, the file is saved.

The user requests saving of his "temporary" or "working copy" files via the /SAVE command. For example, /SAVE PROG1 saves the source program PROG1. To save the object or data files of PROG1, issue /SAVE PROG1@O or /SAVE PROG1@D. To save everything (including the O file if it exists), just use the @ALL modifier, e.g., /SAVE@ALL PROG1. Note that saving does not destroy the "working copy". If a permanent copy already exists, it is replaced (and in the case of a permitted file, unpermitted, replaced, and re-permitted). Also, if one is saving an empty file, it will not really be saved; moreover, if there is already a permanent copy of that file, it will be destroyed (after all, why save an empty file and incur unnecessary permanent file tenancy charges?). See the /SAVE command writeup for more details.

Note: The files created are MTS sequential files that are formatted for efficiency of usage in BASIC in such a way that they cannot be listed via the MTS $LIST command.

Examples:

```
/SAVE@A PROG1
/SAVE PROG1@D
SA PROG2@O
```

X SHARING BASIC FILES AMONG USERS

Files (Sharing Among Users)
Sharing Files Among Users

It is possible for BASIC users to share (obtain copies of) users' source, object, or data files. All that the user having the desired files need do is permit them for others' use. Whoever wants copies of the files need only indicate to BASIC that they are sharing them and give the user id of the "owner". At any time the "owner" of the files wishes to disallow this access, an unpermit should be issued. See the /PERMIT, /SHARING, and /UNPERMIT command writeups for details; however, the following summarizes the facility.

PERMITTING: If the user with id Q123 has three files for program PROG1, namely the source, object, and data files, then all may be permitted via the command:

```
/PERMIT@ALL PROG1
```

If access is to be granted to only one or two of the files, then they must be explicitly referenced by name. For example:

```
/PERMIT PROG1@D
/PERMIT PROG1@O
PE PROG1
```

SHARING: Assume the above permitting was done by user Q123. User Z456 wishes to obtain a copy of Q123's source program and data (e.g., PROG1@S and PROG1@D). User Z456 may have permanent files by these names but must not have temporary core copies by these names. If this condition is met, the user would then issue the commands:

```
/SHARING PROG1@D Q123
/SHARING PROG1 Q123
```

For each file needed, a temporary or "working copy" would be opened (but not made "active") and the corresponding file of Q123 copied (completely unaltered) to the "working copy" under Z456. It is Z456's job to explicitly /SAVE these "working copies" if he wishes his own permanent copies. To share all of Q123's PROG1 files, the @ALL modifier could be used on the /SHARING command. Note that if a user tries to share a file that has not been permitted, BASIC will complain. If user Q123 has permanent files

(S, O, and D) under the name PROG1, the files obtained from user Q123 can be renamed to PROG2 by the following:

```
          /SHARING@A PROG1 Q123
          /RENAME PROG1 TO PROG2
          /REN PROG1@D PROG2@D
          /REN PROG1@O PROG2@O
```

Now /SAVE@A PROG2 can be issued.

UNPERMIT: To disallow access to any file, Q123 need only issue the /UNPERMIT command. For example:

```
          /UNPERMIT PROG1@D
          UNP@A PROG1
```

MTS DETAILS: All files permitted by BASIC are permitted READ to OTHERS. This augments any permit status given to the files by the user via the MTS file-permitting facility. Users can copy such files outside of BASIC by means of the MTS $COPY command (with TRIM *off*). The user must create a sequential file (without line numbers) of a size of at least two pages (issue $FILESTATUS on the file in question) with name BSC.TXXXXXXX where T is either S, O, or D and XXXXXXX is the BASIC file name. The other user's file may then be copied to the file just created. For example, there exists under the user id W070 a BASIC source file named GUNNER. To get a copy of this file in MTS, one would issue the MTS commands:

```
          $CREATE BSC.SGUNNER TYPE=SEQ SIZE=2P
          $COPY W070:BSC.SGUNNER@-TRIM TO BSC.SGUNNER@-TRIM
```

X.1 INPUT FROM AND OUTPUT TO MTS

MTS I/O in BASIC
Input from MTS
Output to MTS

The BASIC input and output streams are initialized to the MTS pseudo-device names *SOURCE* and *SINK*, respectively, but the assignments may be changed by the user while running in BASIC through the use of the BASIC pseudo-device commands (see Section XIII) which may be given either as input lines (from the user input stream) or output lines (e.g., from a running program). Some useful MTS-supported devices are paper tape, magnetic tape , line printers, and MTS files. The user should consult the appropriate MTS manual for the device he wishes to use.

X.1.A INPUT FROM MTS

Input from MTS (File, Paper Tape, Magnetic Tape,etc.)

The input stream may be specified as any valid MTS input device via the BASIC #INP pseudo-device command. For example, a set of BASIC commands and data lines could be stored in an MTS file (e.g., MTF1), and the command #INP=MTF1 could be used to direct BASIC to read from MTF1. The direction may given *any* time BASIC is expecting user input (e.g., command mode prompting for commands or program prompting for data) from the current input stream (e.g., a terminal or an MTS file). When either an end-of-file is encountered or an attention is given, BASIC will automatically redefine the input stream to be *MSOURCE*.

X.1.B OUTPUT TO MTS (E.G., FILE AND PRINTER OUTPUT)

Printer Listings from BASIC
Output to MTS (File, Paper Tape, Magnetic Tape, etc.)

The output stream may be specified as any valid MTS output device (e.g., a file, *PRINT*, *PUNCH*, etc.) via the BASIC #OUT pseudo-device command (e.g., #OUT=MTF2). To revert back to the normal definition, an *explicit* #OUT=*SINK* must be given. This facility is most useful in producing output listings on a remote printer via *PRINT* as illustrated in the following terminal session. The first character of each line in the example is either the BASIC or MTS command mode prompting characters (:) and (#), respectively.

Printer Listings from BASIC, Example

```
1)  :#OUT=*PRINT*    (Output stream is *PRINT*)
     (The MTS *PRINT* receipt is printed here.)
2)  :MTS    (Pass control to MTS)
3)  # (At  this  point,  the user may qualify[1] *PRINT* with a hold
      status, routing, multiple copies,  uppercase  and  lowercase
      printing, etc.)
4)  #$RESTART    (Return control to BASIC)
5)  :LIST X    (List a BASIC file)
6)  :MTS    (Pass control to MTS)
7)  # (At  this  point, the user should release[2] *PRINT* if he put
      it in hold status in 3).)
8)  #$restart    (Return control to BASIC)
9)  :#OUT=*SINK*    (Back to normal output stream)
     (*PRINT* will be released from BASIC here.)
```

Example 11.1 Printer Listings from BASIC Command Mode

If no special conditioning of *PRINT* (routing, multiple copies, etc.) is desired, only lines 1), 5) and 9) are necessary.

```
1)  :#OUT=*PRINT*    (Output stream is *PRINT*)
5)  :LIST X    (List a BASIC file)
9)  :#OUT=*SINK*    (Back to normal output stream)
```

In <u>all</u> cases, line 9) is necessary to release *PRINT* from BASIC. If 3) specified a hold on *PRINT* then 7) must release it since the user's hold in 3) is independent of BASIC's hold resulting from 1).

----

[1]This is accomplished by the MTS control command as illustrated by the following example.

```
$CONTROL *PRINT* HOLD PRINT=TN ROUTE=CNTR COPIES=2
```

[2]$CONTROL *PRINT* RELEASE

X.1.C MTS I/O FROM A RUNNING BASIC PROGRAM

Output to MTS from a Running BASIC Program
Input from MTS to a Running BASIC Program

When a BASIC program is running, it may use #OUT and #INP assignments by executing a PRINT statement with the required pseudo-device command as a string. The string must be referenced in the output list as a single output line. The string will be interpreted but not printed. The following sequence illustrates the usage.

Printer Listings from a Running BASIC Program, Example

```
        .
        .
        .
150 PRINT "#OUT=*PRINT*"  /* REASSIGN OUTPUT STREAM
155 REM THE *PRINT* RECEIPT IS PRINTED AT THE TERMINAL.
160 I=CMD("/LIST F1")  /* LIST SOME FILE
164 P'T "1*** RESULTS FOLLOW:"  /* TITLE WITH PAGE SKIP
165 P'T A,B,TAB(40);C,DD  /* PRINT SOME RESULTS
170 PRINT "#OUT=*SINK*"  /* RESET THE ASSIGNMENT
        .
        .
        .
```

Example 11.2 Printer Listings from a Running BASIC Program

A more complex example follows.

Input from MTS Files to a BASIC Program, Example
Output to MTS Files from a BASIC Program, Example

```
5 INPUT A,B,C  /* READ FROM TERMINAL
10 * WRITE A,B,C, INTO MTS FILE F1
15 P'T "#OUT=F1"
20 P'T A,B,C
30 * READ DATA PAIRS (ONE PER LINE) FROM MTS FILE F2
40 * AND PRINT NUMBERS AND PRODUCT INTO MTS FILE F3.
50 P'T "#OUT=F3":"#INP=F2"  /* : FORCES NEW OUTPUT LINE
60 INPUT A,B   /* READ FROM MTS FILE F2
70 PRINT A,B,A*B   /* WRITE INTO MTS FILE F3
80 GOTO 60
```

Example 11.3 MTS File I/O from a Running BASIC Program

In the above example, the program terminates when an end-of-file is encountered on reading MTS file F2. The #OUT assignment remains as last set by the program. The user may issue a #OUT command to reassign the output stream. The #INP assignment is changed to *MSOURCE* due to the end-of-file on F2. If BASIC is being run at a terminal (*MSOURCE*), the user will be prompted (via a colon prompting character) for further input to BASIC.

X.1.D I/O STREAM-CHANGING DURING INPUT DATA PROMPTING

The input and output streams may also be changed when a BASIC program is prompting the user for data input (regardless of the mode of the data being requested). The user simply types the appropriate BASIC pseudo-device command. It is processed, and if it does not change the input stream, the user will be reprompted for the data. If the input stream has been changed, the data for which the user was originally prompted will be obtained from the newly defined input stream.

In summary, BASIC pseudo-device commands may be given to BASIC from wherever BASIC (or a running program) reads input lines or by whatever user-accessible BASIC facility that causes printing of output lines (e.g., PRINT statements in a running program). The facility is restricted to the I/O programming language statements and commands which normally deal with terminal I/O (i.e., *not* BASIC file I/O). See Section XIII for a complete specification of BASIC pseudo-device commands.

X.2 BASIC LIBRARY FACILITY

Library Facility

The BASIC library is a collection of programs and data files (utility, demonstration, etc.) which are accessible to all users in the same manner as their own BASIC private files. These files are not accessible from MTS (they are not MTS *-files). The user must be using BASIC at the time references to the library are made. All program or file names in the library are two to seven characters in length, the first being an asterisk (*) library prefix (e.g., *SORT); hence, they may be called BASIC *-files. They are accessible from either BASIC command mode (e.g., /RUN *SORT) or from within a running BASIC program (e.g., CALL *SORT or CMD function reference). Library data files may be referenced from command mode by using the @D modifier or from a running BASIC program via the usual facilities such as READ FILE, WRITE FILE, BACKSPACE FILE, etc. For example, if *CDATA@D were in the library, then its name *CDATA (without the @D modifier) could be used in a FILE statement for reference by the file manipulation statements.

The user may obtain information about the contents of the library by issuing the BASIC command /LIBRARY. As is necessary, the user will interactively be led through categorical elimination to the set of programs or data files which are desired (if they exist). The library does contain the set of complete examples shown in Appendix O for demonstration purposes.

XI BASIC IN MTS BATCH

Batch Operation of BASIC

BASIC at the University of Michigan was developed mainly as a "conversational" program. Consequently, the facilities for error recovery in batch are almost non-existent. In fact, what happens on an error in "batch mode" is that the user is immediately returned to MTS, i.e., execution of BASIC terminates. Take note of the fact that *any* error, caused by a program *or* by an illegal command, will cause the system to terminate operation. The user may force BASIC out of "batch mode" while running in batch by issuing the $NOBATCH command. This is discouraged, however, since the results of an error are unpredictable.

The other major differences in running BASIC in batch are the default settings of various system parameters. In particular, the printing of prefixes is disabled, the user is not queried before execution of the EMPTY, FREE, or DESTROY commands (i.e., CONFIRM=OFF), and command and data lines are echoed to the printer (ECHO=ON and DATAECHO=ON). In addition, the type of output device will not be checked. It will be assumed to be a printer with a PN print train (DEVICE=PNPTR, LINELEN=131, BACKSPACE=OFF).

The user may negate any or all of the above behavior in batch by issuing the appropriate commands and/or specifying certain options in the PAR= field to the $RUN *BASIC command. Also, see Section III on "Entering and Leaving BASIC".

XII FORPRT - A SIMPLE TEXT-PROCESSOR FOR BASIC

FORPRT Text-Processor

One of the most important parts of any interactive program is the set of diagnostics (error messages) which aid the user in correctly communicating with it. The FORPRT (FORmatted PRinTout) processor in BASIC was written to improve the quality of error messages in BASIC in one particular area, the area of readability. This processor performs such functions as conversion to lowercase with specified capitalization, underlining of characters, selective carriage control, overstriking of characters, and centering of a line within the output line. These functions are performed selectively and in different ways depending upon the type of output device being currently used. For instance, with a Model 35 Teletype, lowercase conversion, backspacing, underlining, and overstriking cannot be done. With an IBM 2741 terminal, however, all of these functions can be performed.

FORPRT is in essence a simple one-line text processor. It is accessible to the BASIC user via the $TITLE command and the #FMT "pseudo-device command" (see Section XIII which follows). From within a running BASIC program, the $TITLE command can be issued via the CMD function or the #FMT pseudo-device command can be issued as a single line output string via a PRINT statement. Either command can be issued in command mode. Either command will cause the specified text to be passed to the FORPRT processor for format control interpretation and direction of the resultant print line or lines to the current BASIC output stream.

There are two basic ways in which a line is processed by this routine. One is to perform the functions which affect the line _as a whole_. These are specified at the beginning of the line in what will be called the _global control field_. This field is signaled by the occurrence of an at-sign (@) as the first character of the line. The "global control field" _must be terminated_ as well as initiated by an at-sign. The field, however, need not be present. The various functions indicated in this field are signaled by either a single letter or a letter followed by a number in parentheses. The letters and the functions they signal are detailed below. If a number is allowed after them, it will be indicated by (#) after the letter. All numbers must be positive integers. The legal range applicable to the various functions will be indicated below in parentheses following the given letter. The "global control field" will not be in the final form of the output line (it is removed). The ordering of function specifications within the "global control field" is irrelevant.

C

will center the entire line. This centering takes place with respect to the left margin (0 or as defined by the global tab function T) and the right boundary of the line. The right boundary is initialized according to the starting output device. It may be altered by the user via the /SET command through the LINELEN parameter.

L

will convert the entire line to lowercase, excluding the global control field, if applicable to the given output device. The user may override the initial assignment of the lowercase option with the $LC and $UC commands or with the /SET DEVICE=dev command.

O(#) (1≤#≤4)

will overstrike the entire message the number of times specified by #. This option will only be performed for those devices which have a backspace character or for a printer. The initial assignment of the backspace option may be overridden by the user via the SET BACKSPACE=ON (OFF) command. The number of overstrikes for a particular column may be less than asked for if local overstriking and/or underlining were specified for that column. In any case, the number of total overstrikes plus local underlining for a given column never exceeds 4. Local overstriking and underlining will be performed before this global overstriking. This overstriking also differs in that each character is overstruck with itself.

P

will cause a skip to the top of the next page for a printer and a skip of 10 lines for any other output device. In batch mode, the output device is assumed to be a printer. In conversational mode, printer will never be the initial assignment. The user may assign the SET command parameter DEVICE either the value PNPTR (without lowercase) or TNPTR (with lowercase), forcing the output device to be a printer.

Q

will capitalize the first letter of every "word" in the line. A "word" in this case is any letter or sequence of letters not preceded by a letter or a digit. This function is not performed unless the L global function (lowercase conversion) was performed.

S(#) (0≤#)

causes the specified number of lines to be skipped. This function is performed after the P function (skip to the top of the page). Any number of lines up to the maximum integer which can be converted on the computer (2,147,483,647) may be skipped.

T(#) (0≤#≤Ml-l)

causes a left margin to be created the number of spaces specified from the beginning of the output line. In the expression Ml-l, Ml refers to the maximum possible line length (assigned initially according to the output device and changeable via the command /SET LINELEN=l), and l refers to the length of the current line after removal of the global control field. This control effectively pushes the current line over the specified number of spaces to the right. If the number specified exceeds Ml-l, it is set to it.

U

causes every non-blank character in the output line to be underlined. If backspacing is allowed for the given output device or if the device is a printer, underlining will be done using the underline character (_) on the same line. If it is not allowed, underlining will be done with the minus or hyphen character (-) on the <u>next</u> line. This function is always performed!

Z

inhibits "local processing" of the line and "global control" functions L and Q.

The other major way in which this routine processes a line is to perform functions on the individual characters to be printed.

These include such things as capitalizing a given letter, capitalizing a number of consecutive letters, underlining a number of consecutive characters, overstriking a given character with another, and local tabbing. These various functions are detailed below. Their presence in the line is signaled by _the at-sign (@) which must precede them_. After the function is performed, all characters defining it (this does _not_ mean the character(s) being operated on) are removed from the line. If an unknown function is encountered, the character defining it and the at-sign are left in the line. The function scan is performed from left to right. It is performed only once. The rules for interpreting the prototypes below are the same as detailed for the "global control function" prototypes.

@C(#) (0≤#≤*)

capitalizes the previous number of characters specified. Non-letters are, of course, unchanged. In the expression above, * refers to the current position in the line (number of characters to the left of the at-sign).

@K

capitalizes the previous character.

@L

starts or stops capitalizing of every character from the current position to the next occurrence of @L. This function is ignored (and deleted) if the "global function" L is not specified.

@O

causes the previous character to be overstruck with the character immediately following the O. The overstriking character is effectively removed from the line as an independent character. If backspacing is not allowed _and_ the device is not a printer, this function is ignored (and deleted, including the overstrike character). The overstrike character will have had no case conversion done on it, if it is used.

@T(#) (0≤#≤132)

will cause the remainder of the line to be moved to the position specified relative to the beginning of the line before centering and global tabbing have been performed. If the specified number is greater than 132, it is set to 132. Blanks are used to fill any vacancy created when moving to the right. Characters are deleted when moving to the left, but overstrikes and local underlining done to them are not. The function scan continues with the character immediately to the right of the right parenthesis.

@U(#) (0≤#≤*)

underlines the previous number of characters specified. In the expression above, * refers to the current position in the line (number of characters to the left of the at-sign). This function is not performed if backspacing is off _and_ the device is not a printer.

@Z

performs the same function as @O except that the overstrike character is first converted to lowercase.

@@

puts a single at-sign (@) in the line in place of @@. The function scan is continued just to the right of this at-sign.

The following sample program shows how the FORPRT routine can be used from a BASIC program:

```
10 INPUT AA           /* READ IN A LINE
20 AA="$TI@NE """+AA /* CONSTRUCT THE COMMAND LINE
30 AA=AA+""""
40 I=CMD(AA)          /* PASS COMMAND TO CLI
50 IF I¬=0 THEN: PRINT "BASIC MESSED UP."
60 GO TO 10           /* GET ANOTHER LINE
```

Example 12. Sample Program to Use FORPRT

Notice the use of abbreviation for the $TITLE command and the NOECHO command modifier in line 20. Line 50 is really unnecessary since the $TITLE command should never generate an error. Therefore, the return code from the CMD built-in function would always be zero, for this example. Finally, it should be noted that input lines for the statement at line 10 must be begun and ended with quotes (") if they contain blanks or commas. A sample set of input lines to the program and the resultant output lines follows.

INPUT LINE	OUTPUT LINE
"TEST"	TEST
"@L@TEST"	test
"@L@T@KHE TEST@C(4)."	The TEST.
"@L@T@KHE @LTEST@L."	The TEST.
"@LU@T@KHE TEST@C(4)."	The TEST.
"@L@T@KHE TEST@C(4)@U(4)."	The TEST.
"THE @LTEST."	THE test.
"@L@T@KHE @T(10)TEST."	The test.
"@L@A@KN @@-SIGN."	An @-sign.
"@L@A@KN O@O/VERSTRIKE."	An overstrike.[1]
"@O(3)@B@KOLD."	Bold.[2]
"@LQ@THE TEST."	The Test.
"@ZL@T@KHE TEST@C(4)."	T@KHE TEST@C(4).
"@T(5)L@T@KEST."	Test.
"@C@TEST"	TEST[3]

The input lines could be given in command mode via the $TITLE command as in:

```
     $TITLE "@L@T@KHE TEST@C(4)."
```

or be stored in an MTS file (e.g., FIL1) and be accessed via the #FMT pseudo-device command to generate output for all lines referenced, as in the following example.

```
     #FMT=FIL1(5,10)
```

----

[1]where the / would be printed superimposed on the o.

[2]where the line would be overstruck three times (printed four times).

[3]where the text is centered using the current output line length.

XIII BASIC PSEUDO-DEVICE COMMANDS

Pseudo-Device Commands
Commands (Pseudo-Device)

The terminal user of BASIC may issue commands to the input/output service routine. This routine is the part of BASIC which interfaces with the system of which BASIC is a subsystem (MTS). It monitors input and output lines (all I/O except BASIC file I/O) and processes attentions. Commands to this part of BASIC should not be confused with regular BASIC commands or data lines. Since the "input/output service routine" monitors all input and output lines, it can intercept commands intended for it at any time. Such commands are called "pseudo-device commands" because they deal with functions related to the device(s) from which BASIC obtains input lines and to which output lines are sent.

These commands perform such functions as redefining the source for all input to BASIC, redefining the sink (where it goes) for all BASIC output (output here referring to what BASIC would print on the terminal), enabling or disabling the printing of prefixes, or defining an output translate table. All commands are of the form #xxx=parm, where xxx is a three-letter mnemonic defining the command and parm is the command parameter. Any command which cannot be recognized or for which the parameter is undecipherable will be passed on unchanged to the component of BASIC requesting input. These commands may be issued whenever the system is requesting input. After the command is processed, the user will again be prompted for the input line expected by BASIC. It is permissible to enter any number of device commands one after the other, before finally entering the requested input line.

The six "pseudo-device commands" currently implemented are detailed below. For prototypes, the material within braces ({}) in lowercase refers to a generic type to be replaced by an item supplied by the user. Brackets ([]) are used to indicate alternatives, separated by vertical bars (|). Material in upper-case is to be typed as is.

#FMT={fdname}

reads input lines from the MTS file or device specified by fdname and sends them directly to the FORPRT routine which processes them and then outputs them. Input will be continually read until either an end-of-file is encountered or an attention is given. For details on what the FORPRT routine does, see Section XII entitled "FORPRT - A Simple Text Processor for BASIC".

#INP={fdname}

redefines the source of input to BASIC to be the MTS file or device specified by fdname. After execution of this command, _all_ input to BASIC will be from the given source until another #INP command is encountered, an attention is given, or an end-of-file is encountered. After either an attention or an end-of-file, the source will be switched to the MTS pseudo-device *MSOURCE*.

#OFF=[{n}|OFF]

defines the offset for output lines to be n or disables any offset currently in effect (turns if OFF). Offset here refers to some number of blanks added to the front of any output line before it is printed. This offset is applied after the prefix, if any, is printed. This feature is intended for use mainly in batch mode. Successive #OFF commands override previous ones.

#OUT={fdname}

redefines the sink (where it goes) for output from BASIC to be the MTS file or device specified by fdname. After execution of this command, _all_ output (except BASIC file I/O) from BASIC will be to the given sink until another #OUT command is encountered.

#PFX=[ON|OFF]

enables (turns ON) or disables (turns OFF) the printing of _all_ prefixes by BASIC.

#TRT=[{fdname}|OFF]

defines the translate table for BASIC output (not BASIC file output) to be the one contained in the MTS file or device specified by fdname, or disables the translation of output lines (turns it OFF). The translate table is expected to be a regular 256-character (EBCDIC) translate table as defined for use with the computer instruction TR, which is used to translate the output lines. For a description of how this instruction works, the user should consult the IBM System/370 "Principles of Operation" manual, GA22-7000. The 256 characters defining the translate table may be on any number of lines from the given file or device. Extra characters are ignored. Extra lines are not read. The feature once enabled continues until explicitly disabled. The table may be redefined at any time.

Some complete in-context examples of pseudo-device command usage are given in Section X.1.

XIV REFERENCES

1. Michigan Terminal System, Volume 1, MTS and the Computing Center, 3rd ed., University of Michigan Computing Center, Ann Arbor. (Especially useful sections on batch and conversational (terminal) usage.)

2. Basic Interim Improvement by Call-A-Computer (BIICAC) Reference Manual, Pillsbury-Occidental Co., 1968 and 1969. (Basis for U or M BASIC implementation; not recommended.)

3. Forsythe, A. I., et al., Computer Science: Basic Language Programming, John Wiley and Sons, Inc., 1970. (Fundamental algorithmic programming approach.)

4. Kemeny, John G. and Kurtz, Thomas E., Basic Programming, 2nd ed., John Wiley and Sons, Inc., 1971. (Highly recommended for numerous excellent application examples.)

5. Spencer, Donald D., A Guide to BASIC Programming: A Time Sharing Language, Addison-Wesley, 1970. (Highly recommended; comprehensive coverage with excellent examples and problems.)

6. Boettner, Donald W. and Alexander, Michael T., MTS - Michigan Terminal System, SIGOPS Operating System Review, Volume 4, Number 4, December 1970, pp. 6-19.

XV GLOSSARY OF COMPUTING TERMS

Glossary of Computing Terms

Absolute Value The quantitative value of a number, exclusive of its sign.

Allocation The process of reserving blocks of computer storage for specific data. (In BASIC, the DIM statement performs this function.)

Alphanumeric Consisting of alphabetic and numeric characters combined.

Application The problem for which a computer operation is designed.

Array An ordered arrangement of items, such as a list or table. (In BASIC, it is possible to have arrays of either strings or numbers.)

Batch Mode The user's job has been turned in to the Computing Center input window or a remote batch station for processing. The deck is read into the computer and saved on a special disk for later execution, and the deck and a job receipt are returned to the user. Batch mode then refers to the state of programs run under such a job. In this mode the user cannot be informed of errors and prompted for corrections or new directions. See Conversational Mode.

Binary Operator An operator which requires two operands, e.g., A+B.

Character Any single letter of the alphabet, numeric digit, or special symbol.

Code A set of symbols and rules for representing information.

Coding The writing of computer instructions in a programming language for acceptance by a computer.

Compiler A built-in computer program that translates symbolic-language programs (such as those written in BASIC) into machine-language programs.

Concatenation The operation of adjoining two strings to form a single composite result.

Conditional Transfer A variation of program control based on some condition. (The IF-THEN statement in BASIC is an example.)

Constant A quantity that does not change during execution of a program.

Control Statements Program elements that direct the flow of the program, e.g., GOTO and IF-THEN.

Conversational Mode Running of a user's job in a condition where the computer can report directly to him and ask him for direction. In this mode, errors detected by the computer may be reported to the user, and he can be prompted for corrections or new directions. This is the mode a user is in when running from a terminal. (Conversational mode is sometimes called interactive mode.)

Data Cell A storage device in which information is recorded on magnetic strips (sheets) which reside in a container called a cell. This device is slower than a disk since the strips are moved in and out of the cell for reading and writing information.

Debugging The process of eliminating errors from a computer program (sometimes called program checkout).

Diagnostics Messages printed on the terminal by BASIC indicating the errors detected.

Disk A storage device in which information is recorded magnetically on surfaces of rotating disks.

Edit To rearrange data or information.

Exponentiation Representation of a number in terms of a power of a base.

Hardware The physical equipment of a computer system.

Hexadecimal (Hex) A number system in which the 16 digits are 0 through 9, A, B, C, D, E, and F in numerically ascending order. For example, 2B in hex is 2*16+B or 2*16+11 or 43 decimal. Characters on the Computing Center's computer are representable in hex. For example, C1 hex is the letter A.

Initialize To preset variables, counters, etc., to proper initial values before beginning a calculation.

Input To enter information into the computer. Also the information so entered.

I/O(Input/Output) The process of getting information into and out of the computer. Also the information so processed.

Instruction A symbol or group of symbols recognized by the computer as an order to perform an operation.

Integer A whole number that contains no fractional part or decimal point.

Line Number The number that identifies and is associated with every statement in a BASIC program. The legal range extends from 0 to 9999 inclusive.

Loop A sequence of statements repeated a controlled number of times within a program.

Machine Language The coding system in which instructions and data are represented internally within a computer.

Matrix A rectangular (two-dimensional) array of elements (also commonly called a table). Its name consists of a letter or a letter followed by one or two digits.

Memory The storage area of a computer which consists of cells (called _cores_) to contain instructions and data.

Nesting A method of grouping items completely within another group of items, e.g., two nested FOR-NEXT loops.

Operator A symbol that specifies an action.

Output To transfer information from the computer to some usable form, such as a printout on a teletypewriter. Also the information so produced.

Power A symbolic representation of the number of times a number is multiplied by itself. The process is called exponentiation.

Program An ordered list of statements that directs the computer to perform certain operations in a specified sequence to solve a problem.

Relational Operator A symbol that specifies a comparative relationship between two items of data, e.g., less than, equal to, etc.

Routine A sequence of statements or instructions that carry out a well-defined function.

Simple Variable A letter or a letter followed by one or two digits (also called an unsubscripted variable), e.g., A or A12.

Source Language Any symbolic language in which a program is written.

Source Program A program coded in any language other than machine language.

String In BASIC, a sequence of 0 to 127 characters.

String Variable A symbol in BASIC that may take on the assignment of a string. It is denoted by a pair of identical letters or a pair of identical letters followed by a digit, e.g., AA or BB1.

Subroutine A group of statements to perform a particular operation that may be used repeatedly within a program. A subroutine is not itself a complete program. In BASIC, it terminates its execution with a RETURN statement that transfers control back to the program which invoked it.

Subscript An expression that represents the position of an element in a list or table.

Subscripted Variable A symbol whose value can change. It is denoted by a letter followed by a quantity in parentheses, e.g., CC(2), A(23) or W(2,4). This is sometimes called an array reference since the symbol is a way of denoting a position in a table.

Time-sharing The simultaneous use of a computer by several operators at several terminals.

Unary Operator An operator requiring only one operand, e.g., -A.

Unconditional Transfer A shift of program control to a specified location in a program. In BASIC, the GOTO statement is an example of an unconditional transfer.

Variable A symbol whose numeric value can change.

Vector A linear (one-dimensional) array or list of cells.

XVI APPENDICES

The following appendices contain useful summaries of BASIC facilities and usage protocol.

Appendix A - Lexicographic Character-Ordering

Character-Ordering (Lexicographic)

The table is ordered from low to high moving down the column.
EBCDIC - Extended Binary-Coded Decimal Interchange Code
ASCII - American Standard Code for Information Interchange
The ASCII equivalents here are 7 bit octal (no parity).

EBCDIC	ASCII	Hexadecimal	EBCDIC	ASCII	Hexadecimal	EBCDIC	ASCII	Hexadecimal	EBCDIC	ASCII	Hexadecimal	EBCDIC	ASCII	Hexadecimal
	040	40	_	137	6D	j	152	91	B	102	C2	T	124	E3
¢	000	4A	>	076	6E	k	153	92	C	103	C3	U	125	E4
.	056	4B	?	077	6F	l	154	93	D	104	C4	V	126	E5
<	074	4C	:	072	7A	m	155	94	E	105	C5	W	127	E6
(	050	4D	#	043	7B	n	156	95	F	106	C6	X	130	E7
+	053	4E	@	100	7C	o	157	96	G	107	C7	Y	131	E8
|	174	4F	'	047	7D	p	160	97	H	110	C8	Z	132	E9
&	046	50	=	075	7E	q	161	98	I	111	C9	0	060	F0
!	041	5A	"	042	7F	r	162	99	J	112	D1	1	061	F1
$	044	5B	a	141	81	s	163	A2	K	113	D2	2	062	F2
*	052	5C	b	142	82	t	164	A3	L	114	D3	3	063	F3
)	051	5D	c	143	83	u	165	A4	M	115	D4	4	064	F4
;	073	5E	d	144	84	v	166	A5	N	116	D5	5	065	F5
¬	176	5F	e	145	85	w	167	A6	O	117	D6	6	066	F6
-	055	60	f	146	86	x	170	A7	P	120	D7	7	067	F7
/	057	61	g	147	87	y	171	A8	Q	121	D8	8	070	F8
,	054	6B	h	150	88	z	172	A9	R	122	D9	9	071	F9
%	045	6C	i	151	89	A	101	C1	S	123	E2			

Appendix B - BASIC Built-In Functions (Table)
Functions (Built-In, Table of)

For ARGuments:
N =: numeric expression.
Nv =: simple numeric variable.
S =: string variable or constant.
	For return VALUEs:
	N =: number.
	S =: string.

NAME	ARG	VALUE	MEANING
ABS	N	N	Absolute value of arg
ATN	N	N	Arctangent of arg (value in radians)
BFR	S	N	Arg becomes "scan string" (value is 0)
CLK		N	Time of day in hours past midnight
CLS		S	Time of day (HH:MM.SS)
CMD	S	N	Send arg to BASIC CLI (value is return code)
COS	N	N	Cosine of arg (arg in radians)
CRP	N	N	Is user conversational? (0=YES;
DAT		S	Date (MM-DD-YY)
EDT	S	S	Replace in arg the first occurrence of the "scan string" with the "replace string" (result is value, arg not changed)
EXP	N	N	e raised to the power arg
HEX	S	S	Internal hex equivalent of given characters
INP	Nv	N	Input a number into arg (value is number)
INT	N	N	Integer part of arg (arg truncated)
ISN	S	N	Is arg a number? (0 for YES; 1 for NO)
LEN	S	N	Length of arg ($0 \le value \le 127$)
LOG	N	N	Natural logarithm of arg ($arg \ge 0$)

For ARGuments:	For return VALUEs:
N =: numeric expression.	N =: number.
Nv =: simple numeric variable.	S =: string.
S =: string variable or constant.	

NAME	ARG	VALUE	MEANING
NTS	N	S	Convert arg to a string
OUT	N	N	Print arg (value is arg)
RND	Nv	N	Generate a random number between 0 and 1
RPB	S	N	Arg becomes "replace string" (value is 0)
SCN	S	N	Find first occurrence of "scan string" (value is number of first character)
SGN	N	N	+1 if arg>0;-1 if arg<0;0 if arg=0
SIN	N	N	Sine of arg (arg in radians)
SKP	N	N	Skip over N data items on input
SQR	N	N	Square root of arg (arg≥0)
STN	S	N	Convert arg to number (0 for non-numeric)
TAN	N	N	Tangent of arg (arg in radians)
TIM		N	Elapsed time in seconds from signon to BASIC
UID		S	User’s signon id
XTS	S	S	Character equivalent of given hex digits

Appendix C - BASIC Statement Abbreviations

Abbreviations (Statement, List of)
Statement Abbreviations List

Statement	Abbreviations	
BACKSPACE	B'E	
BACKSPACEFILE	BACKSPACE#	B'E#
CALL	C'L	
CHAIN	C'N	
DATA		
DEF		
DIMENSION	DIM	
END		
FILE	F'E	
FOR		
GOSUB	G'B	
GOTO	GO	
IF		
IFENDFILE	I'E	
INPUT	I'T	
LET		
LINPUT	L'T	
MATINPUT	M'I	
MATLET	MAT	
MATPRINT	M'P	
MATREAD	M'D	
MATREADFILE	MATREAD#	M'D#
MATWRITE	M'E	
MATWRITEFILE	MATWRITE#	M'E#
NEXT	N'T	
PAUSE	P'E	
PRINT	P'T	
READ	R'D	
READFILE	READ#	R'D#
REMARK	REM	
RESTORE	R'E	
RESTOREFILE	RESTORE#	R'E#
RETURN	R'N	
STOP	S'P	
WRITE	W'E	
WRITEFILE	WRITE#	W'E#

In addition, the symbol asterisk (*) is recognized as a synonym for REMARK. A few other synonyms have been defined. ABORT will be recognized for STOP, and WAIT for PAUSE.

Appendix D - BASIC Command Prototypes
Command Prototypes and Abbreviations (Table of)
Abbreviations (Command, List of)

$^{+}$ Legal only in debug mode. ° Illegal in debug mode. _ Shows minimum abbreviation.		Applicable Modifiers					
Commands	Parameters	ALL	CON	ECH	ERR	TER	VER
ALTER	['str1'str2'] [l#] [file]	x		x	x		x
BLIST^{+}				x	x		
BREAKLIST^{+}				x	x		
BREAKPOINT^{+}	l# [, l#] •••			x	x		
BYE	[PAR=STAT]			x			
CATALOG		x		x	x		
CLEAN^{+}				x	x		
COMMENT	[text]			x			
COMPILE°	[file] [PAR=parm •••]			x	x		
COMPLAIN	[text]			x			
CONTINUE^{+}				x	x		
COPY	[file1 [TO] [file2]]			x	x		
DEBUG°	[file] [PAR=parm •••]			x	x		
DESTROY	file	x	x	x	x	x	
DISPLAY^{+}	var [, var] •••			x	x		
EDIT	['str1'str2'] [l# [file]]	x		x	x		x
EMPTY	file		x	x	x	x	
EXECUTE°	[file] [PAR=DEBUG]			x	x		
FILESNIFF	[file]			x	x		

$^{+}$ Legal only in debug mode. ° Illegal in debug mode. _ Shows minimum abbreviation.		Applicable Modifiers					
Commands	Parameters	ALL	CON	ECH	ERR	TER	VER
FREE	file	x	x	x	x	x	
GET	file			x	x		
GOODBYE	[PAR=STAT]			x			
GOTO^{+}	l#			x	x		
HELP	[name]			x			
INCLUDE	[fdname]			x			
LIBRARY				x			
LINERANGE	[file]			x	x		
LINE#	[l#]			x	x		x
LIST	[file]			x	x		
LOGOFF	[parm] [PAR=STAT]			x			
L#	[l#]			x	x		x
MODIFY^{+}	var value [var value] •••			x	x		
MTS				x			
MTSCMD	[MTScommand			x			
NUMBER	[start [, incr]]			x	x		
OBJSCAN	[file]			x	x		
OPEN	file			x	x		
PERMCATALOG				x			

$^{+}$ Legal only in debug mode. ° Illegal in debug mode. _ Shows minimum abbreviation.		Applicable Modifiers					
Commands	Parameters	ALL	CON	ECH	ERR	TER	VER
PERMIT	file	x		x	x	x	
PREFIX	prefix			x	x		
PRINT	keywrd [, keywrd] •••			x	x		
QUIT	[PAR=STAT]			X			
RELEASE				x			
RENAME	file [TO] filenm			x	x		
RENUMBER	filenm(s,l) or s,l			x	x	x	
RESET°				x	x		
RESTORE^{+}	l# [, l#] •••			x	x		
RUN°	[file] [PAR=parm •••]			x	x		
SAVE	[file]	x		x	x	x	
SCAN	['str'] [file]	x		x	x		x
SET	keywd=val [keywd=val] •••			x	x		
SHARING	file usid	x		x	x	x	
SIGNOFF	[PAR=STAT]			x			
SINGLESTEP^{+}	[switch]			x	x		
START^{+}				x	x		
STATUS				x			
STOP^{+}				x	x		
SYSTEM				x			

$^{+}$ Legal only in debug mode. ° Illegal in debug mode. _ Shows minimum abbreviation.		Applicable Modifiers					
Commands	Parameters	ALL	CON	ECH	ERR	TER	VER
TRACE	keywrd [, keywrd] •••			x	x		
TUTOR				x			
/UNNUMBER				x			
UNPERMIT	file	x		x	x	x	
WHAT?$^{+}$				x	x		
WHERE^{+}				x	x		

Appendix E - Carriage Control

Carriage Control

Char-acter	Effect Before Printing	Exceptions		
		Printer	Terminal: Hard-Wired Controller	Terminal: Data Con-centrator
blank	single space			
0	double space			
-	triple space			
+	overprint previous line--print without spacing first		ss[1]	undef[2]
&	suppress carriage return after printing	undef		
9	single space and suppress overflow[3]		ss	ss
1	skip to top of next page[4]		skip 6[5]	skip 6
2	skip to next 1/2 page[6]		skip 6	undef
4	skip to next 1/4 page[6]		skip 6	undef
6	skip to next 1/6 page[6]		skip 6	undef
8	same as 6		skip 6	undef
;	skip to top of next physical page (at perforation)		undef	undef
<	skip to bottom of physical page (at perforation)		undef	undef

[1]ss = single space.
[2]undef = undefined, single spacing; character is printed.
[3]Printer normally skips first and last 3 lines of a page.
[4]"Top" is physically 3 lines down from perforation.
[5]skip 6 lines.
[6]The logical page is divided into halves, quarters, sixths.

Appendix F - How to Use a Teletype

Teletype (How to Use)
Terminal (Teletype, How to Use)

1. Check the "duplex" switch. It should be set on half duplex.

2. Depress the "origin" button.

3. Dial 763-0300 (only 3-0300 when calling from within the U of M Centrex telephone system).

4. Adjust the speaker control: if there is no answer or a busy signal is heard, hang up by pressing the "clear" button and try again later.

5. If your call is answered, wait until the MTS acknowledge has been typed. Then, when a # appears, you may sign on.

NOTE: ALL INPUT LINES YOU TYPE MUST BE TERMINATED WITH AN END-OF-LINE CHARACTER (see 10a).

6. You sign on by typing: `$SIGNON USID` where USID is your user id.

7. You are then asked to give your password. At this point you should flip the "duplex" switch from half to full duplex. Then enter your password and flip the switch back. This prevents your password from printing (see 12 below).

8. Once signed on, you can enter any MTS command when a # appears prompting you for a response, e.g., $RUN *BASIC.

9. If you get in trouble, you can do several things:
   a. press the "clear" button which hangs you up.
   b. press the "break" button which returns control to you at the teletype. In this case, you must press the "break release" button to unlock the keyboard.

10. A list of the "control" buttons follows (where the c following the character means that you also press the control key when you type the character):

    a. Q(c) or S(c) : to send a line in.

    b. A(c) : to delete a character.

    c. N(c) : to delete a line.

d. C(c) : end of file.

11. You can sign off in the normal way whenever prompted with the # by typing: $SIG or ($SIG $ for a short signoff).

12. If you wish to change your password, issue the MTS command $SET PW=XXXXXXXXXXXX where XXXXXXXXXXXX is the password you want to use the next time you sign on. It can be anywhere from 1 to 12 characters.

Appendix G - BASIC Implementation Differences

"Standard" BASIC is defined as the original BASIC System implemented at Dartmouth College. U or M BASIC differs from standard BASIC in that ** is used instead of an up arrow for exponentiation , unary minus has precedence just above rather than just below exponentiation, and the line continuation character is a minus sign (the MTS default - may be changed by the user) as the last character of the line to be continued. The RND function is also invoked in a different manner.

Other versions of BASIC have a few common facilities that U of M BASIC does not have, namely multiple statements per line and a multiple assignment statement. However, U of M BASIC has all the important statements with many extensions. See Appendix K for a statement summary.

Appendix H - Useful Constants (Double-Precision)

Constants (Double-Precision, List of e, pi, etc.)
e (Double-Precision List of Powers, Logs)
pi (Double-Precision List of Powers, Logs)
Radians to Degrees (Conversion Constant)
Degrees to Radians (Conversion Constant)
Logarithmic Conversion (Base 10 from Base e)

e	2.7182 81828 45904 524
e^2	7.3890 56098 93065 023
e^{-1}	0.3678 79441 17144 2322
e^{-2}	0.1353 35283 23661 2692
pi	3.1415 92653 58979 324
pi^2	9.8696 04401 08935 862
pi^{-1}	0.3183 09886 18379 0672
pi^{-2}	0.1013 21183 64233 7771
log10e	0.4342 94481 90235 1828
log10pi	0.4971 49872 69413 3854
1 radian	57.295 77951 30823 209°
1°	0.0174 53292 51994 32958 radians

Appendix I - Detection of System Errors in BASIC

System Errors in BASIC
Errors (in the BASIC System)

If during the execution of the BASIC system a program interrupt occurs, the BASIC system will enter in its system error log the user id, time, date, contents of the PSW and general registers, the location of the interrupt, the last BASIC command issued (if any), and , if a BASIC program was running, the name of the program and the line number of the statement being executed when the interrupt occurred. The user is then notified that the system error has occurred, is requested to report the error to the BASIC system staff, and, if a program was running at the time of error, is requested to permit the files involved.

Appendix J - BASIC Command Summary
Command Summary (Name-Function, List of)

The minimum number of characters needed to specify the command are underlined.

COMMAND	FUNCTION
ALTER	String replacement within a line (editing).
BLIST	List breakpoints (debug mode).
BREAKLIST	List breakpoints (debug mode).
BREAKPOINT	Set breakpoints (debug mode).
BYE	Sign off BASIC and return to MTS.
CATALOG	List catalog of "temporary" file copies.
CLEAN	Restore all breakpoints (debug mode).
COMMENT	Issue a comment.
COMPILE	Compile a source program.
COMPLAIN	Give feedback about BASIC to the staff.
CONTINUE	Continue the program (debug mode).
COPY	Copy one file to another.
DEBUG	Enter interactive debugging at program start.
DESTROY	Destroy permanent and temporary file copy.
DISPLAY	Display contents of variables (debug mode).
EDIT	String replacement within a line (editing).
EMPTY	Empty a "temporary" file copy.
EXECUTE	Execute the program's O file.

COMMAND	FUNCTION
FILESNIFF	Obtain file statistics.
FREE	Destroy only the "temporary" file copy.
GET	Open a file and make it the "active file".
GOODBYE	Sign off BASIC and return to MTS.
GOTO	Transfer program control (debug mode).
HELP	Obtain information about BASIC.
INCLUDE	Input an MTS file to BASIC.
LIBRARY	Obtain BASIC Library information.
LINERANGE	Obtain line range of a file.
LINE#	Set "line" and "scan" pointers (editing).
LIST	List contents of a file.
LOGOFF	Sign off both BASIC and MTS.
L#	Abbreviation for LINE#.
MODIFY	Modify values of variables (debug mode).
MTS	Pause in BASIC and return to MTS.
MTSCMD	Issue an MTS command and continue.
NUMBER	Enter "numbering mode".
OBJSCAN	Dump contents of an object file.

COMMAND	FUNCTION
OPEN	Open a file and make it the "active file".
PERMCATALOG	List catalog of permanent BASIC files.
PERMIT	Permit access to a file to other users.
PREFIX	Change input prefix.
PRINT	Selectively print BASIC statistics.
QUIT	Sign off BASIC and return to MTS.
RELEASE	Deactivate the "active file".
RENAME	Rename a file.
RENUMBER	Renumber a BASIC file.
RESET	Reset BASIC system parameters.
RESTORE	Selectively restore breakpoints (debug mode).
RUN	Run a program.
SAVE	Save a permanent file copy.
SCAN	Search file for substring (editing).
SET	Set system parameters.
SHARING	Share another user’s file.
SIGNOFF	Signoff BASIC and return to MTS.
SINGLESTEP	Slow the program down.

COMMAND	FUNCTION
START	Start a program from its beginning (debug mode).
STATUS	Print all the BASIC statistics.
STOP	Stop program running (debug mode).
SYSTEM	Pause in BASIC and return to MTS.
TRACE	Trace program action and report.
TUTOR	Enter the BASIC tutorial system.
/UNNUMBER	Leave "numbering mode".
UNPERMIT	Unpermit a file.
WHAT?	Retrieve an error message (terse mode).
WHERE?	Retrieve position in program (debug mode).

Appendix K - BASIC Statement Summary
Statement Summary (Name-Function, List of)

The following summary is divided into the specific subsections of the language. The statement or operation is followed by an explanation of its function and examples.

General Computation	
LET	Computes and assigns value. Ex: `Y(I)=A*X(I)+4`
INPUT	Read data from terminal. Ex: `INPUT A,B(I),CC`
PRINT	Print data on terminal. Ex: `PRINT A,B,"Y =";A*B-3`
GOTO	Transfers control. Ex: `GOTO 10` `GOTO (10,20),I`
IF	Compare numeric data. Ex: `IF A > B THEN 100` `IF E*F <> 0 THEN:  PRINT X`
FOR	Sets up and operates loop. Ex: `FOR I=1 TO M STEP 2`
NEXT	Closes the loop. Ex: `NEXT I`
PAUSE	Pause in the program. Ex: `PAUSE THAT REFRESHES`
STOP	Stop a program. Ex: `STOP THIS MESS`
REMARK	Document the program. Ex: `REMARK  READ IN THE X,Y PAIRS`
*	Document the program. Ex: `*  THIS SECTION READS DATA`
/*	Document the program. Ex: `INPUT A,B   /* DATA INPUT`

Matrix or Vector Operations (Arrays)	
DIMENSION	Declare array sizes. Ex: DIM A(5),B(2,3),C(0,5)
MAT INPUT	Read entire matrices from terminal. Ex: MAT INPUT A,B(3,4)
MAT PRINT	Print entire matrices on terminal. Ex: MAT PRINT A,B;C
MAT +	Addition. Ex: MAT LET D=E+F
MAT -	Subtraction. Ex: MAT D=E-F
MAT *	Multiplication. Ex: MAT D=E*G
MAT /	Division. Ex: MAT E=W/Z
MAT Scalar *	Scalar multiplication. Ex: MAT E=(3)*F
MAT =	Assignment. Ex: MAT E=F
MAT (RDM)	Redimensioning. Ex: MAT A=RDM(4,5)
MAT (CON)	Generate matrix of ones. Ex: MAT A=CON(3,2)
MAT (IDN)	Generate identity matrix. Ex: MAT A=IDN(4,4)
MAT (ZER)	Generate matrix of zeroes. Ex: MAT A=ZER(3,2)
MAT (TRN)	Transpose. Ex: MAT A=TRN(B)
MAT (INV)	Inversion. Ex: MAT A=INV(B)

String Operations	
DIMENSION	Declare size of vector of strings. Ex: DIM AA(5)
LET	Assignment. Ex: LET AA(1)="STRING" SS=TT
+	Concatenation. Ex: AA(1)=BB+"END""
*	Substring extraction. Ex: AA=BB*(4) AA(3)=(5)BB
NTS	Number to string conversion. See also STN and ISN built-in functions. Ex: AA(4)=NTS(X)
HEX	Hexadecimal string conversion. Ex: AA=HEX("C1F1")
XTS	Internal hex to string conversion. Ex: SS=XTS(AA) See AA in HEX above.
CLS	Time of day. Ex: AA(1)=CLS()
DAT	Date. Ex: AA(2)=DAT()
UID	User id. Ex: AA(3)=UID()
EDT	String editing. See BFR, RPB, and SCN built-in functions. Ex: SS(1)=EDT(XX)
IF	String comparisons. Ex: IF AA = "HELLO" THEN 100 IF AA <> BB THEN (10,20),I
LINPUT	Read non-quoted strings from terminal. Ex: LINPUT AA,BB(3)

File Operations (Line-Oriented Stream I/O in D-Files)

DATA	Define program’s data file. Ex: DATA 1,2,4.5,"STRING"
READ	Read from data file. Ex: READ A,B,C(I),SS
WRITE	Write into data file. Ex: WRITE A,B,C(I),SS
MAT READ	Read entire matrices from data file. Ex: MAT READ A,B(4,2)
MAT WRITE	Write entire matrices into data file. Ex: MAT WRITE A,B
FILE	Define program’s indexable file list. Ex: FILE FONE,FTWO,F3
READ FILE	File list read file. Ex: READ FILE 2,X,Y(I)
WRITE FILE	File list write file. Ex: WRITE FILE 3,X,Y(I)
MAT READ FILE	File list entire matrix read. Ex: MAT READ FILE 1,A,B(4,2)
MAT WRITE FILE	File list entire matrix write. Ex: MAT WRITE FILE 3,A,B
IF ENDFILE	End of file test. Ex: IF ENDFILE 2 THEN 100
RESTORE	Restore program’s data file. Ex: RESTORE
BACKSPACE	Backspace programs’s data file one item. Ex: BACKSPACE
RESTORE FILE	File list restore. Ex: RESTORE FILE 2
BACKSPACE FILE	File list backspace of one item. Ex: BACKSPACE FILE 3

Procedures (Subroutines and Functions)	
Built-In Functions	See Appendix B. Ex: Y=SQR(X)
DEF	Define one-line function. Ex: DEF FNA(X)=P*X+Q and invokes it by Y=FNA(7.5)
GOSUB	Transfer to internal subroutine. Ex: GOSUB 100
RETURN	Return from internal subroutine. Ex: RETURN
CALL	Invoke external procedure. Ex: CALL PROG2
CHAIN	Relinquish control to external procedure. Ex: CHAIN PROG3,20
END	Return from external procedure. Ex: END
Command Function	
CMD	The CMD function issues commands to BASIC. Ex: I=CMD("/EMPTY@NC FILE@D")

Appendix L - What Is MTS?

MTS (Condensed Summary of)

MTS, the Michigan Terminal System, is a terminal-oriented time-sharing system that offers both batch and terminal usage. It was developed by the University of Michigan Computing Center and is currently running on a high-speed computer with over two million bytes of core storage, high-speed paging drums, and disk storage. The facilities include 7-track and 9-track tape drives, a paper-tape reader, and paper-tape punches. For primarily batch operation, there are card readers, card punches, and high-speed printers. There is also a queued digital plotting facility. Batch operation is through HASP spooling feeding MTS. There are facilities for remote batch operation as well. Terminal users communicate with MTS via telephone lines through hard-wired transmission controllers, through the Data Concentrator, a programmable multi-tasking message routing system, or through an audio-response unit. Various types of terminals are supported.

The basis of the system is the "supervisor" which processes all interrupts, does all physical input/output, and does device, CPU, and core storage allocation and scheduling for the jobs (tasks) running. The "supervisor" is the only part of the system that runs in "supervisor state". MTS is a re-entrant program which is activated as a separate job, once for each terminal user line and once for each batch task. Each user can use up to 8,388,608 bytes (2048 pages) of "virtual memory". A job called the paging drum processor handles external page storage and the transferring of pages between core storage and the drums.

MTS is a file-oriented system which allows creation and manipulation of temporary or permanent user (private) files and limited access of permanent public (*) files. The user communicates with MTS via $ prefixed commands to manipulate the system resources. There is an abundance of computer language and general utility facilities in the collection of MTS public files. The MTS command $RUN *BASIC will invoke the BASIC subsystem of MTS which is stored in the public file *BASIC. As one can see, the BASIC user is greatly removed from the complexity of the machine.

The above explanation was condensed and modified from reference 6 (Section XIV).

Appendix M - BASIC System Implementation

Currently, the BASIC System is a re-entrant, sharable subsystem of MTS which consists of two parts, the system interface which is MTS-dependent and the remainder which is independent of MTS. In the latter part there are the following subdivisions.

1) Command Language Interpreter (the CLI) is the heart of the system. It processes BASIC commands and data lines and performs file handling and general resource manipulation. It is responsible for the text-editing, debugging facilities, and program running.

2) Virtual File Handler maintains a catalog of all "virtual" (in-core) files and performs primitive file operations. All editing operations are efficient since the entire file resides in core. The maximum file size is 8 pages (4096 bytes per page).

3) Lexical Scanner lexically scans an S file containing the source program and produces a symbol table plus a postfix file (type P) containing Polish postfix representation of the statement syntactic forms. The P file exists only during lexical scanning and compilation.

4) Compiler (plus hard-wired operator definitions) produces sharable, re-entrant machine code in a non-standard format which resembles CSI (control section) images now implemented in MTS (we were first!). This form which admits recursive programs is acceptable to the BASIC recursive loader. The compiler performs general and floating-point register optimization on a per-statement basis. All floating-point operations are in double-precision.

5) Loader loads and controls all program invocations via CALL, CHAIN, or CLI request. A BASIC program consists of re-entrant, sharable code and a dsect consisting of variable, constant, and temporary storage, and parameter list and address constant storage. The array adcons point to the separately allocated array storage which is also considered part of the program’s dsect. Each time a program is invoked, its dsect is put on a pushdown stack (the dsects are linked together on a list) and a use count (program level) is

recorded. If that program is already loaded (use count > 1), the code is available to be shared, in which case only a dsect is allocated for the program. Hence, a recursive program which calls on itself twice will have three dsect copies and one copy of the code. Since the object file (O) is a virtual file and the program is in CSI form, loading is a rapid core-to-core transfer.

6) Execution Support Routines support I/O conversion, common input/output list handling with special routines being invoked for files or terminal I/O. Special programs to handle GOSUBs, RETURNs, ENDs, computed GOTOs, etc., reside here.

7) Matrix Support handles, except for I/O, all matrix operations in double-precision and checks for conformability of matrices for those operations. Especially noteworthy is the system solver to implement the matrix division A=C/B to solve BA=C, where A and C are not restricted to column vectors but in fact may be matrices.

8) String Support handles all string operations. Especially noteworthy are the extended functions for hex/string and number/string conversion, and string scan and edit, the latter two being similar to the PL/I INDEX function and the SNOBOL4 string scan with replacement, respectively.

9) Built-In Function Support performs all built-in functions with the floating-point operations in double-precision.

The MTS-dependent portion of BASIC consists of system initialization, storage management, permanent file manipulation, sharing and permitting of the MTS files corresponding to the BASIC files, the /INCLUDE command support, terminal I/O, the FORPRT text processor, and the /HELP command processor. The HELP items are stored in an MTS file with a directory at the beginning.

All BASIC files are stored on MTS disks as sequential files without line numbers. The names are of the form:

```
BSC.TXXXXXXX  or  BSC.T*XXXXXX
```

where T is either S, D, or O and XXXXXXX is the name the user refers to (with possibly @T). The second form is for BASIC library files which are stored under a private BASIC staff id.

Appendix N - Postfix Operator Codes

When BASIC translates a "source program" into machine code, it actually goes through two phases of processing (sometimes referred to as "passes"). The first of these phases is called a lexical scan. The lexical scanner produces for its work two tables, a "symbol table" and a "postfix table". The "symbol table" contains an entry for each variable, function name, constant, program name, and line number (of executable statements only) mentioned in the "source program".

The "postfix table" is actually a BASIC file of type "Postfix" (it cannot be referenced by the user). This file will exist only while translation is in progress. Each line in the file represents a modified form of "Polish postfix" for the corresponding executable "source statement". Users interested in seeing the postfix lines corresponding to their "source statements" may use the POSTFIX parameter on the /COMPILE command.

The entries (postfix lines) are of variable length with each being composed of an integral number of words (a word is a 32-bit unit of storage). The first two words of each entry in the postfix file are control words. The first word contains a packed decimal line number which corresponds to the line number of the BASIC statement. This line number will also be the line number where the postfix statement is located in the postfix file. The second word is divided into two halfwords. The first halfword contains a code indicating the type of statement this entry represents. The remaining words of the line contain the postfix decomposition of the statement, one element per word.

Each element may be either an operand or an operator. The high-order byte (8 bits) of each element contains an operator code or a zero indicating that this is an operand element. If the element is an operand, the remaining three bytes will be an absolute pointer to a symbol table entry. If the element is an operator, the remaining three bytes will be an operand count.

In BASIC "source programs" the same symbol may be used for a variety of operators according to its context, i.e., the "+" in A(I)+2.3 and AA1+"THE" denote different operations. Operators in the postfix output will uniquely correspond, not to the external operators, but to the operators and their contexts. That is, the operator codes for the two uses of "+" above will be different.

EXAMPLES:

1. Source: 10 LET A=B+C*D

   Parsed: ABCD*+=

   Postfix output:

00010C00	000A	000F	00	P(A)	00	P(B)	00	P(C)

00	P(D)	03	2	01	2	06	2	4E	

2. Source: 35 GOTO (1,2,27), I+3

   Parsed: 1 2 27I3+ goto-computed

   Postfix output:

00035C00	000A	000B	00	P(LN1)	00	P(LN2)	00	P(LN27)

00	P(I)	00	P(3)	01	2	24	4	4E	

Example 13. Sample Polish Postfix Lines

The following tables completely detail all of the postfix operator codes produced by the lexical scanner in BASIC. The operand type designators should be interpreted as follows:

F	a pointer to a function name entry in the symbol table.
FN	a pointer to a file name entry in the symbol table.
LN	a pointer to a line number entry in the symbol table.
M	a pointer to a matrix entry in the symbol table.
N	a pointer to an undimensioned numeric entry in the symbol table, or the result of another numeric computation.
S	a pointer to a string entry in the symbol table.
V	a variable number of string or numeric entries. For the matrix operators, this should be interpreted as a variable number of matrix entries.

Note that string entries may be string variables or constants. Numeric entries may be numeric variables or constants or numeric expressions (resulting from a previous postfix operation). Each of the designators specified above (except for V) may be preceded by a quantifier which is either a number or the letter V, which indicates a variable number of operands may be involved.

Hex Code	NAME	Operand Types
01	Numeric Addition	2N
02	Numeric Subtraction	2N
03	Numeric Multiplication	2N
04	Numeric Division	2N
05	Numeric Exponentiation	2N
06	Numeric Assignment	2N
07	Numeric Unary Negation	N
08	Matrix Addition	2M
09	Matrix Subtraction	2M
0A	Matrix Multiplication	2M
0B	Matrix Division	2M
0C	Matrix Assignment	2M
0D	Matrix Unary Negation	M
0E	Scalar Matrix Multiplication	N,M
0F	String Concatenation	2S
10	String Assignment	2N
11	Left Extraction	N,S
12	Right Extraction	S,N
13	Numeric Equivalence	2N
14	String Equivalence	2S
15	Numeric Greater Than	2N
16	String Greater Than	2S
17	Numeric Less Than	2N

Hex Code	NAME	Operand Types
18	String Less Than	2S
19	Numeric Not Equal	2N
1A	String Not Equal	2S
1B	Numeric Greater Than Or Equal	2N
1C	String Greater Than Or Equal	2S
1D	Numeric Less Than Or Equal	2N
1E	String Less Than Or Equal	2S
1F	Numeric Subscription	2N
20	String Subscription	S,N
21	Return From User-Def. Fcn.	
22	Call User-Defined Function	F,N
23	Goto	LN
24	Computed Goto	VLN,N
25	If False Goto Next	
26	For	4N
27	Next	N
28	Gosub	LN
29	Return (Call or Gosub)	
2A	Call	FN
2B	UID	
2C	Chain	FN,LN
2D	Redefine Dimensions	M,2N
2E	INV	M

Hex Code	NAME	Operand Types
2F	TRN	M
30	ZER	M
31	CON	M
32	IDN	M
33	ABS	N
34	ATN	N
35	CLK	N
36	COS	N
37	EXP	N
38	INT	N
39	LEN	S
3A	LOG	N
3B	RND	N
3C	SGN	N
3D	SIN	N
3E	SQR	N
3F	TAN	N
40	TIM	N
41	Pause	
42	End/Stop	
43	Read	V
44	Input	V
45	Print	V

Hex Code	NAME	Operand Types
46	Restore	N
47	Matrix Read	V
48	Matrix Print	V
49	TAB	N
4A	Go To Next Print Field (,)	N|S
4B	Continue This Print Field (;)	N|S
4C	Print Matrix In Columns (,)	M
4D	Print Matrix Packed (;)	M
4E	End-Of-Line Terminator	
4F	Illegal Statement	
50	Double Subscripts	2N
51	Flag Between Parts In FOR Stm	N
52	Write File	V
53	NTS	N
54	EDT	S
55	Backspace	N
56	If Endfile	N
57	Matrix Input	V
58	Stop	
59	Formal Argument In DEF Stm.	
5A	Matrix Write File	V
5B	File Designator	
5C	End-Of-Record Mark (:)	N|S

Hex Code	NAME	Operand Types
5D	RDM	M,2N
5E	HEX	S
5F	XTS	S
60	CMD	S
61	STN	S
62	ISN	S
63	RPB	S
64	BFR	S
65	SCN	S
66	DAT	
67	CLS	
68	INP	N
69	OUT	N
6A	String Input Argument	S
6B	String Input Skip (LINPUT)	N
6C	CRP	N

Appendix O - Complete Sample Programs

The following programs are representative of most of the important features of BASIC. They have all been tested and are given here with the results they produce. These samples were prepared on an IBM 2741 terminal, which has both lowercase and uppercase capability. Two of these sample programs were motivated by examples from reference 4 (Section XIV).

Numerical Integration (Runge-Kutta-Gill) Subroutine (Numerical Integration Example)

```
:/list rkdeq
   10 * RKDEQ: RUNGE-KUTTA-GILL SUBROUTINE
   20 * FOURTH ORDER METHOD
   30 DIM Y(8),F(8),Q(8)
   40 A3=1.70710678
   50 A2=0.29289321
   55 RESTORE
   60 READ K,N,S,X
   70 MAT READ Y(N,0),F(N,0),Q(N,0)
   80 GOTO (100,110,180,260,330,410),K+1 /* WHERE TO GO.
   85 * INITIALIZATION.
   100 MAT Q=ZER
   110 K=K+1  /* COMMON EXIT POINT.
   120 RESTORE
   130 WRITE K,N,S,X
   140 MAT WRITE Y,F,Q
   150 RESTORE
   160 RETURN
   170 *   ***SECOND TIME***
   180 FOR I=1 TO N
   190 F1=0.5*(F(I)-2*Q(I))
   200 Y(I)=Y(I)+S*F1
   210 Q(I)=Q(I)+3*F1-0.5*F(I)
   220 NEXT I
   230 X=X+S/2
   240 GOTO 110
   250 *   ***THIRD TIME***
   260 FOR I=1 TO N
   270 F1=A2*(F(I)-Q(I))
   280 Y(I)=Y(I)+S*F1
   290 Q(I)=Q(I)+3*F1-A2*F(I)
   300 NEXT I
   310 GOTO 110
   320 *   ***FOURTH TIME***
   330 FOR I=1 TO N
   340 F1=A3*(F(I)-Q(I))
   350 Y(I)=Y(I)+S*F1
   360 Q(I)=Q(I)+3*F1-A3*F(I)
   370 NEXT I
   380 X=X+S/2
   390 GOTO 110
   400 *   ***FIFTH(FINAL) TIME***
   410 FOR I=1 TO N
   420 F1=(F(I)-2*Q(I))/6
   430 Y(I)=Y(I)+S*F1
   440 Q(I)=Q(I)+3*F1-0.5*F(I)
```

```
     450 NEXT I
     460 K=1   /* FLAG PRINTING.
     470 GOTO 120
  &End-Of-File
  :/compile rkdeq  par=noex
  :/list rkmain
     10 * PROGRAM TO SOLVE THE SECOND ORDER EQUATION:
     20 * Y''=A*Y'+B*Y WHERE Y IS A FUNCTION OF X.
     30 * DECOMPOSITION INTO TWO FIRST ORDER EQUATIONS YIELDS:
     40 * Y(1)'=Y(2) AND Y(2)'=A*Y(2)+B*Y(1)
     50 * WHICH ARE SOLVED VIA RUNGE-KUTTA-GILL METHODS.
     60 * Y(1) AND Y(2) ARE Y AND Y', RESPECTIVELY.
     70 * A,B, THE INTERVAL, STEP S ARE READ IN WITH
     80 * STARTING VALUES FOR Y(1) AND Y(2).
     90 DIM Y(2),F(2),Q(2)
     95 N=2
     100 FILE RKDEQ
     105 PRINT "0***RUNGE-KUTTA-GILL INTEGRATION***"
     106 PRINT "    SOLVES Y''(X)=A*Y'(X)+B*Y(X)"
     110 PRINT "0ENTER A,B,X0,X1,S,Y(X0),Y'(X0)"
     120 INPUT A,B,X,X1,S,Y(1),Y(2)
     125 PRINT "0A =";A;",  B =";B;",  STEP =";S
     128 PRINT " X RANGES FROM";X;" TO ";X1
     130 K=0   /* 0 MEANS RKDEQ INITIALIZATION.
     150 PRINT "0";TAB(3);"X";TAB(19);"Y";TAB(34);"Y'"
     160 RESTORE FILE 1
     165 WRITE FILE 1,K,N,S,X
     170 MAT WRITE FILE 1,Y,F,Q
     180 CALL RKDEQ  /* INITIALIZE FIRST TIME, SOLVE THEREAFTER.
     190 READ FILE 1,K,N,S,X
     200 MAT READ FILE 1,Y,F,Q
     210 IF K=1 THEN 260
     220 IF X>X1 THEN 110
     230 F(1)=Y(2)     /* COMPUTE DERIVATIVES.
     240 F(2)=A*Y(2)+B*Y(1)
     250 GOTO 160
     260 PRINT X,Y(1),Y(2)
     270 GOTO 160
  &End-Of-File
  :/run rkmain


     ***RUNGE-KUTTA-GILL INTEGRATION***
        SOLVES Y''(X)=A*Y'(X)+B*Y(X)

     ENTER A,B,X0,X1,S,Y(X0),Y'(X0)
  ?0,-1,0,4,0.2,0,1

     A = 0,  B = -1,  STEP = 0.2
     X RANGES FROM 0 TO  4
```

```
      X              Y              Y’
     0              0              1
     0.2            0.1986667      0.9800667
     0.4            0.3894131      0.9210622
     0.6            0.5646351      0.8253389
     0.8            0.7173474      0.696713
     1              0.8414619      0.5403121
     1.2            0.9320307      0.3623714
     1.4            0.9854433      0.1699847
     1.6            0.9995706      -2.917843E-2
     1.8            0.9738491      -0.2271782
     2              0.9093043      -0.4161211
     2.2            0.8085094      -0.5884748
     2.4            0.6754828      -0.7373683
     2.6            0.5155277      -0.856866
     2.8            0.3350208      -0.942204
     3              0.1411582      -0.9899802
     3.2            -5.833161E-2   -0.99829
     3.4            -0.2554958     -0.9668022
     3.6            -0.4424742     -0.8967721
     3.8            -0.6118129     -0.7909916
     4              -0.7567611     -0.6536777

     ENTER A,B,X0,X1,S,Y(X0),Y’(X0)
  ?/endfile
  + At Line "120" in Program "RKMAIN"
  + Program Ends
  :
```

Sorting (Shuttle Exchange)

```
:/list sort
   10 * SORTING PROGRAM READS IN AN ARRAY OF
   20 * NUMBERS, SORTS THEM USING A SHUTTLE
   30 * INTERCHANGE METHOD WITH OPTIMIZATION, AND
   40 *PRINTS THEM OUT IN DESCENDING ORDER.
   50 DIM X(100)    /* MAX OF 100 NUMBERS ALLOWED.
   55 PRINT "0HOW MANY NUMBERS TO SORT?"
   60 INPUT N
   70 IF N>0 THEN: IF N<=100 THEN 95
   80 PRINT "ILLEGAL! 1 TO 100 ALLOWED. TRY AGAIN!"
   90 GOTO 60
   95 PRINT "ENTER THE ";N;" NUMBERS."
   100 MAT INPUT X(N,0)
   110 IF N=1 THEN 230
   120 M1=N
   130 S=0    /* SWAP SIGNAL OFF.
   140 M1=M1-1
   150 FOR I=1 TO M1
   160 IF X(I)>=X(I+1) THEN 210
   170 S=1     /* SAY THERE WAS A SWAP.
   180 T=X(I)
   190 X(I)=X(I+1)
   200 X(I+1)=T
   210 NEXT I
   220 IF S<>0 THEN: IF M1>1 THEN 130
   230 PRINT "0ELEMENTS IN DESCENDING ORDER:"
   240 MAT PRINT X;   /* PACKED TOGETHER
   250 GOTO 55
&End-Of-File
:/run sort

   HOW MANY NUMBERS TO SORT?
?6
  ENTER THE  6 NUMBERS.
?-2,0,15,7,10,3

   ELEMENTS IN DESCENDING ORDER:
   15 10 7 3 0 -2

   HOW MANY NUMBERS TO SORT?
?-2
  ILLEGAL! 1 TO 100 ALLOWED. TRY AGAIN!
?/endfile
+ At Line "60" in Program "SORT"
+ Program Ends
:
```

Curve Plotting (Terminal Output)
Plotting (of Data Curves on Terminal) Example

```
:/list plot
   10 * PLOTTING PROGRAM TO PLOT THE
   20 * FUNCTION P*SIN(X)+Q VERSUS X.
   30 * X RANGES FROM A TO B IN STEPS OF S.
   40 * Y RANGES FROM C TO D OVER N DIVISIONS.
   50 * POINTS OUTSIDE THIS RANGE ARE IGNORED.
   60 AA="-------------------------"
   70 AA=AA+AA
   100 PRINT "0*** PLOT PROGRAM FOR P*SIN(X)+Q ***"
   110 DEF FNR(X)=INT(X+.5)
   120 DEF FNX(X)=INT(100*X+.5)/100
   125 PRINT "0ENTER XMIN,XMAX, AND STEP SIZE"
   130 INPUT A,B,S
   135 PRINT "0ENTER P,Q AND NUMBER OF Y DIVISIONS"
   140 INPUT P,Q,N
   150 IF N>0 THEN: IF N<=50 THEN 180
   160 PRINT "SORRY, 1 TO 50 SUBDIVISIONS ALLOWED"
   170 PRINT "TRY AGAIN"
   175 GOTO 135
   176 *
   180 D=P+Q
   182 C=-P+Q
   184 IF D>=C THEN 188
   186 D=C
   187 C=P+Q
   188 H=(D-C)/N
   189 PRINT "0"
   190 PRINT "&Y-AXIS:  FROM ";C;" TO ";D
   195 PRINT " IN STEPS OF ";H
   200 BB=(N-1)*AA  /* GET N-1 MINUS SIGNS.
   210 PRINT "0";TAB(8);"’";BB;"’"
   255 *
   260 FOR X=A TO B STEP S
   270 Y=P*SIN(X)+Q
   280 Y1=8+FNR((Y-C)/H)
   290 PRINT FNX(X);TAB(Y1);"*"
   300 NEXT X
   310 GOTO 125
&End-Of-File
```

```
:/run plot

   *** PLOT PROGRAM FOR P*SIN(X)+Q ***

   ENTER XMIN,XMAX, AND STEP SIZE
?0,6.5,0.5

   ENTER P,Q AND NUMBER OF Y DIVISIONS
?1,0,20


   Y-AXIS:  FROM  -1 TO  1 IN STEPS OF  0.1

          '-------------------'
   0                *
   0.5                   *
   1                        *
   1.5                        *
   2                         *
   2.5                    *
   3                 *
   3.5          *
   4        *
   4.5    *
   5      *
   5.5       *
   6             *
   6.5                *

   ENTER XMIN,XMAX, AND STEP SIZE
?/endfile
+ At Line "130" in Program "PLOT"
+ Program Ends
:
```

Concordance (String Manipulation)
String Manipulation (Concordance) Example

```
:/list concord
   1 * CONCORDANCE PROGRAM READS ONE SENTENCE
   2 * PER LINE WHICH ENDS WITH A PERIOD. THE WORDS,
   3 * WHICH MAY BE SEPARATED BY BLANKS OR COMMAS, ARE
   4 * TABULATED AND WORD FREQUENCIES ARE RECORDED.
   5 * A MAXIMUM OF 100 WORDS IS ALLOWED.  THE COMMAND
   6 * "STOP!" PRODUCES THE FREQUENCY LIST.
   10 DIM WW(100),C(100)  /* WORDS,COUNT
   15 PRINT "0*** CONCORDANCE PROGRAM ***"
   16 P'T "ENTER A SENTENCE AFTER EACH $ PREFIX."
   18 PRINT "STOP! PRODUCES THE WORD FREQUENCY LIST."
   20 LINPUT AA   /* THE SENTENCE
   30 IF AA="STOP!" THEN 310
   35 AA=AA+" "  /* SURE OF TRAILING BLANK.
   40 AA=EDT(AA(BFR(".")+RPB(" ")))  /* PERIOD TO A BLANK
   50 AA=EDT(AA(BFR(",")))   /* BLANK OUT COMMAS
   60 IF SCN(AA)<>0 THEN 50
   70 * NOW WE HAVE WORDS FOLLOWED BY BLANKS.
   80 R=BFR(" ")   /* BLANK BREAK CHARACTER.
   85 IF AA="" THEN 20   /* DONE WITH THE LINE?
   90 I=SCN(AA)   /* FIND THE BLANK.
   100 SS=(I-1)*AA   /* GET THE WORD
   110 AA=AA*(LEN(AA)-(I-1))  /* REMOVE IT.
   120 BB=(1)*AA  /* REMOVE BLANK SEQUENCE.
   130 IF BB<>" " THEN 170  /* HIT A NON-BLANK.
   140 AA=AA*(LEN(AA)-1)  /* REMOVE THE BLANK
   150 IF AA<>"" THEN 120  /* MAYBE AT END.
   170 FOR I=1 TO N   /* WORD SEARCH.
   180 IF WW(I)=SS THEN 250
   190 NEXT I
   200 N=N+1   /* NOT PRESENT, ADD IT.
   210 IF N>100 THEN 270
   220 WW(N)=SS   /* ENTER IT.
   230 C(N)=1   /* INITIALIZE COUNT.
   240 GOTO 85
   250 C(I)=C(I)+1
   260 GOTO 85
   270 PRINT "YOU HAVE TOO MANY WORDS!"
   280 PRINT """";AA;""" NOT PROCESSED."
   290 PRINT "PARTIAL CONCORDANCE FOLLOWS:"
   300 GOTO 320
   310 PRINT "0FULL CONCORDANCE FOLLOWS:"
   315 IF N=0 THEN 380
   320 PRINT "0WORD";TAB(20);"COUNT"
   330 PRINT " ----";TAB(20);"-----"
   340 FOR I=1 TO N
```

```
     350 PRINT " ";WW(I),TAB(21);C(I)
     360 NEXT I
     370 STOP
     380 PRINT "NOT A WORD WAS FOUND!"
  &End-Of-File
  :/run concord

     *** CONCORDANCE PROGRAM ***
    ENTER A SENTENCE AFTER EACH $ PREFIX.
    STOP! PRODUCES THE WORD FREQUENCY LIST.
  $now, now, i say, now.
  $how, how, i say, how, cow.
  $stop!

     FULL CONCORDANCE FOLLOWS:

     WORD               COUNT
     ----               -----
     NOW                  3
     I                    2
     SAY                  2
     HOW                  3
     COW                  1
  +Stop!
  + At Line "370" in Program "CONCORD"
  + Program Ends
  :
```

Random Distributions (Card Dealing)

```
:/list deal
   5  REM PROGRAM DEALS FOUR POKER HANDS ON REQUEST.
   6  REM THE USER JUST TYPES THE COMMAND "DEAL".
   7  DIM L(51),VV(12),PP(4),SS(3)
   8  PRINT "0I DEAL POKER HANDS."
   9  PRINT "0TYPE DEAL FOR A DEAL; OTHERWISE TYPE STOP."
   10 GOSUB 1100    /* PROGRAM INITIALIZATION
   14 PRINT "0"
   15 INPUT AA    /* READ USER REQUEST.
   16 IF AA="DEAL" THEN 20
   17 PRINT "THE HOUSE BIDS FAREWELL!"
   18 STOP
   20 GOSUB 1020   /*  GATHER THE CARDS.
   21 FOR H=1 TO 4
   22 PRINT "0";PP(H)
   25 FOR C=1 TO 5
   30 GOSUB 2010  /* DEAL A CARD.
   40 GOSUB 3010  /* PRINT IT.
   50 NEXT C
   55 NEXT H
   60 GOTO 14   /*GET NEXT REQUEST.
   70 *
   1000 REM SET UP THE DECK.
   1020 FOR I=1 TO 51
   1030 LET L(I)=I
   1035 NEXT I
   1045 RETURN
   1050 *
   1100 FOR S=0 TO 3
   1110 READ SS(S)   /* READ SUITS FROM DATA FILE
   1120 NEXT S
   1130 FOR V=0 TO 12
   1140 READ VV(V)   /* READ CARD RANKS TOO.
   1150 NEXT V
   1155 FOR H=1 TO 4
   1156 READ PP(H)    /*  THE PLAYER IDENTIFICATION.
   1158 NEXT H
   1160 RETURN
   1170 *
   1200 DATA CLUBS,DIAMONDS,HEARTS,SPADES
   1205 DATA DEUCE,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT
   1210 DATA NINE,TEN,JACK,QUEEN,KING,ACE
   1220 DATA NORTH,SOUTH,EAST,WEST
   1230 *
   2000 REM DEAL A CARD.
   2010 LET I=INT(52*RND(D))
   2015 * NOTE THAT D IS INTIALLY 0 FOR RND.
```

```
     2020 X=L(I)
     2030 IF X<0 THEN 2010 /* DISALLOW SAME CARD.
     2040 L(I)=-1   /* MARK IT AS BEING DEALT.
     2050 RETURN
     2060 *
     3000 REM PRINT A CARD.
     3010 S=INT(X/13)
     3020 V=X-13*S
     3030 PRINT TAB(4);VV(V);" OF ";SS(S)
     3040 RETURN
  &End-Of-File
  :/run deal

     I DEAL POKER HANDS.

     TYPE DEAL FOR A DEAL; OTHERWISE TYPE STOP.

  ?deal

     NORTH
        QUEEN OF CLUBS
        THREE OF CLUBS
        EIGHT OF HEARTS
        TEN OF CLUBS
        KING OF DIAMONDS

     SOUTH
        SEVEN OF HEARTS
        NINE OF HEARTS
        DEUCE OF CLUBS
        FIVE OF HEARTS
        THREE OF SPADES

     EAST
        QUEEN OF DIAMONDS
        QUEEN OF HEARTS
        SIX OF SPADES
        DEUCE OF DIAMONDS
        DEUCE OF HEARTS

     WEST
        JACK OF SPADES
        EIGHT OF SPADES
        ACE OF HEARTS
        SEVEN OF DIAMONDS
        FOUR OF HEARTS


  ?stop
    THE HOUSE BIDS FAREWELL!
  + Stop!
  + At Line "18" in Program "DEAL"
  + Program Ends
```

Reader’s Comment Form

BASIC in MTS
Volume 10
December 1980

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD.

Date ______________________

Name ____

Address ____

____

____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Update Request Form

BASIC in MTS
Volume 10
December 1980

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you desire to have these updates mailed to you, please submit this form.

Updates are also available in the memo files at both the Computing Center and NUBS. There you may obtain any updates to this volume that may have been issued before the Computing Center receives your form. Please indicate below if you desire to have the Computing Center mail to you any previously issued updates.

Name ____

Address ____

____

____

Previous updates needed (if applicable):__________

The completed form may be sent to the Computing Center by Campus Mail or U.S. Mail, or dropped in the Suggestion Box at the Computing Center, NUBS, or BSAD. Campus Mail addresses should be given for local users.

Publications
Computing Center
The University of Michigan
Ann Arbor, Michigan 48109

Users associated with <u>other MTS installations</u> (except the University of British Columbia) should return this form to their respective installations. Addresses are given on the reverse side.

Addresses of other MTS installations:

The University of Alberta
Information Coordinator
352 General Services Bldg.
Edmonton, Alberta
Canada T6G 2H1

Information Officer, NUMAC
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne
England NE1 7RU

Rensselaer Polytechnic Institute
Documentation Librarian
130 Amos Eaton Hall
Troy, New York 12181

Simon Fraser University
Computing Centre
User Services Information Group
Burnaby, British Columbia
Canada V5A 1S6

Wayne State University
Computing Services Center
Academic Services Documentation Librarian
5950 Cass Ave.
Detroit, Michigan 48202