BCC-500 BASIC INSTRUCTIONS

Numbers

Integer (no decimal point)
Floating Point (has a decimal point)
Scientific Notation (decimal point and power of 10)

Simple Variables

All variables are a letter or a letter followed by a number (and a $ is a string)

Arrays

Only one and two dimensional arrays permitted. Array names must be a letter, subscripts are any expression

Arithmetic Operators

-	Negation
↑	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

Related Operators

=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
><	Not equal to

Commands

CHAIN	Loads another program segment, retains variables
CONTINUE	Continues program execution
DELETE	Deletes statements from memory
LET	Assigns a value to a variable
LIST	Lists statements stored in memory
LOAD	Reads a file into memory
QUIT	Returns program control to the Executive
RUN	Start executing program statements
SAVE	Save program memory on designated file
SET	Assigns a value to a variable
~~(esc)~~ control/K	One or more ~~(esc)~~ control/K s returns control to BASIC or EXEC

Miscellaneous Program Statements

DIM	Reserves memory for an array
END	Last statement in program (not required)
EXECUTE	Automatic program running upon completion of loading
PAUSE	Stops program execution until CONTINUE command given
REM	Non-executable remark statement
STOP	Terminates execution

Controlling Statement Execution Order

FOR	Start loop, TO set range, STEP set increment
NEXT	Designated end of a FOR loop
GO TO	Begins execution at statement specified
GOSUB	Begins execution at subroutine statement specified
IF GO TO	Conditional GO TO statement
IF THEN	Conditional GO TO statement
ON GO TO	Multiple conditional GO TO statement
ON GOSUB	Multiple conditional GOSUB statement

Matrix Statements

Substitution	Sets elements of one matrix to another
Constant Multip.	Multiplies elements of a matrix by a constant
Multiplication	Multiplies two matrices
Subtraction	Subtracts two matrices
Addition	Adds two matrices
CON	Sets elements of a matrix to 1
IDN	Sets elements of a matrix to the identity matrix
INV	Creates matrix which is the inverse of given matrix
ONE	Sets elements of a matrix to 1
TRN	Creates matrix which is the transpose of given matrix
ZER	Sets elements of a matrix to Ø

Strings

CHANGE	Changes String-variables to variable or visa-versa

Input

DATA	Data stored within program to be used with READ
READ	Reads data from a DATA statement
RESTORE	Sets pointer to first DATA statement
INPUT	Reads data from the terminal
INPUT FILE	Reads data from a symbolic file
READ FILE	Reads data from a binary file
INPUT USING	Reads formatted data from the terminal
INPUT FILE USING	Reads formatted data from a symbolic file
ACCEPT	Identical to INPUT
MAT	Mat added to any INPUT or READ statement changes the statement to read matrices

Output

PRINT	Writes data on the terminal
PRINT FILE	Writes data on a symbolic file
WRITE	Writes data on a binary file
WRITE FILE	Writes data on a binary file
PRINT USING	Writes formatted data on the terminal
PRINT FILE USING	Writes formatted data on a symbolic file
TYPE	Identical to PRINT
MAT	MAT added to any PRINT or WRITE statement changes the statement to write matrices

Formatting Input/Output

,	Spaces to multiple of 15 positions before printing
i.	Spaces to multiple of 3 positions before printing
<string>"	No spacing before printing next value
:<picture>	Designates a picture string to format input/output

Files

OPEN	Opens files
EOF	Check for parity error or end-of-file
CLOSE	Closes files

Functions and Subroutines

GOSUB	Call to subroutine
ON GOSUB	Multiple conditional call to subroutine
RETURN	Designates end of subroutine
DEF	User defined function
SIN(X)	Sine of X
COS(X)	Cosine of X
TAN(X)	Tangent of X
ATN(C)	Arc tangent of X
EXP(X)	Exponential (base e)
ABS(X)	Absolute value of X
LOG(X)	Natural log of S
SQR(X)	Square root of X
LGT(X)	Log base ten of X
INT(X)	Integer part of X
RND(X)	Random number
EXT(X)	Exponentiation (base 1Ø)
TAB(X)	see Formatted Input/Output
TIM(X)	CPU or terminal time
EOF(X)	see Files
DAT(X)	Current date and time
CHR(X)	Returns one character string
STR(X)	Returns 2-13 character strings
[redacted]ED(X)	Returns first character of argument
TAL(X)	Returns all but first character of argument
LEN(X)	Length of argument
NUM(X)	Returns floating point of argument
ASC(X)	Returns internal decimal value

Chapter 1 - INTRODUCTION

Using this Manual

This manual is intended to be a reference manual for XDS 940 BASIC and a tutorial for learning BASIC.

Chapters one through four provide an introduction to BASIC. Beginners may read these chapters and begin writing programs immediately. If the user needs only the elementary features described, he may use just these without worrying about the extra features described in chapters five through eight. These more advanced features, although always available, cost the user almost nothing if they are not used. However, these other features of BASIC can be of great value. In particular, the matrix package and the formatted input/output capabilities can be very helpful. These can be learned and used as necessity dictates.

For clarity, certain typographical conventions have been used throughout this manual. To distinguish text output by the computer from text entered by the user, we have underlined user entered text. Actual examples of BASIC commands or programs are preceeded by |. Nonprinting control characters are indicated by an ampersand (&) preceding the letter (e.g. &D to indicate a control-D). The use of the RETURN key is indicated by (ret) and (esc) indicates the use of the ESCAPE key.

An idea is enclosed by <>. Some of the concepts used in the manual and their abbreviations are:

<cons>	constant
<stn>	statement number
<fln>	file number
<fnm>	file name
<rel-expr>	relational-expression
<expr>	expression
<var>	variable
<let>	letter
<ilist>	input list in form: <var 1> [,<var 2>]...
<olist>	output list in form: <expr> [{<comma>,;} <expr>]...
<mlist>	output list for matrix in form: <var 1> [{<comma>,;} <var 2>]...

Braces {} are used to denote a grouping. For example,

```
LOAD {(ret), <fnm>}
```

The grouping indicates that a choice is to be made. The above example indicates that the word LOAD must be followed by either a (ret) or a file name.

Brackets [] denote options. Items enclosed in brackets may or may not appear. For example,

[<stn 1>] GO TO <stn 2>

denotes that the words GO TO may or may not be preceded by a statement number; however, the words GO TO must be followed by a statement number.

Ellipsis marks (...) indicate that the preceeding item may occur one or more times in succession. For example,

<expr 1> [, <expr 2>]...

states that <expr 1> must occur, <expr 2> might occur and that it may be followed by <expr 3>, <expr 4>, etc.

In all program examples, each line is terminated by a (ret) which is not indicated in the examples.

Chapter 1 - INTRODUCTION

Using this Manual

This manual is intended to be a reference manual for XDS 940 BASIC and a tutorial for learning BASIC.

Chapters one through four provide an introduction to BASIC. Beginners may read these chapters and begin writing programs immediately. If the user needs only the elementary features described, he may use just these without worrying about the extra features described in chapters five through eight. These more advanced features, although always available, cost the user almost nothing if they are not used. However, these other features of BASIC can be of great value. In particular, the matrix package and the formatted input/output capabilities can be very helpful. These can be learned and used as necessity dictates.

For clarity, certain typographical conventions have been used throughout this manual. To distinguish text output by the computer from text entered by the user, we have underlined user entered text. Actual examples of BASIC commands or programs are preceeded by |. Nonprinting control characters are indicated by an ampersand (&) preceding the letter (e.g. &D to indicate a control-D). The use of the RETURN key is indicated by (ret) and (esc) indicates the use of the ESCAPE key.

An idea is enclosed by <>. Some of the concepts used in the manual and their abbreviations are:

<cons>	constant
<stn>	statement number
<fln>	file number
<fnm>	file name
<rel-expr>	relational-expression
<expr>	expression
<var>	variable
<let>	letter
<ilist>	input list in form: <var 1> [,<var 2>]...
<olist>	output list in form: <expr> [{<comma>,;} <expr>]...
<mlist>	output list for matrix in form: <var 1> [{<comma>,;} <var 2>]...

Braces {} are used to denote a grouping. For example,

LOAD {(ret), <fnm>}

The grouping indicates that a choice is to be made. The above example indicates that the word LOAD must be followed by either a (ret) or a file name.

Chapter 2 - GENERAL CONCEPTS

Program definition

A computer program is a set of statements which describe the specific directions for solving a particular problem. By means of a program you communicate your directions to a computer and, as with all attempts at communication, you must be careful to restrict yourself to terms that are clearly understood. A program consists of one or more statements that are to be executed in a certain order by the computer.

Statement Number

Statements entered with a statement number are analyzed by BASIC to see if it can understand what you wish done. If so, it will save this statement for later execution with the other statements you may enter. If not, it will print an error message and discard the line. These statements are actually executed only after you have instructed BASIC to do so by entering a RUN command. In BASIC, order is determined by the statement number. BASIC will execute the statements in increasing numerical sequence starting with the smallest statement number and proceeding to the largest.

Command

If a statement which has no statement number is entered, BASIC assumes that you wish this statement to be executed directly after a (ret) is typed, and therefore these statements are called commands.

The arithmetic components of the BASIC language are constants, variables and expressions.

Constant

A constant is a fixed value used in the program rather than being calculated by the program. A constant may be any decimal number, positive or negative, with or without a decimal point, that can be expressed in twelve digits or less. Only twelve significant digits of accuracy are retained. There are three forms in which a constant may be expressed.

Integer

A constant may be expressed as an integer number with twelve or fewer significant digits and no fractional part. Examples are:

1 -7 130

Floating Point

A constant may be expressed as a floating point number containing a fractional part. Examples are:

1.5 10.2 155.275

Scientific Notation

A constant is expressed in scientific notation if it is too large or too small to be expressed in eight digits. Scientific notation consists of a number, either integer or floating point., followed by the letter E, followed by a 1 to 3 digit exponent. The exponent represents the integral power of ten by which the number to the left of the E must be multiplied. For example, all of the quantities 2.59, .259E1, 25.9E-1, and 259E-2 express the same number. The largest number which BASIC will accept is 5.7896044E76; the smallest is 4.318084E-77.

Variable

BASIC permits the user to use a symbol to represent a constant. Such symbols are called variables because the value represented by the symbol may be changed. In BASIC, an arithmetic variable may be a single letter or a letter followed by a digit. The one exception is that PI is also a valid variable name. The value of this variable is initialized by BASIC to be 3.1415926.

Array

It is often convenient to keep data in a list. Such a list is called an array. The individual values in the list are called array elements. We refer to an array element by using the name of the array and the position of the element in the array. For example, we can refer to the fourth element in array A by writing A(4). In this example, the 4 is called the subscript. Note that the name of an array must be only one letter, while a subscript may be any well-defined expression, which may also include an array element. (Expressions are discussed below.)

Dimension

An array may have one or two dimensions. A two dimensional array may be thought of as having columns and rows. There is always one subscript for each dimension; thus, a two dimensional array is written as A(X,Y) where X represents a row number and Y represents a column number. See the DIM statement below.

Expression

Arithmetic expressions are formed by combining variables and/or constants with arithmetic operators. There are six arithmetic operators in BASIC:

- Negation
↑ Exponentiation
* Multiplication
/ Division
+ Addition

- Subtraction

To make sure that the computer evaluates the expression the way the user meant it to be evaluated, there is an established rule of precedence:

first - Functions
second - Exponentiation
third - Negation
fourth - Multiplication or Division
fifth - Addition or Subtraction

The computer calculates from left to right if operators of the same precedence (for example, multiplication and division) appear in the same line. To alter this order, parentheses must be used. Expressions within the parentheses are evaluated first using the above rules of precedence. Parentheses which do not alter the normal precedence operations have no effect.

Relational Expression

A relational expression consists of two arithmetic expressions separated by one of the relational operators. The relational operators available in BASIC are:

Operator	Relation
=	equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
<>	not equal to
><	not equal to

In BASIC, a relational expression is either "true" or "false", depending on whether the answer to the question implied by the relational expression is "yes" or "no". If the answer to the question is "yes", the relational expression is "true"; if the answer is "no", the relational expression is "false".

Function

The functions available in BASIC are:

SIN(<expr>)	sine of <expr> in radians
COS(<expr>)	cosine of <expr> in radians
TAN(<expr>)	tangent of <expr> in radians
ATN(<expr>)	arctangent of <expr> in radians
EXP(<expr>)	natural exponential of <expr>
ABS(<expr>)	absolute value of <expr>
LOG(<expr>)	natural log of <expr> (base e)
SQR(<expr>)	square root of <expr>
LGT(<expr>)	common log of <expr> (base 10)
INT(<expr>)	integer part of <expr>
RND.	random number
EXT(<expr>)	exponentiation (base 10)
TAB(<expr>)	tabs output in a PRINT statement
TIM(<expr>)	time since system start, or user's CPU time
EOF(<expr>)	check for end-of-file

The <expr> enclosed in parentheses is called the argument. The argument need not be enclosed in parentheses where no confusion can result from the precedence implied in the following expression.

Some functions require additional description.

RND

The RND function is a pseudo-random number generator. When called, it will produce a number between zero and one. When called repeatedly, it will produce a sequence of pseudo-random numbers. The same sequence of pseudo-random numbers will occur in every program in which the RND function is used. This feature is helpful in debugging. The argument in the call to the RND function is meaningless and may be called with an argument or simply followed by a period; either X = RND. or X = RND(Z) is acceptable.

INT

The INT (integer) function is used to determine the integer part of a number. INT always returns as its value the next smaller integer value whether the value of the expression is positive or negative; thus INT(7.8) equals 7 and INT(-7.8) equals -8.

TAB

When used in a PRINT statement, the TAB function spaces over to the Nth print position, where N is the integer part of the argument. If N is less than or equal to the current position, nothing happens. Spacing occurs such that after a TAB(N), the next character will be printed in the Nth position.

TIM

The TIM function returns the time since system start-up in 1/60ths of a second if the argument is Ø, and the CPU time accumulated since login (in 1/60ths of a second) if the argument is 1.

EOF

When reading from either a symbolic or binary file, there will be a certain point when there is nothing more to read, because nothing more has been written. This is the end-of-file. Reading past it is regarded as an error, so the end-of-file condition should be detected before an error occurs.

The EOF function has a file number as its argument, returning -1 if the file is closed, Ø if the file is open and everything is normal, 1 if the file is at the end-of-file and 2 if a hardware error occurred on the last operation.

When the end-of-file is reached, variables will be set to either zero or the null string until the input list is exhausted. Then the file will be flagged as at the end-of-file. If an attempt is made to use the file again without closing and reopening it, the error message "FILE AT EOF" will be printed. If the EOF function is called after the end-of-file is detected, the file will be reset to its closed state. Therefore the end-of-file can only be checked once on each file read.

Chapter 3 - PROGRAM PREPARATION AND EXECUTION

Preparation

There are several methods for preparing a BASIC program for execution: typing directly into BASIC, QED, or using paper tape input. The simplest and most direct is to type the program directly into BASIC. This is the usual procedure for small or medium sized programs. BASIC does not have complete provisions for line editing like those available in QED and for the programmer who has mastered QED it is usually far simpler to prepare the program and make revisions while in QED, write the program to a file and load it into BASIC for debugging and execution. The most economical method is to prepare the program off line on paper tape, log-in and read the paper tape into a file. Complete information regarding the preparation of paper tapes can be found in the "Timesharing User's Guide".

LOAD

LOAD <fnm>

If the program was prepared in QED or on paper tape, then the program is stored in a file which needs to be brought into memory by the LOAD command. If the user types a (ret) after LOAD instead of the file-name, BASIC responds with "FROM FILE>", and waits for a file-name to be supplied.

Errors

As the program is loading or while typing a statement directly into BASIC, the system prints out any line containing an error. It also prints a pointer to the position in the line where BASIC detected the error although this may not be the location of the error.

Editing

Any line can be corrected by retyping it using the same statement number. The editing control characters &A, &W and &Q may be used while entering BASIC commands or programs. &A deletes the previous character, &W deletes the previous word and &Q returns to the beginning of the current line. An uparrow, back slash or underline is printed respectively so that you know that the editing action has taken place.

Remember that a space is ignored in BASIC statements except in a literal string (text surrounded by quotes) although spaces are usually inserted for readability. Since each BASIC statement is limited to a single line of 80 characters, completion of a complex statement may be more important than the readability of that line.

BASIC interprets lower-case letters as upper-case letters in all cases except when included in a literal string.

LIST

LIST {[<stn 1> [-<stn 2>, <comma><stn 2>...]]}

When editing the program, it is useful to list some or all of the statements of the program. LIST prints statements in ascending numerical order regardless of input order.

Command	Meaning
LIST	list entire program
LIST 10	list statement number 10
LIST 10 - 30	list statement numbers 10 thru 30
LIST 10, 20, 30	list statement numbers 10, 20, 30

DEL

DEL {ALL, [<stn 1> [-<stn 2>, <comma><stn 2>...]]}

When editing the program in BASIC, it is possible to delete one or more statement numbers of the program by the DEL command. To delete one statement, the user may simply type the statement number followed by a (ret).

Command	Meaning
DEL ALL	delete entire program
DEL 10	delete statement number 10
DEL 10 - 30	delete statement numbers 10 thru 30
DEL 10, 20, 30	delete statement numbers 10, 20, 30

REM

<stn> REM <text>

An important part of any program is the description of what is done and what data should be supplied. One way of documenting a program is to supply remarks along with the program itself. BASIC provides this capability with the REM (remark) statement.

```
100 REM I NEVER REMEMBER WHAT MY PROGRAM DOES
```

Comment

Line-by-line comments may be included by putting an exclamation point and the comment at the end of the statement. It is not recommended that commands be commented in this manner. The comment will be ignored, and the ! will be treated as though it were a (ret). Exclamation points may still be used in literals, of course. CHAIN statements should not be commented with the !, as BASIC has no easy way of determining if the ! belongs with the file name or a comment.

```
10 PRINT "FAR OUT!" ! comment of surprise and awe
```

PAUSE

<stn> PAUSE

The PAUSE statement is useful while debugging a program. When executed it interrupts the program and returns to the BASIC command mode after typing the message

PAUSE AT <stn>

At this point commands may be used to change or examine variables. Program execution may then be resumed using the CON command. Execution may also be begun at some other point in the program using the GOTO command. However, GOTO, unlike CON will perform a certain amount of initialization. Program statements should not be changed or new statements added after a PAUSE statement unless the program is restarted with the RUN command.

CON

CON

The CON (continue) command is used to resume execution after a PAUSE.

STOP

<stn> STOP

The STOP statement is used to stop program execution at some point in the program other than the last statement.

END

<stn> END

An END statement indicates the termination point of a program. Some versions of BASIC require an END as the last statement of the program. While NOAA-BASIC does not, the END statement is provided for consistency with other systems.

RUN

RUN [<fnm>]

The user types a RUN command to begin execution. The program always begins by executing the statement with the smallest statement number and executes by ascending statement numbers. An important point to remember about BASIC is that the values of simple variables are not automatically reset to zero when a program has been loaded, therefore, any variable whose value must be zero at some stage of the program should be initialized to guarantee the desired results.

If an escape is typed while the program is running, the current statement is finished and the message:

(ESC) AT <stn>

will be printed out. <stn> will be the number of the next statement to be executed. You may resume execution with the CON command.

Two consecutive escapes will normally return command to the Executive. However, if the current statement is an INPUT, one escape will only cause BASIC to wait for more input. Two must be used to return to the command level of BASIC, but this will bypass the (ESC) message. Attempts to continue execution will probably meet with failure. Typing any input value followed by an (esc) and (ret) will cause termination with the proper message.

Automatic Run

If the <fnm> is included in the RUN command, the file will be loaded into BASIC and execution will automatically begin when loading is completed.

If "BASIC <fnm>" is typed while in the Executive, BASIC will load the file and automatically run it. For example:

-BASIC "XYPLOT"

will call BASIC, which will load and run public file "XYPLOT". The file does not need to have a RUN command at the end to start running.

EXECUTE

<stn> EXECUTE

Include an EXECUTE statement somewhere in the program, if you wish the program to start running each time it is loaded.

SAVE

SAVE <fnm>

To store the currently loaded version of a BASIC program onto a permanent disc file the SAVE command is used. This must be done if you have made changes to a program and wish to save the corrected version of the program on a file.

If the user types a (ret) after SAVE instead of the <fnm>, BASIC responds with "ON>" and waits for a file name to be typed.

If there are any doubts about the current BASIC program, it would be best to list the program before saving it, since BASIC only writes the version of the program that is currently loaded.

CHAIN

[<stn>] CHAIN [<fnm>]

If the user types a (ret) after CHAIN instead of the <fnm>, BASIC responds with "FROM FILE>" and waits for a file name to be supplied.

This statement provides the user with the capability to chain several programs together as though they were one program. Although BASIC can now handle single programs of up to 12250 characters, programs larger than this may be run using the CHAIN statement. Perhaps more useful is the capability of breaking a large program into smaller chains and after examining the output of one chain, chaining to the other chains to perform additional calculations.

When the CHAIN statement is executed, the current program is deleted (like DEL ALL) and a new program is loaded. The values of all array variables and simple variables are preserved. Two restrictions on the use of the CHAIN statement are:

1) The contents of string variables (simple and array) will be destroyed when a CHAIN statement is executed. In other words, all string variables will be undefined at the beginning of a program chain. If the values of string variables need to be retained from one program chain to the next, then they must be written onto a disc file before the CHAIN is executed and read back into the variables at the beginning of the next chain. This problem will be fixed in a future version of BASIC.

2) All input and output files are closed before a new chain is loaded.

Chapter 4 - BASIC COMMANDS AND STATEMENTS

Commands

Individual statements without a statement number may be used as one-line programs or commands. These commands are executed immediately and the results stored in the computer or printed out. This capability makes BASIC extremely powerful as a desk calculator, able to evaluate complicated mathematical expressions.

LET
SET

[<stn>] LET <var> = <expr>
[<stn>] SET <var> = <expr>

If a variable is to be used as part of an expression in a BASIC statement, the user must be able to assign a value to the variable. The LET or SET commands are used to indicate replacement of the current value of a variable by a new value. In BASIC neither the words LET nor SET are required but either is accepted to retain compatibility with other systems. For example, the command:

```
X = 2 + 3
```

assigns the value 5 to variable X; the command:

```
SET Y = 10.2
```

assigns the value of 10.2 to the variable Y and:

```
LET Z = SQR (Y + X + .8)
```

assigns the value of 4 to the variable Z.

PRINT
TYPE

[<stn>] PRINT <olist>
[<stn>] TYPE <olist>

Numeric expressions in <olist> must be separated by a comma or a semicolon, while character strings, which are always enclosed between quote characters ('... '), may be separated from other strings or expressions by nothing, a comma or a semicolon.

TYPE is identical to PRINT except that TYPE is not used in other BASIC systems. Type has been added for FORTRAN II similiarity.

Zoned format

The use of a comma will result in BASIC spacing to the next multiple of 15 print positions before the next value (or character string) is printed. Each numeric value occupies

one print zone of 15 positions. There are five print zones per line. If all five zones have been printed, BASIC goes to the first print zone of the next line to print the next value. A character string is not restricted to 15 positions. The entire string is always printed; overprinting will occur at the right margin depending on the line length of the output device being used.

If a PRINT statement is terminated by a comma, the output of a (ret) is suppressed in printing.

```
>PRINT "FOR X = ",.5, TAN(.5)
FOR X =          .5              .5463025
```

There will always be at least one space between fields. If a field ends in the column adjacent to the column where the next field would normally begin, BASIC will space to the next zone.

```
>10 PRINT "12345678901234","X"
>20 PRINT "1234567890123 45","X"
>RUN
1234567890123 4 X
1234567890123 45                X
```

Packed format

The user may specify that output is to be printed in packed format by separating the expressions with a semicolon instead of a comma. The semicolon signals BASIC to type two spaces and then enough additional spaces to move to the next multiple of three print positions on the print line. One exception to this rule is the case in which a semicolon is used to terminate the print list. This serves only to suppress the output of a (ret) which BASIC normally types at the end of a print statement.

```
>PRINT "FOR X = ";.5;TAN(.5)
>FOR X =       .5     .5463025

>10 PRINT "THE BEGIN";
>20 PRINT "ING THE END"
>RUN
THE BEGINNING THE END
```

Compressed format

If two character strings (or an expression and a character string) appear in an output statement with neither a comma nor a semicolon separating them, BASIC will type them as closely together as possible; no extra spaces are inserted in the line. Remember that BASIC always leaves one space for the sign when outputting a numeric field.

```
>A = 3
```

```
>B = 5

>PRINT A"COMP""RESSED" B
 3COMPRESSED 5
```

Special Characters

In order to include control characters or function characters in a character string, it is necessary to know the "internal octal code" of the characters. A list of these octal codes is in the "Timesharing User's Guide" Having determined the octal code of any character, include that character by using the form &nnn where n is an octal digit. For example, the internal octal code of the bell character (control-G) is 147. To cause BASIC to ring the bell during program execution, use the statement PRINT "&147". This form for special characters may be mixed with other conventional characters in a string. When BASIC scans the string, the presence of the character & signals it to examine the next 1-3 characters to see if they are octal digits. If not, the & is treated like any other character. Control characters may also be indicated by & followed by a letter.

```
>PRINT "FOR X = &352";.5; "&352" TAN(.5)
FOR X =
                .5
                        .5463025
```

To space up a single line include a PRINT statement with nothing in its list. This may also be accomplished by including an &155 or &M in a string, as 155 is the internal octal form of a carriage return, which is the same as control-M.

Statements

If the user wants to execute a number of commands in a row without stopping each time to enter the command, then he composes a program. This program contains any number of statements. For example, consider the following statements for a program to calculate the hypotenuse of a right triangle.

```
>100 A = 4
>110 B = 3
>120 C = SQR(A^2 + B^2)
>130 PRINT "A= A, B= B, "C= "C
>RUN
```

Note that each statement has a unique statement number (which may be any integer in the range 1 through 99999). The presence of the statement number tells BASIC that these

statements are not to be immediately executed, but are to make up a program. When the RUN command is given, telling the computer to execute the program, the statements are executed one at a time in ascending numerical sequence. The result printed by the computer for the above example would be:

```
 A= 4            B= 3            C= 5
```

Although BASIC executes statements according to the numerical sequence of statement numbers, the statements of a program need not be prepared in numerical sequence. For example, the above program could have been prepared as:

```
130 PRINT "A=" A, "B=" B, "C=" C
100 A = 4
110 B = 3
120 C = SQR(A↑2 + B↑2)
```

and the results would have been identical.

GO TO

[<stn 1>] GO TO <stn 2>

<stn 1> is the optional statement number of the GO TO statement and <stn 2> is the statement number that is to be executed next.

As we have seen, BASIC executes the statements of a program in ascending numerical sequence by statement number. However, in writing programs, it is sometimes necessary to change the normal sequence of execution. This can be accomplished by using the GO TO statement.

```
100 GO TO 105
GO TO 215
```

IF THEN
IF GO TO

<stn 1> IF <rel-expr> THEN <stn 2>
<stn 1> IF <rel-expr> GO TO <stn 2>

It is often convenient to go to a statement only under certain conditions. This type of statement is called the IF (or conditional GO TO). This means, "If the relational expression is true, go to <stn 2>; otherwise (that is, if the relational expression is false), go to the next statement number in numerical sequence after <stn 1>". For example, if we want to say, "If X is greater than 5, go to statement 100", we would write

```
70 IF X > 5 THEN 100
```

Other examples are:

```
100 IF A = 10 GO TO 500
500 IF C(5) > 10 THEN 2300
2300 IF D <= E THEN 100
```

DATA
READ

<stn> DATA <cons 1> [,<cons 2>]...
[<stn>] READ <ilist>

Values can be assigned to variables in several ways. Using LET or SET is one method. Another method involves the combined use of the DATA and READ statements. All the constants that are to be assigned to variables throughout the program are written together in DATA statements. Each time a READ statement appears, the computer automatically assigns each constant in the DATA list to the corresponding variable in that READ statement. For example, the statements

```
100 READ A, B, C
200 DATA 1, 2, 3
```

would be equivalent to the statements

```
100 A = 1
150 B = 2
155 C = 3
```

Generally a program uses more than one value for a variable in order to prevent excessive use of constants and assignments. For example, consider the program:

```
10 G = 100
20 P = 20
30 D = G * P * .01
40 A = G - D
50 PRINT D, A
55 G = 150
60 P = 5
70 D = G * P * .01
80 A = G - D
90 PRINT D, A
```

Another way of writing this program using the READ and DATA statements is:

```
10 READ G, P
30 D = G * P * .01
40 A = G - D
50 PRINT D, A
60 READ G, P
```

```
70 D = G * P * .01
80 A = G - D
90 PRINT D , A
96 DATA 100, 20, 150, 5
```

Note that all the data that is to be assigned to G and P is now located in statement number 96.

Once all the data has been assigned by READ statements, additional READ statements will result in the error message "OUT OF DATA <stn>" where <stn> is the statement number of the READ statement.

RESTORE

<stn> RESTORE

RESTORE instructs the computer to reread DATA values beginning with the first DATA statement. Thus in the example below, the values for E, F and G (statement 35) are the same as for A, B and C because statement 35 follows a RESTORE statement.

```
>10 READ A, B, C
>15 PRINT "A=" A; "B=" B; "C=" C
>20 READ D
>25 PRINT "D=" D
>30 RESTORE
>35 READ E, F, G
>40 PRINT "E=" E; "F=" F; "G=" G
>45 DATA 1, 3
>50 DATA 5, 7, 9, 11, 13
>RUN
A= 1  B= 3  C= 5
D= 7
E= 1  F= 3  G= 5
```

INPUT
ACCEPT

[<stn>] INPUT <ilist>
[<stn>] ACCEPT <ilist>

To use the READ and DATA statements or the LET statement, the user must assign values to all variables when the program is written. To assign values to variables at execution time the user may use the INPUT statement. Each time the INPUT statement is encountered during execution, the program is halted and a ? is output to the terminal. At this time numbers separated by commas, spaces or (ret) must be typed in. The values are automatically assigned to the respective variables and execution is continued.

ACCEPT functions identically to INPUT except that ACCEPT is not compatible with other BASIC systems and has only been

added for FORTRAN II similarity.

The editing control characters &A, &W and &Q may also be used while entering data from the teletype. &A deletes the previous character, &W and &Q delete the current input field.

Now the sample program given for the READ and DATA statements could be written as:

```
>10 INPUT G, P
>20 D = G * P * .01
>30 A = G - D
>40 PRINT D, A
>50 INPUT G, P
>60 D = G * P * .01
>70 A = G - D
>80 PRINT D, A
>RUN
? 100,? 20
 20              80
? 150,? 5
 7.5             142.5
```

When the RUN command is given, BASIC executes the first INPUT statement and waits for the values of G, P and (ret). Upon receiving these, execution continues until the next INPUT statement.

Program loops

Note that the first four statement numbers of the sample program for the READ statement are exactly like the second four statements. This makes it possible to represent the program in the following way:

```
10 READ G, P
20 D = G * P * .01
30 A = G - D
40 PRINT D, A
50 GO TO 10
60 DATA 100, 20, 150, 5
```

The computer will perform statements 10 through 50 in the normal fashion, but after completing statement 50 it will go back to statement 10 and repeat statements 10 through 50. This process is repeated over and over until all data defined in the DATA statement (or any higher numbered DATA statement) has been used. At this time, the error message "OUT OF DATA 10" would be typed. This technique, often called a loop, is a very important programming capability.

The following example shows the statements necessary to set

up a loop to print all numbers between 1 and 100.

```
10 I = 1
15 IF I > 100 THEN 60
20 PRINT I;
45 I = I + 1
50 GO TO 15
60 PRINT "FINISHED."
```

First, a variable, I, was selected to be the counter. Second, an initial value of 1 was assigned to the counter variable. Third, the value of the counter variable was tested to see if it exceeded the upper limit of 100. Fourth, the value of the counter variable was increased each time the loop was repeated.

FOR
NEXT

<stn> FOR <var> = <expr 1> TO <expr 2> [STEP <expr 3>]
<stn> NEXT <var>

A second and more concise method of constructing program loops is to use the FOR and NEXT statements. The FOR statement assigns the value of <expr 1> to the variable, <var>, uses <expr 2> as an upper limit for the value of the variable and <expr 3> as the increment to be added to the variable when the NEXT statement is executed. The increment is assumed to be 1 if the optional [STEP <expr 3>] clause is omitted.

The NEXT statement must appear somewhere after the FOR statement. The variable must be exactly the same variable given in the FOR statement. The purpose of NEXT is to increment the value of the variable by <expr 3> and to compare its incremental value with the value that <expr 2> had when the FOR statement was first encountered. If the incremented variable is less than or equal to that value, BASIC interprets the NEXT statement as "GO TO the statement after the previous FOR statement". However, if the incremented value of the variable is greater then the initial value of <expr 2> , BASIC interprets the NEXT statement as "GO TO the next statement in numerical sequence after the NEXT statement".

The FOR loop is always executed at least once, even when <expr 1> is initially greater then <expr 2>.

Thus, using the FOR and NEXT statements, the program given above could also be written as:

```
10 FOR I = 1 TO 100
20 PRINT I;
30 NEXT I
```

```
40 PRINT "FINISHED."
RUN
```

Exactly the same looping procedure is followed; however, it happens automatically.

In this sample program the "body" of the loop consists of one statement, statement 20. The body of the loop may be any number of statements, but it is always terminated by the NEXT statement.

In some program loops it is necessary to increment the counter variable by a value other than 1. For example, to find and print all even numbers in the range 50 through 56, the following program could be used:

```
>10 FOR X = 50 TO 56 STEP 2
>20 PRINT X;
>30 NEXT X
>40 PRINT "&MFINISHED."
>RUN
 50    52    54    56
FINISHED.
```

The arguments in the FOR statement may be any expression so that:

```
100 FOR X = 1.5 TO -1.5 STEP -.1
```

is a valid statement. In this case the loop is repeated until <expr 1> has been reduced to a value equal to or less than <expr 2>. With a fractional argument, successive addition of the increment may not produce exact values of the variable. The above statement terminated with X = -1.4 while the statement:

```
100 FOR X = 1.5 TO -1.5 STEP -.5
```

terminated properly with X = -1.5.

It is often useful to have loops within loops. These "nested" loops can be expressed with FOR and NEXT statements. A loop is valid as long as the <var> used in one loop is not identical to <var> in a nested loop and the NEXT statement for the first loop does not fall between the FOR and NEXT statements of the second loop.

Arrays

The concept of subscripting and arrays becomes extremely useful in relation to programming loops. Consider the following table, which lists the quantity of each type of item sold by each of five salesmen in one week.

	Jones	Smith	Brown	Doe	White
Item 1	40	20	37	29	42
Item 2	10	16	3	21	8
Item 3	35	47	29	16	33

The price of each item is Item 1- $1.25, Item 2- $4.30 and Item 3- $2.50.

In the following discussion, the quantities of items in the first table are regarded as the two dimensional array Q(I,S) where I is the item number and S is the salesman. The prices of the items are regarded as the one dimensional array P(I) where I is the item number. The following program calculates the total sales in dollars for each salesman using data from the preceding tables:

```
>10 FOR I = 1 TO 3
>20 READ P(I)
>30 NEXT I
>40 FOR I = 1 TO 3
>50 FOR S = 1 TO 5
>60 READ Q(I,S)
>70 NEXT S
>80 NEXT I
>90 FOR S = 1 TO 5
>100 T = 0
>110 FOR I = 1 TO 3
>120 T = T + P(I) * Q(I,S)
>130 NEXT I
>140 PRINT "TOTAL SALES FOR SALESMAN";S;"$" T
>150 NEXT S
>200 DATA 1.25, 4.30, 2.50
>210 DATA 40, 20, 37, 29, 42
>220 DATA 10, 16, 3, 21, 8
>230 DATA 35, 47, 29, 16, 33
>RUN
TOTAL SALES FOR SALESMAN 1     $ 180.5
TOTAL SALES FOR SALESMAN 2     $ 211.3
TOTAL SALES FOR SALESMAN 3     $ 131.65
TOTAL SALES FOR SALESMAN 4     $ 166.55
TOTAL SALES FOR SALESMAN 5     $ 169.4
```

Statements 10 through 30 read in the values of the list P. **Statements** 40 through 80 read in the values of the table Q. **Statements** 90 though 150 compute T, the total sales for each of the five salesmen, and print each answer as it is computed.

DIM

<stn> DIM <let 1> (<expr> [,<expr>]) [,<let 2>]...

The DIM statement is used to provide storage for subscripted variables. A DIM statement may appear anywhere in the program. In some other BASIC systems, <expr> can not be an expression and must be a set value.

<expr> can take on any non-negative value. If the value of the expression is not an integer, the value is truncated to the next smaller integer value before the statement is executed.

DIM A(N) defines an array of N+1 elements: A(Ø), A(1), A(2)...A(N). DIM A(M,N) defines an array of (M+1) x (N+1) elements:

```
A(Ø,Ø)  A(1,Ø)  A(2,Ø)  ...  A(M,Ø)
A(Ø,1)  A(1,1)  A(2,1)  ...  A(M,1)
  .       .       .           .
A(Ø,N)  A(1,N)  A(2,N)  ...  A(M,N)
```

BASIC automatically establishes dimensions of A(1Ø) or A(1Ø,1Ø) for an array A, if a DIM statement does not contain A. The first usage of the array determines if the array will be singly or doubly dimensioned. Thus:

```
1Ø DIM A(1Ø,1Ø)
2Ø A(5,6) = 3
```

has exactly the same effect as:

```
1Ø A(5,6) = 3
```

If a previously dimensioned array is redimensioned to a smaller size, the memory difference is reclaimed by BASIC and used whenever needed. For example, if an array was dimensioned with 1Ø DIM A(3ØØ) and later by 9Ø DIM A(1ØØ); 2ØØ array locations would be reclaimed by BASIC for further use. It should be noted that continually redimensioning arrays is a very inefficient operation and should be avoided whenever possible.

QUIT

[<stn>] QUIT

When the QUIT statement is executed, control is returned to the Executive.

DEF

[<stn>] DEF FN<let> (<var>) = <expr>

where <let> must be A through Z and <var> must be an unsubscripted variable name. This variable is treated as a dummy argument and its value is not changed by a call to

the function. The expression may be any valid BASIC expression, but it can not contain a reference to another user-defined function.

The DEF statement permits the user to define his own arithmetic function. The DEF statement must be executed prior to the first call to the function in the program. The meaning of a function may be changed in a program and the interpretation of the function when it is called will be that defined in the most recently executed DEF statement.

```
>20 DEF FNL(X) = SIN(X*PI/180)
>30 Z = FNL(30)
>40 PRINT Z
>50 DEF FNL(X) = SQR(X*X + Y*Y)
>60 Y = 30
>70 S = FNL(40)
>80 PRINT S
>RUN
 .5
 50
```

The use of DEF is limited to those cases where the value of the expression can be computed within a single BASIC statement. Often much more complicated functions, or perhaps even sections of a program that are not functions, must be calculated at several places within the program. For this the GOSUB statement may be useful.

GOSUB
RETURN

<stn 1> GOSUB <stn 2>
<stn> RETURN

The GOSUB statement transfers control to <stn 2>. Statements are then executed in statement number order until a RETURN statement is encountered. Then the computer automatically returns to the statement immediately following the most recently executed GOSUB statement.

The following program will calculate the factorial of any number X that is input. Note that the subroutine is recursive, that is, GOSUB 110 calls itself.

```
> 5 DIM F(20)
>10 INPUT X
>20 IF X<1 THEN 10
>30 X = INT(X)
>40 MAT F = CON(X)
>50 MAT F = (-1) * F
>60 F(0) = 1
>70 F(1) = 1
```

```
>80 GOSUB 110
>90 PRINT X, F(X)
>100 GO TO 10
>110 IF F(X) > -1 THEN 180
>120 X = X - 1
>130 GOSUB 110
>140 X = X + 1
>170 F(X) = F(X-1) * X
>180 RETURN
>RUN
? 1
 1                 1
? 3
 3                 6
? 5
 5                 120
```

ON GOTO
ON GOSUB

<stn> ON <expr> GOTO <stn 1> [, <stn 2>]...
<stn> ON <expr> GOSUB <stn 1> [, <stn 2>]...

The effect of this statement is to execute a GOTO or GOSUB to the Ith statement number, where I is the truncated value of <expr>. This statement is used as a multi-branch GOTO or GOSUB statement.

```
10 ON I GO TO 20, 30, 40, 50, 60
20 ON N - INT(N/2) * 2 + 1 GOSUB 30, 90
```

In the last example, the GOSUB at statement number 30 will be executed if N is even and if it is odd the GOSUB at statement 90 will be executed.

Chapter 5 - FILE STATEMENTS

In addition to the terminal, data may be input from a file or written onto a file. By use of files, data may be set up beforehand or stored for use at any later date. Any file in the user's file directory or some public files may be used as input. A BASIC program may write on any file in the user's file directory.

OPEN

<stn> OPEN [#<fln>,] <fnm>, [<type>]<use>

where <fln> is an expression that must evaluate in the range 2 through 9, <fnm> is either the name of the file or an unsubscripted string variable name, <type> may be either SYMBOLIC or BINARY (default SYMBOLIC) and <use> is either INPUT or OUTPUT.

Before input or output to a file can be made in a BASIC program the file must have been opened by an earlier statement in the program for input or output by the use of an OPEN statement. An exception to this are files numbered Ø and 1 which are always open for terminal input and output, respectively.

If you only have two files, one input and one output, you do not need to specify [#<fln>,] since files 8 and 9 are assigned when no file number is given. File 8 is the default output file number and file 9 the default input file number. The following pairs of statements are equivalent:

```
5 OPEN #8, /SORT-OUT/, OUTPUT
6 OPEN /SORT-OUT/, OUTPUT

7 OPEN #9, /72-07/, INPUT
8 OPEN /72-07/, INPUT
```

To use more than one file either for input or output, all but one of the files must be opened by assigning a file number 2 through 7. The system only allows three disc files to be open at one time, but file numbers 2 through 9 are allowed for compatibility with FORTRAN-II. The following example opens three files for input, each with their own unique file number. The first uses the default number 9.

```
10 LET X = -1
15 OPEN /1/, INPUT
20 OPEN #1+ABS(X), /2INPUT/, INPUT
```

```
25 OPEN #3, /3/, SYMBOLIC INPUT
```

A file which is open may be reopened, in which case the file is first closed and then opened. This will reset the file's location counter to the first line in the file in much the same way the RESTORE statement resets the data statement pointer to the first data statement.

The two sets of statements:

```
1 OPEN /A/, INPUT
2 OPEN /B/, INPUT

1 OPEN #9, /A/, INPUT
2 OPEN /B/, INPUT
```

both first open file /A/. Since the default input file number is 9, the first statement of each group opens /A/ as file 9. But then the second statement opens file /B/ for input (as file 9, since no file number is specified), with the result that file /A/ will be closed and no longer accessible unless it is reopened.

Symbolic File

Input or output files are ordinarily symbolic. This means that all data is converted from internal machine representation to normal printing characters, or vice versa. Because of the conversion from machine representation, extra CPU time is used for writing or reading a symbolic file.

Binary File

To eliminate input/output conversion time, binary files should be used. Binary files do not convert data to normal printing characters, but instead, leave the data in the internal machine representation.

INPUT FILE
ACCEPT FILE

```
[<stn>] INPUT FILE [#<fln>,] <ilist>
[<stn>] ACCEPT FILE [#<fln>,] <ilist>
```

Once a file has been opened for symbolic input, the user may read from it by using the INPUT FILE statement. Each time the INPUT FILE statement is encountered during execution, the next value appearing on the file specified is read and assigned to the next variable in the list of variables in the INPUT FILE statement. If the input file was opened without specifying a file number or by specifying file number 9, then either of the following statements will read from the input file:

```
10 INPUT FILE A, B, C
20 INPUT FILE #9, A, B, C
```

If a file was opened by specifing a file number between 2 and 7, the INPUT FILE statement must include that same file number.

```
10 OPEN #6, /A/, INPUT
30 INPUT FILE #6, A, B, C
```

Only legal BASIC numbers may be read from a file (excepting into String variables as outlined in Chapter 7). The numeric fields in a data file may be separated by commas, spaces or (ret)s. Since BASIC treats spaces in a file as an "end-of-field" character, the user may not imbed spaces in the numeric fields. The only non-numeric characters allowed are: comma, space, E (for scientific notation), plus sign, minus sign and carriage return. A carriage return ends a line and is also a valid "end-of-field" character. A line consisting of spaces and a carriage return will be ignored.

PRINT FILE
TYPE FILE

[<stn>] PRINT FILE [#<fln>,] [<olist>]
[<stn>] TYPE FILE [#<fln>,] [<olist>]

Once a file is opened for symbolic output, the user may write on the file using the PRINT FILE statement. Each time the PRINT FILE statement is encountered during execution, the value of each expression appearing in the list of expressions is appended to the file specified for output in the same order that it appeared in the list of expressions. When a file is first opened for output, the file location pointer always points to the beginning of the file.

If the output file was opened without specifying a file number or by specifying file number 8, then either of the following statements will write on the output file:

```
10 PRINT FILE A, B, C
20 PRINT FILE #8, A, B, C
```

If a print statement is only to write a carriage return onto the file, the comma and output list may be omitted:

```
10 PRINT FILE #3
```

If a file was opened by specifying a file number between 2 and 7, then the PRINT FILE statement must include that same file number.

```
10 OPEN #5, /A/, OUTPUT
20 PRINT FILE #5, A, B, C
```

The file output list may also contain strings of characters enclosed in quotes.

```
15 PRINT A;B " POINTS,„DATA =" N
16 PRINT FILE #1, A;B " POINTS, DATA =" N
```

READ FILE
WRITE
WRITE FILE

[<stn>] READ FILE [#<fln>,] <ilist>
[<stn>] WRITE <expr> [, <expr>]...
[<stn>] WRITE FILE [#<fln>,] <expr> [,<expr>]...

Once a file has been opened for binary input or output, then the above statements will handle binary data in the same manner as the INPUT FILE and PRINT FILE.

IF EOF

<stn 1> IF EOF (<fln>) = <var> THEN <stn 2>

See EOF function for detecting end-of-file condition.

CLOSE

[<stn>] CLOSE [<fln>]

where <fln> is an expression. If no expression is given, or if the expression is negative, all files are closed. If the expression is 0 or 1, or if the file is already closed, nothing will happen. Otherwise, the file will be closed. Some examples are:

```
2 CLOSE
CLOSE -1
99 CLOSE 5
CLOSE (A + B) / 2
```

Chapter 6 - MATRIX STATEMENTS

Dimensions

In BASIC, there are a number of statements involving Matrix operations. Any array used in a MAT statement must have been dimensioned in a DIM statement. No default dimensions are assumed for matrix arrays, but once the dimensions have been established a MAT statement can change the dimensions of an array as long as they do not exceed the dimensions specified in the DIM. Maximum dimensions are about 56 by 56 for a two dimensional array or about 3152 elements for a one dimensional array.

There are several ways to establish the values of the elements in a matrix:

1) By reading values into the elements with any of the input statements.
2) By setting the elements to zero with the ZER statement.
3) By setting the elements to one with the CON or ONE statement.
4) By setting a matrix to the identity matrix with the IDN statement.

MAT READ

[<stn>] MAT READ <ilist>

The MAT READ statement will result in the input of all of the elements of a matrix. The values of the elements are processed row by row from a DATA statement. More than one matrix may be included in the input list but the list may not contain string variables or simple variables.

```
10 DATA 2,3,4,2,2,3,3,4,5,4,5,6,7,8,9
15 DIM A(3,3), B(3,2)
20 MAT READ A, B(2,2)
```

MAT INPUT
MAT ACCEPT

[<stn>] MAT INPUT <ilist>
[<stn>] MAT ACCEPT <ilist>

These statements function in the same manner as the MAT READ statement, except that input values are from the terminal instead of from DATA statements.

```
>15 DIM A(3,3), B(3,2)
>20 MAT INPUT A, B(2,2)
>RUN
?1,?2,?3,?4,?5,?6,?7,?8,?9,?10, ?11,?12,?13(ret)
```

MAT INPUT FILE [<stn>] MAT INPUT FILE <ilist>
MAT ACCEPT FILE [<stn>] MAT ACCEPT FILE <ilist>

These statements function in the same way as the MAT READ statement, except that input values are from a disc file previously opened for symbolic input.

If the file is binary, then the following statement is used:

MAT READ FILE [<stn>] MAT READ FILE <ilist>

MAT PRINT [<stn>] MAT PRINT <mlist>
MAT TYPE [<stn>] MAT TYPE <mlist>

These statements type the elements of a matrix on the terminal, row by row. Subscripts may not be included.

```
>10 DATA 2,3,4,2,2,3,3,4,5,4,5,6,7,8,9
>15 DIM A(3,3), B(3,2)
>20 MAT READ A, B(2,2)
>50 MAT PRINT A, B;
>RUN
 2             3             4
 2             2             3
 3             4             5
 4     5
 6     7
```

To get an extra blank line between the matrices use a string containing a carriage return. It is important here not to put a comma or semicolon after the string, because if you do, the first line of the second array will be indented a few spaces.

```
>10 DATA 2,3,4,2,2,3,3,4,5,4,5,6,7,8,9
>15 DIM A(3,3), B(3,2)
>20 MAT READ A, B(2,2)
>50 MAT PRINT A, "&M" B;
>RUN
 2             3             4
 2             2             3
 3             4             5

 4     5
 6     7
```

MAT PRINT FILE [<stn>] MAT PRINT FILE [#<fln>,] <mlist>
MAT TYPE FILE [<stn>] MAT TYPE FILE [#<fln>,] <mlist>

These statements are like MAT PRINT, except that output is to a file previously opened for symbolic output. If the file is binary, use either of the following statement forms:

MAT WRITE
MAT WRITE FILE

[<stn>] MAT WRITE <ilist>
[<stn>] MAT WRITE FILE <ilist>

ZER

[<stn>] MAT <var> = ZER[(<expr> [,<expr>])]

This statement sets all elements of the matrix to zero.

```
>10 DIM A(3,3)
>20 MAT A = ZER
>40 MAT PRINT A
>RUN
 0             0             0
 0             0             0
 0             0             0
```

If subscripts are used, the statement will set the dimensions to the values of the expression(s).

```
>10 DIM A(3,3), B(6)
>20 MAT A = ZER(2,2)
>30 MAT B = ZER(3)
>40 MAT PRINT A; &M" B
>RUN
 0     0
 0     0

 0
 0
 0
```

CON
ONE

[<stn>] MAT <var> = CON [(<expr> [,<expr>])]
[<stn>] MAT <var> = ONE [(<expr> [,<expr>])]

This statement sets all elements of the matrix to one. Like the ZER statement, if subscripts are used the statement will set the dimensions to the values of the expression(s).

```
>10 DIM A(3,3)
>20 MAT A = CON
>40 MAT PRINT A
>RUN
 1             1             1
 1             1             1
```

```
| 1              1              1
```

IDN

[<stn>] MAT <var> = IDN [(<expr> [,<expr>])]

This statement sets all elements of the principal diagonal of the square matrix to one and all off diagonal elements to zero.

```
|>10 DIM A(3,3)
|>20 MAT A = IDN
|>40 MAT PRINT A;
|>RUN
| 1      0      0
| 0      1      0
| 0      0      1
```

Substitution

[<stn>] MAT <var 1> = <var 2>

This statement sets the values of the elements of <var 1> to the values of the elements of <var 2>. If the dimensions of <var 2> are (M,N) then the dimensions of <var 1> are set to (M,N).

```
|>10 DATA 2,3,4,2,2,3,3,4,5,4,5,6
|>15 DIM A(3,3), C(4,3)
|>20 MAT READ A
|>25 MAT C = A
|>30 MAT PRINT C;
|>RUN
| 2      3      4
| 2      2      3
| 3      4      5
```

Constant Multiplication

[<stn>] MAT <var 1> = <expr> * <var 2>
[<stn>] MAT <var 1> = <var 2> * <expr>

The elements of a matrix may be multiplied, element by element, by a constant value resulting from the evaluation of any legal BASIC expression. The dimensions of <var 1> will be set to the dimensions of <var 2>. <var 1> need not be distinct from <var 2>.

Addition

[<stn>] MAT <var 1> = <var 2> {+,-} <var 3> Subtraction

Matrix addition or subtraction sets the elements of <var 1> equal to the sum or difference of the corresponding elements of two (not necessarily distinct) matrices, <var 2> and <var 3>. The two matrices to be added or subtracted

must have the same dimensions and <var 1> will have its dimensions reset.

```
>15 DIM A(3,3), B(3,3), C(3,3)
>20 DATA 2,3,4,2,2,3,3,4,5,4,5,6,7,8,9,10
>21 DATA 11,12,13,14,15,16,17,18
>25 MAT READ A, B
>30 MAT C = A + B
>35 MAT PRINT C, "&M"
>40 MAT C = A - B
>45 MAT PRINT C
>RUN
 6              8              10
 9              10             12
 13             15             17

-2             -2             -2
-5             -6             -6
-7             -7             -7
```

Multiplication

[<stn>] MAT <var 1> = <var 2> * <var 3>

Matrix multiplication uses three distinct matrices. If <var 2> has dimensions (M,N) then <var 3> must have dimensions (N,P). After multiplication the dimensions of <var 1> will have been reset to (M,P).

```
>10 DIM A(5,5), B(5,5), C(5,5)
>20 MAT READ A(3,2), B(2,4)
>30 MAT C = A * B
>40 MAT PRINT C;
>50 DATA 3,2,4,1,5,2,-5,7,9,6,4,5,3,1,2
>RUN
-7     31     33     20
-16    33     39     25
-17    45     51     32
```

TRN

[<stn>] MAT <var 1> = TRN(<var 2>)

This statement is used to calculate the transpose of a matrix. Both matrices must be distinct. If <var 2> has dimensions (M,N), then the TRN statement sets the dimensions of <var 1> to (N,M) and each element (I,J) of <var 1> is equal to the element (J,I) of <var 2>.

```
>10 DATA 1,2,3,4,3,4,3,4,5,6,7,8,9,10,11
>15 DIM A(5,5), C(5,5)
>20 MAT READ A(4,3)
>25 MAT C = TRN(A)
>40 MAT PRINT C;
```

```
>RUN
 1      4      3      6
 2      3      4      7
 3      4      5      8
```

INV

[<stn>] MAT <var 1> = INV(<var 2>)

This statement performs two calculations: the inverse and the determinent. <var 2> must be square and need not be distinct from <var 1>. The dimensions of <var 1> will be reset to the dimensions of <var 2>. The value of the determinent is stored in element (0,0) of <var 1>.

```
>10 DATA 1,2,3,4,3,4,3,4,5,6,7,8,9,10,11
>15 DIM A(5,5), B(5,5), C(5,5)
>20 MAT READ A(3,3)
>25 MAT B = INV(A)
>30 MAT C = A * B
>40 PRINT "DETERMINENT = " B(0,0)
>45 MAT PRINT B, C
>RUN
DETERMINENT =  4
-.25            .5             -.25
-2             -1               2
 1.75           .5             -1.25
 1             -.1455191E-10   -.2910383E-10
 .5820766E-10   1              -.2910383E-10
 0             -.2910383E-10    1
```

Sometimes when the original matrix is multiplied with its inverse, the result does not give a perfect identity matrix since the computer is limited to 11 significant digits. If, during inversion, an element of the matrix is reduced by a factor of 10 to the ninth or greater, the element will be set to an exact zero. If this element later appears as a pivotal element in the inversion process, execution of the program will be terminated and the error message "MATRIX NEARLY SINGULAR" will be typed.

Chapter 7 - STRINGS

BASIC has been extended to handle groupings of characters, called strings. Strings may now be used in INPUT or PRINT statements and some character by character manipulations may be done on strings.

Literal

"[<characters>]"

A string consisting of from 0 to 300 characters enclosed between quotation marks is called a literal.

```
10 PRINT "THIS IS A LITERAL STRING"
20 PRINT ""
```

Variable

<let>$[(<expr>)]

A group of characters may be assigned as the value of a variable which is then designated as a string variable. Note that numeric quantities may not be assigned to string variables. String variable names consist of a letter, A-Z, followed by a dollar sign. Initially the value of a string variable is the null string or, no characters.

```
10 A$ = "THE VALUE OF A IS THIS STRING"
20 B$=""
```

A string variable may be a singly dimensioned array. Double subscripts may not be used. If a subscripted string variable, A$, is referenced but has not been previously dimensioned, it is assigned the default dimension A$(10). If more then 11 locations are to be used the array must be dimensioned in a DIM statement.

```
10 DIM B$(15)
20 A$(8+1) = "NEED NOT BE DIMENSIONED"
30 B$(13) = "MUST BE DIMENSIONED"
40 B$((N+M) /2) = ""
```

Each element of a string array is a string and is independent of all other string variables.

```
>10 A$(1) = "EACH "
>20 A$(2) = "ELEMENT IS A"
>30 A$(3) = " UNIQ"
>40 A$(4) = "UE STRING"
>50 PRINT A$(2)
>RUN
```

```
|ELEMENT IS A
```

Expression

<operand> & <operand>

String expressions are similar to numeric expressions in that they are constructed by repeating the form <operand> & <operand>, where <operand> is either a string literal, string variable, or string function. & is the amperand string concatenation operator. String expressions, therefore, make it possible to form one string from multiple strings.

```
|>10 A$(1) ="EACH "
|>20 A$(2) = "ELEMENT IS A"
|>30 A$(3) = " UNIQ"
|>40 A$(4) = "UE STRING"
|>50 B$ = A$(1) & A$(2) & A$(3) & A$(4)
|>60 PRINT B$
|>RUN
|EACH ELEMENT IS A UNIQUE STRING
```

Notice that in all of the above examples on strings, the contents of the string variable have been assigned by a string assignment statement in the form:

<var> = <expr>

where <var> is any valid string variable name and <expr> is any valid string expression, be it a string literal or another string variable.

IF THEN
IF GOTO

[<stn 1]> IF <rel-expr> THEN <stn 2>
[<stn 1>] IF <rel-expr> GOTO <stn 2>

Strings may also appear in the <rel-expr> portion of the IF THEN statement. If the <rel-expr> is true, transfer will go to <stn 2>; otherwise, transfer will go to the statement following the IF THEN statement.

```
20 IF A$ = B$ THEN 10
25 IF A$(N+1) >= A$(N) GO TO 12
30 IF A$ & B$ < "LITERAL" THEN 100
```

Two strings are equal if they are of the same length and contain indentical characters. A string X is greater than Y if, during left-to-right character-by-character comparison, a character in X is higher in the collating sequence than the corresponding character in Y. If all charcters match throughout the length of the shorter string, then the longer string is greater. See Appendix A which lists the character collating sequence.

READ

String variables may be assigned by READing string literals from DATA statements. The following sequence of statements would assign the alphabetic month names to string array M$ and the corresponding numeric days of the month to the array, D.

```
10 DIM M$(12), D(12)
20 FOR I = 1 TO 12
30 READ M$(I), D(I)
40 NEXT I
50 DATA "JAN",31,"FEB",28,"MAR",31,"APR",30,"MAY",31
60 DATA "JUN",30,"JLY",31,"AUG",31,"SEP",30,"OCT",31
70 DATA "NOV",30,"DEC",31
```

INPUT
ACCEPT

Strings may also be inputed from the terminal by using the INPUT statement. The character strings may be entered in any of the following four ways.

```
"[<characters>]"
'[<characters>]'
[<characters>](&D)
[<characters>](ret)
```

With the first two input forms, BASIC will accept and store characters into the string variable until the ending double-quote (") (if the beginning delimiter was a double-quote) or single-quote character is entered. The delimiter quotes will not be included in the string. If the string being entered is not begun with quotes, it must be terminated with either a Control-D (&D) or a RETURN (ret).

```
>10 INPUT A$(1), A$(2), A$(3)
>20 INPUT A$(4)
>30 PRINT A$(1) "&M" A$(2)
>40 PRINT A$(3) "&M" A$(4)
>RUN
? "DOUBLE QUOTES"? 'SINGLE QUOTE'? CONTROL D(&D)
? CARRIAGE RETURN(ret)
DOUBLE QUOTES
SINGLE QUOTE
CONTROL D
CARRIAGE RETURN
```

The input list may be a mixed arrangement of string and numeric arrays.

```
30 INPUT A$, X, Y, A$(3)
```

INPUT FILE
ACCEPT FILE

Strings may also be input from a file previously opened for symbolic input with the INPUT FILE statement. The character strings may be entered in any of the following three ways.

```
"[<characters>]"
'[<characters>]'
[<characters>](ret)
```

The method of terminating a string with a control D does not work as with input from the terminal since the system regards it as the end-of-file.

PRINT
TYPE

Strings may be output to the teletype by using the PRINT statement. Literals may be included in the output list.

```
>10 DIM M$(12), D(12)
>20 FOR I = 1 TO 4
>30 READ M$(I), D(I)
>35 PRINT "MONTH" I "IS "M$(I)" WITH "D(I)" DAYS"
>40 NEXT I
>50 DATA "JAN",31,"FEB",28,"MAR",31,"APR",30,"MAY",31
>60 DATA "JUN",30,"JLY",31,"AUG",31,"SEP",30,"OCT",31
>70 DATA "NOV",30,"DEC",31
>RUN
MONTH  1 IS JAN WITH  31 DAYS
MONTH  2 IS FEB WITH  28 DAYS
MONTH  3 IS MAR WITH  31 DAYS
MONTH  4 IS APR WITH  30 DAYS
```

PRINT FILE
TYPE FILE

Strings may also be output to a file previously opened for symbolic output.

```
10 PRINT FILE A$, "EXAMPLE"
20 PRINT FILE #8, A$, "EXAMPLE"
30 PRINT FILE #6, A$, "EXAMPLE"
```

Strings are printed without the quote delimiters, so if there is more than one string per line and the strings are to be read later as BASIC input, then quotes will have to be added to differentiate between strings. Note that &002 is the internal value of a double quote character.

```
>10 OPEN /A/, OUTPUT
>20 PRINT FILE "&002FIRST&002&002SECOND&002"
>30 OPEN /A/, INPUT
>40 INPUT FILE A$, B$
>50 PRINT A$, &M B$
```

```
>RUN
FIRST
SECOND
```

CHANGE

[<stn>] CHANGE <string var> TO <var>
[<stn>] CHANGE <var> TO <string var>

In either CHANGE statement, <string var> will contain a string of length N, <var> will have the zeroth element equal to N and the first through Nth elements equal to the internal decimal code of the corresponding character of the string. <string var> must be eitner a subscripted or unsubscripted string name and <var> must be a singly subscripted array previously dimensioned by a DIM statement; no default values are available.

String to <var>

Changing a <string var> to a <var> stores the length of the string in the zeroth element of the <var> and the internal decimal value of each character of the string into successive elements of the array.

```
>5 DIM M(10)
>10 A$ = "String"
>15 CHANGE A$ TO M
>20 PRINT "LENGTH IS " M(0)
>30 PRINT "VALUES ARE " M(1); M(2); M(3); M(4)
>RUN
LENGTh IS  6
VALUES ARE  51    84    82    73
```

If the string is longer than the dimensioned length of the array, the message "ARRAY OUT OF BOUNDS" will be printed.

Presumably some manipulation of the string is desired, such as making sure that all letters are upper case. Since the value of a lower case character is between 65 and 90, it can be converted to upper case by subtracting 32.

```
>5 DIM M(10)
>10 A$ = "String"
>15 CHANGE A$ TO M
>20 FOR I = 1 TO M(0)
>30 IF M(I) < 65 THEN 45
>35 IF M(I) > 90 THEN 45
>40 M(I) = M(I) - 32
>45 NEXT I
>50 PRINT "NEW VALUES " M(1); M(2); M(3); M(4)
>RUN
NEW VALUES  51    52    50    41
```

<var> to string

Perhaps a string is desired from an array. To change an array to a string makes a string whose length is the value of the zeroth element of the array and converts the numbers in each element of the array to the next character of the string.

```
>5 DIM M(10)
>10 A$ = "String"
>15 CHANGE A$ TO M
>25 FOR I = 1 TO M(0)
>30 IF M(I) < 65 THEN 45
>35 IF M(I) > 90 THEN 45
>40 M(I) = M(I) - 32
>45 NEXT I
>55 CHANGE M TO B$
>60 PRINT "UPPER CASE A IS " B$
>RUN
UPPER CASE A IS STRING
```

OPEN

<stn> OPEN [#<fln>,] <fnm>, [<type>] <use>

The <fnm> file name in the OPEN statement can be replaced by an unsubscripted string variable, the value of which is the file name. If an unslashed file name is to be used, its name must be spelled out in full (no substituting "TEL" for "TELETYPE"). String expressions however, are not allowed in lieu of file names.

For example, assume that a program must sequentially read 3 files: /A/, /B/ and /C/. The following statements will open the files in order and transfer to statement number 200 when the last file has been opened:

```
10 A$(1) = "/A/"
20 A$(2) = "/B/"
30 A$(3) = "/C/"
40 N = 1
50 IF N > 3 THEN 200
60 A$ = A$(N)
70 OPEN #3, A$, OUTPUT
80 N = N + 1
90 PRINT FILE #3, "OPENED"
100 GO TO 50
110 QUIT
```

CHAIN

<stn> CHAIN [<fnm>]

The <fnm> file name in the CHAIN statement can be replaced by an unsubscripted string variable, the value of which is the file name, just as in the OPEN statement.

String Functions

DAT(X) This function will always return the current date as an 18 character string of the form YY/MM/DDbhh:mm:ssb where YY=year, MM=month, DD=day, hh=hours, mm=minutes, ss=seconds and b represents a blank. The variable name and value of the numeric argument, X, is unimportant, but must be present nevertneless.

```
>10 A$ = DAT(0)
>20 PRINT A$
>RUN
73/04/23 15:32:11
```

CHR(X) This function returns a one character string equivalent to the internal decimal value of the argument X. If X is less then 0 or greater than 255, the function returns the null string.

```
>PRINT CHR(55), CHR(21)
W              5
```

STR(X) This function will return a 2-13 character string equivalent to the value of its numeric argument, X. The string will contain 7 significant digits, a minus sign if negative and scientific notation if needed.

This function, in conjunction with function HED, would be useful if a numeric value needed to be printed vertically (along the left margin of a plot printout possibly).

```
>PRINT STR(PI*-1,0E-11), STR(5+10)
-.3141592E-10   15
```

HED(S) This function returns a one character string consisting of the first character of the argument string, S. The argument remains unchanged. If the argument string is null, then the function returns the null string.

```
>10 A$="TEST"
>20 PRINT HED(STR(PI*-1,E-11)),HED("STRING");HED(A$)
>RUN
-              S  T
```

TAL(S) This function returns a string consisting of all characters in the argument string, S, except the first character. The argument remains unchanged. If the argument is either null or only one character in length, TAL will return the null

string. The example shows how to print a vertical heading with either alpha or numeric values.

```
>10 A$="HEADING"
>15 B$ = STR(PI*-1.0E-11)
>20 FOR I = 1 TO 13
>25 C$ = HED(A$)
>30 A$ = TAL(A$)
>35 D$ = HED(B$)
>40 B$ = TAL(B$)
>45 PRINT C$, D$
>50 NEXT I
>RUN
H               -
E               .
A               3
D               1
I               4
N               1
G               5
                9
                2
                E
                -
                1
                0
```

LEN(S)

This function returns a numeric value equal to the number of characters in the argument string, S. If the string, S, is null then zero is returned.

```
>PRINT LEN(DATA(0)), LEN(TAL("X"))
 18              0
```

NUM(S)

This function returns a floating point value equivalent to its string argument, S. S must contain a string which has a numeric counterpart.

```
>PRINT STR(PI*-1.0E-11), NUM(STR(PI*-1.0E-11))
-.3141592E-10  -.3141592E-10
```

If the argument string is not valid (contains a bad character for instance), an error message will be displayed and .5798604E77 (positive machine infinity) will be returned.

```
>PRINT NUM("3.14$56")
BAD STR CHAR
 .5789604E+77
```

ASC(S) This function returns the internal decimal value of the first character of its string argument, S.

```
|>PRINT ASC("TEST"), ASC(CHR(51))
| 52             51
```

If the string is null, ASC returns the value -1; else it returns a value in the range Ø - 255.

Chapter 8 - FORMATTED INPUT/OUTPUT

Formatted Input/Output may also be done in BASIC. A line is formatted according to a picture statement.

Picture

<stn> :<picture string>

The colon denotes a picture statement, <picture string> is the actual picture or image by which the data is to be input or output. The picture string begins in the first character space following the colon.

Numeric

Pictures are divided into fields: numeric, string or literal. A numeric field consists of mainly libras (#), with an optional sign at the front, an optional decimal point somewhere in the libras, and an optional exponent (denoted by 4 exclamation points) at the end. If the optional exponent is used, the decimal point must appear somewhere in the libras. A numeric field may be terminated by a commercial at sign (@), which acts like a libra in that it reserves a place for a digit, but the @ will always be the last character of the field. Examples:

```
10 : #####
20 :+#####
30 : -####
40 : #.#
50:+#.##!!!!
60 :###.@+####
```

String

A string field consists of any number of dollar signs and may be terminated with a single quote ('), which marks the last character of a string field.

```
10 : $$$$$
20 :$$'$$$$
```

Literal

A literal field consists of any characters that do not signify numeric or string fields.

```
10 : this is a literal picture field
```

Note that double quotes (") are not used for literal fields in picture strings. Different field types may be mixed in a picture string.

```
10 :##.##!!!!That is numeric and this is literal
20 :$$$##$$String, numeric and literal mixed.
```

It is also possible to specify the picture string directly in the input or output statement instead of referencing another statement number. Any string expression may be used in place of the statement number and it will act as the picture string. For example, instead of using the following two statements:

```
10 :###@$$$$
20 INPUT USING 10, A, B$
```

Either of the following INPUT statements could be used and would be identical.

```
20 INPUT USING "###@$$$$", A, B$
30 F$(3) = "###@$$$$"
40 INPUT USING F$(3), A, B$
```

Runtime formats should be used carefully because they cause a substantial amount of overhead in CPU time each time the format is used.

INPUT USING
ACCEPT USING

[<stn 1>] INPUT USING <stn 2>, <ilist>
[<stn 1>] ACCEPT USING <stn 2>, <ilist>

where <stn 2> is the statement number of the picture string and <ilist> contains variables that may be either string or numeric. INPUT USING reads formatted input from the terminal.

Numbers

As many characters as the field is wide are read from the input line. Once the field has been read, it is converted to binary using the free field converter - the picture only reserves a particular width. Thus signs, exclamation points and decimal points in the picture have no effect.

```
>10 :###@+#@-#@#.@##.!!!!
>20 INPUT USING 10, A, B, C, D, E
>30 PRINT A; B; C; D; E
>RUN
? 3-5,-40,32,55E3
 3    -5    -40    .32   550
```

Strings

If the input variable is a string, the characters are stored as the value of the string.

```
>10:$$$'$$$$$$
>20 INPUT USING 10, A$, B$
>30 PRINT A$ &M B$
>RUN
? STRING
STRI
```

```
NG
```

Literals

If there is a literal string in the picture, as many characters in the input line will be skipped as the literal is long.

```
>10 :LITERAL$$$$
>20 INPUT USING 10, A$
>30 PRINT A$
>RUN
? SOME WILL BE MISSING
LL B
```

(ret)

A carriage return (&M or &155) appearing in the picture string skips to the beginning of the next input line. Input then proceeds according to the remainder of the picture.

```
>10 :##&M##
>20 INPUT USING 10, A, B
>30 PRINT A, B
>RUN
? 1234
5678
 12                56
```

```
>10 :##literal&M##
>20 INPUT USING 10, A, B
>30 PRINT A, B
>RUN
? 1234
5678
 12                56
```

```
>10 :##&155li##
>20 INPUT USING 10, A, B
>30 PRINT A, B
>RUN
? 1234
5678
 12                78
```

When the end of a picture is reached without the input list being exhausted, the picture is simply rescanned.

```
>10 :#
>20 INPUT USING 10, A, B, C
>30 PRINT A; B; C
>RUN
? 5710
 5     7     1
```

If a (ret) is read from the input line while getting a field, the field is immediately terminated. Thus INPUT USING may be used to read a whole line as a string. No trailing spaces will be appended.

```
>10 :$$$$$$$$$$$$
>20 INPUT USING 10, A$
>30 PRINT A$"IS ALL"
>RUN
? RETURN
RETURNIS ALL
```

This also means that if a (ret) is read in the middle of a numeric field, no zeros will be added at the end.

```
>10 :########
>20 INPUT USING 10, A
>30 PRINT A
>RUN
? 333
 333
```

When a (ret) is read on the input line before the input list is exhausted, all remaining input variables will be either zero or the null string.

```
>10 :###@###@$$$$
>20 INPUT USING 10, A, B, A$
>30 PRINT A, B, A$
>RUN
? 0007
 7              0
```

When the variable list is exhausted, any remaining characters on the current input line are skipped.

PRINT USING
TYPE USING

[<stn 1>] PRINT USING <stn 2>, <ilist>
[<stn 1>] TYPE USING <stn 2>, <ilist>

where <stn 2> is the statement number of the picture string and the variables in <ilist> may be either string or numeric. These statements write formatted output on the terminal.

Numbers

A numeric value may be output in the form of a rounded integer by using a field of multiple '#' characters, one '#' for each integer digit desired. The value will be right-justified in the indicated field.

```
>10 :####@###
>20 PRINT USING 10, 55, -3
```

```
>RUN
  55 -3
```

Numeric values may be output as fixed decimals by using the field form <integer>.<decimal>; where <integer> is a string of '#'s denoting places to the left of the decimal point and <decimal> is a string of '#'s indicating the number of places to the right of the decimal.

```
>5 A=1987.6
>10 :#####.#@##.##
>20 PRINT USING 10, A, 17.865
>RUN
 1987.6017.86
```

Scientific notation output format is denoted by including four exclamation points (!) or up arrows (↑) after a fixed point format. The exponent will be output as the character 'E' followed by either a plus or minus sign and two digits which represent the exponent.

```
>5 A = 1987.6
>10 :#.###!!!!###.##↑↑↑↑
>20 PRINT USING 10, A, A
>RUN
1.988E+0319 8.76E+01
```

In all of the above pictures, no explicit provision has been made for signs. When no sign is specified BASIC will output a minus sign if the number is negative. If the number is positive, no sign will be output. If it is desired to specify the sign, either a plus or minus sign may be put before the number format. If a plus sign is used, a plus sign will appear if the number is positive or zero and a minus sign will appear if the number is negative. A minus sign prints a space if the number is greater than or equal to zero; it prints a minus sign if the number is less than zero.

```
>10 :##.@###.#
>20 PRINT USING 10, -3, 5
>30 :+#.@+#.#
>40 PRINT USING 30, -3, 5
>50 :-#.@-#.#
>60 PRINT USING 50, -3, 5
>RUN
-3.0 5.0
-3.0+5.0
-3.0 5.0
```

Strings

String variables and expressions may be output with a picture format by using a field consisting of multiple

dollar signs ($). The string value will be left-justified in the field with trailing blank fill if the string length is less than the field width. If the string length is greater than the field width, only the left-most string characters will be output.

```
>10 A$ = "HI"
>20 B$ = A$ & " HOW ARE YOU"
>30 :$$$$$$'$$$$$
>40 PRINT USING 30, A$, B$
>RUN
HI     HI HO
```

Character strings within a picture will be output literally unless there is an ampersand (&) in the string. If the & is followed by a 1-3 digit octal number, then the corresponding character will be output. The ampersand may also be followed by letter A - Z, in which case the two characters will be replaced by the corresponding control-character.

```
>10 :$$$$$$$$$$$$$$$$$$$$$$$$$$
>20 PRINT USING 10, "THIS WI&54L BE O&MN 2 LINES"
>RUN
THIS WILL BE O
N 2 LINES
```

(ret)

When a picture does not contain as many numeric or string fields as the list contains variables, the picture is simply repeated from the beginning. No (ret) is output when the picture is repeated; but the user may include a &M or &155 at the end of the picture and the (ret) will be output only when the picture is repeated.

```
>10 PRINT USING 30, 1, 2, 3, 4, 5
>20 PRINT USING 40, 1, 2, 3, 4, 5
>30 :#
>40 :#&155
>RUN
1 2345
1
2
3
4
5
```

BASIC normally terminates the print line with a (ret) after the last expression has been written. However, if the last expression in the PRINT USING list is followed by a comma, BASIC will not output the (ret).

INPUT FILE USING [<stn 1>] INPUT FILE [#<fln>,] USING <stn 2>, <ilist>
ACCEPT FILE USING [<stn 1>] ACCEPT FILE [#<fln>,] USING <stn 2>, <ilist>
PRINT FILE USING [<stn 1>] PRINT FILE [#<fln>,] USING <stn 2>, <ilist>
TYPE FILE USING [<stn 1>] TYPE FILE [#<fln>,] USING <stn 2>, <ilist>

By using the above statements, input or output may be formatted when used with a file previously opened as symbolic.

MAT INPUT USING [<stn 1>] MAT INPUT USING <stn 2>, <ilist>
MAT ACCEPT USING [<stn 1>] MAT ACCEPT USING <stn 2>, <ilist>
MAT PRINT USING [<stn 1>] MAT PRINT USING <stn 2>, <ilist>
MAT TYPE USING [<stn 1>] MAT TYPE USING <stn 2>, <ilist>

Formatted matrix input or output can use the terminal by following the same rules as for formatted input/output to the terminal.

In addition, the MAT PRINT USING statement will cause the picture to be rescanned as often as is necessary to print a matrix row. A (ret) will be output after each row.

```
>10 DIM X(3,5)
>20 MAT READ X
>30 DATA 11,12,13,14,15,21,22,23,24,25
>35 DATA 31,32,33,34,35
>40 L$ = "=========================="
>50 PRINT L$
>60 MAT PRINT USING 80, X
>70 PRINT L$
>80 :##.# ##
>RUN
==========================
11.0 12  13.0 14  15.0
21.0 22  23.0 24  25.0
31.0 32  33.0 34  35.0
==========================
```

MAT INPUT FILE USING [<stn 1>] MAT INPUT FILE [#<fln>,] USING <stn 2>, <ilist>
MAT ACCEPT FILE USING [<stn 1>] MAT ACCEPT FILE [#<fln>,] USING <stn 2>, <ilist>
MAT PRINT FILE USING [<stn 1>] MAT PRINT FILE [#<fln>,] USING <stn 2>, <ilist>
MAT TYPE FILE USING [<stn 1>] MAT TYPE FILE [#<fln>,] USING <stn 2>, <ilist>

Formatted matrix input/output may be to a file previously opened for symbolic output by following the above rules for formatted matrix output.

Chapter 9 - ERROR MESSAGES

Note that error messages are followed by the statement number of the last statement executed before the error.

ARRAY DIM TOO LARGE	The size of an array dimensioned by a DIM statement is too large for the available storage.
ARRAY OUT OF BOUNDS	You are trying to place values into an array where the subscript is bigger than the dimensioned value. The dimensioned value may be either a default value or an assigned value in a DIM statement.
BAD EXPONENT	The exponent of a number has an illegal character in it.
BAD FILE NAME	A file name that is not in your directory or a non-existent public file name follows the command LOAD.
BAD INPUT CHARACTER	You typed a character that is illegal in a numeric field in response to an input statement. Retype the field.
BAD STR CHAR	The function NUM was called with a string which has a character that is illegal in a numeric field. Positive machine infinity is return and the program continues running.
CHANGE STMT ILLEGAL	The variable name used in a CHANGE statement is not a valid array name.
DIRECT ONLY	You may not insert a command like RUN, LIST, DEL, LOAD or SAVE into a program as a program statement.
ERR IN FORMAT	There is a runtime format processing error.
ERROR IN PICTURE	This error occurs from a PRINT USING statement where the picture statement number is a string or string variable and that string contains a format error such as only three !'s. This error also occurs if a formatted input/output statement references a statement which is not a picture.
EXCESSIVE GOSUB NESTING	Too many GOSUB statements without executing a RETURN statement. The maximum number allowed is calculated by the following:

	(<no of FOR>) * 7 + (<no of GOSUB>) <= 2048 It may mean that subroutines are being exited by GOTO or IF-THEN statements instead of RETURN.
FILE AT EOF	Attempt to read past the end of the file on a READ FILE or INPUT FILE statement.
FILE CLOSED	Attempt to read or write on a file that has not been opened by an OPEN statement.
FILE TYPE WRONG	A file was opened as symbolic and you are trying to use it as binary or else it was opened as binary and you are trying to use it as symbolic.
FILE USE WRONG	You are trying to write on a file that was opened as input or else you are trying to read from a file that was opened as output.
FORMAT NOT FOR NUMBER	You are attempting formatted reading/writing while designating a numeric variable in the statement, but indicating a non-numeric field in the picture.
FORMAT NOT FOR STRING	You are attempting formatted reading/writing while designating a string variable in the statement, but indicating a non-string field in the picture.
FUNCTION ILLEGAL	You are trying to call a BASIC function with an incorrect name or you are using a three character name for a subscripted variable and the computer thinks it is a function.
ILLEGAL	The computer does not recognize what you have typed as a legal BASIC statement or command. Sometimes it can tell you what character is wrong. The statement needs to be retyped.
IMPROPER NESTING IN FOR LOOP	You have either a FOR or a NEXT for a second loop inbetween the FOR-NEXT of the first loop. It could also mean more than one NEXT for a single FOR.
INCOMPLETE	You did not finish the command or statement.
INCONSISTENT DIMENSIONS	The dimensions of two matrices to be added or subtracted are not identical. It could also be that you are trying to multiply two matrices with the first having dimensions (M,N) and the second not having dimensions (N,P).
LINE NUMBER REQUIRED	The command you have just typed is not a valid command and must be a program statement.

LINE NUMBER TOO BIG	You have used a statement number that is greater than 99999.
LINE TOO LONG	You are trying to enter a statement which exceeds the maximum number of characters allowed in a line. This number varies but will be less than 90.
LOG OF NUMBER <= Ø	THE argument of a LOG function is non-positive.
MATRICES NOT INDEPENDENT	Matrix multiplication requires three distinct matrices and yours may not be distinct. The MAT TRN statement also requires distinct matrices and may be your source of trouble.
MATRIX STATEMENT ILLEGAL IN	Illegal syntax in a MAT statement.
MATRIX NEARLY SINGLULAR	A MAT INV statement has encountered a matrix with zero or nearly zero pivotal elements. The matrix being inverted is singular or nearly so.
NEGATIVE SUBSCRIPT	The expression in a subscripted variable was negative.
NO PROGRAM	It is possible to get this from a RUN, LIST or DELETE command. Perhaps you forgot to load it from a file or maybe you accidentally deleted it.
NO STMT.	You have entered a DEL command referencing a statement number that does not exist.
NUMB READ FOR STRING	A number was input where the READ statement expected a string variable.
NUMBER ILLEGAL	The computer is interpreting something in the statement as a number that is illegal or in the wrong place. Perhaps you have left out an arithmetic operator or you concatenated a number to form a string variable.
NUMBER TOO BIG	The argument of the INT function may be greater than the largest acceptable integer or the file number specified by an input or output statement is greater than 9.
OUT OF DATA	A READ statement for which there is no data has been encountered. This may mean a normal end of your program and should be ignored. Otherwise, it means that you have not supplied enough data in DATA statements.
OUT OF MEMORY	Program (or arrays in it) is too big and no storage is available (22,800 characters available), or FOR loops

	nested too deep. The maximum number is figured according to: (<no of FOR) * 7 + (<no of GOSUB>) <= 2048
OVERFLOW DURING INPUT	A number larger than 5.78960E76 has been input. The computer supplies machine infinity and continues running the program.
RANGE ERROR	The expression in an ON-GOTO or ON-GOSUB statement evaluates to I where I does not have a corresponding statement number. Therefore there is no statement to transfer control to. Another cause can be the argument of the EOF function is greater than 9.
RETURN BEFORE GOSUB	The computer cannot execute a RETURN before it has executed a GOSUB because the GOSUB sets up the place to which the RETURN statement must go.
SQRT OF NEG NUMBER	The argument of the SQR function is negative. The computer returns the square root of the absolute value of the argument and continues running.
STNG BAD	The computer is interpreting something in the statement as a string that is illegal or in the wrong place, such as a string on the right hand side of a numeric expression.
STRING READ FOR NUMB	A string was input where the READ statement expected a numeric value.
STRING TOO LONG	You are trying to input a string which is more than 300 characters. Another possible reason for this message is that you are concatenating two or more strings and the resulting string is more than 300 characters.
UNABLE TO OPEN FILE	You have an OPEN statement that refers to a file name that does not exist.
UNDEF LINE NO IN	The statement number appearing in a GOTO, GOSUB, ON GOTO, ON GOSUB, or IF-THEN statement does not appear as a number for a statement in the program.
UNDEFINED FUNCTION	You are calling a programmer defined function that you have forgotten to define or else the DEF statement is after the statement where you use the function.
UNDEFINED MATRIX	A matrix has been used on the right hand side of a MAT statement before having values assigned to its elements.

UNDEFINED SUBSCRIPTED VARIABLE	The program is trying to use a subscripted variable that has no previous value. Before any variable can appear on the right hand side of an assignment statement, it must first appear on the left side of one, or in an input statement.
UNDERFLOW DURING INPUT	A number in absolute size smaller than 4.31809E-78 has been input. The computer supplies 0.863616E-77 and continues running.
VARIABLE ILLEGAL	There are some places where certain variable names are not allowed: 1) subscripted variables as the iterative variable in a For statement, i.e. FOR A(I) = ... 2) subscripted variables as the dummy variable in a DEF statement, i.e. DEF FNA(A(0)). 3) a letter-number combination variable name of a subscripted variable, i.e. DIM A1(50).
WRONG NUMBER SUBSCRIPTS	A one-dimensional array was referenced with two dimensions or a two-dimensional array was referenced with one dimension.
0 STEP SIZE IN FOR LOOP	The increment value of a FOR loop contains a zero value.

XDS 940 and CDC 3800 Character Codes

XDS 940 ASCII					CDC 3800 BCD					
INT DEC	INT OCT	EXT OCT	TELE TYPE	LINE PRINTER	INT DEC	INT OCT	EXT OCT	CARD PRINT	CARD PUNCH	LINE PRINTER
00	000	040	sp	SP	48	60	20	sp	sp	SP
01	001	041	↓	▵	42	52	52	nd	-0	∨
02	002	042		'	29	35	75	nd	+58	≥
03	003	043	#	▵	62	76	36	nd	068	≡
04	004	044	$	≠	43	53	53	$	-38	$
05	005	045	%	▵	14	16	16	nd	68	%
06	006	046	&	▵	63	77	37	nd	078	∧
07	007	047	'	'	12	14	14	'	48	≠
08	010	050	(	(	60	74	34	(	048	(
09	011	051	)	)	28	34	74	)	+48	)
10	012	052	*	*	44	54	54	*	-48	*
11	013	053	+	+	16	20	60	+	+	+
12	014	054	,	,	59	73	33	,	038	,
13	015	055	-	-	32	40	40	-	-	-
14	016	056	.	.	27	33	73	.	+38	.
15	017	057	/	/	49	61	21	/	01	/
16	020	060	0	0	00	00	12	0	0	0
17	021	061	1	1	01	01	01	1	1	1
18	022	062	2	2	02	02	02	2	2	2
19	023	063	3	3	03	03	03	3	3	3
20	024	064	4	4	04	04	04	4	4	4
21	025	065	5	5	05	05	05	5	5	5
22	026	066	6	6	06	06	06	6	6	6
23	027	067	7	7	07	07	07	7	7	7
24	030	070	8	8	08	10	10	8	8	8
25	031	071	9	9	09	11	11	9	9	9
26	032	072	:	:	10	12	00	:	82	:
27	033	073	;	;	31	37	77	nd	+78	;
28	034	074	<	<	26	32	72	nd	+0	<
29	035	075	=	=	11	13	13	=	38	=
30	036	076	>	>	47	57	57	nd	-78	>
31	037	077	?	▵	13	15	15	nd	58	<

XDS 940 ASCII					CDC 3800 BCD					
INT DEC	INT OCT	EXT OCT	TELE TYPE	LINE PRINTER	INT DEC	INT OCT	EXT OCT	CARD PRINT	CARD PUNCH	LINE PRINTER
32	040	100	@	↓	61	75	35	nd	058	←
33	041	101	A	A	17	21	61	A	+1	A
34	042	102	B	B	18	22	62	B	+2	B
35	043	103	C	C	19	23	63	C	+3	C
36	044	104	D	D	20	24	64	D	+4	D
37	045	105	E	E	21	25	65	E	+5	E
38	046	106	F	F	22	26	66	F	+6	F
39	047	107	G	G	23	27	67	G	+7	G
40	050	110	H	H	24	30	70	H	+8	H
41	051	111	I	I	25	31	71	I	+9	I
42	052	112	J	J	33	41	41	J	-1	J
43	053	113	K	K	34	42	42	K	-2	K
44	054	114	L	L	35	43	43	L	-3	L
45	055	115	M	M	36	44	44	M	-4	M
46	056	116	N	N	37	45	45	N	-5	N
47	057	117	O	O	38	46	46	O	-6	O
48	060	120	P	P	39	47	47	P	-7	P
49	061	121	Q	Q	40	50	50	Q	-8	Q
50	062	122	R	R	41	51	51	R	-9	R
51	063	123	S	S	50	62	22	S	02	S
52	064	124	T	T	51	63	23	T	03	T
53	065	125	U	U	52	64	24	U	04	U
54	066	126	V	V	53	65	25	V	05	V
55	067	127	W	W	54	66	26	W	06	W
56	070	130	X	X	55	67	27	X	07	X
57	071	131	Y	Y	56	70	30	Y	08	Y
58	072	132	Z	Z	57	71	31	Z	09	Z
59	073	133	[	[	15	17	17	nd	78	[
60	074	134	\	\	30	36	76	nd	+68	¬
61	075	135	]	]	58	72	32	nd	028	]
62	076	136	↑	^	45	55	55	nd	-58	↑
63	077	137	_	←	46	56	56	nd	-68	↓

NOTES

sp= space, blank or no punch.
nd= not defined, varies with particular unit used.
: (colon) is not transmitted on BCD (even parity) tape, changed to 0.

XDS 940 ASCII Character Codes

INT DEC	INT OCT	ASCII OCT	TELE TYPE	940 USE	INT DEC	INT OCT	ASCII OCT	TELE TYPE	940 USE
064	100	140	`	`	096	140	000	NULL	Blank tape
065	101	141	a	a	097	141	001	SOM	&A Delete character
066	102	142	b	b	098	142	002	EOA	&B
067	103	143	c	c	099	143	003	ETX	&C
068	104	144	d	d	100	144	004	EOT	&D End function
069	105	145	e	e	101	145	005	ENQRY	&E
070	106	146	f	f	102	146	006	ACK	&F
071	107	147	g	g	103	147	007	BELL	&G Ring bell
072	110	150	h	h	104	150	010	Bksp	&H
073	111	151	i	i	105	151	011	Tab	&I
074	112	152	j	j	106	152	012	LF	&J Line feed
075	113	153	k	k	107	153	013	VT	&K Vertical tab
076	114	154	l	l	108	154	014	FORM	&L Page eject
077	115	155	m	m	109	155	015	CR	&M Carriage return
078	116	156	n	n	110	156	016	SO	&N
079	117	157	o	o	111	157	017	SI	&O
080	120	160	p	p	112	160	020	DLE	&P
081	121	161	q	q	113	161	021	DC1	&Q Paper tape on
082	122	162	r	r	114	162	022	DC2	&R
083	123	163	s	s	115	163	023	DC3	&S Paper tape off
084	124	164	t	t	116	164	024	DC4	&T
085	125	165	u	u	117	165	025	NAK	&U
086	126	166	v	v	118	166	026	SYNC	&V Next chr literal
087	127	167	w	w	119	167	027	ETB	&W Delete word
088	130	170	x	x	120	170	030	CANCL	&X
089	131	171	y	y	121	171	031	EM	&Y
090	132	172	z	z	122	172	032	SUB	&Z
091	133	173	{	{	123	173	033	Prefix Escape	
092	134	174	|	|	124	174	034	FS	
093	135	175	}	mb	125	175	035	GS	
094	136	176			126	176	036	RS	
095	137	177	DELETE	eof	127	177	037	US	

Eofwd=27657537B (3 137's)

NOTES

mb=multiple blank character.
esc=escape or prefix.
eof=end of file character.

NOTES

&A, &B, etc. means Control A, Control B, etc.

TTY names (NULL, SOM, etc.) are standard communication names.

Prefix is used on model 37 Teletypes for special functions; it can be input only with &V in QED. The characters following the prefix perform the following functions: 1=set horizontal tab, 2=clear horizontal tabs, 3=shift to red ribbon, 4=shift to black ribbon, 5=set vertical tab, 6=clear vertical tab, 7=reverse line feed, 8=half reverse line feed, 9=half forward line feed.

The multiple blank character (135B) will always be followed by another character whose octal value will be used as a blank (space) count. Thus, if the characters 135B 12B are read from a symbolic file, it means that 10 spaces in a row were stored in that file.

End of file character (137B) will generally be read as the last character in a file. However, it is possible to have 137B's in the middle of a file. Therefore, any time a 137B is encountered, the file word should be tested for the EOF bit (same as the sign bit) being set to make sure the true end of file has been reached.

Form feed (&L or 154B) should be followed by at least 6 (for model 33 and 35 Teletypes) or 45 (for model 37's) non-printing characters to allow time for the page to stop at the top of the next form (up to three seconds for a 37).

Special Characters

INT DEC	INT OCT	ASCII OCT	TELE TYPE	940 USE
221	335	375	}	Right brace
234	352	212	LF	Line feed with no return
237	355	215	CR	Carriage return with no line feed

352B and 355B are used to produce special output, such as plotting or underlining. They inhibit the normal carriage return and line feed processing of the monitor, which normally puts out a carriage return, rubout, line feed when a carriage return is output; and a line feed, carriage return, rubout for a line feed. These characters can be put in a file by using QED. In QED, the control-shift-L (&\) character will put in single line feed character (352B), and control-shift-M (&]) will put in a single carriage return character. Note that these two characters are treated exactly like any normal character in QED. That is, they are not considered to mark the end of a logical line or terminate the edit mode.

335B is used to produce a right brace on output. It is not possible to output this character as 135B, because that is the multiple blank character. 335B may be entered into a file using &V in QED (Version 6.0 or higher.)

Acknowledgment

The BASIC language and compiler were originally developed at Dartmouth College for time-sharing computer users with no previous knowledge of computers, as well as for users with considerable programming experience. A simple straightforward language, BASIC closely resembles standard mathematical notation.

BASIC was orginally supplied by Scientific Data Systems, now Xerox Data Systems. Modifications to the BASIC language have been sponsored by the NOAA Computer Division, Boulder, Colorado. The matrix operations were introduced in 1968 by Dr. R. J. Slutz, J. R. Winkelman and Thomas B. Gray of the NOAA Space Disturbances Laboratory. Strings and formatted input/output were added by Howard E. Bussey, Jr. in 1972.

The first BASIC Manual was also originally supplied by Scientific Data Systems, then rewritten completely by Thomas B. Gray. Supplements were issued for Matrix operations and extensions to the language. This manual is a complete revision which supercedes all previous manuals and supplements.