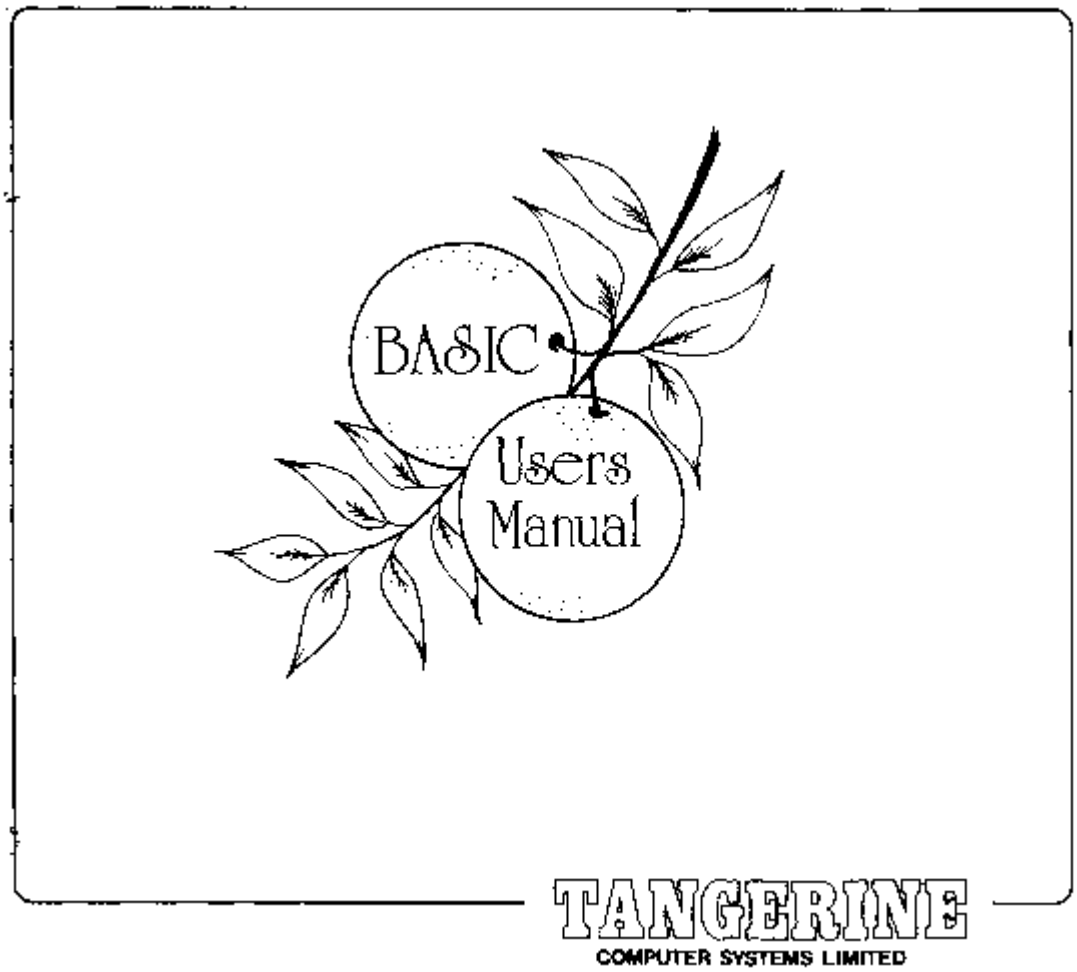
BASIC Users Manual



FOREWORD

This manual describes the 10K Microsoft BASIC available for the Microtan system and describes the enhancements to this package by TANGERINE. Microsoft BASIC is recognised as being "standard" BASIC and as such it is possible to run the many published programs for computers supporting BASIC. Microsoft BASIC for the Microtan consists of three Read Only Memories (ROM) which must be inserted into the TANEX board. Cassette SAVE and LOAD commands are available and use filenames and may be operated at two speeds, 300 Baud and 2400 Baud. Please note that XBUG must be used if you wish to use the SAVE and LOAD commands.

CHAPTER 1: Microtan's 10K Extended MICROSOFT BASIC

STARTING BASIC

To start BASIC, enter TANBUG by pressing BREAK or issuing a reset, then type:

GE2ED<CR> (<CR> = Carriage Return) (note1)

BASIC responds with:

MEMORY SIZE?

Type either <CR> in which case BASIC calculates available memory space automatically or

<Decimal Number><CR>

where Decimal Number is a number representing the amount of store which you require BASIC to use. Store is then allocated from location 400 HEX upwards to the limit specified. The remaining store may be used to contain, for example, user subroutines.

Beware of typing in a number larger than the area you have available - Microsoft Basic uses the top part for variables and print statements, which will be assigned the wrong value.

BASIC next prompts:

TERMINAL WIDTH?

For the MICROTAN VDU, responds with <CR> since the program looks after VDU scrolling.

BASIC responds with:

OK

You can now use BASIC.

Exiting from BASIC

To exit from BASIC and return to TANBUG it is necessary to cause a microprocessor reset by pressing BREAK on the ASCII keyboard. The message

TANBUG

is issued, followed by the cursor prompt.

Note that when you exit from BASIC, all your program information is destroyed (note 2). You should therefore SAVE any programs required before exiting.

INSTALLATION

The minimum system requirement to run BASIC is as follows: MICROTAN 65 and TANEX connected via mini-motherboard or system motherboard, ASCII keyboard, 2K RAM (1K on MICROTAN, 1K on TANEX).

If you wish to use the cassette SAVE and LOAD functions, XBUG must be present in socket G2 on TANEX.

Procedure:

1. Install TANEX as per instructions in TANEX manual.
2. If SAVE and LOAD are required, install the XBUG EPROM as per the XBUG manual.
3. Ensure power is off
4. Install the ROM marked BASL in TANEX socket J2.
5. Install the ROM marked BASM in TANEX socket H2.
6. Install the EPROM marked BASH in TANEX socket D3.
7. Power up the system and refer to the instructions for starting BASIC.

PROGRAM STORAGE USING CASSETTES

Note that in order to use this facility, the XBUG PROM must be resident in slot G2 of TANEX. If it is not, reference to these two commands will cause BASIC to crash with resultant loss of your current program. Cassette record and replay levels must have been set up in accordance with the instructions given in the XBUG manual.

These routines dump the BASIC code in its machine format, that is it is not in text form but in the packed-up form in which it is stored. Thus a considerable time saving is achieved during SAVE and LOAD functions. It is not possible to dump selected parts of programs, nor is it possible to load in additional lines to an existing program since memory is always cleared before a load occurs. To save the program which is in memory type:

SAVE<CR>

BASIC responds with:

FAST?


If you require the dump to be at fast speed, type:

Y<CR>

otherwise just type <CR> to obtain slow speed.

BASIC responds with:

FILENAME?

Type in the required filename - this can be up to 6 alphanumeric characters - start the cassette recorder in record mode, and type carriage return, for example:

FILE<CR>

BASIC appends a .B to this name and responds with:

FILE.B

The file is now dumped on to cassette, BASIC responding with:

OK


when the dump is complete.

You may return to BASIC from the SAVE function at any time up to typing the final carriage return after the filename by typing CTRL C.The SAVE is then no executed. However, once you have started the SAVE, you cannot return to BASIC until the SAVE is complete.

To recover a program dumped to cassette, type:

LOAD<CR>

BASIC responds with:

FAST?

Type

Y<CR>

if the required file was recorded at fast speed, carriage return otherwise. BASIC responds with:

EXAM?


Type

Y<CR>

if you wish to compare what is on tape with what is in memory, carriage return only if you wish the tape to actually read into memory (see section on errors for result of an EXAM call). BASIC responds with:

FILENAME?


Type in the filename that was given to the file when it was saved, omitting the .B, followed by carriage return. For example:

FILE<CR>

BASIC responds with the filename it is looking for:

FILE.B

Start the cassette recorder in playback mode. Now, as each file on tape is encountered, BASIC prints this out. When the correct file is found, BASIC loads or verifies it.When the load or verification is complete, the prompt

OK


appears on the screen.

If any of the M, F, P, or C errors appear before this prompt, then read or verification errors were present - try reading the tape again. If errors persist, the recording is probably corrupt.

Errors may take the following forms:

F (HEX number)	A filename error occurred - unable to read the filename header. Check that you are reading back at the same speed at which you recorded.
P (HEX number)	A parity error occurred - try reading the tape again.
M (HEX number)	If loading, the memory failed to be loaded with the required number - probably caused by a hardware fault. If examining, the memory contains a different number to that recorded on tape.
C (HEX number)	A checksum error occurred. Similar fault as for P.

Note that within the context of BASIC the hexadecimal status numbers are meaningless.

At any time until you actually start the tape search, you can return to BASIC by typing CTRL C. Once the tape search has been started, the only way to stop it, should it not find the correct file, is to type BREAK on the keyboard and then restart BASIC.

EDITING BASIC PROGRAMS

MICROTAN BASIC allows you to edit programs either in the line mode or the character (screen edit) mode.

Line edit mode is usually used to enter new lines, or programs, delete complete existing lines, or modify existing lines where a large number of changes are required.

To enter a line in line edit mode, type in a line number, followed by the program statement, for example:

```
10 PRINT A<CR>
```

BASIC responds with a cursor character if line syntax is correct, and the line is entered into program memory. If syntax is incorrect, the message

```
SYNTAX ERROR
OK
```

is generated, and the line does not enter program memory.

MICROTAN BASIC allows a maximum program line length of 80 characters (VDU line length is immaterial because scrolling is done automatically). If more than this number is typed, the BELL character, or a question mark, will be displayed on the screen and the character ignored. All characters with ASCII codes of less then HEX 20 or greater than HEX 7E are also ignored. When entering a line, the DELETE key deletes the character previous to the cursor. The @ deletes the whole of the line currently being entered.

To delete a line within the stored program, type the line number followed by carriage return, for example:

```
10<CR>
```

deletes line 10 from program storage.

To edit a line in line edit mode, type in the whole new line. For example, say you wish to change 10 PRINT A to 10 INPUT A type:

```
10 INPUT A<CR>
```

The old line 10 is overwritten by the new line 10.

The LIST command is used to list parts of or a whole BASIC program. To list the whole program, type:

LIST<CR>

The first 5 lines of the program are displayed on the VDU. To obtain the next 5, type carriage return and so on until the end of the program. Should you wish to inhibit the paging pauses, type line feed - the program now lists to the end without pausing. Typing CTRL C at any time stops the listing and returns you to the BASIC prorgam. To list a single line, type LIST followed by the line number

LIST 10<CR>

To type a selected group of lines, type

LIST 10-90<CR>

and use the paging delimiters CR, LF and CTRL C as required.

If there is only one error in a long line of BASIC code, it is tedious to type in the whole line to correct it - MICROTAN BASIC includes a screen (cursor) editor to resolve this problem.

To enter screen edit mode, type in the required line number followed by CTRL E (hold down the CTRL key and depress key E). For example:

90 CTRL E

If your program does not contain a line 90, the message

NO SUCH LINE

is printed. Otherwise line 90 is displayed at the screen center between two sets of dashed lines. Above and below these lines appear the lines of code immediately above and below that being edited. A flashing cursor appears between the two dotted lines. The bottom line of the VDU displays the number of the line which you are going to change.

You may now move the cursor around within the two dotted lines and edit the text within them by using any of the following commands:

CTRL U Moves the cursor up.

CTRL D Moves the cursor down.

CTRL L Moves the cursor left.

CTRL R Moves the cursor right.

CTRL E Erases the character over which the cursor is flashing. The rest of the line is shifted back one space to fill the gap left by this character.

Any character

Character is inserted immediately before that over which the cursor is currently flashing. The rest of the character string is shifted right one place to make room. The cursor is also shifted one place to allow text to be inserted as it is written.

RUBOUT

Deletes the character immediately prior to that over which the cursor is flashing. It can therefore be used to correct badly typed character insertions. The rest of the line, plus the cursor, are shifted back one place.

Once the line has been edited to satisfaction and appears to be correct, one of the following commands can be used to make this line update the original program.

CTRL C Returns to BASIC without updating the main program, i.e. the edited line is ignored. Use this for gross error correction.

Carriage Return

First, the edited line is checked for syntax errors. If any occur, the line is not updated in program storage and control is returned to the main BASIC program. If the line is satisfactorily, the line whose number is displayed at the bottom of the screen is deleted and the line in the edit buffer inserted. The program then returns to BASIC command mode. This command may therefore be used to change line numbers of existing line by editing line numbers within the edit buffer.

Line Feed

Performs as for carriage return, except that, on completion of a successful update the BASIC stays in cursor edit mode and the next line is opened for editing between the dashed lines. When the end of the program is reached, control is returned to BASIC command mode. If lines are not changed between each step, line feed may obviously be used to step downwards through the program.

Escape

Escape functions as for line feed, but steps up the program one line at a time.

Note that, with either line feed or escape, if the current line number is changed so that the new line is inserted somewhere different, the old nect line and not the new is displayed.

CTRL K

Control K is used as a line delete command. The line whose number is displayed at the bottom of the screen is deleted, and no new line is entered. It may therefore be used to delete selected lines while stepping through a program using the line feed and escape commands. After a CTRL K, the next line is opened for editing.

CTRL I

Control I is used as a line insert command, and is mainly used to duplicate lines. In this case, the line whose number appears at the page bottom is not deleted (unless the edited line has the same number), but the new line (whose number you have changed) is inserted at the appropriate place within program memory, (program lines located according to line numbers). You may also insert completely new lines by first erasing the information in the edit buffer (depress CTRL E and RPT until the buffer is blank), typing in a new line and depressing CTRL I. The next old line is displayed on completion, i.e. the editor does not jump to display the new line.

---

note1: or BAS<CR> if you have TANBUG v2
note2: with TANBUG v2, you can enter BASIC again without losing your program with WAR<CR> (Warm Entry)

CHAPTER 2 : Introduction To The BASIC Language

INTRODUCTION

This chapter will serve as an introduction for those of you unfamiliar with BASIC. To help get you started writing programs, the text introduces the primary concepts and uses of MICROTAN BASIC.

If your MICROTAN 65 does not have MICROTAN BASIC loaded and running, follow the procedure in Chapter 1 to bring it up. We recommend that you try each example in the chapter as it is presented. This will enhance your "feel" for BASIC and how it is used. If you are already accustomed to programming in BASIC, turn to Chapter 3 for more information about BASIC's characteristics.

When MICROTAN displays "OK", you are ready to use MICROTAN BASIC.

NOTE: All commands to BASIC end with a carriage return. The carriage return tells BASIC that you have finished typing the command. If you make a typing error, type a back-arrow or RUBOUT to eliminate the last character. An at-sign (@) eliminates the entire line that you are typing.

PRINT STATEMENT

Now, try typing in the following:

```
PRINT 10-4  (end with carriage return)
```

BASIC will immediately print:

```
6
OK
```

The PRINT statement you typed in was executed as soon as you hit the carriage return key. BASIC evaluated the formula after the PRINT and then typed out its value, in this case 6. Now try typing in this:

```
PRINT 1/2,3*10  (* means multiply, / means divide)
```

BASIC will print:

```
.5    30
```

As you can see, BASIC can do division and multiplication as well as subtraction. Note how a comma was used in the PRINT command to print two values instead of just one. The comma divides the 72 character line into 5 fields, each 14 characters wide. The last two of the positions on the line are not used. The comma causes BASIC to slip to the next field on the line, where the value 30 was printed.

PROGRAMS AND LINE NUMBERS

Commands such as the PRINT statement you have just typed in are called direct commands. There is another type of command called an indirect command. Every indirect command begins with a line number. A line number may be any integer from 0 to 64000. Try typing in the following lines:

```
10 PRINT 2+3
20 PRINT 2-3
```

Notice that BASIC did not print either of the values. That is because BASIC recognizes by the line numbers that these lines are to be saved and executed as a sequence later. A sequence of indirect commands is called a program. BASIC saves indirect commands in memory. When you type RUN, BASIC executes the program, beginning with the lowest numbered line. If we type RUN now, BASIC will type out:

```
5
-1
OK
```

In the above example, we entered lines 10 and 20 in numerical order. However, it makes no difference in what order you enter indirect statements. BASIC always puts them into correct numerical order according to line number. To

see a listing of the complete program currently in memory, type LIST. BASIC will reply with:

```
10 PRINT 2+3
20 PRINT 2-3
OK
```

Sometimes it is desirable to delete a line of a program altogether. This is accomplished by typing the line number of the line to be deleted, followed only by a carriage return. Type in the following:

```
10
LIST
```

BASIC will reply with:

```
20 PRINT 2-3
OK
```

We have now deleted line 10 from the program. There is no way to get it back except to re-type it. To insert a new line 10, just type in 10 followed by the new line. Type in the following:

```
10 PRINT 2*3
LIST
```

BASIC will reply with:

```
10 PRINT 2*3
20 PRINT 2-3
OK
```

There is an easier way to replace line 10 than by deleting it and then inserting a new line. You can do this by just typing the new line 10 and hitting the carriage return. BASIC throws away the old line 10 and replaces it with the new one. Type in the following:

```
10 PRINT 3-3
LIST
```

BASIC will reply with:

```
10 PRINT 3-3
20 PRINT 2-3
OK
```

It is not recommended that lines be numbered in small increments. It may become necessary to insert a new line between two existing lines. An increment of 10 between line numbers is usually sufficient.

If you want to erase the complete program currently stored in memory, type in NEW. If you are finished running one program and are about to type in a new one, be sure to type NEW first. This should be done in order to prevent a mixture of the old and new programs. Type in the following:

```
NEW
```

BASIC will reply with:

```
OK
```

Now type in:

```
LIST
```

BASIC will reply with:

```
OK
```

NUMBERS, OPERATORS, AND STRINGS

Often it is desirable to include text along with answers that are printed out in order to explain the meaning of the numbers. Type in the following:

```
PRINT "ONE THIRD IS EQUAL TO";1/3
```

BASIC will reply with:

```
ONE THIRD IS EQUAL TO .333333333
OK
```

As explained earlier, including a comma in a PRINT statement causes it to space over to the next fourteen-column field before the value is printed. Using a semi-colon instead of a comma causes the value to be printed immediately, without spacing to the next field.

NOTE: Numbers are always printed with at least one trailing space. Any text to be printed must always be enclosed in double quotes. Try the following examples:

a) PRINT "ONE THIRD IS EQUAL TO ";1/3

```
ONE THIRD IS EQUAL TO .333333333
OK
```

b) PRINT 1,2,3

```
1             2             3
OK
```

c) PRINT 1;2;3

```
1 2 3
OK
```

d) PRINT -1;2;-3

```
-1 2 -3
OK
```

We will digress for a moment to explain the format of numbers in BASIC. Floating point numbers are stored internally to over nine digits accuracy. When a number is printed, only nine digits are shown. Any number may also have an exponent.

The largest number that may represented in MICROTAN BASIC is 1.70141*10E38, while the smallest positive number is 2.93874*10E-39.

When a floating point number is printed, the following rules are used to determine the exact format.

1. If the number is negative, a minus sign (-) is printed. If the number is positive, a space is printed.
2. If the absolute value of the number is an integer in the range 0 ti 999999999, it is printed as an integer.
3. If the absolute value of the number is greater than or equal to .1 and less than or equal to 999999999, it is printed in fixed point notation, with no exponent.
4. If the number does not fall under categories 2 or 3, scientific notation is used.

Scientific notation is formatted as follows:

```
SX.XXXXXXXXESTT
```

Each X is an integer, 0 to 9. The leading S is the sign of the numbers: a space for a positive number and a minus sign for a negative number. One non-zero digit is printed before the decimal point. This is followed by the decimal point and then the other five digits of the mantissa. An E is then printed (for exponent), followed by the sign (S) of the exponent; then the two digits (TT) of the exponent itself. Leading zeros are never printed; i.e. the digit before the decimal is never zero. Also, trailing zeros are never printed. If there is only one digit to print after all trailing zeros are suppressed, no decimal point is printed. The exponent sign will be "+" for positive and "-" for negative. Two digits of the exponent are always printed; that is, zeros are not suppressed in the exponent field. The value of any number expressed thus is the number to the left of the E times 10 raised to the power of the number to the right of the E.

No matter what format is used, a space is always printed following a number. BASIC checks to see if the entire number will fit on the current line. If not, a carriage return/linefeed is executed before printing the number.

The following are examples of various numbers and the output format BASIC will use:

NUMBER	OUTPUT FORMAT
+1	1
-1	-1
6523	6523
-23.460	-23.46
1E20	1E20
-12.3456E-7	-1.23456E-06
1.23456789E-10	1.23456789E-10
999999999	999999999

A number input from the terminal or a numberic constant used in a BASIC program may have as many digits as desired, up to a maximum length of a line (72 characters). However, only 9 digits are significant, with this version of BASIC. The final digit is rounded up.

```
PRINT 1.2345678901234567890
1.23456789
OK
```

So far we have used several different operators in order to inform BASIC of the calculations we wish to perform. Whenever a combination of these operators is used, it is necessary to know which operations are to be performed first. As in standard algebra, we can either specify which operations have the highest priority, or we can rely on BASIC's precedence of operators. That precedence is as follows:

Priority Of Operations

1) Parentheses - any expression enclosed in parentheses is always evaluated first.
2) Exponentiation.
3) Negation.
4) Multiplication and Division (of equal priority).
5) Addition and Subtraction (of equal priority).
6) Relational operators (all of equal priority).

= Equals
<> Not Equal
< Less Than
> Grater Than
<= Less Than or Equal
>= Greater Than or Equal

7) Logical Operators in the order NOT, AND, then OR.

BASIC's relational and logical operators are described in detail in Chapter 3.

INPUT AND GOTO STATEMENTS

The following is an example of a program that reads a value from the terminal and uses that value to calculate and print a result:

```
10 INPUT R
20 PRINT 3.14159*R*R
RUN
?10
314.159
OK
```

Here is what is happening. When BASIC encounters the INPUT statement, it types a question mark on the terminal and then waits for a response. When you type in 10, execution continues with the next statement in the program, with the variable R set to 10. In executing line 20, BASIC prints 314.159.

As you can see, the program calculates the area of a circle with radius R. To calculate the area of different circles, we could keep re-running the program over for each value of R. But, there is an easier way to do this; simply add another line to the program as follows:

```
30 GOTO 10
RUN
?10
314.159
?3
28.3743
?4.7
69.3977
?
OK
```

By putting a GOTO statement on the end of our program, we have caused it to go back to line 10 after it prints each answer. This could have gone on indefinitely, but we decided to stop after calculating the area for three circles. This was accomplished by typing a carriage return to the input statement (thus a blank line).

VARIABLES

The letter R in the program we just used was termed a variable. A variable name consists of one or two characters, and the first character must be a letter. The second character may be a letter or a number (0-9). Any character after the first two are ignored.

Variable names may not be the same as BASIC reserved words, nor may then contain BASIC reserved words. Reserved words are those words used as BASIC commands, statements, or functions. For example, TO would be an illegal variable name and FEND would be illegal because it contains the reserved word END.

Here is a list of MICROTAN BASIC reserved words:

ABS	AND	ASC	ATN
CHR$	CLEAR	CONT	COS
DATA	DEF	DIM	END
EXP	FN	FOR	FRE
GOSUB	GOTO	IF	INPUT
INT	LEFT$	LEN	LET
LIST	LOG	MID$	NEW

NEXT	NOT	NULL	ON
OR	PEEK	POKE	POS
PRINT	READ	REM	RESTORE
RETURN	RIGHT$	RND	RUN
SGN	SIN	SQR	SPC
STEP	STOP	STR$	TAB
TAN	THEN	TO	USR
VAL	WAIT		

MICROTAN BASIC also has two other variable forms; "string" and "integer". String variables are used for character lists, and are described in this chapter.

Integer variables are very useful for saving memory space. A floating point number requires five bytes of memory space for storage in this particular version of BASIC. One byte stores the exponent of the number, and the other bytes are used to store the mantissa. BASIC requires only 2 bytes of memory to store integers, because of their smaller range; +32767 to -32767.

Integer variables are distinguished from numeric variables by a "%" after the variable name. You may assign values to an integer variable in the same way that you assign values to floating point variables. For example:

```
X%=6
D4%=5*3
```

If you assign a non-integer value to an integer variable, the number will be truncated; that is, any value to the right of a decimal point will be lost. For example:

```
B%=4.283
PRINT B%
4
OK
```

During program execution, when BASIC encounters a calculation to be done with an integer variable, these steps are followed:

1. BASIC retrieves the variable from memory.
2. The number is converted into its floating point equivalent.
3. The calculation is performed.
4. The number is converted back into an integer.
5. The result is stored in memory.

If you use integer values in a program and wish to get accurate results from calculations, you must be very careful to ensure that values are not truncated. When you define a variable as an integer, be certain that it will never receive a floating point number that you may wish to recall. Although integer variables can save significant amounts of memory space, their use may cause some loss of execution speed.

LET STATEMENT

Besides having values assigned to variables with an INPUT statement, you can also set the value of a variable with a LET or assignment statement. Try the following examples:

```
A=5
OK
PRINT A,A*2
5          10
OK
LET Z=7
OK
```

```
PRINT Z,Z-A
7          2
OK
```

As can be seen from the examples, the LET is optional in an assignment statement. BASIC "remembers" the values that have been assigned to variables using this type of statement. This "remembering" process uses space in memory to store the data. The values of variables are thrown away and the space in memory used to store them is released when one of four things occurs:

1. A new line is typed into the program or an old line is deleted.
2. A CLEAR is executed.
3. A RUN is executed.
4. A NEW is executed.

Another important fact is that if a variable is encoutered in an expression before it is assigned a value, it is automatically assigned the value zero. Zero is then substituted as the value of the variable in the expression. For example:

```
PRINT Q,Q+2,Q*2
0             2             0
OK
```

REM STATEMENT

REM is short for remark. This statement is used to insert comments or notes into a program. These comments can help avoid confusion - especially when you write long programs. They are also helpful to anyone else who reads your program.

REM statements often note the purpose of a program and describe the techniques used in accomplishing that purpose. REM statements are useful only to humans. No information is passed to BASIC through the notes in a REM statement. When a REM is encountered during execution of a BASIC program, everything on the rest of the line is ignored.

IF AND THEN STATEMENTS: RELATIONAL OPERATORS

Suppose we want to write a program to find out if a number is zero or not. With the statements we have gove over so far this could not be done. What is needed is a statement which can be used to conditionally branch to another statement. The IF and THEN statements do just that. Try typing in the following program:

```
10 INPUT B
20 IF B=0 THEN 50
30 PRINT "NON-ZERO"
40 GOTO 10
50 PRINT "ZERO"
60 GOTO 10
```

When this program is typed into the computer and run, it will ask for a value for B. Type any value you wish. The computer will then come to the IF statement. Between the IF and THEN portion of the statement there are two expressions separated by a relational operator. A relational operator is one of the following six symbols:

```
=    Equal
>    Greater Than
<    Less Than
<>   Not Equal
<=   Less Than or Equal
>=   Greater Than or Equal
```

The IF statement is either true or false, depending upon whether the two expressions satisfy the relation or not. For example, in the program we just did, if 0 were typed in for B, the IF statement would be true because 0=0. In this

case, since the number after the THEN is 50, execution of the program would continue at line 50. Therefore, ZERO would be printed and then the program would jump back to line 0 (because of the GOTO statement in line 60).

Suppose a 1 were typed in for B. Since 1=0 is false, the IF statement would be false and the THEN clause would not be executed. The program would continue execution with the next line. Therefore, NON-ZERO would be printed and the GOTO in line 40 would send the program back to line 10.

Now try the following program for comparing two numbers:

```
10 INPUT A,B
20 IF A<=B THEN 50
30 PRINT "A IS BIGGER"
40 GOTO 10
50 IF A<B THEN 80
60 PRINT "THEY ARE THE SAME"
70 GOTO 10
80 PRINT "B IS BIGGER"
90 GOTO 10
```

When this program is run, line 10 receives two numbers from the terminal. At line 20, if A is greater than B, A<=B will be false. This will cause the next statement to be executed, printing "A IS BIGGER", and then line 40 sends the computer back to line 10 to begin again.

At line 20, if A has the same value as B, A<=B is true so we go to line 50. At line 20, if A is smaller than B, A<=B is true so we go to line 50. At line 50, A<B will be true so we go back to the beginning.

Try running the last two sample programs several times. Remember, to stop these programs, just give a carriage erturn to the input request. The try writing a few programs of your own using the IF-THEN statement. Writing programs of your own is the quickest and easiest way to understand how BASIC works.

FOR AND NEXT STATEMENTS

One advantage of computers is their ability to perform repetitive tasks. Let us take a closer look and see how this works. Suppose we want a table of square roots from 1 to 10. The BASIC function for square root is SQR, and the form is SQR(X). We could write the program as follows:

```
10 PRINT "1",SQR(1)
20 PRINT "2",SQR(2)
30 PRINT "3",SQR(3)
40 PRINT "4",SQR(4)
50 PRINT "5",SQR(5)
60 PRINT "6",SQR(6)
70 PRINT "7",SQR(7)
80 PRINT "8",SQR(8)
90 PRINT "9",SQR(9)
100 PRINT "10",SQR(10)
```

This program will do the job. But imagine how time consuming this would be if we wanted to make a table of 100 square roots, or 1000 square roots! The inefficient method above could be improved by using the IF statement as follows:

```
10 N=1
20 PRINT N,SQR(N)
30 N=N+1
40 IF N<= THEN 20
```

When this program is run, its output will look exactly like that of the ten-statement program above. Let us look at how it works. Line 10 has a LET statement which sets the value of the variable N to 1. Line 20 prints N and the square root of N using its current value. Line 30 is a LET statement that sets the value of N to N+1. In a LET

statement, an equals sign means "to be replaced with".

Thus, the first time line 30 is executed, N becomes 2. At line 40, since N now equals 2, N<10 is true, so the THEN portion branches back to line 20, with N now at a value of 2. The overall result is that lines 20 through 40 are repeated, each time adding 1 to the value of N. When N finally equals 10 at line 20, line 30 will increment it to 11. This results in a false condition at line 40, and since there are no further statements to the program, it stops. This technique is referred to as "looping" or "iteration". Since it is used quite extensively in programming, there are special BASIC statements for using it. We can show these with the following program.

```
10 FOR N=1 TO 10
20 PRINT N,SQR(N)
40 NEXT N
```

The output of the program listed above will be exactly the same as the previous two programs. But notice that the first method we used would require 1000 statements to create a table of 1000 square roots, whereas the final example could do it with just 3 lines.

At line 10, N is set to equal 1. Line 20 causes the value of N and the square root of N to be printed. At line 30 we see a new type of statement. The NEXT N statement causes 1 to be added to N, and then if N<=10, the program returns to the statement following the FOR statement. The overall effect is the same as with the previous program. Notice that the variable following the FOR is exactly the same as the variable after the NEXT.

It is also possible to add a STEP clause to the FOR statement to change the increment of the FOR loop variable.

```
10 FOR N=10 TO 20 STEP 2
20 PRINT N,SQR(N)
30 NEXT N
```

This tells BASIC to start with N=10 and to add 2 to N each time, instead of 1 as in the previous program. If no STEP is given in a FOR statement, BASIC assumes that 1 is to be added each time. The STEP can be followed by any expression.

Suppose we want to count backwards from 10 to 1. A program for doing this would be as follows:

```
10 I=10
20 PRINT I
30 I=I-1
40 IF I>=1 THEN 20
```

Notice that we are now checking to see that I is greater than or equal to the final value. The reason is that we are now counting by a negative number. In the previous examples it was the opposite, so we were checking for a variable less than or equal to the final value. The STEP statement previously shown can also be used with negative numbers to accomplish this same purpose. For example:

```
10 FOR I=10 TO 1 STEP -1
20 PRINT I
30 NEXT I
```

FOR loops can also be "nested". An example of this procedure follows:

```
10 FOR I=1 TO 5
20 FOR J=1 TO 3
30 PRINT I,J
40 NEXT J
50 NEXT I
```

Notice that the NEXT J comes before the NEXT I. This is because the J-loop is inside of the I-loop. The following program is incorrect; run it and see what happens.

```
10 FOR I=1 TO 5
20 FOR J=1 TO 3
30 PRINT I,J
40 NEXT I
50 NEXT J
```

It does not work because when the NEXT I is encoutered, all knowledge of the J-loop is lost. This happens because the J-loop is "inside" the I-loop.

ARRAYS - DIM STATEMENT

An array is a table of values. The name of this table, which is called the array name, can be any legal variable name, such as "A". The array name A is distinct and separate from the simple variable A, and both can be used in a single program. To identify an element of the array, a subscript is added in parentheses to the array name. A(I) represents the Ith element in the array A.

BASIC must be told how much space to allocate for the entire array A. This is done with a DIM (dimension) statement, for example,

```
DIM A(15)
```

In this example, space is reserved for the array A to have elements from A(0) to A(15). Array subscripts always start with zero. Therefore, an array dimensioned at 15 has space for 16 elements.

If "A(I)" is used in a program before it has been dimensioned, BASIC reserves space for 11 elements (0 through 10).

The following program illustrates the use of arrays. It asks you to input 8 numbers, which it then sorts into ascending order.

```
10 DIM A(8)
20 FOR I=1 TO 8
30 INPUT A(I)
50 NEXT I
70 F=0
80 FOR I=1 TO 7
90 IF A(I)<=A(I+1) THEN 140
100 T=A(I)
110 A(I)=A(I+1)
120 A(I+1)=T
130 F=1
140 NEXT I
150 IF F=1 THEN 70
160 FOR I=1 TO 8
170 PRINT A(I),
180 NEXT I
```

When line 10 is executed, BASIC sets aside space for 9 numeric values, A(0) through A(8). Lines 20 through 50 get the unsorted list from the user. The sort is accomplished by comparing adjacent numbers. If they are already ordered, the program procedes. If they are not ordered, their positions in the array are reversed. If any pairs are reversed, the variable "F" is set to 1 and at line 150, BASIC is told to repeat the comparison process. If all the numbers are in order, lines 160 through 180 print out the sorted list. Notice that a subscript can be any expression, as illustrated in line 110.

Arrays may also be dimensioned as integer arrays. If all elements of an array can be represented by integers, a significant amount of memory space can be saved. An integer array is dimensioned exactly like a numerice array, except that a % must follow the array name. For example:

```
DIM A%(9)
```

This array will have 10 elements, each one an integer. As with other integer variables, BASIC will perform all computations in floating point arithmetic.

INTRINSIC FUNCTIONS

BASIC provides functions to perform a number of standard mathematical operations. These functions would be time-consuming for you to calculate and code in BASIC, especially if you used them more than once in your program. A BASIC function has a three- or four-letter call name followed by an argument in parentheses. A function may be used anywhere in a program. A list of the BASIC functions, numeric and string, appears in Chapter 3. Each function is defined and the parameters for its argument are given.

USER-DEFINED FUNCTIONS - DEF STATEMENT

You may also define your own functions when you require a reasonably simple calculation at more than one place in your program. The DEF statement allows you to create a function and use it in the same way that you would use any of BASIC's intrinsic functions. A legal user-defined function name begins with FN followed by any acceptable BASIC variable name. A function need be defined only once and can be defined anywhere in the program. For example:

```
DEF FNA(S) = S*S
```

The dummy argument must be a simple variable, but the expression after the equals sign may contain the argument variable or any other program variable. If, after defining the above function, BASIC encountered a line such as:

```
20 PRINT FNA(4)+5
```

BASIC should respond with:

```
21
OK
```

First 4*4 was calculated using the formula we defined above. Then 5 was added to this value. More information about particular user-defined functions appears in Chapter 3.

SUBROUTINES - GOSUB AND RETURN

A program may perform the same action in several different places. If this is expressed by a relativeley simple calculation, you can probably use a BASIC function or a user-defined function to perform it. However, often it is a more complicated series of calculations requiring many lines of programming. In such cases the "GOSUB" and "RETURN" statements can be used to avoid entering the same lines over and over. When a "GOSUB" statement is encountered, BASIC branches to a specified line number. BASIC keeps track of the point at which it branched from the main program, and when a "RETURN" is encountered, BASIC goes back to the statement following the last executed "GOSUB". Observe the following program.

```
10 PRINT"WHAT IS THE NUMBER";
30 GOSUB 100
40 T=N
50 PRINT"WHAT IS THE SECOND NUMBER";
70 GOSUB 100
80 PRINT"THE SUM OF THE TWO NUMBERS IS",T+N
90 STOP
100 INPUT N
110 IF N=INT(N) THEN 140
120 PRINT"YOU MUST ENTER AN INTEGER. TRY AGAIN"
130 GOTO 100
140 RETURN
```

This program asks for two numbers which must be integers and then prints the sum of the two. The subroutine in this program consists of lines 100 to 130. The subroutine asks for a number, and if it is not an integer, asks for

another number. It will continue to ask for numbers until an integer is entered.

The main program prints "WHAT IS THE NUMBER" and then calls the subroutine to determine if the number is an integer. When the subroutine returns to line 40, the value input is saved in the variable T. This is done so that when the subroutine is called a second time, the value of the first number will not be lost. The program then prints "WHAT IS THE SECOND NUMBER" and the second value is entered when the subroutine is called again. When the subroutine returns the second time, "THE SUM OF THE TWO NUMBERS IS" is printed, followed by the value of their sum. T contains the value of the first number that was entered and N contains the value of the second number.

The next statement in the program is the "STOP" statement. It causes the program to stop execution at line 90. If the STOP statement were not included in the program, we would "fall into" the subroutine at line 100. It would then ask that another number be input. When program execution reached line 130, the subroutine would try to return to the main body of the program. But since it was never called by a GOSUB, a RETURN WITHOUT GOSUB error would occur. Each GOSUB executed in a program must have a corresponding RETURN to be executed later. The opposite also applies, i.e. a RETURN should be encountered only if it is part of a subroutine which has been called by a GOSUB.

Either "STOP" or "END" can be used to separate a program from its subroutines. STOP will print a message indicating the line at which the STOP was encountered.

READ, DATA, AND RESTORE STATEMENTS

Suppose that a program required that the same list of numbers be entered into the program every time it was run. BASIC contains two statements for accomplishing this purpose, the "READ" and "DATA" statements. These same statements also make it easy to change the list of numbers whenever necessary. Consider the following example:

```
10 PRINT"GUESS A NUMBER";
20 INPUT G
30 READ D
40 IF D=-999999 THEN 90
50 IF D<>G THEN 30
60 PRINT"YOU ARE CORRECT"
70 END
90 PRINT"BAD GUESS, TRY AGAIN."
95 RESTORE
100 GOTO 10
110 DATA 1,393,-39,28,391,-8,0,3.14,90
120 DATA 89,5,10,15,-34,-999999
```

When the program is run and the READ statement is encountered, the effect is similar to that of an INPUT statement. But instead of getting a number from the terminal, a number is read from the DATA statements.

The first time a number is needed for a READ, the first number in the first DATA statement is returned. The second time one is needed, the second number in the first DATA statement is returned. When the entire contents of the first DATA statement have been read in this manner, the second DATA statement will be used. DATA is always read sequentially in this manner, and there may be any number of DATA statements in a program.

The above program plays a game whose object is to guess one of the numbers contained in the DATA statements. For each guess that is typed in, the program reads through all of the numbers in the DATA statements until one is found that matches the guess. If more values are read than there are numbers in the DATA statements, an OUT OF DATA error occurs. Line 40 checks to see if -999999 was read. This number is used as a flag to indicate that all of the data has been read. If -999999 is read, then the guess given was incorrect.

Before returning to line 10 to make another guess, we need to make certain that the READs will begin with the first piece of data again. This is the function of the "RESTORE" statement. After the RESTORE is encountered, the next piece of data read will be the first piece in the first DATA statement.

DATA statements may be placed anywhere within the program. Only READ statements make use of the DATA

statements in a program, and any DATA statements encountered at any other time during program execution will be ignored.

STRINGS AND STRING FUNCTIONS

A list of characters is referred to as a "string". HELLO, BASIC 6502, and THIS IS A TEST are all strings. Like numeric variables, string variables can be assigned specific values. String variable names are distinguished from numeric variable names in that they have a "$" as the last character of the variable name. For example:

```
A$="BASIC 6502"
OK
PRINT A$
BASIC 6502
OK
```

This example assigns the string value "BASIC 6502" to the string variable A$. Note that a character string to be assigned to a variable must be entered in quotes.

Now that A$ has a string value, it is possible to use various string functions to learn more about that value. String functions are similar to numeric functions, and are called and passed arguments in the same way. One of these functions is the "LEN" or length function. When called, it returns an integer equal to the number of characters in a string. For example:

```
PRINT LEN(A$),LEN("HELLO")
10            5
OK
```

The number of characters in a string may range from 0 to 255. A string which contains 0 characters is called the "NULL" string. Before a string variable is set to a value in the program, it is initialized to the null string, in much the same way that numeric variables are initialized to 0. Printing a null string on the terminal will cause no characters to be printed, and the print head or cursor will not be advanced to the next column. For example:

```
PRINT LEN(Q$);Q$;3
0 3
OK
```

Another way to create the null string is by using quotation marks with nothing between them, i.e. Q$="". Setting a string variable to the null string can be used to free up the string space used by a non-null string variable.

It is often desirable to be able to access parts of a string and manipulate those parts. The string A$ was previously defined. If we wished to see only some of its contents we might use:

```
PRINT LEFT$(A$,5)
BASIC
OK
```

"LEFT$" is a string function which returns a string composed of the leftmost characters of its string argument. Another example might read:

```
FOR N=1 TO LEN(A$):PRINT LEFT$(A$,N):NEXT N
B
BA
BAS
BASI
BASIC
BASIC
BASIC 6
BASIC 65
BASIC 650
```

```
BASIC 6502
OK
```

Since A$ has 10 characters, the loop will be executed with N=1,2,3,...,9,10. The first time though only the first character will be printed. The second time, the first two character will be printed, etc.

There is a similar string function called "RIGHT$" which returns the right N characters from a string expression. Try substituting RIGHT$ for LEFT$ in the previous example and note the result.

There is also a string function which allows us to take characters from the middle of a string. For example:

```
FOR N=1 TO LEN(A$):PRINT MID$(A$,N):NEXT N
BASIC 6502
ASIC 6502
SIC 6502
IC 6502
C 6502
 6502
6502
502
02
2
OK
```

MID$ returns a string starting at the Nth position of A$ to the end (last character) of A$. The first position of the string is position 1 and the possible position of a string is position 255. Very often it is desirable to extract only the Nth character from a string. This can be done by calling MID$ with three arguments. The third argument specifies the number of characters to return. For example:

```
FOR N=1 TO LEN(A$):PRINT MID$(A$,N,1),MID$(A$,N,2):NEXT N
B             BA
A             AS
S             SI
I             IC
C              C
              6
6             65
5             50
0             02
2             2
OK
```

Strings may also be concatenated (joined together) through the use of the "+" operator. Try the following:

```
B$="HELLO"+" "+A$
OK
PRINT B$
HELLO BASIC 6502
OK
```

Concatenation is especially useful if you wish to take a string apart and then put it back together with slight modifications. For instance:

```
C$=LEFT$(B$,5)+"-"+MID$(B$,7,5)+"-"+RIGHT$(B$,4)
OK
PRINT C$
HELLO-BASIC-6502
OK
```

Sometimes it is desirable to convert a number to its string representation and vice-versa. "VAL" and "STR$" perform these functions. Try the following:

```
STRING$="567.8"
OK
PRINT VAL(STRING$)
 567.8
OK
STRING$=STR$(3.1415)
OK
PRINT STRING$;LEFT$(STRING$,5)
3.1415 3.14
OK
```

"STR$" can be used to perform formatted I/O on numbers. A number may be converted to a string and then LEFT$, RIGHT$, and MID$ can be used to re-format the number as desired. STR$ can also be used to find out how many print columns a number will requires. For example:

```
PRINT LEN(STR$(3.157))
 6
OK
```

VAL is useful in many different applications. If a program accepted a question from a user such as:

"WHAT IS THE VOLUME OF A CYLINDER OF RADIUS 5.36 FEET, OF HEIGHT 5.1 FEET?"

VAL could be used to extract the numeric values 5.36 and 5.1 from the question.

The following program sorts a list of string data and prints out the sorted list. This program is quite similar to the one given earlier in this chapter that sorted a numeric list.

```
100 DIM A$(15):REM ALLOCATE SPACE FOR STRING MATRIX
110 FOR I=1 TO 15:READ A$(I):NEXT I
115 REM READ IN STRINGS
120 F=0:I=1
125 REM SET EXCHANGE FLAG TO ZERO AND SUBSCRIPT TO ONE
130 IF A$(I)<=A$(I+1) THEN 180
135 REM DON'T EXCHANGE ORDERED ELEMENTS
140 T$=A$(I+1): REM USE T$ TO SAVE A$(I+1)
150 A$(I+1)=A$(I): REM EXCHANGE TWO CONSECUTIVE ELEMENTS
160 A$(I)=T$
170 F=1:REM FLAG EXCHANGE OF ELEMENTS
180 I=I+1:IF I<15 GOTO 130
185 REM ONCE PASS IS MADE THRU ALL ELEMENTS,
187 REM CHECK TO SEE IF WE EXCHANGED ANY.
188 REM IF NOT, SORT COMPLETED.
190 IF F THEN 120
200 FOR I=1 TO 15:PRINT A$(I):NEXT I:REM PRINT LIST
210 REM STRING DATA FOLLOWS
220 DATA APPLE,DOG,CAT,BASIC,MICROSOFT,COMPUTER
230 DATA MONDAY,"***ANSWER***","FOO"
240 DATA RANDOM,BISCUIT,BELLEVUE,SEATTLE,BYTE,BIT,SUGAR
```

Strings may also be elements of arrays. These are dimensioned exactly like numeric arrays, except that the variable used as the array name must be a string variable (habing character "$"). For instance

```
DIM A$(10,10)
```

creates a string array of 121 elements, eleven rows by eleven columns (rows 0 to 10 and columns 0 to 10). Each

array element is a complete string, which can be up to 255 characters in length.

CHAPTER 3 FEATURES OF 10K MICROSOFT BASIC

BASIC COMMANDS

A command is usually given after BASIC has typed OK. This is called the "command level". Commands may be used as program statements. Certain commands, such as LIST, NEW, and LOAD will terminate program execution. The form and function of each command is described below.

CLEAR	Clears all variables: resets FOR and GOSUB state, RESTOREs data.
LIST x-y	Lists program starting at line specified by x, up to and including line specified by y. If no y is specified, the program will be listed from x to the end of the program. If neither x nor y is specified, the entire program will be listed.
NULLx	Sets the number of null (ASCII 0) characters printed after a carriage return/linefeed. X may be specified in the range 0 to 71. NULL must be specified for hardcopy terminals that require a delay after a CR/LF. We recommend that you use a null setting of 0 or 1 for Teletypes. A setting of 0 will work with Teletype compatible CRTs. (Teleype is a registered trademark of Teletype Corporation).
RUNx	If x is not specified, starts execution of the program currently in memory at the lowest numbered statement. RUN deletes all variables (does a CLEAR) and restores DATA. If you have stopped your program and wish to continue execution at some point in the program, use a GOTOx statement to stat execution of your program at the desired line. X may be used optionnally to start execution at a specified line number.
NEW	Deletes current program and all variables.
CONT	Continues program execution after a Control/C is typed or a STOP statement is executed. You cannot CONTinue after an error, after modifying your program, or before your program has been run. One of the main purposes of CONT is debugging.
LOAD	Loads the program from the cassette tape. A NEW command is automatically done before the LOAD command is executed. When the LOAD is complete, BASIC will type out the usual OK.
SAVE	Saves on cassette tape the current program in memory. The program in memory is left unchanged. More than one program may be stored on cassette using this command.

BASIC STATEMENTS

The following summary of BASIC statements defines the general format of each statement, and includes an example of each statement's application. In the following statements and descriptions, an argument of:

V or W denotes a numeric variable.

X denotes a numeric expression.

X$ denotes a string expression.

I or J denotes an expression that is truncated to an integer before the statement is executed.

Truncation means that any fractional part of the number is lost, e.g. 3.9 becomes 3, 4.01 becomes 4. An expression is a series of variables, operations, function calls and constants which, after the operation and function calls are performed using the precedence rules, evaluates to a numeric or string value. A constant is either a number (such as 3.14) or a string literal (such as "GOOD").

DATA	10 DATA 1,3,-156,.04,"NUMBER",CAT

Specifies data to be read from left to right. Information appears in statements in the same order as it will be read in the program. Strings may be read from DATA statements. If you want the string to contain leading spaces (blanks), colons (:) or commas (,), you must enclose the string in double quotes. It is impossible to have a double quote within string data or a string literal.

DEF 100 DEF FNA(V)=V/B+C

BASIC provides many commonly used functions for you, but you may define your own functions using this statement. A function is named with FN followed by any legal variable name, for example: FNX,FNQ2,FNNY. User defined functions are restricted to one line. A function may be defined by any expression, but may have only one argument. In the above example, B and C are variables that are used in the program. Executing the DEF statement defines the function. User defined functions can be redefined by executing another DEF statement for the same function. V is called the dummy variable. User defined string functions are not allowed.

DIM 133 DIM A(3),B$(2,4,4),Z(2*1)

Allocates space for arrays. All array elements are set to zero by the DIM statement. Arrays can have more than one dimension. Up to 255 dimensions are allowed, but due to the restriction of 72 characters per line, the practical maximum is about 34 dimensions. Arrays can be dimensioned dynamically during program execution. If an array is not explicitely dimensioned with a DIM statement, it is assumed to be a singly dimensioned array whose subscript may range from 0 to 10 (eleven elements). All subscripts start at zero, which means that an array dimensioned at n contains n+1 elements.

END 999 END

Terminates program execution without printing a BREAK message. Using the command CONT after an END statement causes execution to resume at the statement after the END statement. END can be used anywhere in a program. It is not necessary to have an END statement at the end of a program.

FOR 150FOR V=1 TO 6.5 STEP .5

(Also see NEXT statement). V is assigned the value of the expression following the equals sign, in this case, 1. This is called the initial value. Then the statements between this FOR statement and the corresponding NEXT statement are executed. The final value is the value of the expression following the TO. The initial value is incremented by the step value until it reaches the final value.

If no STEP is specified, it is assumed by BASIC to be 1. If the STEP is positive and the new value of the variable is <= the final value (6.5 in this example), or the step value is negative and the new value of the variable is => the final value, then the first statement following the FOR statement is executed. Otherwise the statement following the NEXT statement is executed. All FOR loops execute the statements between the FOR and NEXT loops at least once, even in cases like FOR V=1 TO 0. Expressions may be used for the initial, final, and step values in a FOR loop. The values of the expressions are computed only once, before the body of the FOR...NEXT loop is executed. When the statement after the NEXT is executed, the loop variable is never equal to the final value, but is equal to whatever value caused the FOR...NEXT loop to terminate. Do not use nested FOR...NEXT loop with the same index variable. The depth of FOR loop nesting is limited only by the available memory.

GET GET A
GET B$

Fetches a single character from the keyboard without displaying it on the screen and without requiring that the return key be pressed.

GOTO 50 GOTO 100

Branches to specified line number.

GOSUB 10 GOSUB 910

Branches to the specified statement until a RETURN is encountered, when a branch is then made to the statement after GOSUB. GOSUB nesting is limited only by the available memory.

IF...GOTO 32 IF X=Y+23.4 GOTO 92

Branches to a specified line number if expression is true. If expression is not true, program execution continues at next line number in sequence.

IF...THEN IF X<12 THEN PRINT "INCORRECT"
IF X=5 THEN 200

If the relation is true, the THEN clause is executed. The THEN clause may be a statement or a line number for branching. If the relation is not true, the next numbered statement will be executed, and any further statements on the original line will be ignored. For example: 100 IF X=5 THEN Y=2: PRINT "INCORRECT". If X is not equal to 5, then "INCORRECT" will never be printed.

INPUT 5 INPUT V,W,W2
10 INPUT "VALUE";V

Requests that data be entered from the terminal. Each value must be separated from the last one by a comma. The last value typed in should be followed by a carriage return. A "?" is given by BASIC as the prompt character. Only constants may be entered in response to the INPUT statement, such as 4.5E-3 or "CAT". Expressions are not legal responses. If more data items are requested by an INPUT statement than are entered from the terminal, a "??" is printed and BASIC will wait for the rest of the data. If more data items are typed in than requested, the extra data items will be ignored. BASIC gives the message "EXTRA IGNORED" in this event. Strings must be input in the same format as they are specified in DATA statements. In the second example above, the prompt VALUE? will be printed on the terminal when data is requested. If carriage return is typed in response to an INPUT statement, BASIC returns to the command level. Typing CONT after an INPUT command has been interrupted will cause execution to resume at the INPUT statement.

LET 300 LET W=X
310 W=X

Each of the above statements accomplishes the same thing. Any value which previously resided in W is lost, and it is replaced with the value of X. The use of LET is optional.

NEXT 550 NEXT V
550 NEXT
550 NEXT V,W

Marks the end of a FOR loop. If no variable is given, as in the second example above, the most recent FOR loop will be matched. As in the third example, a single NEXT statement may be used to match multiple FOR statements. It is equivalent to NEXT V: NEXT W.

ON...GOTO 100 ON I GOTO 10,20,30,40

Branches to the line indicated by the Ith number after the GOTO. That is:

IF I=1 THEN GOTO 10
IF I=2 THEN GOTO 20
IF I=3 THEN GOTO 30

IF I=4 THEN GOTO 40

If I=0 or if I attempts to select a non-existent line (>=5 in this case), the statement after the ON statement is executed. However, if I is >255, or <0, an ILLEGAL QUANTITY error will result. As many line numbers as will fit on a line can follow an ON...GOTO. The following statement: 105 ON SGN(X)+2 GOTO 40,50,60, will branch to line 40 if the expression X is less than zero, to line 50 it it equals zero, and to line 60 if it is greater than 0.

ON...GOSUB 110 ON I GOSUB 50,60

Identical to ON...GOTO, except that a subroutine is called instead of a GOTO. RETURN from the GOSUB returns control of the program to the statement after the ON...GOSUB.

POKE 300 POKE I,J

The POKE statement stores the byte specified by its second argument (J) into the location given by its first argument, (I). The byte to be stored must be => 0 and <=255, or an ILLEGAL QUANTITY error will occur. The address (I) must be =>0 and <=65535, or an ILLEGAL QUANTITY error will result. Be careful not be POKE into locations occupied by BASIC. One of the main uses of POKE is to pass arguments to machine language subroutines. PEEK is the complementary function to the POKE statement.

PRINT 360 PRINT X,Y,Z

500 PRINT

390 PRINT"EQUALS";A

Prints the value of expressions to the terminal. If the list of values to be printed out does not end with a comma or a semi-colon, then a carriage return/linefeed is executed after all the values have been printed. Strings enclosed in quotes may also be printed. If a semi-colon separates two expressions in the list, their values are printed next to each other. If a comma appears after an expression in the list, and the print head is at position 56 or more, then a carriage return/linefeed is executed. If the print head is before print position 56, the spaces are printed until the carriage is at the beginning of the next 14-column field. If there is no list of expressions to be printed, as in the second example (line 500) above, then a carriage return/linefeed is executed.

READ 490 READ V,W

Reads data from a DATA statement into the variables specified in the READ statement. The first piece of data read will be the first piece of data listed in the first DATA statement. Subsequent data will be read until all data in the first statement is exhausted. When this occurs, the program will begin to read data from the second DATA statement. An attempt to READ more values than exist in the program's DATA statements will result in an OUT OF DATA error.

REM 500 REM THIS LINE COMPUTES AVERAGES

Writes comments in the program listing without affecting execution. Even though REM statements are not executed, other statements may branch to them. A REM statement is terminated only by the line's end, not by ":".

RESTORE 510 RESTORE

Allows the re-reading of DATA statements. After a RESTORE, the next item read will be the first item listed in the first DATA statement of the program.

RETURN 50 RETURN

Terminates a subroutine and returns to the statement after the most recently executed GOSUB.

STOP 9000 STOP

Causes BASIC to stop program execution and enter command mode. BREAK IN LINE 9000 (by this example) will be printed after execution is halted. Executing a CONT after a STOP branches to the statement following the stop.

WAIT 805 WAIT I,J,K

This statement reads the status of location I, exclusive ORs K with that status, and then ANDs the result with J until a non-zero result is obtained. Execution of the program continues at the statement following the WAIT statement. If the WAIT statement has only two arguments, K is assumed to be 0. If you are waiting for a bit to become zero, there should be a 1 in the corresponding position of K. I,J, and K must be =>0 and <=255.

BASIC FUNCTIONS - NUMERIC AND STRING

Numeric Functions

BASIC Functions perform certain frequently used calculations so you do not have to supply the formulas or write the code yourself. For these calculations, you may "call" a BASIC function that is already resident in memory. You must specify the argument to the function, which is represented in the descriptions below as "X" or "I". X may be a number or numeric expression. X may contain variable names. All of the following would be acceptable arguments for functions receiving argument "X": 3, 3*5, A, A+2, (A+B)*7. Functions receiving the argument "I" must receive an integer. Passing a non-integer as an argument will cause the integer to be truncated.

ABS(X)	Returns the absolute value of the expression X. ABS gives X if X=>0, -X otherwise.
INT(X)	Returns the largest integer less than or equal to X. For example: INT(.23)=0, INT(7)=7, INT(-.1)=-1, INT(-2)=-2, INT(1.1)=1. The following would round X to D decimal places: INT(X*10^D+0.5)/10^D
RND(X)	Generates a random number between 0 and 1. If X<0 it starts a new sequence of random numbers using X. Calling RND with the same X starts the same random number sequence. X=0 gives the last random number generated. Repeated calls to RND(0) will allways return the same random number. X>0 generates a new random number between 0 and 1. Note that (B-A)*RND(1)+A will generate a random number between A and B.
SGN(X)	Returns 1 if X>0, 0 if X=0, and -1 if X<0.
SIN(X)	Gives the sine of the expression X. X is interpreted as being in radians. Note that: COS(X)=SIN(X+3.14159/2) and that: 1 radian = 57.2958 degrees; so that the sine of X degrees = SIN(X/57.2958).
SQR(X)	Returns the square root of X. An ILLEGAL QUANTITY error will occur if X is less than 0.
TAB(I)	Spaces to the specified print position (column) on the terminal. May be used only in PRINT statements. Zero is the leftmost column on the terminal, 71 the rightmost. If the carriage s beyond position I, then no printing is done. I must be =>0 and <=255.
USR(I)	Calls the user's machine language subroutine with the argument I. See POKE and PEEK.
ATN(X)	Returns the arctangent of the argument X. The result is returned in radians and ranges from -pi/2 to pi/2. (pi/2=1.5708).
COS(X)	Gives the cosine of the expression X. X is assumed to be in radians.
EXP(X)	Returns the constant E (2.71828) raised to the power X. The maximum argument that can be passed to EXP without overflow is 87.3365.
FRE(X)	Returns the number of memory bytes currently unused by BASIC.
LOG(X)	Returns the natural (base e) logarithm of argument X. To obtain the base Y logarithm of X, use the

formula LOG(X)/LOG(Y). For example the base 10 (common) log of 7=LOG(7)/LOG(10).

PEEK(I)	Returns the contents of memory address I. The value returned will be =>0 and <=255. If I is >65535 or <0, an ILLEGAL QUANTITY error will occur. An attempt to read a non-existent memory address will return an unknown value.
POS(I)	Returns the current position of the terminal print head (or cursor on CRT's). The leftmost character position on the terminal is position 0 and the rightmost is 71.
SPC(I)	Prints I spaces or blank characters on the terminal. May be used only in a PRINT statement. I must =>0 and <=255 or an ILLEGAL QUANTITY error will result.
TAN(X)	Returns the tangent of the X. X is assumed be in radians.

String Functions

A string, as described in Chapter 2, is a series of characters. A string may be from 0 to 255 characters in length. All string variables must end in $, for example, A$, B7$, HELLO$. The following functions return values and information pertaining to strings. In the descriptions below, arguments are represented by:

X$ Any string or string expression.

I Any integer or integer expression whose value is >=0 and <=255.

J Any integer or integer expression whose value is >=0 and <=255.

X Any number or numeric expression.

ASC(X$)	Returns the ASCII numeric value of the first character of X$. See Chapter 2, for ASCII/number conversion table. An ILLEGAL QUANTITY error will occur if X$ is the null string.
CHR$(I)	Returns a one character string whose single character is the ASCII equivalent of the value of the argument.
FRE(X$)	When called with a string argument, returns the number of bytes currently unused by BASIC. Identical to FRE with a numeric argument.
LEFT$(X$,I)	Gives the leftmost I characters of the string expression X$.
LEN(X$)	Returns the length of the string expression X$ in characters (bytes). Non-printing characters and blanks are counted as part of the length.
MID$(X$,I,J)	MID$ called with two arguments (X$ and I) returns characters from X$ starting at character position I. If I>LEN(X$), then MID$ returns a null string. MID$ called with three arguments returns a string expression composed of the characters of X$ starting at the Ith character for J characters. If I>LEN(X$), MID$ returns a null string. If J specifies more characters than are left in the string, all characters from the Ith on are returned.
RIGHT$(X$,I)	Returns the rightmost I characters of X$. If I>=LEN(X$) then RIGHT$ returns all of X$.
STR$(X)	Gives a string which is the character representation of the numerical expression X. For instance, STR$(3.1)="3.1".
VAL(X$)	Returns the string expression X$ converted to a number. For example: VAL("3.1")=3.1. If the first non-space character of the string is not a plus (+) or minus (-) sign, or a digit, or a decimal point, then VAL will return a 0.

SPECIAL CHARACTERS

@	(At sign). Erases the current line being typed, and types a carriage return/linefeed. An "@" is usually a shift/P.
RUBOUT	Erases last character typed. If no more characters are left on the line, a carriage return/linefeed is typed.
CR	(Carriage return). A carriage return must end every line typed in. Returns the print head or CRT cursor to the first position (leftmost) on line. A linefeed is always executed after a carriage return.
CTRL/C	(Control C). Interrupts execution of a program or a LIST command. CTRL/C has effect when a

statement finishes execution, or in the case of interrupting a LIST command, when a complete line has finished printing. In both cases a return is made to BASIC's command level and OK is typed. BREAK IN LINE XXXX is printed, where XXXX is the line number of the next statement to be executed.

: (Colon). A colon is used to separate statements on a line. Colons may be used in direct and indirect statements. The only limit on the number of statements per line is the line length. It is not possible to GOTO or GOSUB to the middle of a line.

CTRL/O (Control O). Typing CTRL/O once causes BASIC to suppress all output until a return is made to command level, or until an input statement is encountered, or until another CTRL/O is typed, or until an error occurs.

? (Question mark). The question mark is the equivalent of PRINT. For instance, ?2+2 causes the same output as PRINT 2+2. Question marks can also be used in indirect statements. When a program is listed, BASIC prints all questions marks as PRINT.

BASIC ERROR MESSAGES

After an error occurs, BASIC gives an error message and then returns to command level and types OK. Variable values and the program text remain intact, but the program cannot be continued and all GOSUB and FOR context is lost. The format of error messages is:

Direct Statement: ?XXERROR

Indirect Statement: ?XXERROR IN YYYY

In the above formats, "XX" represents a fully printed message. The "YYYY" is the line number in which the error occurred in the indirect statement. The following descriptions indicate the meaning of BASIC's error codes and messages.

BAD SUBSCRIPT

An attempt was made to reference an array element which is outside the dimensions of the array. This error can occur if the wrong number of dimensions is used in an array reference; for instance, LET A(1,1,1)=Z when A has been dimensioned DIM A(2,2).

REDIM'D ARRAY

After an array was dimensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension 10 because a statement like A(I)=3 precedes the statement DIM A(100) in a program.

ILLEGAL QUANTITY

The parameter passed to a mathematical or string function was out of range. These errors commonly occur due to:

1. A negative array subscript (LET A(-1)=0).
2. An unreasonably large array subscript i.e. >32767.
3. LOG with a negative or zero argument.
4. SQR with a negative argument.
5. A^B with A negative and B not an integer.
6. A call to USR before the address of the machine language subroutine has been patched in.
7. Calls to MID$, LEFT$, RIGHT$, WAIT, PEEK, POKE, TAB, SPC or ON...GOTO with an improper argument.

ILLEGAL DIRECT

INPUT or DEF statement was used as a direct command.

NEXT WITHOUT FOR

The variable in a NEXT statement corresponds to no previously executed FOR.

OUT OF DATA

A READ statement was executed but all of the DATA statements in the program have already been read. The

program tried to read too much data or insufficient data was included in the program.

OUT OF MEMORY
Program too large, too many variables, too many FOR loops, too many GOSUBs, expression too complicated, or any combination of the above.

OVERFLOW
The result of a calculation was too large to be represented in BASIC's number format. If an underflow occurs, zero is given as the result and execution continues without printing any error message.

SYNTAX
Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

RETURN WITHOUT GOSUB
A RETURN statement was encountered without a previous GOSUB statement having been executed.

UNDEFINED STATEMENT
An attempt was made to THEN, GOSUB, or GOTO a statement which does not exist.

DIVISION BY ZERO

CAN'T CONTINUE
Attempt to continue a program when none exists, an error occurred, or after a new line was typed into the program.

STRING TOO LONG
Attempt was made by use of the concatenation operator to create a string more than 255 characters long.

STRING FORMULA TOO COMPLEX
A string expression was too complex. Break it into two or more shorter ones.

TYPE MISMATCH
The lefthand side of an assignement statement was a numeric variable and the righthand side was a string, or vice-versa, or a function which expected a string argument was given a numeric one or vice-versa.

UNDEF'D FUNCTION
Reference was made to a user defined function which had never been defined.

FILE DATA
Some kind of error.

BASIC OPERATORS

Arithmetic operators

BASIC's arithmetic operators perform 6 functions. They are:

SYMBOL	EXAMPLE	FUNCTION
-	-51,X=-A	Negation
+	Z=R+T+Q	Addition
-	J=100-I	Subtraction
*	X=R*B*D	Multiplication
/	C=X/1.3	Division
^	X^3	Exponentiation

Note that negation is quite separate and distinct from subtraction. When you use the minus sign to negate a variable or a number, BASIC treats such usage as zero minus the variable or number. For example: X=-Y is processed as X=0-Y by BASIC. However, X=10-Y is an example of the arithmetic operation subtraction, and as such takes its usual place in the precedence of operators.

Priority of operations

When more than one operation is performed in a single formula, BASIC evaluates each portion using the following schedule of priority:

1) Parentheses - any expression enclosed in parentheses is always evaluated first.
2) Exponentiation.
3) Negation.
4) Multiplication and Division (of equal priority).
5) Addition and Subtraction (of equal priority).
6) Relational operators (all of equal priority).

= Equal
<> Not Equal
< Less Than
> Greater Than
<= Less Than or Equal
>= Greater Than or Equal

7) Logical Operators in the order NOT, AND, then OR.

If, in any given expression, the above rules do not clearly designate the order of priority, then evaluation proceeds from left to right.

Relational Operators

Relational operators allow a comparison of two values and are most frequently used to compare expressions or strings in an IF...THEN statement. The six relational operators are:

MATH SYMBOL	BASIC SYMBOL	EXAMPLE	MEANING
=	=	X=Y	X has the value of Y
<	<	X<Y	X is less than Y
£	<=,=<	X<=Y	X is less than or equal to Y
>	>	X>Y	X is greater than Y
3	>=,=>	X>=Y	X is greater than or equal to Y
1	<>,><	X<>Y	X is not equal to Y

Any expression that contains one of these operations will always have the value of TRUE(-1) or FALSE(0). For example, (4=7)=0, (4=4)=-1, (7<4)=0, and (7>4)=-1. The THEN clause of an IF statement is executed whenever the expression after the IF is not evaluated at 0. For example, IF X THEN... is equivalent to IF X<>0 THEN. When applied to strings, the relational operators test alphabetic sequence. Comparison is made on a character by character basis according to the ASCII codes until a difference is found. If the end of one string is reached before a difference is found, the shorter string is considered smaller than the other string.

Logical operators

The Logical Operators are used for bit manipulation and for performing Boolean operations. These three operators convert their arguments to sixteen bits, signed two's complement integers in the range -32768 to +32767. They then perform the specified logical operation on them and return a result within the same range. If the arguments are not

in this range, an ILLEGAL QUANTITY error occurs.

The operations are performed in bitwise fashion. This means that each bit of the result is obtained by examining the bit in the same position for each argument. The following truth table shows the logical relationship between bits.

		AND			OR		NOT
A	B	A AND B	A	B	A OR B	A	NOT A
1	1	1	1	1	1	1	0
1	0	0	1	0	1	0	1
0	1	0	0	1	1		
0	0	0	0	0	0		

The examples below show some uses of the logical operators.

63 AND 16=16	Since 63 equals binary 111111 and 16 equals binary 10000, the result of the AND is binary 10000 or 16.
15 AND 14=14	15 equals binary 1111 and 14 equals binary 1110, so 15 AND 14 equals binary 1110 or 14.
-1 AND 8+8	-1 equals binary 1111111111111111 and 8 equals binary 1000, so the result is binary 1000 or 8.
4 OR 2=6	Binary 100 OR'd with binary 10 equals binary 110, or 6 decimal.
10 OR 10=10	Binary 1010 OR'd with binary 1010 equals binary 1010, or 10.
NOT 0=-1	The bit complement of binary 0 to 16 places is sixteen ones (1111111111111111) or -1. Also NOT -1=0
NOT X	NOT X is equal to -(X+1). This is because to form the screen bit two's complement number, you take the bit (one's) complement and add one.

A typical use of the bitwise operators is to test bits set in I/O locations which reflect the state of some external device. Bit position 7 is the most significant bit of a byte, while position 0 is the least significant.

CHAPTER 4 : Aids To Efficient Programming

SPACE OPTIMIZATION

The following hints will help you save memory space.

1) Use multiple statements per line. There is a small amount of overhead (5 bytes) associated with each line in the program. Two of these 5 bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number, (minimum number is 0, maximum number is 64000), it takes the same number of bytes. Putting as many statements as possible on a line will cut down on the number of bytes used by your program.

2) Delete all unnecessary spaces from your program. The statement

```
10 PRINT X, Y, Z
```

uses three more bytes than

```
10PRINTX,Y,Z
```

Note: All spaces between the line number and the first non-blank character are ignored.

3) Delete all REM statements. Each REM statement uses at least one byte plus the number of bytes in the comment text. For instance, the statement

```
130 REM THIS IS A COMMENT
```

uses 24 bytes of memory. In the statement

```
140X=X+Y:REM UPDATE SUM
```

the REM uses 14 bytes of memory including the colon before the REM.

4) Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement 10 P=3.14159 into the program, and use P instead of 3.14159 each time it is needed, you will save 40 bytes. This will result in improved execution speed.

5) The END statement is entirely optional at the end of a program.

6) Re-use the same variables. If you have a variable T which is used to hold a temporary result in one part of the program and you need a temporary variable later in your program use T again. Or, if you are asking the terminal user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable A$ to store the reply.

7) Use GOSUBs to execute sections of program statements that perform identical actions.

8) Use the zero elements of arrays; for instance A(0), B(0,X). Simple (non-matrix) numeric variables like V use 6 bytes; 2 for the variable name, and 4 for the value. Simple non-matrix string variables also use 6 bytes, 2 for the variable name, 2 for the length, and 2 for a pointer. Array variables use a minimum of 12 bytes. Two bytes are used for the variable name, 2 for the size of the array, 2 for the number of dimensions, and 2 for each dimension along with 4 bytes for each of the array elements.

String variables also use one byte of string space for each character in the string. This is true whether the string variable is a simple string variable like A$, or an element of a string array such as Q1$(5,3). When a new function is defined by a DEF statement, 6 bytes are used to store the definition.

9) Reserved words such as FOR, GOTO or NOT, and the names or intrinsic functions such as COS, INT and STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each. When a program is being executed, space is dynamically allocated on the stack as follows:

1. Each active FOR...NEXT loop uses 22 bytes.
2. Each active GOSUB (one that has not yet returned) uses 6 bytes.
3. Each parenthesis encountered in an expression uses 4 bytes and each temporary result calculated in an expression uses 12 bytes.

10) Use integer variables. Integer variables require 2 to 4 fewer bytes for storage than floating point variables.

TIME OPTIMIZATION

The following hints should improve the execution time of your BASIC program. Note that some of these hints are the same as those used to decrease the space used by your programs. This means that in many cases you can increase the efficiency of both space and speed at the same time.

1) Delete all unnecessary spaces and REMs from the program. This may cause a small decrease in execution time because BASIC would otherwise have to ignore or skip over spaces and REM statements.

2) Use variables instead of constants. It takes more time to convert a constant to its floating point representation than it does to fetch the value of a simple or matrix variable. This is especially important within FOR...NEXT loops or other code that is executed repeatedly. THIS IS PROBABLY THE MOST IMPORTANT SPEED HINT BY A FACTOR OF 10!

3) Variables that are encountered first during the execution of a BASIC program are allocated at the start of the variable table. This means that a statement such as 5 A=0:B=A:C=A will place A first, B second, and C third in the symbol table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the symbol table to find A; whereas it would search 2 entries to find B, three to find C, etc.

4) Omit the index variable from NEXT statements. NEXT is somewhat faster than its equivalent NEXT I because no check is made to see if the variable specified in the NEXT is the same as the variable in the most recent FOR statement.

DERIVED FUNCTIONS

The following functions, while not intrinsic to BASIC, can be calculated using the existing BASIC functions. Use the DEF statement.

SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=-ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=-EXP(-X)/(EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARGSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARGCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARGTANH(X)=LOG((1+X)/(X))/2
INVERSE HYPERBOLIC SECANT	ARGSECH(X)=LOG((SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARGCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARGCOTH(X)=LOG((X+1)/(X-1))/2

CONVERSION OF BASIC PROGRAMS

Though implementations of BASIC on different computers are in many ways similar, there are some incompatibilities for which you should watch if you are planning to convert some BASIC programs that were not written in Microsoft BASIC.

1) Array subscripts. Some BASIC's use "[" and "]" to denote array subscripts. Microsoft BASIC uses "(" and ")".

2) Strings. A number of BASIC's force you to dimension (declare) the length of strings before you use them. You should remove all dimension statements of this type from the program. In some of these BASICs, a declaration of

the form DIMA$(I,J) declares a string matrix of J elements each of which has a length of I. Convert DIM statements of this type to equivalent ones in BASIC: DIM A$(J). Microsoft BASIC uses "+" for string concatenation, not "," or "&". Microsoft BASIC uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Other BASIC's use A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

OLD	NEW
A$(I)	MID$(A$,I,1)
A$(I,J)	MID$(A$,I,J-I+1)

This assumes that the reference to a substring of A$ is not an assignment. If the reference to A$ is on the lefthand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows:

OLD	NEW
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)

3) Multiple assignments. Some BASIC's allow statements of the form:

```
500 LET B=C=0
```

This statement would set the variables B and C to zero. In Microsoft BASIC this has an entirely different effect. All the "="'s to the right of the first one would be interpreted as logical comparison operators. This would set the variable B to -1 if C equalled to 0. If C did not equal 0, B would be set to zero. The easiest way to convert statements like this one is to rewrite them as follows:

```
500 C=0:B=0
```

4) Some BASIC's use a backslash instead of ":" to delimit multiple statements per line. Change the backslash to ":" in the program.

5) Programs which use the MAT functions available in some BASIC's will have to be re-written using FOR...NEXT loops to perform the appropriate operation.

APPENDICES

ACCESSING USER MACHINE CODE SUBROUTINES

In situations where sections of high-speed processing are required, it may be advantageous to use machine code subroutines. A user machine code subroutine must have the following characteristics:

1. Exit from the subroutine is by instruction RTS.
2. The stack must be at the same level on exit as at entry.
3. The only zero page RAM locations that can be used are (HEX) 35 to (HEX) 80.

Note that these zero page locations are corrupted by BASIC if an INPUT command is used. User subroutines should therefore save any zero page values in RAM adjacent to the subroutine, and restore them to zero-page when next called.

To call a user subroutine from BASIC, the following set of steps is required:

1. Decide whereabouts in memory you are going to put your subroutine - this should be near the top of available memory space.
2. POKE the decimal number corresponding to the lower 8 bits of the subroutine start address into decimal location 34 (see POKE, chapter 3).
3. POKE the decimal number corresponding to the higher 8 bits of the subroutine start address into decimal location 35.
4. Call the subroutine via the USR command.

For example, to call a machine code subroutine whose start is located at address HEX 1F00

Low byte of subroutine address = HEX 00 = decimal 00
High byte of subroutine address = HEX 1F = decimal 31

```
10 POKE 34,0 ; set up low byte
20 POKE 35,31 ; set up high byte
30 X=USR(I) ; call the subroutine
```

Note that X and I (these may be any variable) are dummy locations which have no meaning within the program context.

Transfer of parameters between the user subroutine and BASIC program are effected by using the BASIC PEEK and POKE functions to access a common area of RAM. For example, to access a parameter calculated by the above subroutine call, the following code is required:

User subroutine:

```
1F00 LDA $1EFE ; pickup input parameter
... ; code to calculate parameter
...
STA $1EFF ; result stored in location 1EFF
RTS ; return to BASIC
```

BASIC:

```
10 M=1
20 POKE 7934,M ; poke M to HEX location $1EFE
30 POKE 34,0
40 POKE 35,31
50 X=USR(I) ; call user subroutine at $1F00
60 N=PEEK(7935) ; N = result at HEX $1EFF
```

Note that the BASIC address parameters must be in decimal notation.

To enter a machine code subroutine to memory, you should carry out the following steps:

1. using the TANBUG monitor, enter your user subroutine at the top end of your available memory. You can do this with either the TANBUG "M" command, the XBUG translator, or from cassette.
2. Enter BASIC by typing GE2ED.

3. In response to the "MEMORY SIZE" prompt, type in a decimal number whose HEX value is below the start address of your user subroutine.
4. Enter your BASIC program, either from the keyboard or from cassette.

You are now ready to run the program.

Using Zero Page Locations

BASIC and the associated TANBUG subroutines together use up all zero page locations. Therefore, when employing user subroutines, you cannot use zero page RAM to hold information between one user subroutine and the next. The onlyexception to this rule is if you are not using the BASIC INPUT command, in which case you may use locations $35 (HEX) to $80 (HEX).

You may of course use any zero page location you like within your user subroutine providing you save its contents on entry (either on the stack or in memory allocated to your subroutine) and restore it on exit from the subroutine.

Program Crashes

As will all machine code programs, a system crash occurs if you try to execute incorrectly entered code. Often, it will be impossible to recover BASIC after a crash, and the machine will behave in an unpredictable way until a RESET is executed. For this reason it is recommended that an up-to-date tape dump is kept of all code - minimum effort is then required for program re-entry.

Using Interrupts With BASIC

You can not directly enter a "BASIC" written interrupt routine via an interrupt. There are two ways to handle interrupts in BASIC.

1. The interrupt can set a flag stored in a memory location outside BASIC's program area. The BASIC "WAIT" command (chapter 3) can then be used to poll this location and hold program execution until the flag becomes set.
2. If other program functions are being executed and are required to continue until an interrupt occurs, then the PEEK command may be used to examine the flag, and only take action if it is set.

A user machine code interrupt routine is loaded into memory in a similar way to that described for user machine code subroutines. The MICROTAN 65 manual gives details of the method of adding interrupt driven peripherals.

Note that user interrupt routines must obey the following rules:

1. The accumulator, X and Y registers must be saved on entry and restored on exit (the Processor Status is saved automatically).
2. The stack level must be the same on exit as at entry.

Note also that 6502 interrupts are disabled during cassette file read and write.

USING MICROTAN'S GRAPHICS WITH BASIC

You can use the POKE and PEEK functions in BASIC to produce a graphics (or mixed alphanumeric and graphics) display. The required data is written directly to the memory-mapped VDU. Note, however, that if you use BASIC's PRINT or INPUT command the display is scrolled and the graphics information is lost.

In alphanumeric mode you can consider the display as being made up of 16 rows of 32 characters each. The following sub-routine (which may of course be allocated any line number) puts the character held in string Z$ out on the screen at the co-ordinates specified in X and Y. If X and Y are too large no action is taken.

(The comments shown are not part of the program but are used to explain each step).

```
100 REM SCREEN ADDRESSED CHAR OUTPUT
110 IF X>31 THEN RETURN ; check X size
```

```
120 IF Y>15 THEN RETURN ; ditto Y.
130 ADDR=512+X+((15-Y)*32) ; where to put ut.
140 DTA=ASC(Z$) ; data
150 POKE ADDR,DTA ; write it out
160 RETURN
```

The following subroutine can be used to address the 64x64 chunky graphics space. The input parameters X and Y specify the screen co-ordinates, while if Z is a 0 the pixel is obliterated, if Z is a 1 a pixel is displayed. If Z is negative then the screen is not changed but a value of 0 or 1 is returned in Z to indicate the state of that pixel. If X and Y are out of range then no action is taken.

Warning - do not re-enter command mode with graphics set or you may need to reset to normal mode. Note that, initially the screen contains all spaces and therefore some graphics information is already present when you run a program. You may need to clear it just to use this subroutine.

```
200 REM GRAPHICS SUBROUTINE
210 IF X>63 THEN RETURN ; X bound
220 IF Y>63 THEN RETURN ; Y bound
230 ADDR = 512+X/2+(INT(Y/4))*32 ; calculate address
240 BIT = 2^((X AND 1)+2*(3-(Y AND 3))) ; calculate required bit pattern
250 DTA = PEEK (ADDR) ; read data
260 IF Z<0 GO TO 350 ; skip if Z negative
270 IF Z>0 GO TO 350 ; skip if Z positive
280 DTA = DTA AND (NOT BIT) ; if zero, clear this bit
290 DUMMY = PEEK (49136) ; set graphics
300 POKE ADDR,DTA ; write it
310 POKE 49139,0 ; set char mode
320 RETURN
330 DTA = DTA OR BIT ; output a 1
340 GO TO 290
350 Z = DTA AND BIT ; what is this bit
360 IF Z = 0 THEN 320
370 Z = 1
380 RETURN
```

CASSETTE DATA FILE HANDLING

MICROTAN BASIC contains software to allow you to write data files to and read files from an ordinary cassette player. You can use this either to read in a complete data file, process it, and record the results (single cassette non remote mode) or implement a motor remote control facility and use two cassette units for a read-process-write facility.

Dual Cassette Recorder Connections

On the MICROTAN mini-rack, cassette socket 0 should be used for the replay machine and cassette socket 1 for the record machine. If you are using a single recorder only this should be plugged into socket 0.

The XBUG firmware is necessary to use the data file handling facility. BASIC will crash on these commands if the PROM is not present. Cassette players must have their input/output levels set up as described in the XBUG manual before use with BASIC.

Remote Control Connections

The 6522 on TANEX which handles the cassette I/O has pin PB5 allocated as control for the replay cassette, and PB6 as record cassette control. The cassette drive is off when the pin output is high or three-state.

Different types of cassette recorder have different remote control options, while some have none at all. You should check which type your recorder has and connect it up as required.

WARNING: IF YOU ARE USING A MAINS POWERED RECORDER ENSURE THAT IT IS NOT POSSIBLE FOR ANY EXTERNAL VOLTAGE OR MAINS VOLTAGE TO APPEAR ON YOUR MICROTAN SYSTEM. DO NOT CONNECT THE CONTROL OUTPUTS DIRECT TO THE REMOTE CONTROL INPUT OF YOUR CASSETTE PLAYER, AN INTERFACE CIRCUIT MUST BE USED.

The remote control circuit shown below will be suitable for most portable recorders. If you are using one cassette only but require remote control, you should wire the relays so that your cassette operates when either PB5 or PB6 is asserted. It may be necessary to incorporate an override switch to allow you to rewind your cassette.

BASIC Programming Commands

Data file manipulations are carried out by POKEing into certain zero page locations immediately prior to executing INPUT or PRINT statements. After execution of the statement, normal mode is automatically returned.

The legal operations are:

POKE 14,0	sets fast data format.
POKE 14,1	sets CUTS data format

These do not need to precede I/O statements.

POKE 22,255 PRINT....	Turns the cassette on, records a long header, then all data in the print statement. Turns cassette off.
POKE 22,254 PRINT...	Turns cassette on, records a short header, then all data in the print statement. Turns cassette off.
POKE 22,1 INPUT...	Turns cassette on, looks for a long file header, reads the recorded line, turns the cassette off.
POKE 22,2 INPUT....	As above but does not look for a long file header.

Example

The program

```
10 POKE 14,1
20 POKE 22,255
30 PRINT "FILE1"
40 PRINT "NAMED"
50 STOP
```

sets CUTS speed, records the message "FILE1", prints "NAMED" on the VDU, then exits.

USING THE CASSETTE FILES WITHOUT REMOTE CONTROL

Recording

First, record a filename of your choice as described in the example above. Then record all data generated by your program using the POKE 22,254 command. This may either be done as your program generates it (though if you are doing long calculations this could be inefficient), or from store at the end of the program.

Playback

Rewind the tape. Use the POKE 22,1 command to look for your filename (all individual records with short headers will be ignored). Once you have found this, you can then read in your data records. Note that if you take a long time to process each record then the cassette may play back the next record before you are ready for it.

Example

```
10 POKE 14,1          ; sets CUTS speed
20 POKE 22,1          ; look for the file name
30 INPUT A$
40 IF A$="FILE1" THEN 20       ; is it the one ?
50 POKE 22,2
60 INPUT A            ; get the data
70 A=A+2:PRINT A      ; process it
80 GOTO 50            ; next record
```

Remote Control Operation

Remote control operation is similar to that described above. However, you can now read in one record at a time from your input cassette, process it, and record it on your output cassette.

Errors

If a parity error occurs on read, the message PARITY is printed, and the program returns to the command mode. By typing CONT<CR> you can then enter data manually and the program will continue.

Note that if you exit from a program while the PRINT or INPUT statements are set up for cassette (this may occur if you CTRL C out of a program, or if you do not find the file you are looking for), then it may be necessary to depress CTRL C a few times to regain command mode.

File Formatting

From the command description it is obvious that cassette input-output is entirely free format - the content is totally up to you the user. It is more time efficient if you store as many characters as possible, for example:

```
POKE 22,254
PRINT A;",";B;",";C
```

is almost three times as fast as recording each one separately. If you use this method care should be taken over 4 points:

1. The type of input statement used to read input data should be the same format as the output statement, i.e. errors will occur if you try to read a string with a numeric input statement.
2. You, the user, must ouput commas between each number in a multiple statement as shown above.
3. You must always read the correct number of entries in a record - i.e. if you record three numerics in a record you must read three back.
4. Records must have a maximum length of 80 characters, otherwise only the first 80 will be read.