ICL

BASIC: A User's Guide

2903

2903 Educational System

2903 BASIC

A USER'S GUIDE

Publication 2015/P6
ISBN 0 900482 18 4

First Edition January 1978

ICL will be pleased to receive readers' views on the contents and organisation, etc. of this publication. Please write to

Educational Products Group,
Education Region, International Computers Limited,
Computer House, 322 Euston Road, London NW1 3BD.

Printed by:
ICL Printing Services, Works Road, Letchworth, Herts. SG6 1JY.

Preface

2903 Educational System BASIC was developed by North Staffordshire Polytechnic Computer Centre in conjunction with International Computers Limited, as part of the ICL 2903 Educational System. The design of the system and the specification of the language were influenced by many factors: the needs of the potential user, the characteristics of the 2903, existing implementation of BASIC, both within ICL and elsewhere, and the developments towards an international standard for BASIC.

The BASIC language was introduced at Dartmouth College, USA, in 1963, and its development since then had been centred there. However, many implementors of BASIC systems have introduced new features in their own way, and in 1973 attempts were made to encourage some standardisation, with the publication of *BASIC specification, 6th edition* (Dartmouth College) and *Specification for standard BASIC* (G.M. Bull, W. Freeman, S.J. Garland; NCC). Since then the American National Standards Institute (ANSI) and the European Computer Manufacturers Association (ECMA) have set up committees to define a standard for the BASIC language which will have the full authority of those bodies.

The definition of BASIC for the 2903 Educational System has relied considerably on the documents mentioned above, and in addition the BASIC manuals of a large number of manufacturers and bureaux have been consulted. The aims throughout have been twofold: to define a language which is true to the original spirit of BASIC—that it should be natural in its constructions and easy to learn and use—and which at the same time provides a set of features sufficiently varied and powerful to allow a wide range of educational and applications programs to be written and processed efficiently.

Contents

Introduction

This Guide to 2903 Educational System BASIC is intended to provide an introduction to all the major features of the BASIC language as implemented on the 2903 Educational System.

It is hoped that this guide and the BASIC reference card as an aide memoire will provide all that is necessary to enable one to write BASIC programs successfully.

A full definition of the language syntax can be found in the ICL publication *2903 Educational System BASIC: reference*. Information about setting up the compiling system on disc and then operating it is to be found in the ICL publication *2903 Educational System BASIC: Installation Guide*.

The system

Terminals are connected to an ICL 7502, a small computer which controls messages sent to and from the ICL 2903 central processor. The 2903 has magnetic disc backing storage which is used for holding the BASIC compiler, libraries of software, user programs and data.

Chapter one

A BASIC session

Connection

In order to connect the terminal:

1 The power is switched on at wall switch and terminal.

2 Any necessary telephone connections are made. (Details of this procedure can be obtained from the computer centre).

3 Terminal is switched to "Full Duplex".

4 The key marked CTRL (short for control) is pressed and held down while the A key is pressed — the system should respond by carriage return and line feed. For ICL VT's the 1 key should be pressed instead of the A key.

5 The terminal is now ready for input.

Note:

A further check on readiness can be made by pressing the ACCEPT or RETURN or SEND or NEWLINE keys, to which the system should respond with line feed.

If the appropriate responses are not received, repeat points 1 to 4 and if there is still no response the computer centre should be contacted.

Starting a BASIC session

Once the terminal has been connected to the system, the user can start to use the BASIC system by typing in the HELLO or LOGIN command. The full format for these commands is: —

HELLO *user.id,pass*
or LOGIN *user.id,pass*

where *user* is the username assigned by the computer centre, (up to six characters).
id is the two character identifier which distinguishes one user's programs from others stored under the same username.
pass is the password for the username, (up to four characters).

e.g. HELLO DUNCAN.DS,DUNC.

HELLO *user*
or LOGIN *user*

will elicit the response: —

@@@@ PASSWORD?

The user's password can then be typed in and it is hidden by the previously printed characters. If the username or password is typed in incorrectly the system will print an error message and the HELLO or LOGIN command must be repeated. The correct username and password will produce the response: —

NSP 2903 BASIC SYSTEM
JULY 1 1977 AT 9.45

When the system has completed the message with carriage return and line feed, work may proceed.

e.g.
```
HELLO DUNCAN, DUNC
NSP 2903 BASIC SYSTEM
SEPTEMBER 19 1977 AT 14.22
READY
```

Note:

It is possible that the introductory message shown here may have been modified by the computer centre to provide a local indentification.

Terminating a BASIC session

Before terminating a session the user should ensure that any program or data in main store that he wishes to retain has been stored in his userfile. The session is then ended by typing

```
BYE
```

The system will respond with the number of minutes that the user has been logged on during the session and the terminal will then go off-line. Before the terminal may be used again, the "CTRL and A" (or CTRL and 1 for VT's) and logon procedures must be repeated.

e.g.
```
BYE
0014  MINS.TERM.TIME.
```

Note:

To avoid on-line terminals being left unattended a "time-out" system is used; if a program executing is waiting for input for four minutes it will be broken in to; if a terminal is doing no work and is left idle for four minutes it will be logged off automatically.

Creating a new program in store

Programs can be created in main store simply by typing in lines of BASIC and entering them by pressing the ACCEPT or RETURN or SEND or NEWLINE key after each line.

e.g.
```
10 REM MY PROG
20 PRINT "ANYTHING AT ALL"
30 END
```

All BASIC program statements are preceded by a line number. The statements are executed in line number order. Line numbers must be in the range 1 to 9999. Normally program line numbers ascend in tens i.e. 10, 20, 30, etc. so that extra lines can easily be inserted. The last line in a program must be the statement END.

The statement at line 10 in the above example is used for helping other programmers to understand the program. A REM statement may be inserted anywhere in the program and it will not affect the way the program is executed. This program will print on the terminal words ANYTHING AT ALL.

The programmer could have typed in the lines in any order.

e.g.
```
30 END
10 REM MY PROG
20 PRINT "ANYTHING AT ALL"
```

The computer would execute the lines in numerical order so that the first example would be executed in an identical way to the second.

Typing errors

If a typing error is made the character or the line may be deleted as follows:

the character "←" will delete the previous character. (A number of characters can be deleted by an equivalent number of "←"s.)

pressing the "X" key while holding down the "control" key will delete the line which is currently being typed.

Program errors

If there is a syntax error in a BASIC program statement the system will print out an error message after the accept key is pressed.

e.g. 20 PRNT "ANYTHING AT ALL"
ERROR 86

It is possible to find out more information about the error by typing ?

e.g. ?
STATEMENT NOT RECOGNISED

HELP command

This works in two modes.

When terminal is not logged on; HELP will give information to assist the user to log on.

When the terminal is logged on; HELP may be typed if information is needed on how to use the BASIC system. It will request the user to type in the statement or command which is in question and will respond with details.

Chapter two

Simple programming

Arithmetic

Variables and constants

The BASIC language can be used readily for calculations. The value of 1 + 2 may be calculated by the statement:

```
10 LET A = 1 + 2
```

The letter A is known as a **numeric variable**. It can be set to the value of any number. The values 1 and 2 are known as **numeric constants**.

A numeric constant can be expressed as an integer, a decimal fraction, a mixed number, an integer plus exponent, a fraction plus exponent, or a mixed number plus exponent.

In exponent form A * 10^{x} is written AEX.

e.g.
```
20 LET A = 1027
30 LET B = .1027
40 LET C = 10.27
50 LET D = 102E7
60 LET E = 10.2E-7
```

Any letter can be used as a numeric variable. Numeric variables can also be given names consisting of a letter followed by a single digit in the range 0—9. Thus the full range is A—Z and A0—Z9.

Numeric variables can be used in calculations.

e.g.
```
10 LET B = 1
20 LET A = B + 2
```

All the normal arithmetic operations can be performed.

1) ** or↑ is the symbol for exponentiation
2) / and * are the symbols for divison and multiplication
3) — and + are the symbols for subtraction and addition
4) MAX/MIN

and are executed in the above order if a number of operations occur in any one expression, or from left to right if equal priority is shown.

e.g.
```
30 LET C = 1 + 2 * 3 -4/8 + 3 ** 2
```
Sets C = 1 + 6 — 0.5 + 9 = 15.5

The operators MAX and MIN may also be used. They function in the following way:

A MAX B will access the larger of the 2 variables A and B.

A MIN B will access the smaller of the 2 variables A and B

e.g.
```
20 LET A MAX B = C
```

Brackets can be used, as in arithmetic, to change the order in which the expression is evaluated.

e.g. 30 LET C = (((1 + 2) * 3 — 4)/8 + 3 **2) * 3

Sets C = (((3) * 3 — 4)/8 + 9) * 3
= (5/8 + 9) * 3
= 28.875

Arrays

Variables may be grouped together to form an **array** so that instead of using several variable names one **subscripted** variable can be used

e.g. A, B, C, D, E can be replaced by
A(0), A(1), A(2), A(3), A(4)

The bracketed number, which must be an integer, is the subscript. Each subscripted variable is called an **element** of the array.

Arrays are declared using the DIMENSION statement. This shows how many elements are required and how many **dimensions** (1 or 2). A two dimensional array (or matrix) could be shown pictorially thus:

A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)

and would be declared thus:

DIM (2,3)

Showing a two dimensional array with the highest first subscript equal to 2 and the highest second subscript equal to 3. Since the lowest subscript is 0 this means that 3 × 4 = 12 elements have been declared.

Note that the first subscript shows the row and the second the column in which the element appears.

Examples

10 DIM P(4)
will declare a one dimensional, 5 element array.

20 DIM B (3,3), C(2,4)
will declare 2 two dimensional arrays one with 16 elements and one with 15 elements.

Standard functions

A set of standard functions is available to BASIC programmers to use in calculations.

These are:

SIN (X)	The sine of X
COS (X)	The cosine of X
TAN (X)	The tangent of X
ATN (X)	The arc tangent of X
CPI	The value of π 3.14159265359
LOG (X)	The natural logarithm of X (X>0)
EXP (X)	e^x where e = 2.7182818285
SQR (X)	The square root of X (X⩾0)
ABS (X)	The absolute value of X
INT (X)	The largest integer not greater than X
SGN (X)	The sign of X. If X is positive this is set to + 1, if X is zero this is set to zero, if X is negative this is set to −1.
EPS	An estimate of the precision of arithmetic (about 7.28E−12) ie. The smallest value such that 1−EPS<1<1+EPS
INF	The largest numeric value allowed (about 5E76)
RND	The next value from the random number generator.

Note: All angles must be in Radians.

These numeric functions can be used as shown

```
100 LET A = INT ( CPI + R ** 2)
120 LET C = SIN (A + B/2)
130 LET E = F + SQR(27 - P) - RND
```

The random number generator

The random number generator in the system is initially set to yield a sequence of randomly selected numbers. It will, however, always give the same sequence of random numbers unless the programmer resets it using the statement RANDOMISE.

e.g. 140 RANDOMISE

A different sequence of random numbers can be produced by giving the random number generator a number to work with.

e.g. 150 RANDOMISE 92
or 160 RANDOMISE B

There are additional standard numeric functions which relate to matrix handling which are described in Chapter seven.

Chapter four describes how user's own functions can be written.

Printing

The result of a calculation can be printed on the terminal.

e.g.
```
10 LET X = 10
20 LET Y = 20
30 LET Z = X + Y
40 PRINT Z
50 END
```

When this program is run, the output will simply be the number 30.

It is often helpful to tell the person running the program what the output means.

e.g.
```
10 LET X = 10
20 LET Y = 20
30 LET Z = X + Y
40 PRINT "10 AND 20 ARE:"
50 PRINT Z
60 END
```

This will give:—

```
10 AND 20 ARE:
30
LINE 60 DONE
```

Spaces and any printable characters except quotes (") and backarrow (←) can be output by enclosing them within quotes as shown in the example above.

Program Input

READ, DATA and RESTORE

One method of making data available to the program is by storing it using DATA statements and accessing it using READ statements. Use of variables containing strings (e.g. B$) is described in more detail in the next chapter.

e.g.
```
10 READ A, B$
20 IF A = 99 GOTO 80
30 PRINT B$
40 LET C = A*4
50 PRINT C
60 GO TO 10
70 DATA 25, CAR, 15, BUS, 98, TRAIN, 99, AAAA
80 END
```

Both numbers and strings can therefore be input to the program in this way. The data must be the correct category for the variable into which it is read. If an attempt is made to read a string into area A, the message "INVALID DATA" will appear. DATA statements may be written anywhere in the program, and due to the way the NSP BASIC system is written it is advisable to locate the DATA statements close to the relevant READ statement.

The RESTORE statement is used to reposition the data pointer in the program's DATA statements, if it is necessary to reread some data. If RESTORE is followed by a statement number, the pointer is set to the first DATA statement at or after this number. If RESTORE is written alone, the pointer returns to the start of the first DATA statement.

The INPUT statement

It is often convenient to write a program so that different sets of data can be presented to it when it is being run from the terminal. This can be done using the INPUT command.

e.g.
```
NAME MULT
10 INPUT A, C
20 LET B = A * C
30 PRINT B
40 END
RUN
MULT
? 2,2
4
LINE 40 DONE
```

When this program is run the program will print out ? followed by a space and then the person running the program can input the numbers they choose, separated by a comma. Output will be produced when the PRINT statement is encountered. It is a good practice to output a message to the person running the program so that they know what kind of input is expected. It is also a good practice to check what has been typed in case the person running the program makes an error. Character strings may also be input in this way. If non-numeric characters are typed and the variable is not labelled with a $ or a £ sign, the program will halt with the message:

linenumber BAD INPUT RETYPE FROM ITEM 1

if the first item was wrong. The question mark and space will appear again and an appropriate number can be input.

The LINPUT statement

This statement allows input of whole lines of text from the terminal to a string variable. The statement may be followed by a string-variable list,

e.g. `20 LINPUT A$, B$, C$`

and when the lines of text are input they are assigned in turn to these string variables.

Simple loops and branches

In the simple examples of programs which we have given, the program has started at the first statement, performed each subsequent statement once and ended at the last statement. However, it is often convenient to make the program loop round a sequence of instructions and perhaps halt at some statement other than the last statement.

A program run can be halted using a stop statement. However, the END statement must also be present.

e.g.
```
10 LET A = 20
20 LET B = 10
30 LET A = A + B
40 PRINT A
50 IF A = 80 STOP
60 GO TO 30
70 END
```

This program will print the numbers, 30, 40, 50, 60, 70 and 80. Statement number 30 will give A a new value which is equal to its old value plus the value of B. Every time the statement GO TO 30 is obeyed the program will loop round and execute statements 30, 40 and 50. When A = 80 the program will stop.

More complex formats of the IF statement are given in Chapter four.

A more efficient way of creating program loops is to use the FOR instruction.

FOR loops

The FOR statement is used to set up a loop. It is in the form FOR X = A TO B STEP C

e.g.
```
100 FOR X = 1 TO 10 STEP 2
110 PRINT X
120 NEXT X
```

X is initially set to 1. It is then tested to see whether it is greater than 10, and if not it is printed. The 'NEXT' statement is then encountered and X is set to the next value, i.e. 1 + 2 = 3, and so on.

X will increase in steps of 2 until the test X >10 gives the answer true, i.e. when X = 11 control will then pass to the statement after the NEXT. This example will thus print
1 3 5 7 9

If the STEP is 1 the 'STEP C' part may be omitted.

e.g.
```
100 FOR X = 1 TO 10
```
is equivalent to
```
100 FOR X = 1 TO 10 STEP 1
```

Running programs

The current program will be executed following the command: —

RUN — which will start the program at the first line

or

RUN *statement number* — which will start the program at the given line.

e.g. RUN 40 — will start the program at line 40.

More complex formats of the RUN command are given in Chapter eight.

The program run can be stopped in two ways:

press "A" while holding the "control" key down — during output or processing (CTRL and 1 if using an ICL VT).

press "Z" while holding the "control" key down and then "Return" — in reply to a request from an INPUT statement.

Changing the current program

Adding additional lines and changing lines

The program currently held in store may be altered in several ways. Additional lines may be added by typing in the line numbers and BASIC program statements as normal.

e.g. If the current program was

```
10 REM MY PROG
20 PRINT "ANYTHING AT ALL"
30 END
```

and the following lines were entered

```
10 REM MY NEW PROG
25 PRINT "ADDITIONAL LINE"
```

the current program would then consist of the following: —

```
10 REM MY NEW PROG
20 PRINT "ANYTHING AT ALL"
25 PRINT "ADDITIONAL LINE"
30 END
```

Note that the old line 10 has been overwritten by the new.

A stored program may be added to the end of the current program using the APPEND command.

e.g. APPEND *anyprog* — this would add the program *anyprog* onto the end of the current program.

Note:

The line numbers used in the appended program must have line numbers greater than those in the current program.

Deleting lines

A line may be deleted by entering its line number only, before pressing the ACCEPT key.

Alternatively, one or more lines may be deleted with the DELETE command. This has the same series of formats as the LIST command (see below).

e.g. DELETE *stno,stno* — this would delete all the statements of current program within the given range, inclusive (*stno* = statement number)

Inspecting the current program

Your current program may be inspected by means of the LIST command which has four forms: —

LIST	lists all of the current program
LIST *stno*	lists the given statement of the current program
LIST *stno*,	lists the current program from the given statement to the end
LIST *stno*,*stno*	lists all of the current program between the given limits inclusive.
LIST, *stno*	lists the current program from the beginning to the given statement.

e.g. LIST 20
```
20 PRINT "ANYTHING AT ALL"
```

Naming a program in store

Before a new program is stored on disc, it must be named. The command NAME may be used for this purpose.

e.g. NAME FRED

When this is typed and "accept" is pressed, any BASIC lines that are already in store are named "FRED". Lines may be added or amended and these alterations will be included in the named program.

An alternative to NAME is the NEW command.

e.g. NEW FRED

This command will delete any BASIC lines already in store when it is entered so that only lines typed after the command are included in the named program.

Storing programs on disc

Programs held in store may be stored on disc for use at a later time. This is done as follows: —

SAVE	This will open a new file on disc with the name of the current program and save the current program in it. Users logged on under other usernames cannot access this program.
SAVE RUN	As above but other users can GET and RUN the program.
SAVE SHARE	As above but other users can GET, RUN, LIST and copy the program.

e.g. SAVE
SAVE RUN

Note:

1 Programs to be stored on disc must be named.

2 To store an edited program under the original name, the original stored program must first be deleted.

Listing details of stored programs

Details of all user files currently stored on disc are available to the user. The commands to obtain these details are: —

```
CATALOGUE          lists the names and types of all user files
CATALOGUE  .id     lists the names and types of the sub-users files.
```

More complete details may be obtained by typing

```
        CATALOGUE FULL
or      CATALOGUE  .id/FULL
```

which will produce a table showing file name, file type (e.g. B = BASIC program file, I = internal format file, D = terminal format file), date last used, number of buckets taken and access (e.g. Share, User, Write, Run).

(A bucket is a unit of storage on disc).

In addition to users' files, files of general library programs available to all users can be accessed. The names and type of all library files can be obtained by the command: —

```
LIBRARY
```

Unless access is prevented it is possible to list the catalogue of other users.

e.g. `LIBRARY FRED`

where FRED is the username for the catalogue to be listed.

e.g.
```
LIBRARY
LIBRARY CLASS3
CATALOGUE  .JP/FULL
```

Retrieving stored programs

A file which has previously been saved on disc may be run or edited when it has been loaded into store with one of the commands: —

```
        GET name
or      OLD name
```

where *name* is the program name under which it was stored.

e.g. `GET FRED`

To load a library program for use, a "$" or "£" sign must be typed before the program name.

e.g. `GET $LIBPRO`

If another user's files are not restricted they may also be accessed.

e.g. `GET TEST,LORENS`

where TEST is the name of the program to be loaded and LORENS is the user-name under which the program is stored.

Deleting stored programs

The following commands may be used to delete unwanted stored programs: —

KILL *name*
or UNSAVE *name*

where *name* is the name by which the program is stored.

Once these programs have been deleted they can **not** be recoverd.

e.g. KILL FRED will delete the program named FRED from the user's disc file.

The current program may be deleted from store by means of the SCRATCH command

e.g. GET FRED
SCRATCH

This will cause the program FRED to be loaded into store and then deleted from store.

Note:

All commands except HELP may be abbreviated to their first three letters.

e.g. CAT
LIB

Chapter three

String handling

A string constant is a sequence of characters forming a single data value.

e.g. "BASIC"

String variables are indentified by a £ or $ sign

e.g. A$

The name range for a string variable is thus

A$—Z$ and A0$—Z9$ except where a *string array* is being declared. This would be, say, four strings

e.g. A$(0) = "TOM", A$(1) = "DICK", A$(2) = "HARRY" and A$(3) = "FRED"

Only single letter variable names may be used for arrays hence the range for string arrays is A$(. . .)—Z$(. . .) and a statement such as

10 DIM A3$(4) is invalid.

Note: DIM A$(4,10) would declare an array of 5 × 11 strings = 55 strings.

A string variable can be up to 511 characters in length; if it is of zero length it is known as the null string. Strings may contain any character except "←", which is used to correct mistyped characters.

Use of quotes

Strings must be enclosed in quotes (") except in DATA statements and in response to INPUT.

There is no connection between a numeric and a string variable with the same name, e.g. T and T$ refer to completely different variables, but T$ and T£ refer to the same variable.

The & operator

This operator is used to join together (**concatenate**) two strings. For example, if a surname is held in one string and a first name in another, it may sometimes be necessary to see them as a single string. Thus if A$ = MICHAEL and B$ = JONES, the statement:

```
30 LET C$ = A$ & B$
```

would set C$ equal to "MICHAELJONES",
or if the strings needed a space between the two elements:

```
30 LET C$ = A$ & " " & B$
```

would put C$ equal to "MICHAEL JONES".

The & operator can also be used to make long strings. If card input is used, the 80 column limit only permits strings in DATA statements to be about 70 characters long, but they can be extended by combining them after reading:

```
10 READ A$, B$
20 LET A$ = A$ & B$
30 PRINT A$
```

If after joining the two strings A$ proves to have more characters than can be printed on one line, the string is truncated for printing purposes.

String functions

String functions enable complex string handling to be carried out very easily. The length of a string can be given, the position within it of a particular character, and specific characters can be isolated allowing manipulation of the contents of the string.

Functions giving numeric results

CHR (A$)

This gives the 2903 internal character code of the first character of A$.

```
10 LET A$ = "CAR"
20 LET B = CHR(A$)
```

will set B equal to 35, the code for "C".

LEN(A$)

This gives the number of characters in A$.

```
10 LET A$ = "WEEKEND"
20 LET X = LEN(A$)
```

sets X equal to 7. Spaces are treated as part of the string so that "A CUP OF TEA" has a length of 12. The quotes are excluded. A string constant may be used instead of a variable
e.g. LET D = LEN("WEEKEND")

LIN(A$)

This gives the number of newline characters that appear in a string. If a string A$ is output as:

ONE
newline
newline
TWO
newline
newline
THREE

two lots of two newlines will have appeared, and LIN(A$) will have the value 4.

OCC(A$,B$)

The OCC function will give the number of non-overlapping occurences of B$ in A$, hence if A$ = "THE CAT SAT ON THE MAT" and B$ = "AT", the function will show that there are 3 occurrences by assigning 3 to C in this statement.

```
10 LET C = OCC (A$,B$)
```

The parameters may be literals not variables e.g.

```
10 LET C = OCC ("TRUMPET","T")
```

Because only non-overlapping occurrences are counted, the number of occurrences of "ANA" in "BANANA" would be 1.

POS(A$,B$)

The character position of the start of the first occurrence of B$ in A$. If B$ does not appear in A$, this number will be 0.

```
10 LET A$ = "PROTECTION"
20 LET B$ = "T"
30 LET C = POS(A$,B$)
40 LET D = POS(A$,"O")
50 LET C$ = "Z"
60 LET E = POS(A$,C$)
```

C = 4, D = 3, E = 0.

POS(A$,B$,N)

This gives the position of the start of the Nth occurrence of B$ in A$, and is set to zero if there is no Nth occurrence.

```
25 LET D$ = "ISS"
30 LET E$ = "MISSISSIPPI"
40 LET D = POS(E$,D$,2)
50 LET E = POS(E$,D$,3)
```

In this example, D = 5 since the second occurrence of ISS begins at character 5, and E = 0 since there is no third occurrence.

VAL(A$)

The numeric value of A$, which is a string containing numeric characters.

```
10 LET K = VAL(A$)
```

If A$ = "194" then K would be set to 194.

This function is very useful in processing Terminal Format Files (see Chapter six). Data is input to these files as single strings, and using the string-handling functions, a numeric section of the whole could be isolated. If the VAL function is then applied, the numeric value of this string of numeric characters can be found and normal numeric processing can be performed.

Functions giving string results

CHR$(N)

The character whose 2903 internal character code is N. Thus

CHR$(33) = "A"
CHR$(26) = "*"
CHR$(16) = " "

This function is only designed to decode one character. The statement:

10 LET A$ = CHR$(413544)

will not give the individual character value "ICL" but the character value of the whole number modulo 64, i.e. "H".

DAT$

This gives the date in the format dd/mm/yy

TIM$

This gives the time in the format hh/mm/ss

SEG$(A$,N,M)

The segement of A$ from the Nth character to the Mth character. If M is omitted the segment is from N to the end of A$.

```
e.g.    10 LET A$ = "TRUNCATION"
        20 LET B$ = SEG$(A$,5,7)
```

would set B$ = "CAT" and

```
        10 READ C$, D
        20 LET D$ = SEG$(C$,D)
        30 DATA DRAGONFLY, 7
```

would set D$ = "FLY" taking from character 7 to the end as implied by the SEG$ statement.

SEG$ is a useful function to use in conjunction with the numeric result functions LEN and POS, described earlier. They can be used to scan a string interspersed with a known character, e.g. a space or a comma, to allow specific parts of the string to be accessed. For example, if a string contains a first name and a surname with a space between them, it is possible to isolate and print the surname alone, or the whole name in surname first name order.

e.g. 10 LET A$ = "JOHN EVANS"

The position of the first character of the surname will be the position of the space + 1.

20 LET B = POS(A$, " ") + 1
In this case B = 6

The end of the surname is the end of the string itself, so the function which gives the length of a string will point to the final character position.

30 LET C = LEN(A$)
Here C = 10

We now know that the surname is the section of A$ from the value of B to the value of C and the SEG$ function will isolate this section

40 LET D$ = SEG$(A$, B, C)
D$ now holds the surname only, "EVANS".

These functions can also be used if a single string holds several pieces of information all separated by, for instance, a comma.

```
20 DATA "MR.R JONES,20 PARK DRIVE, WIGAN, LANCS"
30 DATA "MRS.J BUTLER,120 GREEN LANE, CHATHAM, KENT"
40 DATA "MRS.L MITCHELL,57 VAUGHAN ROAD, LEEK, STAFFS"
50 DATA "MR.R ELLIS,91 MELROSE AVENUE, GRAVESEND, KENT"
```

It could be ascertained how many of the householders are men by isolating the section up to the first full stop. To find how many of the people live in Kent LEN, POS and SEG$ can be used to find the value of the section after the third comma.

SEG$ can also be used in conjunction with other functions. If it is necessary to find out how many seconds a program has been running for the following can be used:

```
10 LET A = VAL (SEG$(TIM$,7))
15 PRINT A
20 END
```

This gives the numeric value of the last two characters of the time, the number of seconds. If the time was 11/45/20, it would hold the value 20. This could be done at various points in the program to monitor how long the processing took.

SUB$(A$,N,M)

This function allows a subsection of a string to be isolated. It differs from SEG$ in that M is the number of characters starting from N: it does not end at the Mth character. If M is omitted, the Nth character of A$ is given.

```
10 LET A$ = "EXPORT"
20 LET B$ = SUB$(A$,3)
```

B$ is then equal to "P".

```
10 LET A$ = "SUBSECTION"
20 LET B$ = SUB$(A$,4,4)
```

B$ is then equal to "SECT".

STR$(X)

The number X expressed as a string. This is the opposite of VAL in that it assigns a number to a string.

It is useful in printing numbers with no surrounding spaces. In the normal way a positive number has a leading space and a trailing space and a negative number has a trailing space. Thus

```
        10 PRINT 99                          gives  ∇99∇
but     20 PRINT STR$(99)                    gives  99

        30 PRINT -99;-99;99                  gives -99∇-99∇ ∇99∇
but     40 PRINT STR$(-99);STR$(-99);STR$(99) gives -99-9999.
```

GAP$(N)

This function gives a string of N spaces.

```
10 A$ = "THIRD" & GAP$(3) & "FLOOR"
20 PRINT A$
```

gives "THIRD FLOOR"

LIN$(N)

LIN$(N) gives N new lines. This can be used to space out strings for printing.

```
10 READ A$, B$, C$, D$
20 LET T$ = A$ & LIN$(1) & B$ & LIN$(1) & C$ & LIN$(1) & D$
30 PRINT T$
40 DATA "THE TIME HAS COME THE TEACHER SAID"
50 DATA "TO TALK OF MANY THINGS"
60 DATA "OF SEGS AND SUBS AND MIN AND MAX"
70 DATA "OF MATRICES AND STRINGS"
80 END
```

The assignment to T$ of LIN$ characters in line 20 causes the PRINT statement to output the verse as it appears in the DATA statement, avoiding separate PRINT statements for each string read.

MUL$(A$,N)

This function gives a string of N repetitions of A$

```
10 LET A$ = "JINGLE BELLS"
20 LET B$ = MUL$(A$,2)
30 PRINT B$
40 END
```

This causes the output of the following string "JINGLE BELLSJINGLE BELLS"
or with an & function:

```
10 LET A$ = "THREE BLIND MICE"
20 LET B$ = "SEE HOW THEY RUN"
30 LET C$ = MUL$(A$,2) & MUL$(B$,2)
40 PRINT C$
50 END
```

will give
THREE BLIND MICETHREE BLIND MICESEE HOW THEY RUNSEE HOW THEY RUN

SGN$(X)

SGN$(X) gives + if X is greater than zero.
— if X is less than zero.
space if X is equal to zero.

This can be used to test if X is positive or negative. It may be useful in printing eg.

```
e.g.    10 PRINT "X IS A "; SGN$(X); "VE NUMBER"
```

So that if X is not zero the following will be printed:

```
        X IS A +VE NUMBER
or      X IS A -VE NUMBER
```

DEL$(A$,B$,N)

This function deletes from A$ a single occurrence of B$, the Nth if N is specified, the first if it is not:

```
10 LET A$ = "ACCOUNTING"
20 LET B$ = "COUN"
30 LET C$ = DEL$(A$,B$)
```

C$ is then equal to "ACTING"

```
10 LET X$ = "DESSERT"
20 LET Y$ = DEL$(X$,"S",2)
```

Y$ is then equal to "DESERT"

SDL$(A$,B$,N)

This is also a deletion function, but is used when more than one occurrence of a string is to be deleted. If N is omitted or equals zero, all occurrences of B$ in A$ are deleted. If N appears the first N occurrences are deleted.

```
e.g.    10 LET A$ = "ESTEEMED"
        20 LET B$ = "E"
        30 LET C$ = SDL$(A$,B$,3)

        will set C$ = "STMED"

        10 LET X$ = "PEPPER"
        20 LET Y$ = SDL$(X$,"P")

        will set Y$ = "EER"
```

REP$(A$,B$,C$,N)

Rather than simply deleting part of a string, it may be necessary to replace a section of it with different characters. This function replaces B$ in A$ with C$. It replaces only one occurrence of the string: the Nth if N is specified, the first if it is not.

```
e.g.    10 LET A$ = "CONSONANT"
        20 LET B$ = "ULT"
        30 LET C$ = REP$(A$,"ON",B$, 2)

        will set C$ = "CONSULTANT"
```

SRP$(A$,B$,C$,N)

This performs a similar function to REP$. It replaces one character string with another, but in more than one place. If N is omitted or equals zero, all occurrences of B$ are replaced by C$ in A$; if N appears, the first N occurrences are replaced.

```
e.g.    10 LET A$ = "I CAME, I SAW, I CONQUERED"
        20 LET B$ = "I"
        30 LET C$ = "HE"
        40 LET D$ = SRP$(A$,B$,C$,2)

        will set D$ = "HE CAME, HE SAW, I CONQUERED" and combining the general
        replace function with the specific one:

        10 LET A$ = "THE BOYS WANT TO CLIMB THE TREES"
        20 LET B$ = SRP$(A$,"S","")
        30 LET C$ = REP$(B$," ","S",3)

        will set C$ = "THE BOY WANTS TO CLIMB THE TREE"
```

These string handling functions can be nested together to any depth, the limit being line length. For example:

```
        10 LET A$ = "ABRACADABRA"
        20 LET B$ = "CADAB"
        30 LET C$ = SUB$(A$,OCC(B$,"A"),POS(A$,B$,1))
```

C$ is then equal to "BRACA" since it is five characters of A$ starting from character two.

Using "&", several string handling functions can be put together thus:

```
        10 LET A$ = "STANDARD"
        20 LET B$ = "AND"
        30 LET C$ = TIM$&DAT$&SGN$(-4)&SEG$(A$,POS(A$,B$), LEN(A$))
```

C$ would then be equal to "16/19/3828/02/77-ANDARD".

The CHANGE statement

A statement such as:

```
30 CHANGE W$ TO X
```

places numerical codes for the characters of W$ into consecutive elements of array X. X must be one dimensional and if it is not large enough to hold the string an error will be given. The element X(0) will contain the number of characters in the string. The numerical instructions of BASIC can then be used for working on the characters, and the string reassembled with the statement:

```
50 CHANGE X TO W$
```

Here the value in X(0) will tell the system how many characters to form into a string. The following example shows a four-character string being reversed using CHANGE

```
e.g.    10 DIM X(4) , Y(4)
        20 READ W$
        30 CHANGE W$ TO X
        40 LET Y(0) = X(0)
        50 FOR J = 1 TO 4
        60 LET Y(5-J) = X(J)
        70 NEXT J
        80 CHANGE Y TO W$
        90 DATA RATS
       100 END
```

This sets the reversed W$ to "STAR". Line 40 sets the first element of Y to the number of characters in the string. Line 60 sets the elements of Y equal to the reverse of the elements of X.

Chapter four

Loops and branches

Conditional statements

The IF statement

This condition takes the form

IF *condition action*

If the condition is true the action is peformed, otherwise the next statement is obeyed. One of two types of action may follow an IF statement:

a) The program may be directed to another line with a statement such as

```
10 IF A > 20 THEN 200
or   10 IF A > 20 GO TO 200
```

which are exactly equivalent. If the line number 200 did not exist, the program run would be terminated. If line 200 contained a non-executable statement such as REM, DATA, DEF or DIM, control would pass to the first line after 200 that contained an executable statement.

b) A subsidiary statement could follow the IF. This may be any statement other than DATA, DEF, DIM, END, FNEND, FOR, NEXT and REM.

e.g. 10 IF A = B PRINT C

Logical conditions

These may take several forms.

a) *numeric expression relation operator numeric expression*

The *relation operator* may be

<or = or>

or any two of these together e.g. > =

e.g. IF (X+1) > (Y/2) GOTO 300

b) *string expression relation operator string expression*.

e.g. IF P$ = "CEDRIC" PRINT "GOOD MORNING"

a) & b) above are called **logical terms** and may be used alone or linked together using **logical operators**.

The logical operator may be

AND, OR, XOR, IMP, EQV (AND, OR, EXCLUSIVE OR, IMPLICATION, EQUIVALENCE)
which have the following truth tables.

A	B	A AND B	A OR B	A XOR B	A IMP B	A EQV B
T	T	T	T	F	T	T
T	F	F	T	T	F	F
F	T	F	T	T	T	F
F	F	F	F	F	T	T

e.g. IF (X+1) > (Y/2) AND P$ = "CEDRIC" INPUT A$

This will only give the answer TRUE if both logical terms are true.

A logical term may also be

a) NOT *logical term*

e.g. NOT A = B

This will only be true if A is NOT equal to B.

b) (*logical condition*)

That is any of the forms mentioned above in brackets.

e.g. NOT A = B OR (X+1 > Y/2 AND P$ = "CEDRIC")

The part in brackets is always evaluated first. The condition will be true if either the first part **or** the second part is true.

Note on relation operators

If the relation operator consists of two symbols then it is as if the condition were duplicated, once for each of the symbols with an OR between them

e.g. A = > B

is equivalent to A = B OR A > B
(which is also equivalent to NOT A < B)

In relations between strings < (less than) is interpreted as "earlier in the alphabet than" and > (greater than) as "later".

e.g. "CAR" < "CARD" < "CART HORSE" < "CARTER" < "CAT"

Logical conditions are evaluated in the following order:

parentheses (brackets)
numeric and string expressions
relation operators (> = <)
NOT
AND
OR
XOR
IMP
EQV

Examples

```
40 IF X+1 = > 17 THEN 150
50 IF P$ = "FRED" PRINT P$
60 IF X = 1 AND Y = 1 GOTO 170
70 IF P$ = "FRED" OR NOT Q$ = "HARRY" THEN 180
80 IF NOT (X = 1 AND Y < 1) GOTO 190
90 IF (A MAX B) > 10 THEN 200
```

ON . . . GOTO

The ON statement will transfer control of the program to one of a selection of statements or subroutines.

It takes the form of

ON *numeric-expression* (GOTO) *statement number list*
(THEN)
(GOSUB)

e.g. `60 ON X GOTO 140, 180, 220`

X will be rounded to the nearest integer and control will be transferred to 140, 180 or 220 depending on whether X = 1, 2 or 3.

If X < 1 or X > 3 then control will pass to the next statement after the ON statement.

Subroutines

Subroutines are self contained sections of program. They are accessed using the GOSUB statement.

e.g. `100 GOSUB 2000`

This will transfer control to the specified statement number. On encountering a RETURN statement control will be returned to the statement following the GOSUB.

```
e.g.    100 GOSUB 2000
        110 REM CARRY ON
          .
          .
          .
        2000 PRINT "**********"
        2010 PRINT "ERROR"
        2020 PRINT "**********"
        2030 RETURN
```

Subroutines are useful for frequently used pieces of code. Instead of writing a set of instructions many times it is written just once saving time and storage space.

Subroutines use the same variables and data areas as the main programs, and have no parameters.

FOR– NEXT loops

Simple FOR-NEXT Loops are introduced in Chapter two. In FOR statements the STEP may be fractional or negative or both.

e.g. 100 FOR X = 10 TO 1 STEP - 0.5

If the STEP is negative the test X < *limit* is made (e.g. X < 1)

X will be set equal to 10, 9.5, 9, 8.5. . .2, 1.5, 1, 0.5

When X = 0.5 the test X < 1 is TRUE and the loop ends.
The initial and final values for X may also be negative or fractional or both.

e.g. 100 FOR I = –3 TO –5.75 STEP –0.25

If an impossible loop has been asked for

e.g. 100 FOR N = 1 TO 4 STEP –1

N will be set to the initial value specified and then tested against the final value. In the above case the STEP is negative therefore the test will be N < *limit* i.e. N(= 1) < 4 which is true. So control will be passed immediately to the statement following the corresponding NEXT. Note that N remains set to the initial value.

At the end of any FOR Loop the value of the variable will be the value which satisfied the condition.

e.g. 100 FOR X = 1 TO 10

X will equal 1, 2, 3, ..., 9, 10, 11.

When X = 11 the test X > 10 is true.

X retains the value 11.

FOR statements may be nested within each other

e.g.
```
100 FOR X = 1 TO 3
110 FOR Y = 1 TO 3
120 PRINT A (X,Y)
130 NEXT Y
140 NEXT X
```

The inner loop will run three times for each value of X, so that the following will be printed

```
A(1,1)
A(1,2)
A(1,3)
A(2,1)
A(2,2)
A(2,3)
A(3,1)
A(3,2)
A(3,3)
```

Care should be taken to make sure the NEXT statements are always present to close each loop, and that they are in the right order.

e.g.
```
100 FOR X = 1 TO 3
110 FOR Y = 1 TO 3

200 NEXT X
210 NEXT Y
```

is incorrect.

Function definitions

It is possible to define functions using the DEF statement. Certain functions such as SQR (to find a square root) or SIN (to find the sine) are already set up.

The DEF statement can be used to define a function

e.g. `40 DEF FNA(X) = X*X+1`

From then on in the program to calculate, for example, $P^2 + 1$

```
100 LET B = FNA (P)
```

may be used.

The DEF statement can also be used to introduce a **function definition block** which is a set of statements terminated by an FNEND statement.

This form would be used when the function definition required more than one statement line. The result of a function definition should be a single value—either numeric or string.

All function names take the form FN*variable name*.

It should be indicated in the function name whether the result is a numeric value or a string.

e.g. FNX for a numeric value
FNA$ for a string value.

A function block may contain any statements other than DEF, FNEND or END. Any FOR—NEXT statements must be matched within the block.

It is not possible to jump into or out of a function definition block.

Parameters and local variables

Functions may be defined with or without parameters and with or without local variables. These exist only while the function is being evaluated, thus it is quite permissible to use the same variable names within a definition as outside without fear of corruption.

When a function is used the parameters must be the same in number and type as those defined.

e.g.
```
40 DEF FNA (X, Y$)
.
.
90 FNEND
.
.
200 LET B = FNA (P, Q$)
```

Local variables can be defined for use within a function definition block only. If a variable is used which is not locally defined it is taken to be a main program variable. In the example below A and P$ are local variables.

e.g.
```
 40 DEF FNB (X, Y, Z$) A, P$
 50 LET P$ = Z$ & Z$
 60 LET A = LEN (P$)
 70 IF A > X + Y THEN 100
 80 FNB = X + Y
 90 GOTO 110
100 FNB = A
110 FNEND
.
.
400 REM FUNCTION CALL
410 LET Q = FNB (V, W, T$)
```

Local variables are undefined at the start of the evaluation, even if RUN CLEAR is used.

It is possible to define a recursive function. See the 2903 BASIC: Reference manual for further information.

Chapter five

Program output

Printing— more complex formats

Printing was introduced in Chapter two, but some points need to be made about formatting.

It is often convenient to print several items on the same line. To print the items immediately adjacent, the variables may be separated with semi-colons.

e.g. 30 PRINT A;B;C;D

Numeric fields are preceded either by a space or a minus sign and followed by a space, so in this case if A = 54, B = 2, C = −374 and D = 21, the output will be

∇54∇∇2∇ −374∇∇21∇

Character strings do not have leading or trailing spaces, so a statement such as:

50 PRINT A; "DATA";C;D

will be output as

∇54∇DATA−374∇∇21∇

and

40 PRINT "MORE"; "DATA"

will be output as

MOREDATA

Data can be spaced across a page using a comma as a separator. The page is divided into fifteen-character zones and the comma causes each item to be printed at the beginning of a zone. To skip a zone, two commas may be used as a separator.

e.g. 10 PRINT A, "MORE", "DATA"
will be output as

54	MORE	DATA

and

10 PRINT A,, "MORE"
will be output as

54	MORE

showing that a whole fifteen-character zone has been skipped. A print line may have a mixture of separators if some items need to be printed adjacent to each other and some need to be spaced across print zones.

Printed output may be located at a specific print position by using TAB in the PRINT statement

e.g. 10 PRINT A; TAB(6); "MORE"

The number in brackets is the position where printing should start. A will be output at print position one followed by "MORE" starting at print position six.

If the statement PRINT appears alone on a line, a blank line is printed. This is useful in spacing output.

If a PRINT statement line ends with a comma or a semi-colon, the next PRINT statement in the program will print on the same line

e.g.
```
10 LET X = 1
20 LET A$ = "ABC"
30 PRINT A$;
40 LET X = X + 1
50 IF X < 5 GOTO 30
60 END
```
will produce

ABCABCABCABC

The system will assume that the line length is seventy two print positions so that if the number of characters to be printed exceeds this, the seventy third character will appear in the first print position on a new line. Similarly with a comma separator, the sixth element will appear on a new line, as five zones can begin on each line.

The maximum line length can be reset to a lower or higher number using the MARGIN statement.

e.g. 40 MARGIN 25

will set the maximum line length to 25 print positions. If at any time during a session the maximum line length is altered, it will remain at the new setting throughout all program runs until it is explicitly changed.

Summary of print editing

";" separator	text printed adjacent, with numerics preceded by a space or a minus sign and followed by a space.
"," separator	text printed at the beginning of the next fifteen-character print zone.
TAB(n)	text printed from print position n.

The PRINT USING statement

This statement allows printed output to be formatted. Numerics are introduced by + or — (for sign suppression) and character strings by < to left-justify or > to right-justify. Integer fields are printed right-justified with non-significant zeros suppressed unless an @ sign appears after the + or —. The # sign reserves places for characters within the formatted list. An exponent field is introduced by ↑↑↑↑

e.g.
```
10 FOR I = 1 TO 4
20 READ A$, A
30 PRINT USING "THE COST OF A<#####  =£+@#.##":A$,A
40 NEXT I
50 DATA COAT, 2500, WATCH, 17.50, RING, 47.80, PEN, 02.20
60 END
```

This program will insert the appropriate character string into the first part of the format, left-justified, and the amount with its decimal point and non-significant zeros suppressed into the second part. The result is as follows:

```
THE COST OF A COAT    = £ + 25.00
THE COST OF A WATCH = £ + 17.50
THE COST OF A RING    = £ + 47.80
THE COST OF A PEN     = £ + 02.20
```

The print format in line 30 could have been assigned to a string variable as follows

```
25 LET G$ = "THE COST OF A<#####  = £+@#.##"
30 PRINT USING G$:A$,A
```

Program execution will halt if the variables to be entered do not match the format to which they have been assigned. Program output includes output to files, and MAT PRINT output, which are explained in Chapters six and seven respectively.

Chapter six

File handling

Introduction

Data may be presented to programs in files instead of via DATA statements or INPUT statements. This allows the information to be accessible by more than one program. A file is a linear arrangement of data external to all BASIC programs which can be read and amended by BASIC statements.

Two kinds of file can be accessed by the BASIC system. These are known as **terminal format files** and **internal format files**.

Terminal format files

These are made up of variable length records each of which resembles a line of printed characters. They are accessed serially and can be listed on a terminal.

Statements and commands which process terminal format files

The OPEN command

This command is used outside the program to create a file and takes the form:

OPEN *name*, *recs*, *access*

with all parameters optional except *name*.

e.g. OPEN TFILE,50,SHARE

will open a terminal format file called TFILE with fifty records which may be accessed by all users.

The other access modes are:

USER may only be accessed by the user who created it.
READ may be read by other users but not amended by other users.
WRITE equivalent to SHARE. The file may be read and written to by any user.

The default number of records is 100. The filename may be up to six alphanumeric characters with the first alphabetic. When the file is created by OPEN it contains no data.

The KILL command

When a file has been opened it remains on the disc until it is deleted by the KILL command, which has the format

KILL *filename*

The EXTEND command

If it is found that more records are needed on a file than the number specified in the OPEN command, the file may be extended.

e.g. EXTEND TFILE, 24

will put space for twenty four extra records in TFILE.

The GET command

A terminal format file can be brought into store from the disc by means of the GET command, in the same way as a BASIC program

e.g. GET TFILE
GET TFILE, USER

The LIST command

The contents of a terminal format file can be listed in the same way as a BASIC program. The formats of the LIST command are the same as those described in Chapter two, allowing a whole file or specific records to be listed. The record number, starting from one, is output, followed by the data in the record.

e.g. GET TFILE
LIST 10, 12

```
TFILE
10 TENTH LINE
11 NO. ELEVEN
12 12TH LINE
```

The DELETE command

Similarily, records can be deleted in exactly the same way that lines are deleted from a BASIC program. See Chapter two for formats.

e.g. GET TFILE
DELETE 11
LIST

```
10 TENTH LINE
12 12TH LINE
```

To add new data the procedure is to type the record number followed by the data. This will be incorporated into the file.

BEWARE DELETE with no parameters deletes the **whole** file.

The FILE statment

This is used to link a previously created data file to the program. It assigns to the file a channel number between 1 and 6, and all input and output statements refer to the channel number rather than to the file name. This number is preceded by a #. In the following example a file is opened and the first line of the BASIC program then assigns the file to a channel number.

e.g. OPEN TFILE2, 5, SHARE

```
10 FILE#1: "TFILE2"
```

As the access mode in the OPEN command is SHARE, this file could be read from another username. If TFILE2 had been created in username USERO1, it could be assigned to a program within another username in this way:

```
20 FILE#6: "TFILE2, USER01"
```

The channel number can be simply a number, as in the examples above, or it can be expressed as a numeric variable or arithmetic combination of numeric variables.

e.g. 15 FILE#A + B:"NAME"

where it would be assigned to channel 4 if A + B = 4.

The file statement is also used to end channel associations to free the channel for use by another file. If channel 2 were to be freed the following statement would be used:

50 FILE#2: "*"

The PRINT statement

Terminal format files are intended to resemble lines of print as they would appear on a terminal, thus data can be written to the file using the PRINT statement. In the following example the data is read by the program from DATA statements and then printed into the file.

e.g.
```
OPEN TFILE3
10 FILE#1: "TFILE3"
20 FOR T = 1 TO 3
30 READ A, B, C$
40 PRINT#1 : T;A*B,A; " ";B,C$&"HORSE"
50 NEXT T
60 DATA 2,3, PACK, 1,6, CART, 4,2, CLOTHES
70 END
```

If the file were listed, the following would be output:

```
1  6          2  3          PACKHORSE
2  6          1  6          CARTHORSE
3  8          4  2          CLOTHESHORSE
```

The comma separator causes spacing across the line and the semi-colon separator prints elements adjacent to each other. The maximum line length for a terminal format file is initially 72 characters, but this can be reduced or increased up to 124 by the MARGIN statement described in Chapter five. When MARGIN refers to a file it has the following format.

e.g. 10 MARGIN#2:48
or 10 MARGIN#X:Y + 3

If the current line length is not known the MRG Function can be used to find it.

e.g. 30 Y = MRG (#2)

will set Y equal to the line length of the file occupying channel 2.

The formatted print statement PRINT USING, described in Chapter five, may also be used for writing to terminal format files.

e.g. 40 PRINT#2:USING "£—###.##":X

or if A$ = "£—###.##"

40 PRINT#2:USING A$:X

This will store £∇∇ 12.02 if X = 12.02

The INPUT and LINPUT statements

Data can be read from terminal format files by means of these two statements. INPUT will enable the data from one PRINT statement to be made available in a series of numeric or string variables in the same format in which it was originally stored using PRINT, with each variable separated by commas. LINPUT will present to the program all the characters written in one print statement as a single string, so that

e.g. 20 LINPUT#1:A$

will deposit the next line of data from the terminal format file into the string variable A$. Access to these files is always serial.

Examples

If a terminal format file contains the following data:

```
1 ONE, TWO
2 THREE, FOUR
```

the use of INPUT and LINPUT will result in the formats shown below.

```
10 FILE#1: "FILE01"
20 INPUT#1: A$,B$
30 LINPUT#1: C$
40 PRINT A$
50 PRINT B$
60 PRINT C$
70 END

RUN
ONE
TWO
THREE, FOUR
LINE 70 DONE
```

The IF MORE and IF END statements

These two statements are used to detect the end of the data in a file

e.g. 100 IF MORE#1 GOTO 20

If the pointer for the file has more data following it, the program will go back to statement line 20.

IF END is the inverse of IF MORE.

The SCRATCH statement

This is used to delete all the data held on a file without erasing the file itself. The pointer is set to the start of the file occupying the specified channel.

e.g. 50 SCRATCH #1

The RESET statement

This is used to reset the record pointer to the beginning of the file occupying the specified channel so that the data may be accessed again

e.g. 40 RESET #2

The RESTORE statement is synonymous, but by convention it is used to reset the pointer in program DATA statements, whereas RESET is used when referrring to files.

File handling functions

The LOC function gives the current record number. It can be used for example to count the number of lines input to or output from a file.

e.g. 40 IF LOC(#1) = 20 GOTO 100

will pass control to line 100 when the 20th record is accessed.

The LOF function will give the number of records in the file as stated in the OPEN command, modified by any EXTEND commands.

e.g. 50 PRINT LOF(#6);

The TYP function can be used to identify the type of the next item. It is set to 1 if the item is numeric, 2 if string, 3 if end of record, 4 if end of file where end of file is determined by the size given in the OPEN command and any further extension.

TYP can be used as shown

e.g.
```
 10 FILE : "TFILE"
 20 LET F = TYP(#1)
 30 PRINT "TYP=";F;
 40 ON F GOSUB 100,200,300,320
 50 GO TO 20
100 INPUT#1:X,
110 PRINT " NUMBER=";X
120 RETURN
200 INPUT#1:X£,
210 PRINT " STRING=";X£
220 RETURN
300 PRINT " END OF RECORD";LOC(#1)
305 RESET#1,LOC (#1)+1
310 RETURN
320 END

RUN
TYP = 1   NUMBER = 14
TYP = 1   NUMBER = -23
TYP = 2   STRING = ABCD
TYP = 3   END OF RECORD 1
TYP = 1   NUMBER = 3
TYP = 2   STRING = EF
TYP = 2   STRING = GHIJK
TYP = 1   NUMBER = 4
TYP = 3   END OF RECORD 2
LINE 30 NUMBER NOT PREVIOUSLY WRITTEN TO THIS FILE.
```

Internal format files

These are made up of fixed format, fixed length records. Their fields can be in binary or character form and access to the records can be serial or direct.

Statements and commands which process internal format files

Some of the commands and statements which act on terminal format files also apply to internal format files. It may be assumed that those mentioned here are identical for both files unless variations are mentioned in this section, and that those not mentioned may not be used with internal format files.

The OPEN command

The format for an internal format file is OPEN name (record format), recs, access.
When creating an internal format file, the record format must be specified. This is done in the following way:

N = 1 numeric value

S*n* = 1 string value with *n* characters

*m*N = *m* numeric values

*m*S*n* = *m* string values with *n* characters each.

e.g. OPEN IFILE(3N,2S4,2N,S20,N),30,WRITE

Where the record format is three numeric fields, two four-character strings, two numeric fields, a twenty-character string then one numeric field. The data which is input to the file must then correspond exactly with the original format. If a user forgets the format he gave his file on opening, a program called %FORMAT exists to give this information. It requests the user to type in the name of the file. It is necessary only to type %FORMAT to GET and RUN this program.

e.g. %FORMAT
SUPPLY FILENAME? IFILE
30 RECORDS FORMAT 3N, 2S4, 2N, S20, N

Note: If the format definition ends with a comma then flow from one record to the next is allowed for the file; this means that read and write operations on the file which attempt to go beyond the end of a record pass on to the next record. If the comma is not present this flow is not allowed.

e.g. OPEN IFILE (10N,)

The number of records given will be the number specified on opening — if only fifteen records have been written to the file, %FORMAT will still give thirty as number of records.

Internal format files can be KILLed and EXTENDed. The FILE statement is identical for terminal format and internal format files.

The WRITE statement

Data can be written to an internal format file either serially or record by record.

e.g. WRITE #1: A,B,C,D$,

will write the three numbers and the character string to the next record of the file 1. Clearly this activity will fail if the file has not been opened with a format like 3N, 1S4. If the string D$ contains more than four characters in the example given just the left most four characters will be stored on the file.

Writing will normally start from record 1 on the file.

e.g. WRITE #1,12 : A,B,C,D$

will write data to the 12th record on the file.

The format:

WRITE#X,Y : A,B,C,D$

is also permissable where X and Y are channel and record numbers respectively. To write to the end of the file:

WRITE #1,LOF(#1) : A,B,C,D$

will direct the output to the last record.

The format:

WRITE#1,LOC(#1) + 2 : A,B,C,D$

will direct the output to two records further on from the current file position.

The READ statement

The READ statement is used to move data from an internal format file into a program. Its format is very similar to the WRITE statement

e.g. READ#1 : A,B,C,D$

will read the next record from the file on channel 1 into the numeric variables A, B and C and the string variable D$.

Specific records can be directly accessed.

e.g. READ#1, 7 : A,B,C,D$

will read the 7th record from file #1.

Numeric variables can be used instead of numeric constants for specifying the channel and record number

e.g. IF X = 1 and Y = 3 and Z = 4

READ#X, Y+Z : A,B,C,D$

will read the 7th record from file 1.

The standard file handling function can be used

e.g. READ#2,LOC(#1) : A,D$

would ensure that if the 7th record had just been processed on file 1, the 7th record would be READ on file 2.

LET X = LOF (#2)

would give X the value set up by the command OPEN, modified where appropriate by any subsequent EXTEND commands. If preceding programs have not written to every record on the file the READ action will be terminated by the message

"LINE 40 NUMBER NOT PREVIOUSLY WRITTEN TO THIS RECORD"

where line 40 of the program contains the READ statement. The program can be restarted if necessary using the CONTINUE command but it should be noted that all channel associations will, by then, be lost.

The IF MORE and IF END statements

These two statements are used in the same way for internal format files as terminal format files. They will monitor the end of file as established by the OPEN command, rather than the end of data.

Statements and commands which can be used with all types of files

KILL, EXTEND, FILE, IF MORE, IF END, SCRATCH, RESET are all identical for both types of files.

CHAIN

All the files in a program will normally be closed at the end of a run so that the channel associations set up in the FILE statement disappear. The statement

300 CHAIN "PROG" WITH #2, #1

will enable the program PROG to be run immediately after the current program whilst retaining the channel associations of the current program. All file associations also end on break in or other halts to program execution so that the only file handling statements that can be used with immediate execute (described in Chapter eight) must be preceded by a FILE statement.

Chapter seven

Matrix handling

Declaring

The matrix functions are used to handle arrays. A one dimensional array is called a vector and a two dimensional array is called a matrix. They are declared using the DIMENSION (or DIM) statement.

e.g. DIM A(3) will reserve space for a three element vector

$$[a_1 \quad a_2 \quad a_3]$$

DIM B (3, 4) will reserve space for a matrix with three rows and four columns.

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix}$$

If no bounds are given a 10 × 10 array is assumed.

Note: Generally the lowest bound element of an array is zero. However, when handling a matrix only elements with non zero subscripts are used.

One and two dimensional arrays containing string values may also be used and are declared as:

e.g. DIM A$ (3)
or DIM B$ (2,4) (See Chapter three)

Only certain MAT statements can be used with string arrays; these are denoted appropriately.

The statement OPTION BASE 1 may be used in a program. The effect of this is to set the lower bound of all arrays in that program at one instead of zero. This saves storage space

```
e.g.    10 OPTION BASE 1
        20 DIM A (3), B(4,4)
```

Dimensions and redimensioning

Once declared in a DIM statement the number of dimensions assigned to an array (1 or 2) cannot be changed.

Although the amount of space (i.e. the total number of elements) allowed for an array cannot be increased the current bounds of the array can be changed. This is called **redimensioning**.

If OPTION BASE 1 is **not** used in a program the zero elements in the array must be considered when redimensioning. For example an array declared as A (3,4) has 4 × 5 = 20 elements and may be redimensioned as, say, A (2,5) or A (3,3) as these have 3 × 6 = 18 and 4 × 4 = 16 elements, but not A (2,6) as this has 3 × 7 = 21 elements or A(20) as this has the wrong number of dimensions.

No assumptions should be made about the elements which are no longer used after redimensioning.

Redimensioning is actually done using MAT assignment statements. These are in the form

Line number MAT *destination* = *array expression*.

e.g.
```
120 MAT A = B + C
130 MAT E = IDN
140 MAT F = ZER (2,2)
```

The destination (Left hand side) is redimensioned to the bounds of the array expression (Right hand side) unless this is one of ZER, CON or IDN **with no bounds stated** in which case the array expression is redimensioned to the bounds of the destination.

Examples from above:

If B and C are (4,3) arrays and A is (5,5) then A will be redimensioned to (4,3) and the result of B + C will be assigned to it.

In Line 130 if E is a (4,4) array the RHS will become the 4 × 4 Indentity matrix, which will then be assigned to E. In Line 140 if F is a (4,4) array the RHS will become a 2 × 2 zero matrix, F will be redimensioned to (2,2) and the zero matrix will be assigned to it.

Input and output

Input from and output to a terminal

Vectors, matrices and string arrays can be input from a terminal using the MAT INPUT statement.

e.g.
```
10 DIM A$ (4)
20 MAT INPUT A$
.
.
999 END
RUN
? 1, 2, 3, 4, 5
? 6, 7, 8, 9
```

Items are input to 2 dimensional arrays with the second subscript varying faster than the first.

e.g. $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

in that order.

The MAT LINPUT statement may also be used to input complete string arrays. Each complete line of text will be assigned to one element of the array.

e.g.
```
10 DIM A$ (4)
20 MAT LINPUT A$
.
.
999 END
RUN
? JACK AND JILL WENT UP THE HILL
? TO FETCH A PAIL OF WATER
? JACK FELL DOWN AND BROKE HIS CROWN
? AND JILL CAME TUMBLING AFTER
```

Output to a terminal of both numeric and string arrays can be done using the MAT PRINT or MAT PRINT USING statements. (See Chapter five for PRINT USING formats)

e.g.
```
10 DIM A (3,3), B (4)
.
90 PRINT "MATRIX A AND VECTOR B"
100 MAT PRINT A, B,
.
9999 END
RUN

MATRIX A AND VECTOR B
A(1,1)      A(1,2)      A(1,3)
A(2,1)      A(2,2)      A(2,3)
A(3,1)      A(3,2)      A(3,3)

B(1)        B(2)        B(3)        B(4)
```

When printing, if the last item in the print list is a vector and it is followed by a separator it will be printed as a column vector.

e.g.
```
10 DIM A(4)
20 MAT INPUT A
30 PRINT "THIS IS A ROW VECTOR"
40 MAT PRINT A,
50 PRINT "THIS IS A COLUMN VECTOR"
60 MAT PRINT A
70 END
RUN
? 1, 2, 3, 4,

THIS IS A ROW VECTOR

1       2       3       4

THIS IS A COLUMN VECTOR

1
2
3
4
LINE 70 DONE
```

If ";" is used instead of "," the elements of the row vector will be printed close together.

e.g.
```
1 2 3 4
```

Input from and output to a terminal format file

Matrices, vectors and string arrays can be input from a terminal format file using the MAT INPUT # Channel Number statement, or for strings only the MAT LINPUT # Channel Number statement.

e.g. If MATS is a terminal format file containing nine numbers

```
10 FILE # 1: "MATS"
20 DIM A (3,3)
30 MAT INPUT # 1: A
```

Note that the file must be in the correct format (see Chapter six on files).

Output to a terminal format file of both numeric and string arrays can be done using the MAT PRINT or MAT PRINT USING statements with a channel number.

e.g. If MATS is a terminal format file.

```
10 DIM A (3,3)
20 FILE # 1: "MATS"
.
.
.
140 MAT PRINT # 1: A
```

or If P$ is a suitable print format (see Chapter five)

```
140 MAT PRINT # 1: USING P$ : A
```

This will write the matrix to the file in the same format that would be used to print on a terminal, with a new record for each new line.

Input from DATA statements within the program

This can be done for both numeric and string arrays using the MAT READ and DATA statements

e.g.
```
10 DIM A (3,3)
20 MAT READ A
.
.
200 DATA 1, 2, 3,
210 DATA 4,5
220 DATA 6, 7, 8, 9,
```

Input from the output to an internal format file

Matrices, vectors and string arrays can be input from an internal format file using the MAT READ # Channel Number statement.

e.g. If IMATS is an internal format file containing 9 numeric values

```
10 FILE # 1: "IMATS"
20 DIM A (3,3)
30 MAT READ # 1: A
```

Output to an internal format file of both numeric and string arrays can be done using the MAT WRITE statement.

e.g.
```
10 DIM A (3,3)
20 FILE # 1: "IMATS"
.
100 MAT WRITE # 1: A
```

The internal format file must be set up at OPEN time in the correct format (See Chapter six on files).

The most convenient format for matrices is OPEN filename (N,), No. of records as this is more flexible than the more obvious format of, say, OPEN filename (10 N), 10 for a 10 × 10 matrix.

Similarly for string arrays OPEN filename (Sn,), No. of records could be used.

Matrix functions

ZER

This function is used to set all the elements of the specified matrix or vector to zero.

e.g. 40 MAT A = ZER
will set all of the elements of A to zero.

CON

This function is used to set all the elements of the specified matrix or vector to one.

e.g. 40 MAT A = CON
will set all the elements of A to 1
60 MAT B = CON (4,7)
will redimension B to a (4,7) array and set all the elements to 1

IDN

This function is used to set the specified matrix to the identity matrix. The identity matrix is always square (i.e. the same number of rows and columns) with 1's on the leading diagonal and 0's elsewhere.

e.g. $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ is the 3 × 3 identity matrix.

e.g. 40 MAT A = IDN
will set the square matrix A to the identity matrix.

60 MAT B = IDN (3,3)
will redimension B to a (3,3) array and set it to the identity matrix.

TRN(Z)

This function is used to transpose the specified matrix. That is to replace each column by the corresponding row.

e.g.

MATRIX A = $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

Transpose of A = $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

e.g. 40 MAT B = TRN(A)
will assign to B the transpose of A.

INV(R)

This function is used to invert the specified matrix.

e.g. 40 MAT B = INV(A)
will assign to B the inverse of A.

This can only be performed on square matrices. Certain matrices do not have an inverse, these are called singular matrices.

DET

This function will find the determinant of the last square matrix which was inverted (see INV above).

```
e.g.    40 MAT B = INV(A)
        50 LET X = DET
        will find the determinant of B.
```

COL(A)

This function will give the current second bound for a matrix A.

```
e.g.    40 DIM A(4,7)
        50 LET B = COL(A)
        B will be set equal to 7.
```

Note: This will give 0 (zero) for vectors.

ROW(A)

This function will give the current first bound for a matrix A.

```
e.g.    40 DIM A(4,7)
        50 LET B = ROW(A)
        B will be set equal to 4.
```

These two functions could be especially useful after redimensioning has taken place.

Matrix manipulation and arithmetic

Assignment

To assign the value of one matrix to another, with new bounds if necessary, the MAT assignment statement is used. This is of the form

Line number MAT *destination = array expression*

```
e.g.    40 MAT A = B
        will redimension A to the bounds of B (if necessary) and assign to A the values
        of B.
```

Arithmetic operations

Addition and Subtraction can be performed on complete matrices (or vectors) provided that they are the same size.

```
e.g.    10 DIM A(4,4), B(3,3), C(3,3)
        20 MAT INPUT B, C
        30 MAT A = B + C
        40 MAT PRINT A
        50 MAT A = B - C
        60 MAT PRINT A
        70 END
```

Note: There are no matrix functions that can access a single row or column of a matrix.

This must be done using FOR loops.

e.g. To access the first colum of a 3 × 3 matrix.

```
10 DIM A(3,3), B(3)
20 MAT INPUT A
30 FOR X = 1 TO 3
40 LET B(X) = A(1,X)
50 NEXT X
```

Multiplication can be performed on

a) 2 matrices with the number of **columns** of the first equal to the number of **rows** of the second.

e.g. $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$

b) a vector and a matrix with the number of **columns** of the first being equal to the number of **rows** of the second.

e.g. $\begin{bmatrix} d_1 & d_2 & d_3 \end{bmatrix} \times \begin{bmatrix} e_{11} & e_{12} & e_{13} & e_{14} \\ e_{21} & e_{22} & e_{23} & e_{24} \\ e_{31} & e_{32} & e_{33} & e_{34} \end{bmatrix} = \begin{bmatrix} f_1 & f_2 & f_3 & f_4 \end{bmatrix}$

or $\begin{bmatrix} e_{11} & e_{12} & e_{13} & e_{14} \\ e_{21} & e_{22} & e_{23} & e_{24} \\ e_{31} & e_{32} & e_{33} & e_{34} \end{bmatrix} \times \begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$

c) A scalar (i.e. a number) and a vector or matrix.

e.g. $j \times \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & l_{12} \\ l_{21} & l_{22} \end{bmatrix}$

or $j \times \begin{bmatrix} m_1 \\ m_2 \end{bmatrix} = \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$

Examples in BASIC.

```
10 DIM A(2,3), B(3,2), C(2,2), D(3), E(3,4), F(3)
20 DIM G(3), H(3), K(2,2), L(2,2), M(2), N(2)
30 MAT INPUT A, B, C, D, E, G, K, M
40 INPUT J
50 MAT C = A × B
60 MAT F = D × E
70 MAT H = E × G
80 MAT L = (J) × K
90 MAT N = (J) × M
```

Note that scalars should always appear in brackets to distinguish them from matrices or vectors.

Division It is also possible to divide a matrix or vector by a scalar.

e.g. `40 MAT B = (1/J) * A`
will divide every element of A by J and assign the result to B.

Chapter eight

Advanced system facilities and budgeting

Allocation and control of resources

The Computer Centre staff have control of the usage of the system and they create the files and allocate the resources for the user. Before a user can log on, a file must be created for him on magnetic disc to hold programs and data. This file has a username and password and can only be accessed by those who know them.

The controls which are set on resource usage are as follows:

User name	identifies the user's files.
Password	provides security in that only those who know it can log on.
File access key	may be set to allow all users to list the catalogue or programs in a username or to restrict access to those logged on under the username.
File size	sets the maximum disc storage space that can be used with a user-name.
CPU time	maximum number of minutes of execution for BASIC programs. This is a cumulative total since the creation of the file.
No. logons	maximum number of logons allowed under a username.
Logon time	maximum number of minutes that users under one username may spend logged on to the system. This is a cumulative total since the creation of the file.

The Computer Centre staff can increase the time allocations if a user's budget has been exceeded.

They will also remind a user of his password or allocate him a new one if he should forget it.

Running programs

The current program may be executed as follows: —

RUN	This will start the program at the 1st line.
RUN *st.no*	This will start the program at the specified statement number.
RUN, *size*	This option will start the program with additional store made available to it. The size parameter can have values of 1, 2, 3. . .etc denoting the number of additional areas of 512 words that are allocated to the program for data storage.
RUN CLEAR	This will initialize all program variables before starting it. All numeric variables will be given a value of zero and all string variables will be set to the null string. If CLEAR is not used all program variables should be given values in the program or the program may halt with an "unassigned variable" error message.

These options may be combined but if present must be separated by commas and should be in the order: —

st.no,*size*, CLEAR

Examples

RUN	Run the program
RUN 40	Run the program from line 40
RUN 200,2	Run the program from line 200 with 1024 words (2 × 512) additional data storage.
RUN ,2	Run the program with 1024 words additional data storage.
RUN 10, 1, CLEAR	Initialize the program variables and then run the program from line 10 with 512 words of additional data storage.

Renumbering the program

The line numbers of the current program may be altered as follows:

RENUMBER / RESEQUENCE } *st.no, step*	Renumber the statements in the current program to ascend from the given statement number by the given step.
RENUMBER / RESEQUENCE } *st.no*	As above with the default step value of 10.
RENUMBER / RESEQUENCE }	As above with the default initial value of 10.

e.g. RENUMBER 100,20
will give statement numbers:
100
120
140
160 etc.

Storing compiled programs on disc

Compiled programs may be stored on disc so that each time the programs are loaded and run a recompilation is not necessary. This facility may be useful for storing frequently used programs as the "run" time is reduced.

A named program is compiled and stored as follows: —

CSAVE	This will compile the current program and then store the compiled program on disc with the name of the current program. Users logged on under other usernames cannot access the program.
CSAVE *st.no*	This will store the compiled program on disc in a form where when run it will start at the specified statement number.
CSAVE , *size*	This will store the compiled program on disc with additional data storage made available to it. The *size* parameter can have values 1, 2, 3. . .etc denoting the number of additional areas of 512 words that are allocated to the program for data storage.
CSAVE *access*	where *access* = USER or RUN or SHARE. This will store the compiled program on disc with the specified access restriction i.e. USER — users logged on under other usernames cannot access the program. RUN — other users may GET and RUN the program. SHARE — other users may GET, RUN and copy the Program.
CSAVE CLEAR	This will store the compiled program on disc with all program variables initialized. All numeric variables are set to zero and all string variables are set to the null string. If CLEAR is not used all program variables should be give values in the program or the program may halt with an "unassigned variable" error message.

These options may be combined but if present should be separated by commas and should be in the following order.

st.no., *size*, *access*, CLEAR

Examples

CSAVE	Compile and store the current program.
CSAVE 20, CLEAR	Compile and store the current program. Set up to start from line 20 and initialize the program variables.
CSAVE , 1	Compile and store the current program. Make available 512 words additional data storage.
CSAVE RUN, CLEAR	Compile and store the current program. Initialize the program variables and allow other users to GET and RUN the program.

Note:

CSAVE'd programs require considerably more disc storage than SAVE'd programs.

Dumping programs onto disc

When a program run has been broken into or has halted at a STOP statement or at the termination of the program it may be dumped to disc. It may then be reloaded at a later time and continued. The program is dumped as follows: —

DUMP	Store the current object program on disc. Other users may not access the program.
DUMP RUN	Store the current object program on disc. Other users may GET and continue the program.
DUMP SHARE	Store the current object program on disc. Other users may GET, CONTINUE and copy the program.

When reloaded the program can be continued from any line number but it should be noted that program variables will still have the values they had when the program was dumped.

Loading object programs

CSAVE'd programs and DUMP'ed programs are loaded from disc by GET or OLD commands in the same way as SAVE'd programs.

e.g. GET AA22
will load the program AA22 from the users file.

Any current program is deleted when the program is loaded.

Continuing programs

A program can be started or restarted with the CONTINUE command in the following circumstances: —

(1) The program run has been broken into.

(2) The program run has halted at a STOP statement.

(3) The program run has terminated normally.

(4) The program run has halted with an error message.

(5) A previously DUMP'ed program has been loaded.

(6) A previously CSAVE'd program has been loaded.

The CONTINUE command is used as follows: —

CON Continue from the next line.

CON *st.no* Continue from the given statement number.

Any size parameter specified in an associated CSAVE or RUN command will be retained when the program is continued but any channel associations set up in a previous run will not. Any MAT statement broken into will not be completed upon continuing.

Examples

```
GET ABCD      load the program and RUN it.
RUN
ABCD

     LINE 420 BREAK IN
CON 420       Continue from line number where break in occurred.
ABCD

     LINE 800 DONE
NAM ABCDD  Give program new name.
DUM RUN    Dump program to disc.
SCR
GET ABCDD  Reload
CON 20     Continue from line 20
ABCDD

     LINE 800 DONE
```

The CHAIN statement

This statement allows a program to be run immediately after another. The program name in theCHAIN statement takes the place of the current program and is executed either from its first statement, or from the statement number specified.

e.g. 50 CHAIN "MYPROG" Terminate execution of the current program and execute MYPROG from the first statement.

70 CHAIN "PROG3",40 Terminate execution of the current program and execute PROG3 from statement number 40.

CHAIN also allows channel associations to be retained. This is explained in the chapter on file-handling.

Immediate execute

Many program statements may be executed by entering the required statement without a line number.

a) If a program is present in object form it is possible to access the program variables. This can be very helpful in debugging programs.

e.g.
```
10 LET A = 77
20 END
RUN

LINE 20 DONE
PRINT A

77

DONE
```

b) The facility may be used in calculator mode.

e.g.
```
PRINT 5.2*3.16*2.1

34.5072

DONE
```

Note: The following statements may not be used in immediate execute mode: —

DATA
DEF
DIM
END
FNEND
FOR
NEXT
OPTION BASE
REM

General commands

ACCOUNT

This supplies details of the user's remaining budget. For example:

```
NO LOGINS = 00573 MAX = 99999
TOT TIME USED = 12844 MIN MAX = 99999 MIN
TOT MILL TIME = 00035 SEC MAX = 99999 SEC
DISC USED = 0020 BUCKETS MAX = 0024 BUCKETS
```

DISC

This supplies details of his usage of disc store. For example:
DISC USED = 0020 BUCKETS MAX = 0024 BUCKETS

DATE

This gives the present date and time in the following format:

28/03/77 TIME 14/59/46

TIME

This supplies details of the user's usage of terminal time. For example:

TERM TIME = 0014 MINS TOTAL = 12844 MINS MAX = 99999 MINS

LEN

This gives the length of the current program. For example:

0015 LINES 0002 BUCKETS

Chapter nine

Preparation of program source

Using paper tape on terminals

Note: Programs on Paper Tape may only be read into the system from terminals with the "Autoread" feature.

(a) To prepare a program off line

On some terminals it is possible to punch a program onto paper tape when the terminal is offline and to input this program at a later time when the terminal is online. This technique means that you do not have to spend so much time typing in your programs during your allocation of computer time and therefore allows more economical use of resources.

Your computer centre will advise you if this method of input is possible on your terminals and if so will provide instructions.

(b) To list a paper tape offline

You may obtain a print of a paper tape while the terminal is offline as follows: —

1 Switch terminal to "local".

2 Put tape in paper tape reader and press start.

3 When tape has finished listing press stop on paper tape reader.

(c) To read a program on line

You may read a program on paper tape directly into the system as follows: Remember that any BASIC lines already in store will not be overwritten by the program and so clear your store area with SCR if necessary.

1 Ensure that you are logged on and operating in "FULL DUPLEX".

2 If you want any error messages to be suppressed while the tape is being read, enter the command TAPE.

3 Load paper tape and start the reader.

4 When tape has finished, you may if you used the TAPE facility as in (2) above, list the error messages. You do this by entering the command KEY.

5 Continue with other work.

(d) To Punch a program onto Paper Tape when online

You may wish to punch onto PT a program that you have already stored on the system. This may be done as follows: —

1 Ensure that the program to be punched is in store and that there is blank tape in the punch.

2 Enter the command PUNCH on the terminal.

3 Press START on the PT punch.

4 When the program has all been punched, switch off the punch and carry on.

Using the BASIC system in batch mode

Up to now this guide has talked about using the BASIC system in interactive mode from a terminal but it is also possible to use it in batch mode using punched cards or paper tape as input and the system line printer for output.

The commands and statements take exactly the same form as the terminal. The session is started with a 'HELLO' or 'LOGIN' and ended with a 'BYE' and four asterisks (****) on the next line or card.

It must be remembered that all interactive parts of a program must be catered for in advance.

e.g. A program which requires an input would have to be supplied with appropriate data;

```
HELLO USER,PASS
10 REM BATCH PROG
20 INPUT A
30 PRINT A*A
40 END
RUN
2.5
BYE
****
```

Programs and files which have been created at a terminal may be used in the batch system and vice versa.

Thus the batch system is very useful for performing such tasks as listing a long program, which would monopolize a terminal for some time, or running a program with a lot of output.

Input on cards

Each card would correspond to a single line of input at a terminal. The card that appears after the 'BYE' must contain 4 asterisks in the first 4 columns.

Input on paper-tape

Similarly, for paper tape, each line terminated by a 'newline' character would correspond to a line of input at a terminal. The line after the 'BYE' must contain 4 asterisks in the first 4 characters.

Index