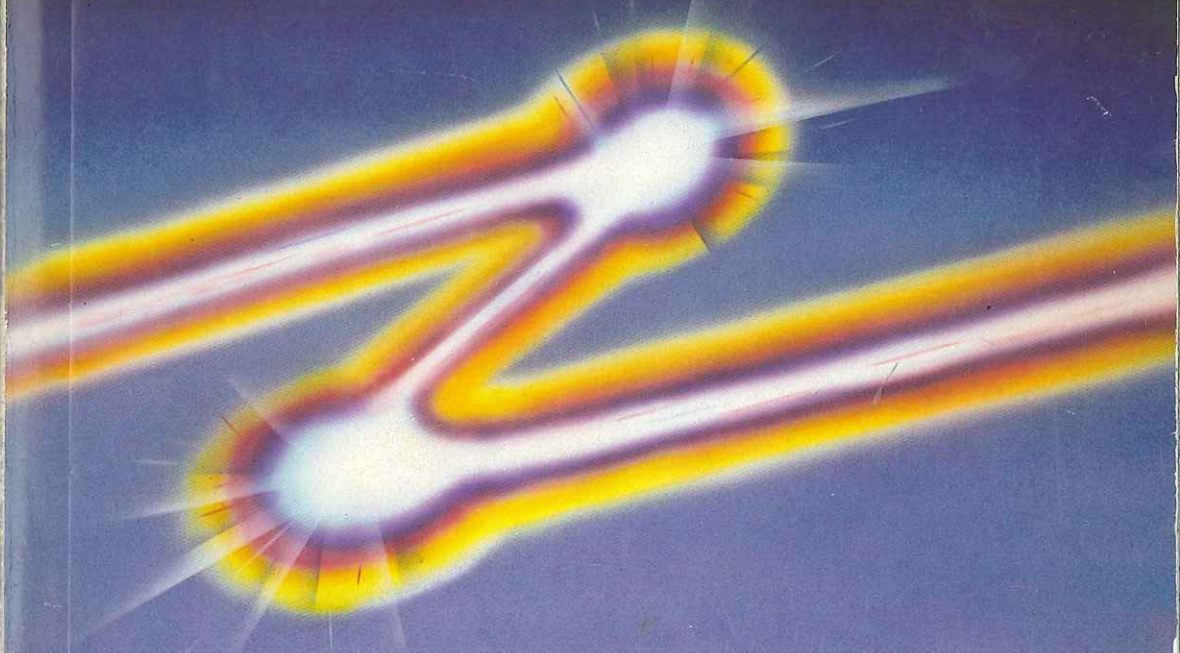


THE CENTURY COMPUTER PROGRAMMING COURSE for the

COMMODORE 64

Edited by
PROFESSOR PETER MORSE and BRIAN HANCOCK
of the Central London Polytechnic





THE CENTURY COMPUTER PROGRAMMING COURSE

for the

COMMODORE 64

A complete guide to programming in BASIC

by

Professor PETER MORSE and BRIAN HANCOCK

of the Central London Polytechnic



CENTURY COMMUNICATIONS LTD
LONDON

Copyright © Peter Morse and Brian Hancock 1985

All rights reserved including the right of reproduction in whole or in part in any form

First published in Great Britain in 1985
by Century Communications Ltd.,
Portland House, 12-13 Greek Street, London W1V 5LE

ISBN 0 7126 0383 2 (*paper*)

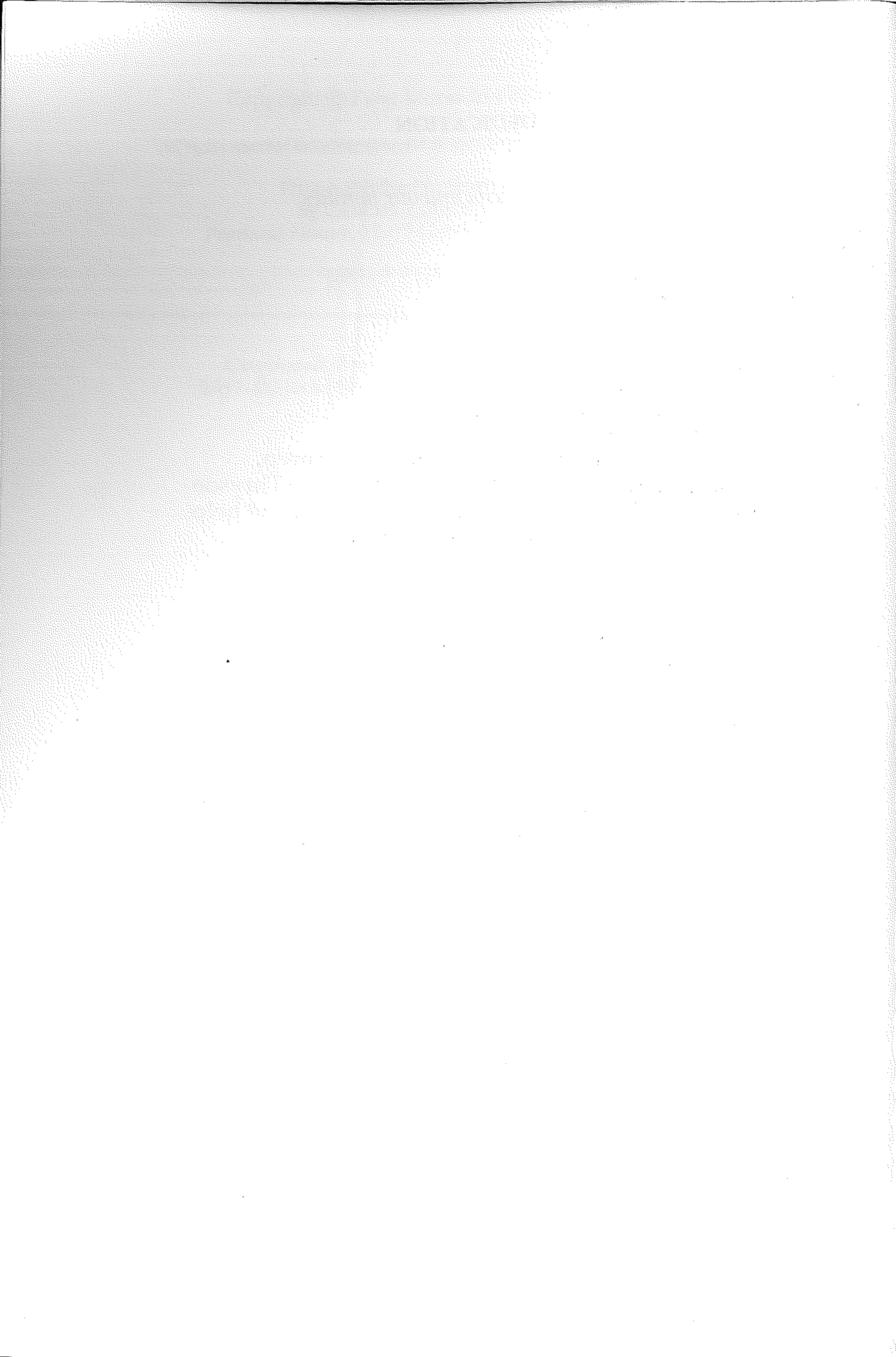
Originated directly from the authors' w-p disks by
NWL Editorial Services, Langport, Somerset, TA10 9DG

Printed in Great Britain by
Butler & Tanner Ltd, Frome, Somerset

DEDICATION

To Vickie, Stella and our families

The authors wish to thank Ben Anrep,
Mark Siegler, Ian Adamson, Anne-Marie Thrysoe
and Jamshid Siabi for their contributions and help
in producing this text.



CONTENTS

Introduction

PART ONE FIRST STEPS

SECTION A: MEET THE COMMODORE 64

A1: The Commodore 64 microcomputer system	3
A2: Function of components	3
A3: Connecting up	4
A4: The keyboard	4

SECTION B: A TEST DRIVE

B1: Printing on the screen	11
B2: Colour	13
B3: Sound	15
B4: Sprite graphics	16
B5: Arithmetic	17

SECTION C: Basic BASIC

C1: The BASIC language	19
C2: A simple program	19
C3: A statement	20
C4: Statement numbers	20
C5: Instructions	20
C6: Numeric variables	21
C7: Strings and string variables	22
C8: Operators and operands	22
C9: Keying in a statement	23
C10: Correcting errors	23
C11: Direct instructions	23
C12: Editing a program on the screen	23
C13: Listing a program on the screen	24
C14: Running the program	24
C15: Error messages	24
C16: How the program works	25
C17: Naming the program	25

SECTION D: SAVING, VERIFYING, LOADING and LISTING

D1: Saving the program on cassette tape	27
D2: Deleting the program from memory	28
D3: Loading the program from cassette tape	29
D4: Program libraries and catalogues	30

SECTION E: IMPROVING THE PROGRAM

E1: Adding comments – the REM statement	31
E2: Using the PRINT statement	32
E3: Adding a loop	32
E4: Stopping the program	33
E5: Testing for a condition	33
E6: Final edit and saving	34

SECTION F: DATA INPUT

F1: Data	37
F2: READ DATA and RESTORE	37
F3: FOR TO STEP and NEXT loops	38
F4: Multiple INPUT	39
F5: Printing on the input line	39
F6: String INPUT	40
F7: READING large data lists	40
F8: Data types and data structures	40

PART TWO ESSENTIALS OF BASIC PROGRAMMING

SECTION G: PROGRAMMING METHODS

G1: Programming	45
G2: Problem analysis	46
G3: Structure diagrams	47
G4: Classifying program modules	49
G5: Control structures	51
G6: The data table	53
G7: Describing the algorithm	54
G8: The pseudocode description	54
G9: Flowcharts	56
G10: Testing the algorithm	64

SECTION H: CONTROL

H1: Control in programs	65
H2: Condition testing	65
H3: IF . . . THEN	66

H4: GOTO instructions	66
H5: ON GOTO	67
H6: Decision structures	68
H7: Logical operators – AND/OR	73

SECTION I: ARITHMETIC AND FUNCTIONS

I1: Arithmetic operations	75
I2: Priority	75
I3: Number	76
I4: The E notation	77
I5: Rounding	78
I6: How numbers are handled	78
I7: Functions	79
I8: List of functions used in Commodore BASIC	80
I9: The standard mathematical functions	82
I10: Trigonometric functions	83
I11: Random numbers	84
I12: User-defined functions	85

SECTION J: LOGICAL OPERATIONS

J1: Logic values and numeric values	87
J2: Boolean operators: the AND operator	87
J3: The OR operator	88
J4: The NOT operator	88
J5: Conditional operators	89
J6: Logic operations on conditional expressions	89
J7: Multiple logic on conditions	91
J8: Logical operations on numbers	92
J9: Priority	94
J10: Logical operations with strings	94
J11: Applications of logical operators	95

SECTION K: STRINGS

K1: Strings	99
K2: Quotes	100
K3: String input	100
K4: Length of a string	101
K5: Null strings	102
K6: String variables and dimensions	102
K7: Multi-dimensional string variables	103
K8: String and string array assignment	104
K9: Substrings and string slices	105
K10: String concatenation	105
K11: Comparing strings	106
K12: VAL and STR\$	108
K13: ASC and CHR\$	109

SECTION L: LOOPS

L1: Loops	111
L2: Counters	112
L3: Counting events	113
L4: Repeat-until loops	113
L5: While-do loops	115
L6: FOR...NEXT loops	116
L7: FOR...NEXT flowcharts	117
L8: FOR...NEXT examples	118
L9: Loops of variable length	121
L10: Nested loops	122

SECTION M: PRINTING AND PLOTTING

M1: The print screen	127
M2: The plot screen	127
M3: PRINT definitions	128
M4: Formatting numbers	129
M5: Word processing	133

SECTION N: SUBROUTINES

N1: Subroutines	135
N2: Subroutine example	136
N3: Nested subroutines	137
N4: Recursive subroutines	139
N5: ON GOSUB	142
N6: Subroutine use	143

PART THREE THE COMPLETE PROGRAMMING METHOD

SECTION O: PROGRAMMING METHODS II

O1: A Recapitulation	149
O2: The rules of coding and design	150
O3: Control structures in 64 BASIC	156
O4: Program development	163
O5: The complete programming method	168
O6: An example of structured design	172
O7: The program	179
O8: The documentation	179

SECTION P: FILES

P1: Introduction	181
P2: Using other devices	181
P3: Files with cassette recorder	182
P4: Structure of files	185
P5: Using a disc drive	186

P6: Loading and saving programs	187
P7: Errors	187
P8: Working with files	188
P9: Random and relative files	188
P10: The disc unit	189
P11: Output to a printer	190

SECTION Q: COLOUR AND SOUND

Q1: Standard character mode	191
Q2: Screen and colour memory	192
Q3: Sound and music	195
Q4: Playing a note	196
Q5: Multiple voices	199
Q5: Filtering	200

SECTION R: GRAPHICS

R1: User defined (programmable) characters	201
R2: Defining characters	202
R3: Multicolour characters	204
R4: Extended background colour	205
R5: High resolution (bit map) mode	206
R6: Sprites	213
R7: Defining a sprite	214
R8: Sprite formation	214
R9: Sprite expansion	218
R10: Sprite priority	219
R11: Multicolour sprites	220
R12: Collision detection	221

SECTION S: LISTS and ARRAYS

S1: Introduction	223
S2: Dimensioning	223
S3: The index variable	224
S4: Lists	224
S5: String arrays	228
S6: Two-dimensional arrays	229
S7: Multidimensional arrays	230
S8: Use of arrays	232

SECTION T: SORTING, SEARCHING and STORING ARRAYS

T1: Searching and sorting	237
T2: Bubble sort with flag	239
T3: Alphabetical sort	240
T4: Insertion sort	241
T5: Shell sort	243
T6: Quick sort	247
T7: Index sort	250

T8: Linear search	253
T9: Binary Search	254

SECTION U: MEMORY and MACHINE CODE

U1: Binary systems	257
U2: The memory map	258
U3: PEEK and POKE	261
U4: The program area	261
U5: Systems variables	262
U6: The hexadecimal system	262
U7: Machine code programs	264
U8: The 6510 microprocessor chip	265
U9: The kernal	267
U10: Input/output memory locations	267
U11: Complete memory maps	267

PART FOUR THE APPLICATION OF PROGRAMMING

SECTION V: APPLICATIONS PROGRAMS

V1: Programming for applications	271
V2: Instructions and input checks	271
V3: Example programs	277
V4: Games programming	309
V5: Example program	310

APPENDICES

I	ASCII and CHR\$ codes	313
II	Screen display characters and codes	315
III	Abbreviations for BASIC keywords	317
IV	Error messages	319
V	Music note values	321
VI	Program library	323

INTRODUCTION

The central conviction behind this book is that programming computers to solve problems is essentially a language-independent activity. This means that there is no reason why Commodore BASIC should not be learned in exactly the same way as other high level languages: that is, with the fundamentals of problem-solving and structured programming introduced at an early stage.

For the majority of readers, Commodore BASIC will be their first introduction to computing; we hope that many will use it as a stepping-stone to more advanced study and application. Good problem-solving and programming habits will make easier both applications programming in BASIC and, eventually, learning a different structured language, such as PASCAL (which has a richer programming environment than BASIC). Once bad programming habits are acquired they are difficult to break; thus, this book has been designed to introduce readers to the elements of computer programming in a systematic manner, with the emphasis on correct rather than merely adequate techniques.

Although the text is intended to be a serious treatment of Commodore BASIC, as an introduction to computing it assumes no prior knowledge of computers and only a minimal understanding of mathematics. (Without the maths you will still be able to make your way through the book, but if you don't know what SIN and COS are, you won't be able to write programs using them!) Above all we intend to give readers a full introduction to the essential control and modular structures present in structured computer languages, and the way in which they operate in Commodore BASIC. We hope that with this behind them, readers will be able to tackle other computer languages with an understanding of the essentials of good programming in any language.

This approach also ensures that the reader who stays with his Commodore machine will be able to maximise its potential. As it runs on one of the world's most popular microcomputers, there can be little doubt that Commodore BASIC will become one of the most commonly used computer languages. This, coupled with the fact that more and more software is becoming available for it, makes it all the more important that users attain a sound understanding of the language. Most published programs in books and magazines have little in the way of documentation; debugging them, normally a tedious and difficult task, becomes easier when the techniques for doing so have been well learned.

This book introduces readers to three main sets of computer rules:

- the rules of using the computer system
- the rules of the Commodore BASIC programming language
- the rules of problem solving, data handling and structured programming using Commodore BASIC

Why this book was written

The wide availability of the Commodore machine demands that it be treated seriously as a means of teaching programming methods to a large number of people, and the programmer of any computer must understand the characteristics of the machine, the high level language by which it is used and controlled (in this case BASIC), and the problem-solving techniques to which it can be applied. We felt that a serious text was needed on Commodore BASIC which gave the first-time user a worthwhile home tutor on computing. So we wrote one!

Who the book was written for

The book has been written for the home user or school user who has access to a Commodore 64 and wants to learn how to program it from scratch. Experience has shown that most Commodore users will buy more than one book on the subject of programming their machine; this book will clear up a few misunderstandings and confusions presented by other texts and will take the reader further into programming techniques.

The text has also been designed as an aid to Commodore BASIC users who are attempting serious applications and are having trouble designing error-free programs.

The structure of the book

The book is a complete self-study text, and should be worked through using the machine, keying in exercises and programs as they arise.

The course has twenty-two sections and is divided into four parts. The table of contents lists all the numbered headings within each section, thus describing the study objectives and BASIC features covered in each part of the book. The course is complete and comprehensive.

Part 1 The book begins with a brief introduction to the 64 and its keyboard, and takes the reader on a 'test drive' of the machine. Then the programming begins immediately: the user is taken through the operation of a simple program, as well as how to save it, load it, list it and add improvements. Programs need data, so the section finishes with an introduction to data handling.

Part 2 The essentials of BASIC programming are introduced, beginning with the fundamentals of problem solving and structured programming in BASIC. The properties and implementation of important language control structures, such as decisions, loops, and subroutines, are introduced, together with the use of arithmetic, functions, logic and strings. A detailed section on printing and plotting on the screen gives an insight into the popular topics of graphics and word processing.

Part 3 Now the book changes up a gear: the complete programming method of structured design, coding, debugging and documentation is first taught. The treatment of graphics continues with a study of the character set, codes, and designing of special characters. This leads naturally into the use of colour and sound, and accelerates into the fields of computer art, animation and some sophisticated graphic toolkits. Theory is thoroughly and carefully covered.

The treatment of data structures and data handling is advanced by investigating lists and arrays, and how to sort and search them. A fundamental unit on the operation of the 64 is then taken, with a detailed consideration of memory, its organisation and handling through peeking and poking, and the discovery of how easy it is to run machine code programs. Some new tricks to protect your programs are given.

The use of a floppy disk drive storage system is considered in some depth, and the important topic of data file handling, so vital to real applications, is covered.

Part 4 Finally, the book takes the student through more examples of applications programming. The methodology taught in the earlier parts of the course is applied to some interesting problems.

Remember, programming is learned most effectively through experience: the exercises which appear in the text are intended to give you practice in programming and to stretch you a little. If you do not understand something or if you get stuck, don't be discouraged; go back over the section again. Take your time, and don't be afraid to experiment with your own variations. See if you can apply your own ideas on the 64!

We hope that you find learning BASIC programming with this book a successful, enjoyable and useful experience, and that the knowledge and programming skills obtained will be a step on the path to a more advanced use of your Commodore or other computers for real applications and enjoyment.

Peter Morse

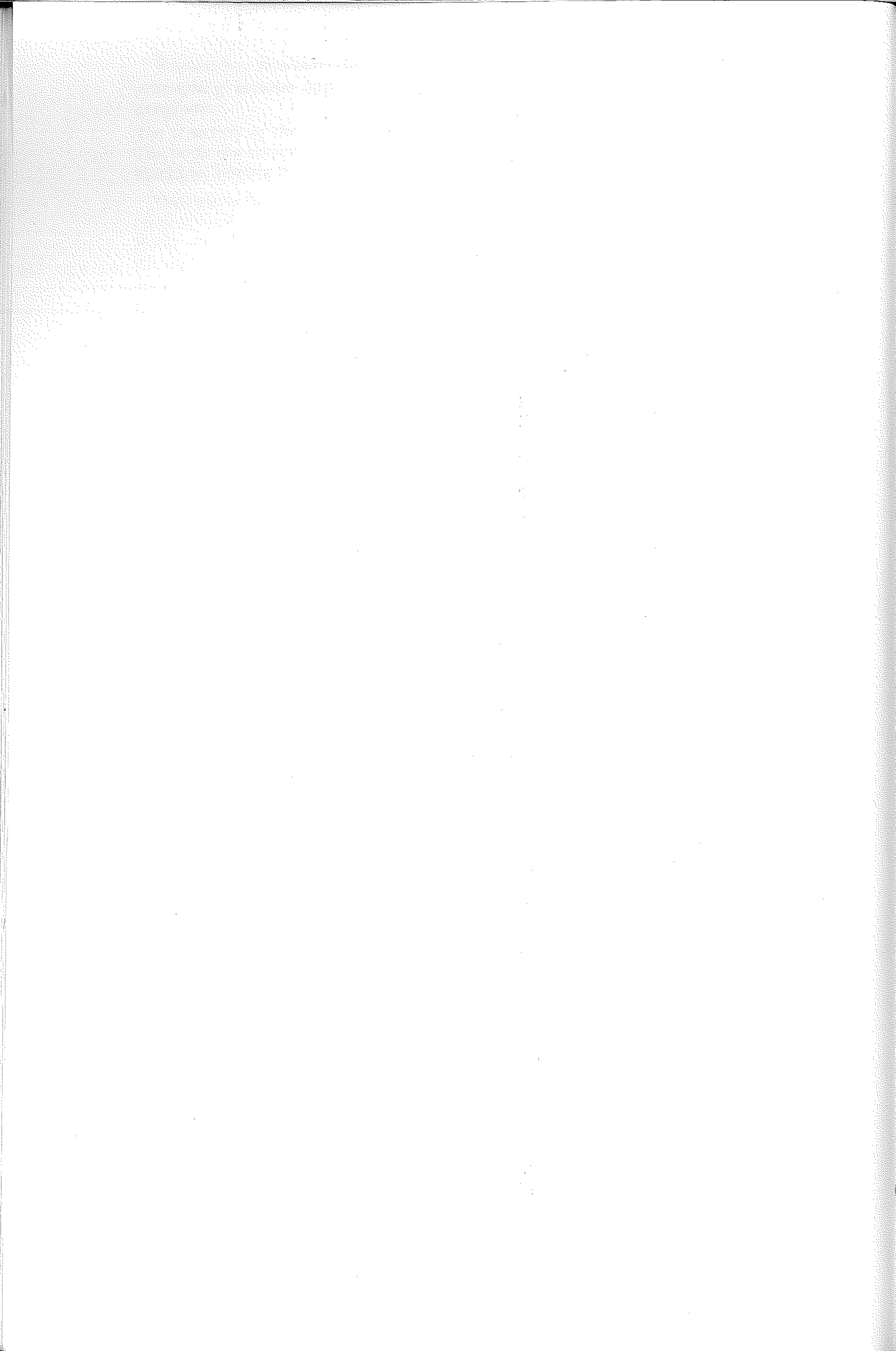
Brian Hancock

The Polytechnic of Central London

1985

PART ONE

FIRST STEPS



Section A: Meet the Commodore 64

A1: The Commodore 64 microcomputer system

The components of the minimum complete operating 64 system consist of:

- The Commodore 64 microcomputer, with keyboard and 64K RAM (Random Access Memory)
- The 64 power supply, with the correct plug attached for your AC power socket
- A domestic TV or a monitor for display
- The aerial/antenna or connecting cable which connects the 64 to the TV or monitor

The computer system can be operated without a recording device, but without a cassette player or a floppy disc unit you will not be able to make a record of your work, or use any of the commercially available software.

A2: Function of components

Here is a brief description of the functions of the components of a Commodore microcomputer system.

<i>Component</i>	<i>Function</i>
Commodore 64 computer board	Data processing and control of information handling: input from keyboard or cassette; output to TV screen (or printer). Holds 64K of RAM memory. (K stands for kilobyte. One byte is eight bits, which are the binary digits 0 and 1 that computers use as on-off switches. A kilobyte is roughly 1000 bytes, actually 2^{10} or 1024, the nearest binary number to 1000.)
Keyboard	Input of information. Programs, data and commands are keyed in.
TV set	Used as VDU (visual display unit) (colour if possible). Provides on-line output of information: visual display of programs, results (data, graphs, pictures) and control commands. A colour set is best, otherwise you will not be able to use all the 64's functions; a purpose-made monitor is best of all because it will give a better picture.
Cassette recorder	Off-line storage of information. Programs and data are stored (recorded, or <i>saved</i>) as coded electromagnetic impulses on cassette tapes. They can be played back (<i>loaded</i>) at any time for use again.
Printer	Output device, to provide a permanent record of the screen display, program listings or information in the computer memory.
Power supply	Supplies the DC current (9 volts at 1.0 amps and 5 volts at 1.5 amps) to run the computer, from the household power supply.
Cables	To interconnect the devices which make up the system.

The printed circuit board inside the Commodore 64 holds and connects the IC (integrated circuit) microchips which provide the computing facilities. These are:

- the 6510 CPU (Central Processing Unit) microprocessor chip, the brain of the system. This is an updated version of the 6502 chip that is used in many other microcomputers, and performs all processing tasks.
- the ROM (Read Only Memory) chip, which holds the 8K BASIC interpreter that translates BASIC instructions into the machine-code instructions that the 6510 operates with. The software or program data in this are fixed, hence the name, and also stable: it remains when the power is switched off.
- the RAM (Random Access Memory) chips, which provide the memory store. This memory is volatile: the data is stored as electrical impulses and is lost when the power is switched off. This memory stores the BASIC program, the values of variables (including some *system variables* that the computer uses to organise its own affairs), a memory picture of the TV screen display, and the stacks which hold the numbers whilst they are being manipulated. The memory organisation is described in Section U of this book.

Also mounted on the circuit board are several other chips for handling graphics and sound, the colour TV signal encoder and modulator circuits, and sockets for the connecting cables to the TV and cassette recorder.

A3: Connecting up

Set aside an area to work in and set up your television, cassette recorder, printer (if you have one), and the 64's power supply.

Connect the 64 to the television aerial with a lead connected to the TV socket on the 64 and connect the Commodore cassette player to the back of the computer.

Your system is now set up. Check that the TV is turned off, and that no cassette keys are depressed. Plug the 64 power supply into AC (household) power supply sockets and switch the sockets on if they have switches.

Switch on the TV and the Commodore 64. Choose a channel with the push button or other channel select control and tune the TV to channel 36 UHF (UK) until

```
(C) COMMODORE 64 BASIC V2
```

appears on the screen. Adjust the tuning until the display is clear, and the brightness, contrast and colour (if you're using a colour TV) controls to get a good picture without it being too bright (since you are going to spend some time looking at it from close up).

On leaving your computer: leave it connected up; switch off the TV and the AC power supply plugs; disconnect power plugs from sockets.

You now have an operating microcomputer system. The system needs no maintenance other than the occasional cleaning of the tape heads on the cassette player.

A4: The keyboard

The keyboard is very similar to that used on an ordinary typewriter, with a few additional keys, as follows:

RETURN	Carriage return
RUN/STOP	Loads and runs a program from cassette when used with the SHIFT key. Stops execution of your program.

CTRL	Control key used with other keys to control the computer.
C=	Expands the range of the control function. Also used when loading from tape recorder.
CLR/HOME	Moves cursor to top of screen. Used with the SHIFT key to clear the screen.
INST/DEL	Deletes a character. Used with SHIFT to insert a character.
CRSR ↑ ↓	Moves the cursor up or down the screen (with SHIFT key for up).
CRSR ↔	Moves the cursor left or right on the screen (with SHIFT key for left).

EXERCISES WITH THE KEYS

RETURN

Type any character(s) on the keyboard. They will be displayed on the screen. Now press **RETURN**: the Commodore will respond with some form of message. In any exercise in this book, by the way, don't worry about doing the wrong thing; you cannot damage the machine by pressing the wrong combination of keys.

CLR/HOME

Enter a few lines, separate them using the **RETURN** key, and press the **CLR/HOME** key. The cursor (the square that flashes on the screen) will move to the top left-hand corner of the screen. Now hold the **SHIFT** key and press **CLR/HOME**. The screen will now clear (blank screen, except for the cursor).

CRSR ↑ ↓ and CRSR ↔

These keys, used with the **SHIFT** key, allow you to move the cursor around the screen. To move the cursor up or left, the **SHIFT** key is also held down; to move the cursor down or to the right, the cursor key alone is used.

INST/DEL

Enter this line:

```
PRINT "HELLO"
```

but do not press the return key. Now press **INST/DEL** key a few times. Notice that each time the key is pressed a character on the screen is removed (deleted). Delete all the characters, or clear the screen as above, and enter:

```
PRINT "GOODBYE" ■
```

Again, do not press **RETURN**. Now using the cursor keys move the cursor back over the second pair of quotes. Hold the **SHIFT** key down and press the **CRSR** ↔ key once, then press **INST/DEL** key four times. Now release the **SHIFT** key and enter a space followed by **NOW**.

CTRL and RESTORE

The main purpose of the **CTRL** key on the Commodore is to change the colour of characters produced on the screen. This can be done in two ways. Firstly, pressing **CTRL** and holding it, press any of the digits marked 1 to 8. Release both keys and now type any characters on the keyboard. The colour you selected will now be displayed. Try this example:

Press **CTRL** and 2

Enter **COMMODORE**

Press the space bar, then **CTRL** and 3

Enter **COMMODORE** again

The screen will show:

```
COMMODORE COMMODORE
```

with the first word in white, the second in red.


```

10 PRINT "KEYBOARD TESTER"
20 PRINT "SCREEN WILL SHOW A CHARACTER"
30 PRINT "PRESS THE CHARACTER AND SEE"
40 PRINT "WHETHER YOU WERE RIGHT"
50 PRINT ""
60 PRINT "PRESS A KEY WHEN READY"
70 PRINT ""
71 FOR A=1 TO 100:NEXT A
75 GET A$
80 PRINT "PRESS A KEY WHEN READY"
81 FOR A=1 TO 100:NEXT A
90 PRINT ""
100 IF A$="" THEN 60
110 PRINT ""
120 S=30+INT(RND(1)*65)
130 PRINT " "CHR$(S)
140 GET A$:IF A$="" THEN 140
150 IF A$=CHR$(S) THEN PRINT "CORRECT":GOTO 120
160 PRINT "INCORRECT TRY AGAIN"
170 GOTO 140

```

```

10 PRINT ""
20 FOR A=1 TO 8
30 READ B
40 FOR C=1 TO 2
50 PRINT CHR$(B);
60 PRINT ""
70 NEXT C,A
100 DATA 5,28,30,31,144,156,158,159

```



```

625 FOR A=1 TO 13*40
630 PRINT " ";
635 NEXT:PRINT "#####"
      ";
650 PRINT "#####"
660 PRINT "#####GRAPHICS PRODUCED USING
SHIFT KEY"
665 PRINT "#####PRESS A KEY TO CONTINUE
      ";
670 GET A$:IF A$="" THEN 670
680 GOSUB 795
690 PRINT "#####GRAPHICS P
RODUCED USING LOGO KEY "
700 PRINT "#####PRESS A KEY TO CONTINUE
      "
710 GET A$:IF A$="" THEN 710
720 PRINT "#####"
730 END
795 PRINT "#####";
800 FOR A=1 TO 16
810 READ B$
820 PRINT B$;"#";
830 NEXT A
840 READ A$:PRINT:PRINT "#####"A$"#";
850 FOR A=1 TO 14
860 READ B$
870 PRINT B$"#";
880 NEXT A
890 PRINT:PRINT "#####";
900 FOR A=1 TO 15
910 READ B$
920 PRINT B$;"#";
930 NEXT A
940 PRINT:PRINT "#####";
950 FOR A=1 TO 15
960 READ B$
970 PRINT B$;"#";
980 NEXT A
985 PRINT "#####"
990 RETURN
1000 DATA "+,1,2,3,4,5,6,7,8,9,0,+,-,.,",
C###","I###"
1010 DATA CT,Q,W,E,R,T,Y,U,I,O,P,5,*,†,R
S
1020 DATA "•","◊",A,S,D,F,G,H,J,K,L,":",
";","=",RE
1030 DATA "•","**",Z,X,C,V,B,N,M,"",".
","/",**,"+","+"
2000 DATA "•"," "," "," "," "," "," "," "," "
"," "," "," ","+","|","" " " "

```


Section B: A Test Drive

B1: Printing on the screen

To display information on the screen the keyword PRINT is used. For example, key in PRINT "1984". This will appear on the screen as you key in the characters. The cursor remains flashing. Now press RETURN and 1984 will be printed immediately below the instruction and the message.

READY

appears beneath. This is the 64 telling you that the last command has been processed satisfactorily.

The screen can be cleared by *embedding* a SHIFT and CLR/HOME within the quotes of a PRINT command.

Type in the following:

```
PRINT "[SHIFT] [CLR/HOME]"
```

A reverse heart symbol appears after the first quote marks, and then the screen clears leaving READY and the cursor at the top left when RETURN is pressed.

Now key in PRINT Commodore and press RETURN. Nothing happens and you get the report:

0

READY

Something's wrong: the word to be printed has not been enclosed in quotation marks. But if you key in PRINT "Commodore" and press RETURN, the word COMMODORE will be printed on the screen. In a PRINT instruction such as this, all letters, words and symbols to be displayed on the screen must be enclosed in quotes.

Correcting mistakes

Now that you are entering commands, symbols and words on the input line, sooner or later you are going to make typing mistakes. These are corrected first by using the **CRSR** arrow keys to bring the cursor to the right of the letter you wish to delete, then by pressing **INST/DEL** to delete the character. You then key in the correction(s) and use the **CRSR** keys if necessary to get back to wherever else you are on that screen.

EXERCISES

Key in some text and use the right and left arrow cursors and the **INST/DEL** key to edit (rewrite) the line.

The screen

The screen is divided up into rows of character-printing positions which may be described as 'cells'. There are 25 rows of 40 cells. Each character or symbol printed on the screen occupies a single cell.

The screen is mapped out as shown in Figure 1:

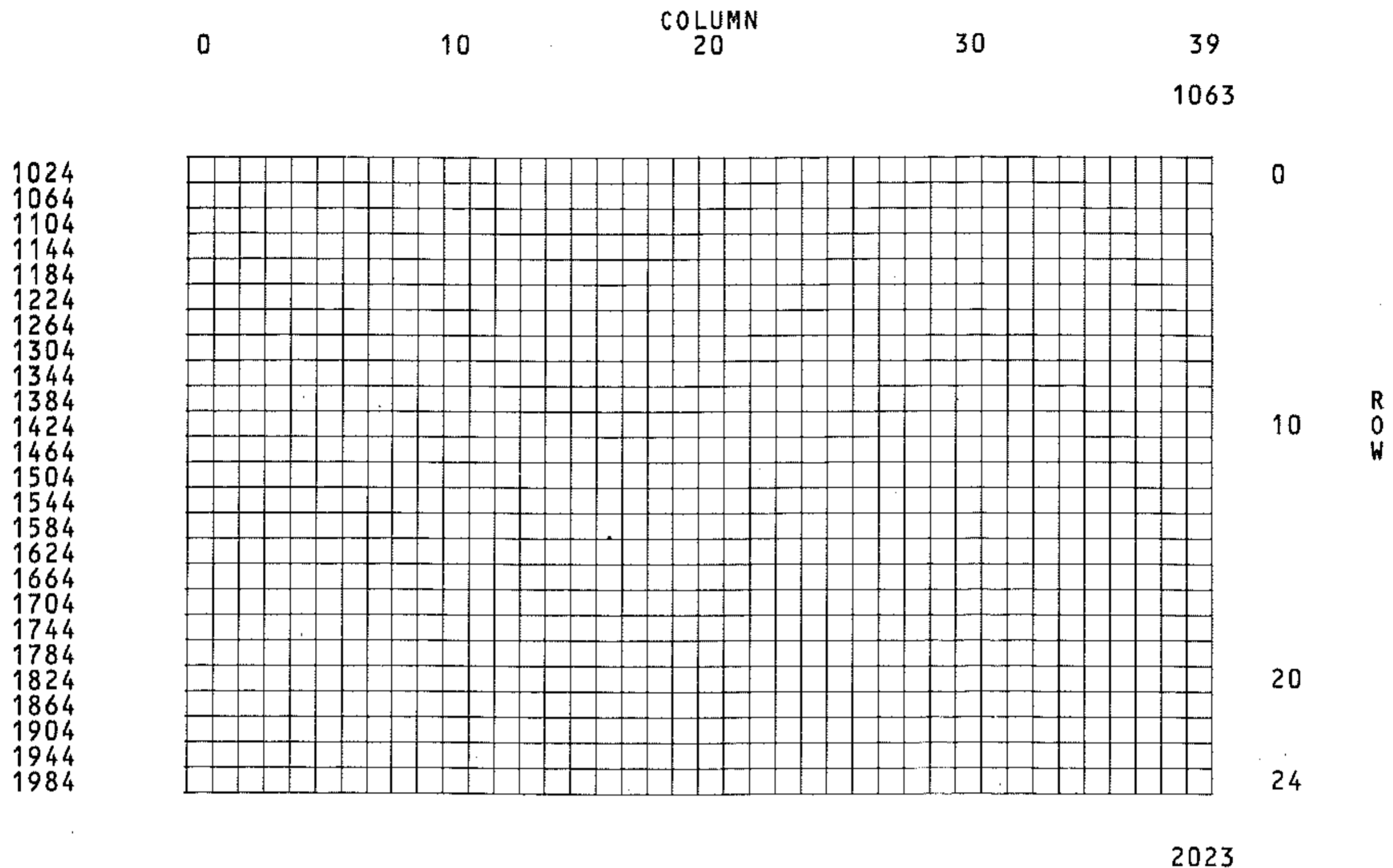


Figure B1: Map of the screen's cells.

A particular cell or character position is referred to by row and column numbers.

Printing at different places on the screen

You can instruct the 64 to print text anywhere on the screen. Enter this line:

`PRINT"`

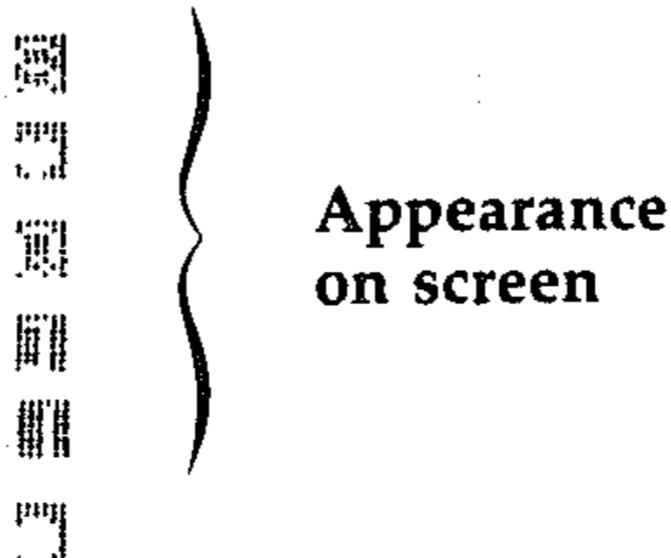
but do not press **RETURN**. Now press any sequence of cursor keys **CRSR** ↑ ↓ **CRSR** ↔; **SHIFT CLR/HOME** or **CLR/HOME** may be used at the start. You will see on the screen symbols representing the cursor movement selected. Now enter any text and then another set of quotation marks; then press **RETURN**. The text will be printed where you specified.

Try this:

`PRINT"[CLR][CD][CD][CD][CD][CR][CR][CR][CR]COMMODORE"`

This clears the screen and prints **COMMODORE** at position (5,4). The cursor control symbols that you will use in this text are represented by the following keys on your Commodore:

HM = CLR/HOME
 CLR = SHIFT CLR/HOME
 CD = CRSR ↑ ↓
 CR = CRSR ↔
 CL = SHIFT CRSR ↔
 CU = SHIFT CRSR ↑ ↓
 ▽ = SPACE



You can print anywhere on the screen using the cursor control keys. Try printing your name at different positions on the screen. Try printing `Commodore` at co-ordinates (0,31). You will see that printing continues on the next line. Alternatively, you could use the following statement, which moves the cursor to the required position on the screen.

```
POKE781,Row:POKE782,Col:SYS65520
```

The locations 781 and 782 are the memory locations that control the cursor row and column respectively, and the SYS command positions the cursor at the specified location.

For example:

```
POKE781,7:POKE782,4:SYS65520:PRINT "Commodore"
```

prints `Commodore` at line 8 column 5. There are several other ways the cursor and characters may be printed at chosen positions on the screen, these again involve specific `POKEs`.

Spaces and overprinting

A space is obtained by pressing the `SPACE` key. `PRINT " "` prints a space on the screen. For example:

```
PRINT "A 6"
```

Try this:

```
PRINT"[CLR][CD][CD][CD][CD][CD][CR][CR][CR][CR]COMMO-  
DORE"  
PRINT"[HM][CD][CD][CD][CD][CD][CR][CR][CR][CR]"
```

Separators

Different items to be printed on the screen can be included in a single `PRINT` statement. The items are separated by the symbols `;` or `,`. These are called *delimiters*. The semi-colon indicates that the next print position is at the next cell, and the comma indicates that next print position is at column 0, 10, 20 or 30 – as appropriate. (`PRINT` on its own means print a line of spaces.)

Try `PRINT "A";"B","C","D"`

TAB

This function works like the tabulator on a typewriter. We have 39 `TAB` positions along a row. For example:

```
PRINT TAB(14) ; "A"
```

```
PRINT TAB(5) ; "A";TAB(10);"B"
```









Note that the semicolon (`;`) after the `TAB()` and between items in quotes may be omitted, and that there must be no space between `TAB` and its (bracket (*ie.* `TAB(5)` and *not* `TAB (5)`).

B2: Colour

So far all the words printed on the screen have been in light blue on a blue background. As an introduction to the powerful colour graphics and display facilities on the 64, you can now print some words on the screen in different colours against coloured backgrounds and borders. Study again the top row of keys; the names of some of the colours available are written on the fronts of the keys.

Changing the colour of characters on the screen

To change the colour of the letters, hold **CTRL** and press any number between 1 and 8 and release **CTRL**. Here are the results:

CTRL 1 black		}	Appearance on screen
CTRL 2 white			
CTRL 3 red			
CTRL 4 cyan			
CTRL 5 purple			
CTRL 6 green			
CTRL 7 blue			
CTRL 8 yellow			









Now try this:

```
PRINT"RAINBOW"
```

but as you type each letter within the quotes, use **CTRL** 1 before typing R, **CTRL** 2 before typing A, **CTRL** 3 before typing I, and so forth. Don't forget the final pair of quotes. Then press **RETURN**. The word *rainbow* will be printed in all the different colours you have specified.

Note that as with cursor symbols, each control code (1 to 8) is shown as a symbol; for example, white is **E** reverse E.

There are also eight more colours that can be accessed using the **C=** key, holding it and pressing 1 through to 8 as an alternative control key. These colours are:

C= 1 orange		}	Appearance on screen
C= 2 brown			
C= 3 light red			
C= 4 grey 1			
C= 5 grey 2			
C= 6 light green			
C= 7 light blue			
C= 8 grey 3			

You can also change the screen colour by using the following command:

```
POKE 53281, X
```

where X is any number between 0 and 15. Try this:

```
POKE 53281, 7
```

This will change the background colour to yellow. You can also change the border colour by using

```
POKE 53280, X
```

where X is any number between 0 and 15. Try this:

```
POKE 53280, 7
```

It will change the border to yellow.

Now you will have yellow background and yellow border. To return to the original border background and printing colour, hold the **RUN/STOP** key and press **RESTORE**. The screen will clear and the border and background will be same as they were before you changed them. Note that the **POKE** value is *one less than* it would be if you had keyed the colour using **CTRL** or **C=**.

Multiple commands

Multiple commands (more than one statement in a line) are possible in Commodore BASIC. For example:

```
10 LET A=20:LET B=10:SUM=A+B:PRINT"SUM=";SUM
```

Note that statements are separated by colons. Also note the line number (10) in front of the commands. When you press **RETURN** the 64 has saved the commands in its memory. You can run the line by using the command **RUN** and

then **RETURN**. You can use the cursor keys to change anything in quotes, then press **RETURN** and enter **RUN** and **RETURN**. This is a very powerful tool. By putting the number 10 in front of the sequence of commands you have created a program line, and you can run it, edit it and run it again as many times as you like until you destroy it, or clear it from the computer's memory with the **NEW** command. Imagine having to key in all those commands every time you wanted to run the sequence again!

B3: Sound

The Commodore has a very sophisticated sound synthesis capability which may be used to play music and to produce sound effects. This is done by setting the parameters of the SID sound chip using **POKEs** like this:

```
POKE 54296,15: REM SET VOLUME TO MAX
POKE 54277,9: REM LOW ATTACK AND HIGH DECAY
POKE 54278,34: REM MID SUSTAIN AND MID RELEASE
POKE 54276,33: REM TRIANGLE WAVEFORM
```

Having set these (and other) parameters, you can now construct a program capable of playing any sequence of notes.

This is done by **POKEing** the low and high frequency values of a note to be played into the registers at locations 54272 and 54273 respectively. These values can be obtained from the table of notes in the Appendix. For example:

```
POKE 54272,37:POKE 54273,17
```

This plays the note middle C. Note that the tone is continuous; to stop it, zeros must be **POKEd** into the above registers.

Here is an example that plays a short tune:

```
10 POKE 54296,15
20 POKE 54277,9
30 POKE 54278,34
40 FOR I=1 TO 11
45 POKE 54276,32
50 READ DE,LF,HF
60 POKE 54272,LF:POKE 54273,HF
65 POKE 54276,33
70 FOR J=1 TO DE
80 NEXT J,I
85 FOR I=0 TO 7:POKE 54272+I,0:NEXT
90 DATA 250,21,154,250,28,214,180,28,214
,150,28,214,250,28,214,180,36,85
100 DATA 150,32,94,250,28,214,180,27,56,
150,24,63,250,21,154
```

The first of the three data items read each time determines how long each note is to be played, while the 2nd and the 3rd data items give the low and high frequency values. To compose your own tune, you would have to change the data items used in the **DATA** statement and change the value of the loop. The above program uses only one of the three available *voices*. Study the program to see if you can understand how it works, and don't be afraid to play with it. Don't worry if you can't figure it out; sound will be covered fully in Section Q.

B4: Sprite graphics

This mode of graphics allows you to define small moveable objects that can be used together with either character mode or high resolution graphics mode. This is the procedure used to create a sprite:

- 1 Draw a 24 by 21 pixel array on a piece of paper.
- 2 Mark all the bits that you want to set.
- 3 Calculate the value for each byte by adding the numbers representing the bits that are marked.
- 4 Use a loop to READ these values and POKE them into their respective blocks (each block is 64 bytes long, i.e. 3 x 21).
- 5 Turn the sprite on by POKEing a particular register (memory location).
- 6 Tell the video chip which block contains data for your sprite.
- 7 Turn on the colour of the sprite.
- 8 Move your sprite onto the screen.

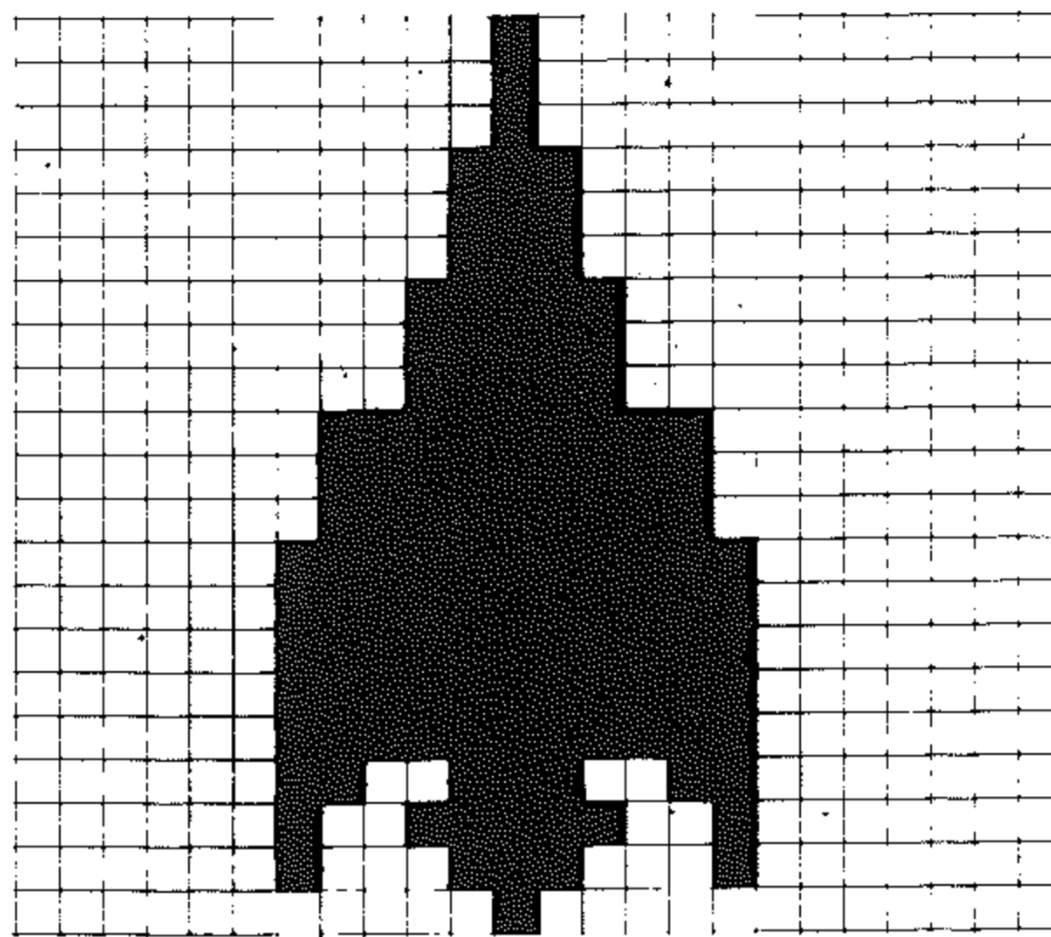


Figure B2: A sprite.

To display the sprite in Figure B1 on the screen, try the following program. To form the data, add all the values in each byte (8 bits). Note that there are 3 bytes in each row. For example, row 12 produces the following data items:

$$\begin{aligned} 0 + 0 + 0 + 0 + 0 + 0 + 2 + 1 &= 3 \text{ for byte 1} \\ 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 &= 255 \text{ for byte 2} \\ 128 + 0 + 0 + 0 + 0 + 0 + 0 + 0 &= 128 \text{ for byte 3} \end{aligned}$$

Here is the program:

```
5 PRINT " "
6 REM STORE SPRITE DATA IN BLOCK 13
10 FOR M=0 TO 62
20 READ SDAT
30 POKE B32+M,SDAT:NEXT M
35 REM TURN ON SPRITE 0
40 POKE53269,1
45 REM DATA IN BLOCK 13
50 POKE2040,13
55 REM COLOUR SPRITE IN CYAN
60 POKE 53294,3
```

```

65 REM MOVE SPRITE AROUND
70 FOR I=249 TO 0 STEP -1
80 POKE53248,150:POKE53249,I
90 NEXT I
100 GOTO 60
110 DATA 0,16,0,0,16,0,0,16,0,0,56,0
120 DATA 0,56,0,0,56,0,0,124,0,0,124,0
130 DATA 0,124,0,1,255,0,1,255,0,1,255,0
140 DATA 3,255,128,3,255,128,3,255,128
150 DATA 3,255,128,3,255,128,3,57,128
160 DATA 2,124,128,2,56,128,0,16,0

```

To create your own sprite, all you have to do is to change all the data items. Try running the program with the following direct commands to expand the sprite in the X and Y direction:

```

POKE 53277,1
POKE 53271,1
POKE 53277,0
POKE 53271,0

```

B5: Arithmetic

Simple arithmetic can be performed in command mode, and quotation marks are not needed:

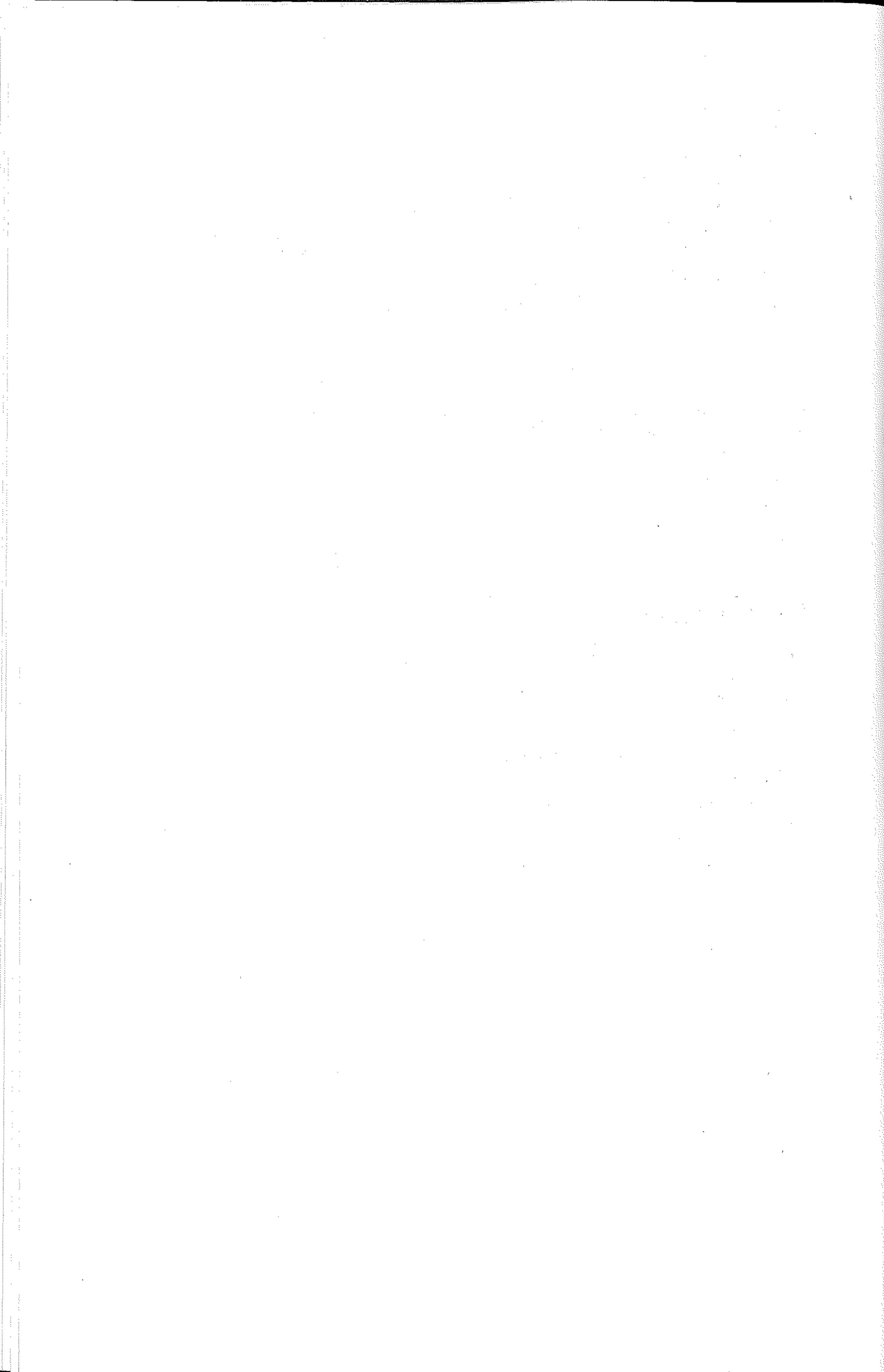
```
PRINT 3+4 RETURN
```

The answer 7 is printed.

Try some other calculations using the operators -, /, * (multiply), and ↑ (raise to the power).

There is a priority of operations in a multiple calculation, which is: brackets, raise to the power, multiplication, division, addition, item subtraction. For example, try this:

```
PRINT 2*(100 - 3 * 5)/5 ↑2
```



Section C: Basic BASIC

C1: The BASIC language

This book is all about BASIC, which is the most commonly used computer language. Just as English is a natural language used to communicate with people, BASIC is a formal language used to communicate with computers. Like natural languages, BASIC has grammatical rules which, although they are fairly simple, must be strictly followed to ensure that the computer understands exactly what it is being instructed to do.

BASIC stands for **B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode. It was invented in 1964 in the USA and is a combination of simple English and algebra. BASIC is the language used throughout this book to write *programs*, which tell the computer what to do, and set the sequence in which particular operations are to be performed.

BASIC is a high-level programming language. Instructions written in BASIC are translated (by the computer's built-in program) into the low-level programming language (the *machine code*) that directly controls the switching of the electrical impulses inside the microchips. High-level languages like BASIC are far easier to write programs in than the low-level languages.

C2: A simple program

A sequence of BASIC *statements* is called a *program*.

```
10 INPUT A
20 INPUT B
30 LET S=A+B
40 PRINT S
```

In this case, line 20 can be deleted and line 10 changed to

```
10 INPUT A, B
```

because more than one variable – the values, here, of **A** and **B** – can be input on a line. Also, the LET operator is optional here (more about that later); line 30 could be `30 S=A+B`.

This simple program prompts for inputs and then adds two numbers and prints the result on the screen.

Before a program is keyed in it should be designed to make the computer do exactly what is required. The way to do this is to write it down line by line on a piece of paper – a process sometimes called *coding*.

A program which doesn't work as intended is said to contain errors, or *bugs*. If it is asked to do something it can't do, or if an instruction has been left out, the computer will respond with an *error message*. If the program runs without error messages, but doesn't do what the programmer wanted it to do, then it is the programmers' fault. In either case it must be corrected (edited, or *debugged*). This is done while the program is in the computer, using the Commodore's powerful editing facilities.

Editing or revising a program is part of the process of *program development*. At any time before or after editing is finished and the program works, it may be *listed* (printed on a printer), as well as *saved* by recording it on cassette tape or magnetic disc, and *stored* so that it can be *loaded* back into the computer without having to key it in again. A partly corrected or edited program – which may not yet run properly – may be saved and worked on later should you so wish.

The complete exercise of designing, coding, developing and documenting a program is called *programming*.

C3: A statement

This is a line of a BASIC program, otherwise called a statement:

```
10 INPUT A
```

A statement is also called a *line*. A statement can simply state something, or instruct the computer to do something. It is composed of:

- a line number (in this case, 10)
- an instruction or command (INPUT)
- and – maybe – variables (A)

Statements are either:

- executable (specifying an action, like INPUT A)
- or
- non-executable (providing information only)

All variables used (such as A) have a start value of 0.

```
10 INPUT A
```

tells the computer to request the user to input a value for the variable A from the keyboard.

C4: Statement numbers

Each BASIC statement or line must begin with a line number, 20 in this example:

```
20 INPUT B
```

The number 20 is called a line number, its value is chosen by the programmer. It may be any number from 0 to 63999 inclusive. The computer uses the numbers to keep the statements in order. Each statement has to have a unique line number – if the same line number is used twice, the second line will replace the first.

Lines may be keyed in in any order; the computer sorts them into the correct sequence according to the line numbers. Statements are usually numbered in tens so that additional lines may be easily inserted later if necessary.

C5: Instructions

A statement gives an instruction to the computer. In this example the instruction is LET.

```
30 LET S=A+B
```

An instruction is called a statement type because it identifies a type of statement. LET tells the computer to let the variable S have a value equal to the sum of the values of variable A and variable B. But, the instruction LET is automatically implied in a statement. Thus:

```
30 S=A+B
```

is equivalent to

```
30 LET S=A+B
```

C6: Numeric variables

A numeric variable is the name given to a storage location which holds a number in the computer's memory. A numeric variable can have a name which is:

- a letter from A to Z, or
- a letter followed by a number, or
- a group of letters and/or numbers.

Only the first two characters are recognised by the Commodore. For example, SUM and SUMTOTAL will represent the same variable name, because they both start with SU. Some care must be taken over the choice of some variable names: the variable name TOTAL will not be accepted by the Commodore, since the first part of it (T0) is a Commodore keyword.

Numeric variables are used to represent numbers inside the computer: values are given (*assigned*) to the variables, and these values are used in calculations. So the variable is simply a symbol or a name chosen to represent the value of a parameter or a quantity, ie. the number stored in the named memory location. Variable names should not be too long or they will be tiresome to key in (which is why single letters are usually used). For example:

S – speed
PRICE – price of fish
SUM – sum of the first set of numbers
R3 – resistor number three

In the sample program the statement

```
10 INPUT A
```

sets up a variable in the computer's memory with the symbolic name A. (It could have been called NUM1, or even FIRST NUMBER.) The statement tells the computer to ask the user to input a value for A when the program is run. If the number 3 is keyed in, the memory cell allocated to A will contain the number 3. This value is then used in all calculations involving A until its value is changed.

In the statement

```
30 LET S=A+B
```

or

```
30 S=A+B
```

S, A and B are the variables in the algebraic equation $S=A+B$. S is the *unknown*, and will take the sum of the values of A and B. The computer will work out the value of $A+B$ and put the result in the memory cell it has allocated to the variable S. The computer will not let you input $LET A+B=S$ (it will respond with a syntax error), because the variable to be given a value must come first. $A+B$ is not a valid variable name.

Variables are so called because their values can vary or change according to the values that are input, or in the course of a program, when the computer is instructed to do something which causes the value to change. (For *constants*, which are quantities that do not change their value, a name or a symbol is set up in the same way, by giving it a value with the equivalent of a LET statement, but it keeps the same value – like a variable that doesn't vary.)

Variable names may be of any length, but they must start with a letter, and must contain only the alphanumeric characters (the letters A to Z and/or the numbers 0 to 9).

C7: Strings and string variables

Strings

A string is a group of characters enclosed by quotation marks.

The following are examples of strings:

```
"PETER"  
"12345"  
"JANUARY 1ST 1982"  
"!$,**."  
"REF:A2"
```

As well as numbers, computers can also handle text or groups of characters. To define a group of characters as a string, quotation marks are placed at the beginning and the end. This tells the computer, for example, that the string "PRICE" means the characters P, R, I, C, and E and not the numeric variable PRICE, which was used above to mean the price of fish.

Strings can contain any character, upper or lower case, which prints on the screen, plus spaces, except that a string cannot contain a pair of quotation marks, because the computer thinks it has got to the end of the string when it gets to the second set of quotes.

String variables

Strings can be handled by string variables, just as numbers can be manipulated with numeric variables. A string variable is used to store a string. It consists of one or two letters, either upper or lower case (depending on display mode), followed by the \$ sign; or a letter followed by a digit/letter and \$ sign. For example:

```
a$,z$,m$,A1$,AA$,AX$
```

Strings are assigned to string variables with LET statements, as with numeric variables. For example:

```
10 LET A$="STRING 1"  
20 PRINT A$
```

The memory store allocated to A\$ will contain the string STRING 1. When the program is run the computer will print the contents of memory store A\$ on the screen. Note that the string is printed without the quotation marks. The string consists only of the characters inside the quotes.

C8: Operators and operands

Operators

Operators perform arithmetic, logical or conditional operations on variables or numbers. In the sample program the line

```
30 LET S=A+B
```

uses two arithmetic operators: = and +.

Operands

Operands are the variables or numbers which are manipulated – operated on – by the operators. In line 10 above, the variables S, A and B are operands.

C9: Keying in a statement

The **RETURN** key must be pressed after each statement has been entered.

```
10 INPUT A      RETURN
20 INPUT B      RETURN
30 S = A + B    RETURN
40 PRINT S      RETURN
```

Pressing **RETURN** informs the computer that the statement (or *line*) is complete, just as the carriage return on a typewriter means that the typist is ready to begin a new line.

C10: Correcting errors

INST/DEL

The **INST/DEL** key acts as a backspace, deleting the character or symbol immediately to the left of it. For example, you may get:

```
10 INPUT S ■
```

by accidentally pressing S instead of A. To correct this, press **INST/DEL** and you get

```
10 INPUT
```

You may now continue and type in A.

CRSR

The cursor control keys move the cursor around the screen. To correct an earlier mistake, use the cursor keys to move the cursor to a position immediately to the right of the character to be changed, then use the **INST/DEL** key as above.

You may use the cursor key to return the cursor to the end of the line, if you have more to key in; otherwise, you may press **RETURN** immediately. You can delete an entire line by holding down **INST/DEL**, but a quicker way is described shortly.

C11: Direct Instructions

Direct instructions are executed immediately. They do not need line numbers, as they are not part of a program. They allow direct control over the computer. Examples are **RUN**, **LIST**, **LOAD**, and **SAVE**. To execute such a command, key in the command and press **RETURN**.

Most commands are also used as instructions in programs. Some instructions that could be used as direct commands are not very useful in that role. Similarly, some commands that could be used in programs never are. However, each command has a role to play in the **BASIC** language, and individual commands will be dealt with as they are encountered in the text.

You have already met the **RETURN**, **INST/DEL** and **CRSR** keys; these are commands that don't print but act instantly. All the others need to be activated.

C12: Editing a program on the screen

There are many editing tricks, depending on what is necessary in an individual program. You will learn these with experience. For example, you can edit a line as described above, deleting characters and replacing them, or you can overwrite an entire line simply by entering a new line with the same line number. As soon as you press **RETURN**, the new line will replace the old one. If you want to move a line (effectively to renumber it), move the cursor to the beginning of the

line, overwrite the line number, key in the new number and press **RETURN**. Then move to a clear screen line and type the 'old' line number, press **RETURN** and it will be cleared from memory – a quick way to delete an entire program line.

C13: Listing a program on the screen

When the program has been entered into memory, to produce a listing on the screen of all lines accepted by the computer key in:

LIST

LIST is a command that prints on the screen and is executed when **RETURN** is pressed.

Parts (only) of a program may be listed by including line numbers representing the range of lines to be listed:

LIST 30 lists line 30

LIST -30 lists all lines up to line 30

LIST 30- lists from line 30 to the end

LIST 30-90 lists lines 30 to 90 inclusive

C14: Running the program

The simple program has been keyed into the computer line by line and entered into memory; now see if it works. Give the computer the **RUN** command: press **RUN** and **RETURN**. The **RUN** command starts execution of a program at the lowest line number unless otherwise instructed (*see below*).

RUN is another command that is keyed in, appears on the screen, and will not be executed until **RETURN** is pressed. When this happens the program starts operating. In the case of the sample program **ADDER**, the computer will almost instantly ask you to input a value for the first variable, **A**. Key in the number 3 in response to the prompt (?) and press **RETURN**. The (?) will appear again immediately: the computer is requesting another number to be assigned to the variable **B**. Key in the number 5 and press **RETURN**.

The result (8) is printed on the screen. Notice the message that also appears: the message **READY** means that the command has been executed and your program has finished.

We can also run the program from a line other than the first program line: **RUN n** will start execution from the line number specified (n).

RUN 20

will start executing the program at line 20.

Note that when the **RUN n** command is used, all statements before the specified statement number (n) will be ignored, and any variables defined in these statements will be considered by the computer to be 0 because it has not run the lines. The program may not work, and an error message may result. All values of variables are initialised to zero by the **RUN** command.

You can **RUN** the program as many times as you like: press **RUN** and **RETURN** again. To get a listing, enter **LIST** and press **RETURN**. The program can now be edited if necessary.

C15: Error messages

The computer tells you that it has finished running the program by giving you a message on the screen:

READY

This tells you not only that the program has finished but also that no errors were found.

There are special diagnostic messages which appear on the screen if a program does not work when it is run. These take the form:

XXXXXX ERROR IN LINE N

where XXXXXX indicates the type of error, and N the number of the line where the program halted due to the error. There is a list of error codes in Appendix II. The message helps in correcting or debugging the program; the line indicated in the error message is not always the one which *caused* the problem, but it is always at least a clue to the nature of the problem.

EXERCISES

- Add an extra line at the beginning of the program. Key in:
5 PRINT "PROGRAM ADDS 2 NUMBERS"
and run the program, starting from different lines by using RUN, RUN 5, or RUN 10.
- Edit the program to obtain the original version.

C16: How the program works

10 INPUT A Line 10 tells the computer that a number must be input and given the name A (ie. assigned to the variable A). The computer reads the line and prints the prompt (?) on the screen, reminding you to input a number. The computer will wait until you key in a number. The number is then stored in memory cell A. The computer goes to the next line.

20 INPUT B Line 20 tells the computer that another number, to be assigned to the variable B, must be input. The (?) appears on the screen and the computer waits until you key in a second number, which is stored in memory cell B. The computer goes to the next line.

30 S=A+B Line 30 tells the computer that a variable S is to be assigned the value of the sum of the variables A and B. The numbers in cells A and B are added and the sum is stored in cell S. The computer goes to the next line.

40 PRINT S Line 40 instructs the computer to output the value of S to the screen. The computer looks for the next line, can find no more statements to execute in the program, and gives a message READY on the screen, telling you that the program has finished. The computer now waits for more commands.

C17: Naming the program

The program needs a name in order to:

- differentiate it from other programs
- store it permanently (SAVE it)
- LOAD it back into the computer in order to run it

The program name can be any combination of characters, but to be used with the SAVE and LOAD instructions it must start with a letter, and can have a maximum of 16 characters in the name. For example, the short sample program you have used is called ADDER. It is sensible to keep the program name short

and relevant to the type of program. Programs which undertake various kinds of statistical analysis could be named:

"STATS 1"

"STATS 2" etc

Programs which perform calculations for experiments in the laboratory could be named:

"OPTICS 3"

"FRACTION"

"TITRATION" etc

If spaces are used in program names, it is easy to misread them, or forget that there should be a space. If the program name is not one word, we can use an asterisk:

"PETER*1"

"FOCAL*LEN" etc

Program names should be written down in a directory, which enables the programmer to access a program library of programs stored on tape or disk.

Section D: Saving, Verifying, Loading and Listing

D1: Saving the program on cassette tape

Programs must be saved because when the power supply is switched off (or disrupted – variations in the mains supply can affect the computer) the RAM memory and the registers in the CPU are cleared and the program is lost. (The Random Access Memory is said to be *volatile*.) This means it would have to be keyed in again – this would be all right for a five-line program, but a 50 liner might take you an hour.

If a copy of the program has been recorded on magnetic cassette tape using the SAVE command, it can be reloaded into the computer quickly, using the LOAD command. (Tape storage is not as quick or reliable as floppy discs for off-line storage, but it works, and it is a lot cheaper.)

Software (programs) stored on tape is available for use when needed, making it *permanent*. Software also has to be *portable*. Programmers want their work to be able to be used by other people with the same computer; software available on cassette may be distributed to many users.

The SAVE command outputs the program to the cassette recorder. If the cassette recorder is in record mode, a copy of the program will be made on the tape.

Saving the program on tape

- 1 Make sure the cassette recorder is plugged into the Commodore.
- 2 Insert a new C5 or C12 computer cassette tape into the recorder. Short cassettes are more convenient than long ones for our purposes.
- 3 Run the tape through on fast forward and then rewind, to tension the whole length of the tape evenly.
- 4 Set the tape counter to zero and run the tape forward five revolutions (about 2 or 3 seconds). This takes the start to a point beyond any tape leaders or opening kinks.
- 5 LIST the program on the screen.
- 6 Key in SAVE "ADDER" (if you are saving the sample program from the last section).
- 7 Press RETURN. The 64 will print a message on the screen which tells you to press record and play on the tape player, putting it into the record mode. Do so.
- 8 The screen will go blank as recording begins. When recording is complete a report will appear on the screen. The recorder will stop automatically – press the STOP key. Make a note of the tape counter reading for when you next save a program, and so that you will be able to locate the program easily on the cassette.
- 9 You can check that the program is properly recorded before wiping it out of the computer's memory. Rewind the cassette to before the start of your recorded program and key in VERIFY, then enter the program name between quotes. Press return and the message "PRESS PLAY ON RECORDER" will be displayed. Press PLAY and the screen will clear. The Commodore will display FOUND "PROGRAM NAME" and the tape will stop. Now press c= to shorten the delay before the recorder re-starts.

- 10 When the program has finished playing back, a **READY** message means the program was correctly **SAVED**, but **LOAD ERROR** means the recording is faulty and you should save it again.
- 11 When the program is saved, run the cassette on for a further five or so more revolutions of the tape counter, ready for the next program.

VERIFY "NAME" checks that the program is **SAVED** as named on tape against the program and variables in the computer memory. If there is a match then the Commodore will display

```
VERIFYING "PROGRAM NAME"  
READY
```

If there is no match then a **VERIFY ERROR** message will be given; **VERIFY " "** will list the actual tape contents on the screen.

EXERCISES

- **SAVE** the program on to the tape.
- **VERIFY** the program saved properly.

D2: Deleting the program from memory

A sure way is to switch off the power, but this is not recommended. This should only be done if the computer needs to be reset. It is much better to use the command **NEW**.

NEW

The **NEW** command deletes any current program and variables from memory. Use the **NEW** command before **LOADING** a program into the computer from cassette tape, to erase old programs and data from memory.

There is another command that only affects the variables store, without also clearing the program store, as **NEW** does.

CLR

The **CLR** command erases all the variables in the current memory. It can be used as an instruction in a program, as can **NEW**, but since it would merely wipe the program, its uses in programs are severely limited. (Try it, if you want to see a program self-destruct.) **CLR** is similarly useless in the middle of most programs – it would mean redefining all variables.

EXERCISES

- Run the **ADDER** program. Enter **PRINT " SHIFT CLR/HOME"** clear screen symbol. Press **RETURN** to clear the screen. Enter **GOTO 40**, then press return again. The computer will print the value of **S**. Now enter **CLR** and press **RETURN** then enter **GOTO 40 RETURN**. You will get a message **0 variable**, indicating that $A+B=0$ because **A** and **B** have not been used, and assumed to be zero, because the computer has wiped the value of **S**. (**GOTO** is an important programming tool; it will be covered in due course.)
- **LIST** the program **ADDER** on the screen. Enter **NEW** and press **RETURN**. Then enter **LIST** and press **RETURN** again. What happens? Why?
- Key in the first line of the program and press return. Switch off the power supply. Switch it on again. What happens?

D3: Loading the program from cassette tape

LOAD "NAME" The command `LOAD "name"` waits for the cassette to play the portion of tape with the program called "name" and copies the program into the computer's memory. When the Commodore finds a program it displays `FOUND "NAME"`. Press `c=` to shorten the pause before loading commences.

This means that you can start the tape, give the command `LOAD "name"` and the computer loads nothing into its memory until the signal it recognises as the name appears on tape. You can thus search a tape for a program. The Commodore will print on the screen the names of any programs it finds on tape, before it encounters the specified program.

LOAD "" This command, with nothing between the quotes, `LOADs` the first program it finds on the tape.

LOADing procedure

- 1 Place the tape with the desired program in the cassette player.
- 2 Position the tape, using the counter, to a point just before the location of the required program.
- 3 As a precautionary measure, clear the computer's memory using the `NEW` command if a program has been `RUN` (*executed*) and is still in memory.
- 4 Key in `LOAD "ADDER"` and press `RETURN`.
- 5 When the 64 has found the program it will print `FOUND ADDER` on the screen and will load after a short pause (shortened further by pressing `c=`). Previous programs will also be listed but will not actually load.
- 6 When the program is correctly loaded, the word `READY` will appear on the screen to indicate that all is well.
- 7 Press the `STOP` key on the *recorder*.
- 8 Remove the tape when finished.
- 9 `LIST` and `RUN` the program.

Causes of misloading

- Loading started in the middle of the program. If a mistake has been made with the start position, rewind the tape completely and let the computer search for the program name.
- The program is not on the tape. Check your directory!
- The program name is incorrect. Try again, making sure you have spelt it correctly in the `LOAD` instruction. If you fail again, run through the tape using the `LOAD ""` command. This will load the next program each time. If each program is not the one you want, repeat for the next program on the tape. The 64 will print the names of all the programs on tape if you use a name that does not exist as a program name such as `LOAD "ZZZ"`.
- Stray electromagnetic fields – from equipment such as a TV or a fluorescent light being turned on – can spoil the program signals.

EXERCISES

- 1 Load the program `ADDER`. `LIST` and `RUN` it.
- 2 Delete it from memory using `NEW`.

D4: Program libraries and catalogues

You should start to keep proper *documentation* of your programs from the beginning, or very soon your program library will be a jumble. A library is a collection of programs stored on tape (or magnetic disc), but you will also need a catalogue or directory, a list of the programs in your library together with other important information about them. Start a notebook and keep a directory of all programs you have entered and saved on tape. This will seem pointless at first, but you will appreciate your systematic habits when you have accumulated a large number of programs.

Each program you write should be:

- Named.
- Saved on a cassette tape.
- Listed (printed out) on a printer, if you have one.
- Documented and catalogued in your notebook.

The documentation, or complete collection of information about the programs and files, should include:

- What the program does.
- How it does it.
- A listing.
- A flowchart.
- How to use it in the program.
- When it was written and by whom.

Flowcharts will be introduced in Section G.

A useful layout for the directory section of notebook would be:

Program name:	MOONLANDER
Cassette name:	GAMES 3
Location:	100 - 120
Program length:	30 lines
Date created:	18.5.84
Author:	PAUL NIXON
Function:	Lands a spaceship on the moon

Use the labels on each side of the cassette. Write on each side:

- Cassette name or code.
- Date.
- Program names. Make sure these are correct in every detail!

Your directory should provide you with the more detailed information, such as precisely where the program is to be found.

Section E: Improving the Program

E1: Adding comments: the REM statement

The REM (remark) statement is used for adding comments to a program. A REM statement and anything following it on the same line is ignored by the computer when the program is RUN. These comment statements are for the users' benefit only. They can be used to explain what the program is doing at any given point. For example:

```
REM **THIS PROGRAM ADDS TWO NUMBERS KEYED IN AND PRINTS THE RESULTS
```

Notice how you can use asterisks as a visual device to separate the text from the instruction.

```
100 REM **END OF PROGRAM**
```

The complete program including all REM statements appears on the screen when using the LIST command.

The saved program ADDER so far looks like this.

```
5 REM ** ADDER **
10 INPUT A
20 INPUT B
30 S=A+B
40 PRINT S
```

Now add some additional REM statements:

```
6 REM **THIS PROGRAM ADDS TWO NUMBERS KEYED IN AND PRINTS THE RESULTS**
60 REM **END OF PROGRAM**
```

Key these extra statements in. LIST the program and RUN it.

Don't worry about the line numbers not being in intervals of 10. You can renumber the program when all the extra lines have been added should you need to.

REM statements can be added at the end of program lines as multiple statements separated by colons. For example:

```
10 INPUT A : REM ENTER A VALUE FOR A
40 PRINT S : REM PRINT SUM
```

This is very useful when individual program lines need comments. But remember never to *begin* a multiple statement line with a REM statement. Any statements after the REM would be ignored, because the computer, finding a REM, will display the rest of the line but otherwise ignore it.

Change line 30 to

```
30 REM SUM : S=A+B
```

and RUN the program to test this.

E2: Using the PRINT statement

Messages and instructions to the user can be printed on the screen like this:

```
7 PRINT "INPUT TWO NUMBERS"
```

The message INPUT TWO NUMBERS is a string, and will be printed without the quotes.

To print the numbers keyed in and the result:

```
40 PRINT A;"+";B;"=";S
```

A and B are the names of the variables to which the numbers keyed in are assigned, and S is the variable that stores the sum of A and B, that is, the result. Variables do not need quotes to be printed. The punctuation (semicolons) tells the computer that you want close printing, with each print item (character or variable) directly after the last, with no spaces between. The quotation marks enclosing the symbols + and = mean that you want those symbols printed. The semicolons are optional in PRINT statements of this type and may be left out. Numeric variables will print with a following space so these are not needed as part of the symbols display.

To leave spaces between lines printed on the screen:

```
8 PRINT
```

The PRINT instruction used on its own will print an empty line on the screen.

Note that you now have a new line 40. Key in the new lines and the program will look like this:

```
5 REM ** "ADDER" **
6 REM **THIS PROGRAM ADDS TWO NUMBERS KEYED
  IN AND PRINTS THE RESULTS**
7 PRINT "INPUT TWO NUMBERS"
8 PRINT
10 INPUT A
20 INPUT B
30 S=A+B
40 PRINT A;"+"B:"=";S
60 REM **END OF PROGRAM**
```

E3: Adding a loop

The statement GOTO n transfers program execution to a specified line number, n. For example:

```
50 GOTO 7
```

When line 50 is inserted into the program, after the computer prints the result on the screen line 50 will send the computer back to line 7 to execute the program again from that line; and when the computer reaches line 50 again it will be sent back to line 7, and so on. A *loop* has been constructed, and the program will carry on looping forever or until it is stopped.

Key in line 50. RUN the program. Remember that with each loop the variables are assigned fresh values. When you are tired of doing that, read on.

```
ON NV GOTO
```

This allows several possible transfers of control to lines of program, depending on the value of the Numeric Variable NV. For example:

```
ON X GOTO 60, 70, 100, 30, 80
```

transfers control to line 60 if X = 1, line 70 if X = 2, line 100 if X = 3, and so on.

E4: Stopping the program

Press **RUN/STOP** to stop program execution. This does not work when executing an input statement; in that case, wait until it is executing any other statement.

The **STOP** command stops a program with the message:

```
BREAK IN LINE N
```

The line number will be the program line the computer was executing when it was stopped.

You can cancel the **STOP** command and continue the program with the **CONT** command as long as no change has been made to the program. This will allow the program to continue from the line it was stopped at. Key in **CONT** and press **RETURN** to continue the program.

E5: Testing for a condition

In a program, decisions can be made that will affect what the computer does next. A decision is made on the basis of whether a *condition* is true or false.

A condition has the form (X) (condition) (Y) where X and Y are numbers, variables or expressions, and the condition is a *conditional operator*. The = (equality) operator will be used here for the moment. The following are all conditions:

```
X = Y
A = 23
B = 2*3
```

Conditions are tested, and the next action is determined by the result of the test, with the **IF** and **THEN** statements used together.

An **IF...THEN** statement has the form **IF** (condition) **THEN** (instruction). For example:

```
IF A = B THEN PRINT "EQUAL"
```

```
IF A = 0 THEN A = 3
```

The instruction can be any valid instruction. The statement means **IF** (the condition is **TRUE**) **THEN** (perform as instruction).

IF (the condition is **FALSE**) the computer ignores the instruction after **THEN** and goes to the next line of the program.

In the simple program **ADDER** the **IF...THEN** statement can be used to insert a conditional test which will stop the loop without using a direct command. You could insert another line:

```
15 IF A = 0 THEN STOP
```

This tells the computer that **IF** $A = 0$ (if it is **TRUE** that A is equal to 0) **THEN** it should **STOP**. **IF** A is any other value (if it is **FALSE** that $A = 0$) it ignores the **THEN STOP** instruction and moves to line 20. Enter this line into the program and **RUN** it. Enter different non-zero values for A to see that if A is not zero then the program continues as before. Enter $.000000001$; only if A is exactly zero will the **STOP** instruction be executed.

(Input 2 for B, and notice that the result is given as 2. This is because calculations are only performed to a certain degree of accuracy. Now enter $.0000001$ for A and input B as 2. The computer returns 2.0000001 as the value of S – the number is within the limit of accuracy.)

Now you have some extra control over the program, but it is still not satisfactory. **IF** $A = 0$ was used here because in this program it is not a value you are interested in seeing added to B (a value used in this way is known as a *dummy* or *sentinel* value – a value used as a signal to the computer, which would not need to be entered in the course of normal inputs). This stops the program

and you can continue it, but the program just goes back into the loop. What is needed is a method of exiting from the loop to end the program, or continuing with more program lines.

This can be done with a *string condition*. The conditional operators can also be used to express relations between strings – either string variables or simple strings.

Insert the following lines:

```
50 PRINT "RUN PROGRAM AGAIN? (YES/NO)"
55 INPUT A$
56 IF A$ = "YES" THEN 7
```

Now when the program gets to line 50 it will print out the message and put the ? on the screen. There is no need to type quotes; whatever characters are typed in will be stored as A\$. The string is entered by pressing **RETURN** after keying in the characters. Line 56 tells the computer to check if the characters in A\$ are the same as the characters of the string YES. If they are it goes to line 7; if they are not, the program will continue to line 60. Notice that any string other than YES will cause the program to continue to line 60.

EXERCISES

- Delete line 15 in the program, which is no longer needed.
- Insert the new lines 50, 55 and 56.
- RUN the program. Enter YES in response to the string input and see that the program loops back to line 7.
- Next, enter NO to see that the program goes to line 60 and gives the message READY. RUN the program again. This time enter anything other than YES or NO, to see that the program goes by default to line 60 if anything other than YES is entered.
- Experiment with the string input. What happens if you press **RETURN** without entering anything? What happens if you try to key in quotes around the string?

E6: Final edit and saving

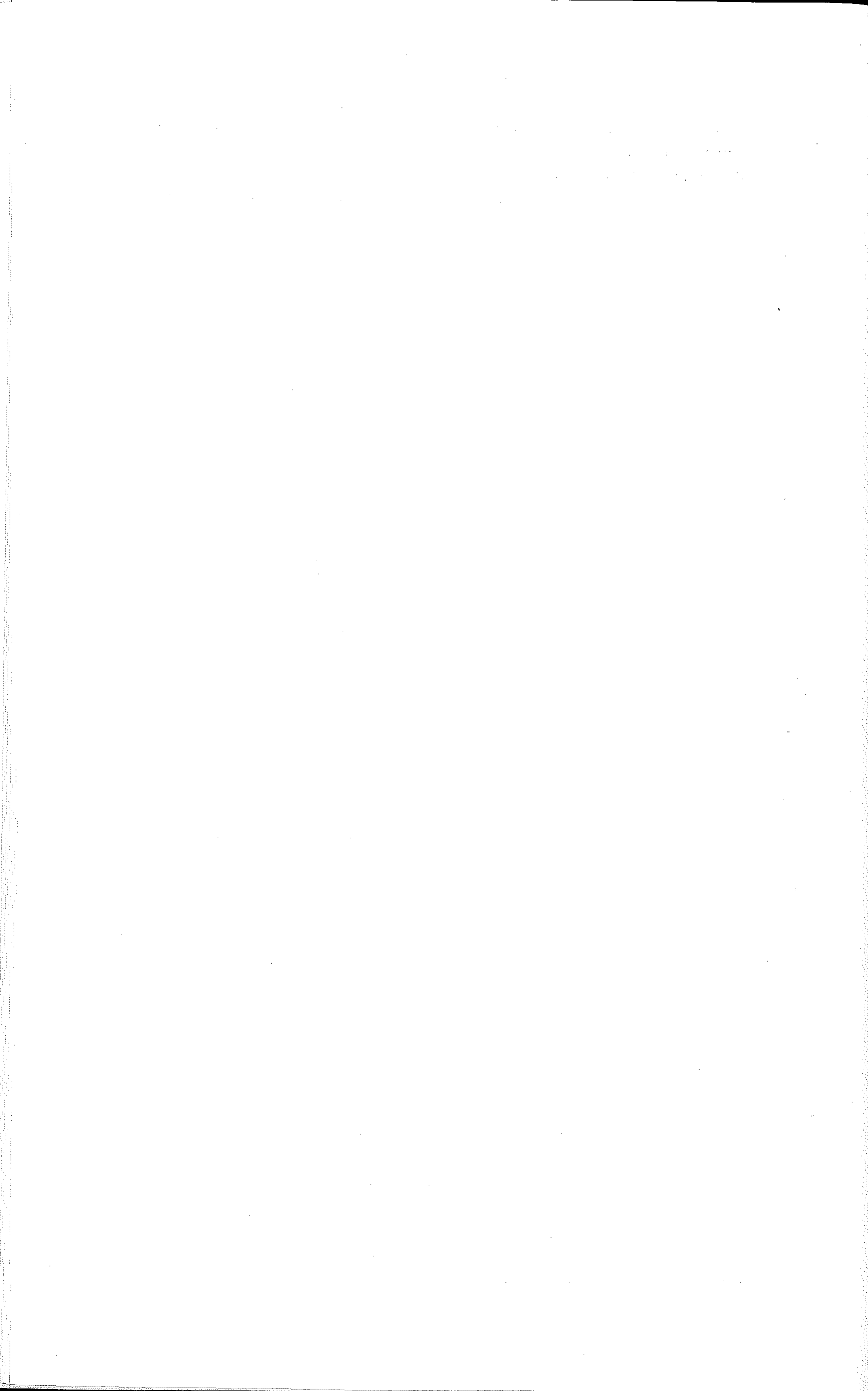
The program ADDER is complete and works. The lines could be renumbered, however. The procedure for this is as follows:

- 1 LIST the program on the screen.
- 2 Use the **CRSR** ↑ and **CRSR** ↔ keys to bring the cursor to line 60 (the bottom line of the program).
- 3 The new *highest* line number is 120.
- 4 Change 60 to 120.
- 5 Press **RETURN**. Line 60 remains, but it is duplicated by the new line 120. Move cursor to free line.
- 6 Delete old line 60 by entering 60 and pressing **RETURN**.
- 7 Change each line number in this way, going from highest to lowest.
- 8 Lines that contain other line's numbers must have these changed to their new numbers. In this program you must change line 56 to:

```
90 IF A$ = "YES" THEN 30
```

where 30 is the new line number corresponding to the old line 7.
- 9 Rename the program ADDER2, at the same time as you change the line number of line 5 to 10. This program must have a different name, both for your reference and for the SAVE and LOAD operations.

- 10 RUN the program to check that it still works, and that we have all the lines, with any GOTO (line number) statements correctly renumbered.
- 11 LIST the new program and SAVE it.
- 12 Write the name of the program on the tape cassette, along with the tape counter readings.
- 13 Put details of the program in your directory, and copy the listing (or stick the printout) in the notebook you are using for documentation.



Section F: Data Input

F1: Data

A program consists of two parts: instructions and the data on which instructions act. So far you have met two types of data – numbers and characters. Groups of characters are called strings. All data used in programs is assigned to variables, which are styled this way:

- numeric: A
- string: A\$
- array: A(I) or A\$(I), where I = 1, or the number of elements in the array.

An example of an array is a list of ten numbers A(10). The first number is called A(1) and the last A(10). You have seen that data is assigned to variables using LET (optionally) and manipulated by INPUT:

```
LET A=3
LET A$="string"
INPUT A (enter 3 on input prompt)
```

Data can also be assigned using READ and DATA instructions:

```
READ A
DATA 3
```

assigns 3 to A.

The READ-DATA structure is powerful when the data to be used is constant, when there is a large quantity of it and/or it is required to be used several times in the program.

F2: READ DATA and RESTORE

Key in the following short program, which READs two numbers from a DATA statement and prints them on the screen.

```
10 READ A,B
12 DATA 1,2
30 PRINT A,B
```

If a program contains READ-DATA statements, the 64 sets up a list of all the data items in the DATA statements, noting the line numbers where they occur. It uses a *data pointer* to keep track of the data values assigned to variables as the program runs. The pointer can be moved to the beginning of the data line using the RESTORE instruction.

When running the program above, the 64 is instructed to read or assign a number to the variable A. It looks at which item in a DATA statement is being pointed to (in this case 1) and makes out assignment: A=1. The pointer moves to the next data item, which is then assigned to B. So the command

```
READ (variable, variable)
```

This assigns sequential data items by numbers, strings, and expressions in a DATA list to the variables named in the READ statement.

BAD DATA ERROR or SYNTAX ERROR occurs if data is of the wrong type.

OUT OF DATA ERROR occurs if there are insufficient data items.

```
DATA item, item
```

This is a statement or line in a program which is part of its data list. Data items are numeric or string. The data list is accessed by a pointer moving sequentially for each READ instruction. DATA items are separated by commas. The 64 does not care where the DATA lines are placed in the program.

EXERCISES

Key in this program, which reads and assigns strings.

```
10 READ A$, B$
20 DATA "COMMODORE",64
30 PRINT A$, B$
```

Remember the two data types: numbers and strings. A string need not be enclosed within quotes in Commodore BASIC.

Controlling the DATA pointer: RESTORE

The DATA list is composed of data items on several DATA lines. The data pointer is initially set to the first data item in the program. It can be moved back to the first item of the first data statement using RESTORE. For example, key in and run this program:

```
10 READ A
20 PRINT A
30 If A=6 THEN RESTORE
40 GOTO 10
50 DATA 1,2,3,4,5,6
```

EXERCISE

Key in and run this program, in which the same data is used by a number of variables.

```
10 READ A$,B$
20 DATA BRIAN, STELLA, PETER, VICKIE
30 PRINT A$,B$
50 READ C$,D$:PRINT C$,D$
```

Then add another line:

```
40 RESTORE
```

Run the complete program a few times and be sure you know how it works.

Position of DATA lines in programs

Program lines containing DATA items are best kept near to the respective READ lines for speed of processing, *unless* a block of data needs to be accessed at several points in a program, when it is best positioned at the end.

F3: FOR TO STEP and NEXT loops

When reading or inputting large amounts of data into a program, as for lists and arrays, the FOR-NEXT repeating structure is a more powerful and elegant method than GOTO n, which was mentioned in Section E3, for repeating the input operation.

Two statements are involved:

- 1 a FOR TO STEP statement, and
- 2 a NEXT statement.

The structure of the statements is:

- FOR (loop counter variable) = (start value) TO (stop value) STEP (counter steps each time)
- (Instructions to be repeated)
- NEXT (loop counter variable)

If there is only one *next* statement referring to a FOR statement, the loop counter (NEXT) variable can be left out. If the STEP value is 1, STEP 1 can be left out of the statement.

The NEXT instruction steps the repeat control variable to the next value.

For example, print the word **COMMODORE** seven times on the screen, using N as the loop control value. 1 is the start value of N, 7 is the stop value, and STEP 1 is the stepping interval.

```
10 FOR N = 1 to 7 : REM STEP 1 IS OMITTED
20 PRINT "COMMODORE"
30 NEXT N :REM could be 30 NEXT
```

When the program is run, N takes the start value of 1 and the program is in the 1st repeat loop. **COMMODORE** is printed, and N steps the loop counter by the step value 1, sending the program back to 10. N now takes the value of 2, *Commodore* is printed a second time – and so on until N takes the value of 7. At line 30 there are no next N's, and control passes to the next line.

If you find FOR-NEXT loops difficult to understand, turn to section L where loops are dealt with more thoroughly. Read it through and come back.

F4: Multiple INPUT

Remember that it is not necessary to have a separate INPUT line for each variable. The INPUT instruction will handle multiple variables. Variables in the same line are separated by commas.

```
INPUT (variable, variable)
```

For example:

```
10 INPUT A,B,C
20 PRINT A,B,C
```

On RUN the prompt ? will be displayed. You can enter these values either separated by commas or one item at a time (BASIC will continue to prompt with ?? until it has all the required values).

F5: Printing on the input line

A message can be printed on the input line by enclosing it in quotes before the variable to be input, like this:

```
INPUT "PRINT ITEMS";A
```

```
10 INPUT "TEST";A
```

F6: String INPUT

Strings can be input without the quotes. Try:

```
10 INPUT A$
```

This assigns the characters entered to the string variable A\$.

F7: READING large data lists

When reading data items into array variables, first dimension the array (tell the Commodore how much space to reserve for the array in its memory) using DIM variable (number of items). For example:

```
DIM A(100)
```

will reserve space for 100 items in a list of variables A(1) to A(100).

```
10 DIM A(100)
20 FOR N = 1 TO 100
30 READ A(N)
40 NEXT
50 DATA (100 items)
```

EXERCISE

Write a program to read in and print 10 numbers. (Use PRINT A(N) in a loop).

Data size variable

```
10 READ Z : DIM A(Z)
20 FOR N=1 TO Z
30 READ A(N)
40 NEXT
50 DATA 17
60 DATA (17 items)
```

Line 10 assigns the value 17 to Z from line 50 and dimensions the list for 17 items. The program loops 17 times to read in the 17 data items in line 60.

EXERCISE

- Write a program for 5 data items and run it.

F8: Data types and structures

The *data types* of information or data which the Commodore uses in programs are numbers, integers, characters, operators (Boolean or logical), or user defined. Data types are organised, through allocation of variables, into *data structures*. These are

- strings
- lists
- tables
- arrays
- trees
- stacks
- queues

Numbers

Numbers are either *integers* – whole numbers, such as 1, 34, 0 and so on – or REAL numbers. REAL numbers have a fractional part and can be represented as decimals: 1.5, 9.36, -0.042.

Characters

Characters are represented in the computer by integers, in an integer code known as the *character code*. Integers thus extend the range of data types to characters. The connection between integers and characters will be explored in Section K.

On the Commodore you can have integer variables. These are specified by using the percent sign (%) in the form:

```
10 INPUT A%
```

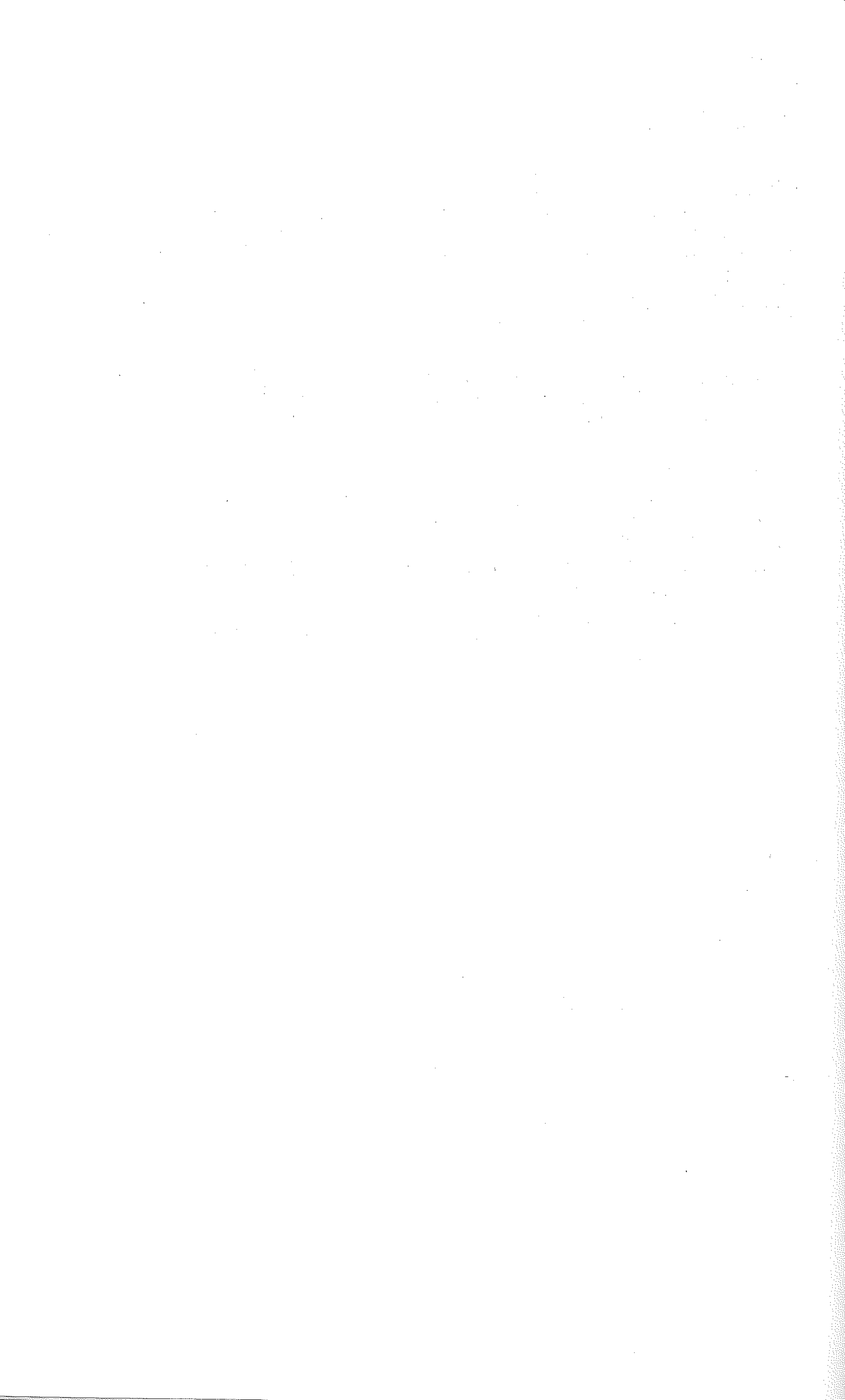
Booleans are the result of a comparison or logical operation; for example, *A is greater than 1* is TRUE or FALSE depending on what is stored in variable location A. TRUE and FALSE are stored in the computer by assigning each an integer:

```
TRUE    = -1  
FALSE   = 0
```

Commodore BASIC allows you to use the results of Boolean or logical comparisons in arithmetic and PRINT statements. Handling this kind of data is covered in Section R.

Data structures are dealt with throughout the text, but most extensively in Sections K (Strings), S (Lists and Arrays) and T (Sorting and Searching).

Now you have met much of the terminology and jargon of programming; in the next sections you will become familiar with some of these data types and how BASIC uses them.



PART TWO

**ESSENTIALS OF BASIC
PROGRAMMING**

1911

1911
1911

Section G: Programming Methods I

G1: Programming

Now that you can operate your computer and have written a short program, we must look in greater detail at how computers solve problems. To enable a computer to solve a problem, you must:

- produce a method for solving the problem, and
- produce a working program.

The method for solving a problem is called an *algorithm*. An algorithm is like a cookbook recipe, and is written down in steps in a brief English style called *pseudocode*, and the method by which we arrive at the recipe is called *structured programming*.

The problem must be broken up into smaller sub-problems, or sub-tasks, in a step by step modular fashion, starting from the simple initial statement of the problem and working down to lower levels of greater complexity (that is, in a *top down* manner). As the problem is refined, the steps become more like the operations the computer can perform. The final description of the lower level of the algorithm will be in terms of the control and other structures of the language.

To help produce the algorithm, *structure diagrams* are used. The simplest of these is a *tree* diagram. The pseudocode description of the algorithm is easily written down from the descriptions of tasks in the tree diagram. This pseudocode description of the algorithm cannot be keyed into the Commodore, and it would not understand it anyway. Each section of the pseudocode must be translated into its equivalent in BASIC, which the computer understands, to produce a program.

For the computer to be able to run the program successfully and produce the results required, there has to be a *logical flow* to the program. This is often difficult to see from the structure diagram, and so another diagrammatic technique is used to illustrate the flow of control through the program (that is, to determine the order in which the program modules or sub-programs are processed and the order of coding the specific instructions within each module.) This technique uses *flowcharts*. These are also important for documentation purposes; they will be described shortly.

Producing a working program means running and *debugging* it. Then the program is tested with sample data and finally documented. In this first section on methodology, problem-solving and coding the algorithm in BASIC will be considered in more detail.

The first half of the activity we call programming is *language independent*. When the problem solving method is produced, the algorithms and their representation in pseudocode and flowchart form can be coded into any computer language; they are thus *portable* from one machine and language to another. But then it is necessary to know the intended language thoroughly, and how the fundamental programming structures used in the algorithm – decisions, loops, subroutines, subprograms, functions – can be implemented in that version of the language which runs on the computer to be used.

Good coding habits are of the greatest importance – there are good and not-so-good ways of turning the solution to a problem into a working program. Style, presentation, ease of understanding, modularity, efficiency are all important. Throughout this book the emphasis will be on correct problem solving techniques and good programming practice, while a thorough knowledge of BASIC is gained.

The first rule of programming is: *program correctly from the start*. Remember, bad habits die hard!

The material in this Section may initially appear dense and difficult to follow. Work through the text carefully, and refer back to this Section as often as you feel necessary, as each of the topics covered in the following Sections (dealing with the essential groundwork of the BASIC language) is introduced. The exercises given in the text should be used to put into practice both the specific techniques involved and the general approach to programming presented here.

G2: Problem analysis

Producing the algorithm, or method of solving the problem, is often the most difficult part of programming, because it involves the most work. Careful planning and organisation from the start are essential. The task is simplified when a structured design method is used, coupled with a diagrammatic representation of the algorithm using a structure diagram or flowchart. The actual coding of the program in BASIC using the available language instructions is then a straightforward matter.

To produce the algorithm, the programmer must:

- 1.1 State the problem.
- 1.2 Research the problem.
- 1.3 Design the algorithm.
- 1.4 Describe the algorithm in pseudocode and flowchart form.

Each of these steps must be broken down further. For example, to *state the problem*:

- 1.1.1 State the problem fully.
- 1.1.2 Understand what is to be done.

To solve any problem you must know what the problem is and what is to be done; later you work out how to do it. A complete statement of the problem should include:

- a What information or data is to be input.
- b What answers or results are to be output.
- c What operations are to be performed on the data.

At this stage a precise description of (c) may not be available.

Sample problems

- Write a program which will print out the sum and the average of five numbers input at the keyboard.
- Design a computerised telephone directory, to contain up to fifty entries, which may be updated and assessed in an enquiry mode.

In the first problem the input data, output data and operations are easy to see. The second is much more complex and needs more researching and information.

In 1.1.1 and 1.1.2 above, the programmer must try initially to specify the problem as exactly as possible. When the problem is analysed further, more information – a more detailed specification – may be necessary

In order to *research the problem*:

- 1.2.1 Research and analyse the problem to see how the computer can handle it.
- 1.2.2 Identify all formulae and relations involved.
- 1.2.3 Identify all data involved.

Here you begin to determine how the computer will solve the problem. You need to find out and write down:

- What formulae and expressions are to be used.
- What kinds of data are involved – numeric, string, etc.
- What functions are involved.
- What input and output data are involved.
- What is the form of this data.
- How much data there is.
- What processing is to be done and how many times.

It is useful at this stage to start a data table (a table of variables, constants and counters) to begin to decide how the data will be stored. Other questions to ask when you are a little more experienced are:

- Have I solved a problem like this before?
- Can I use my solution or modify it?
- Has anyone else solved it?
- Where can I find their algorithm or program?

All the facts obtained from researching the problem should be written down. Otherwise the programmer will find it necessary to research the same information again.

You can now begin to design the algorithm, using structured methods.

- 1.3.1 Break the problem down into sub-problems.
- 1.3.2 Use a structure or tree diagram for clarity.
- 1.3.3 Classify modules or parts of modules for:
 - input
 - processing
 - output
- 1.3.4 Use fundamental control structures.
- 1.3.5 Set up a data table.
- 1.3.6 Refine the algorithm until coding into BASIC is an obvious exercise.

Structured programming means designing the algorithm in a top down, modular fashion, with step by step refinement of the solution starting from the statement of the problem, which is placed at the highest level. The problem is broken into sub-problems at successive lower levels. Structure diagrams or tree diagrams are useful as a representation of this refinement process.

G3: Structure diagrams

These make it easier to break down the problem into distinct tasks and sub-tasks, which eventually become simple enough to be coded directly in BASIC instructions. One type of structure diagram is the tree diagram. This upside-down tree has its 'trunk' at the top of the page; it is called Box 1 and could have the title 'Task to be done' or 'Problem to be solved'. For example, make a cup of tea or find the average five numbers.

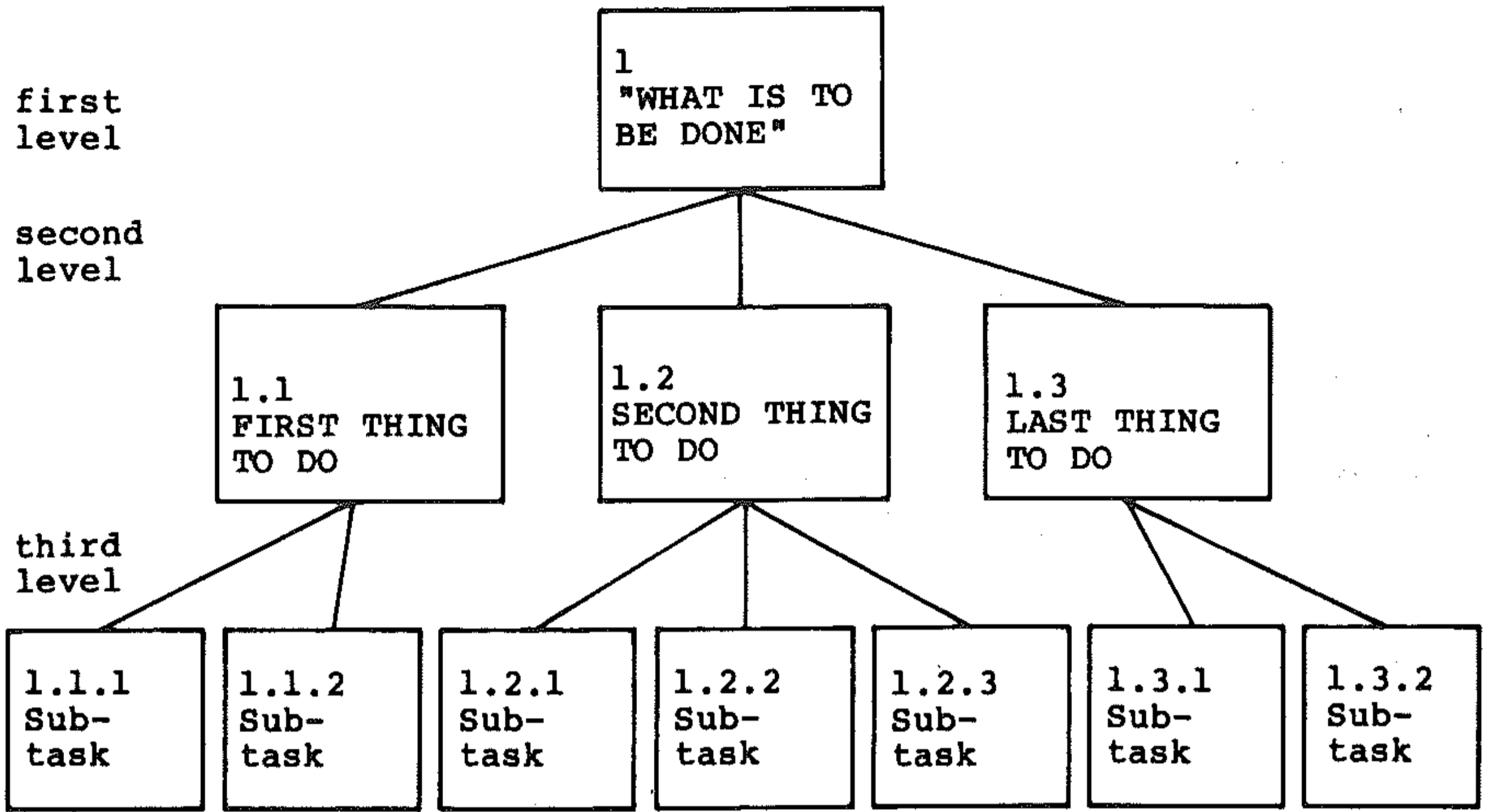
The task is then broken down into things to do. These are sub-tasks and each has its own box. For example:

Box 1.1: First thing to do

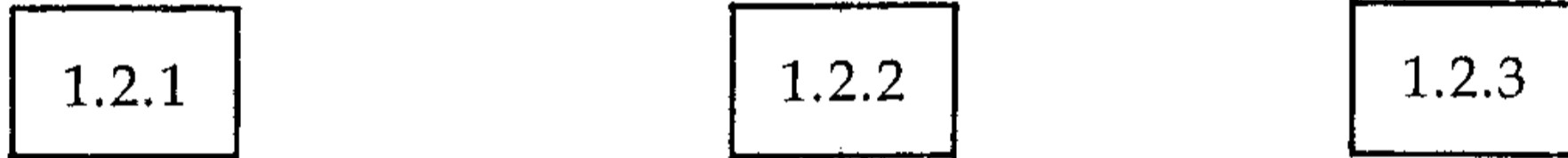
Box 1.2: Second thing to do

Each sub-task is broken down into further sub-tasks: 1.1.1, 1.1.2 etc, each with their own boxes, the things to be placed in them becoming progressively more exact and simple.

Breaking down a task into a tree diagram:



The sort of programs you will start to write in BASIC are *sequential*; that is, things are done one after the other. You indicate the sequence by drawing the boxes containing the tasks to be done in a straight line across the page next to each other, like this:

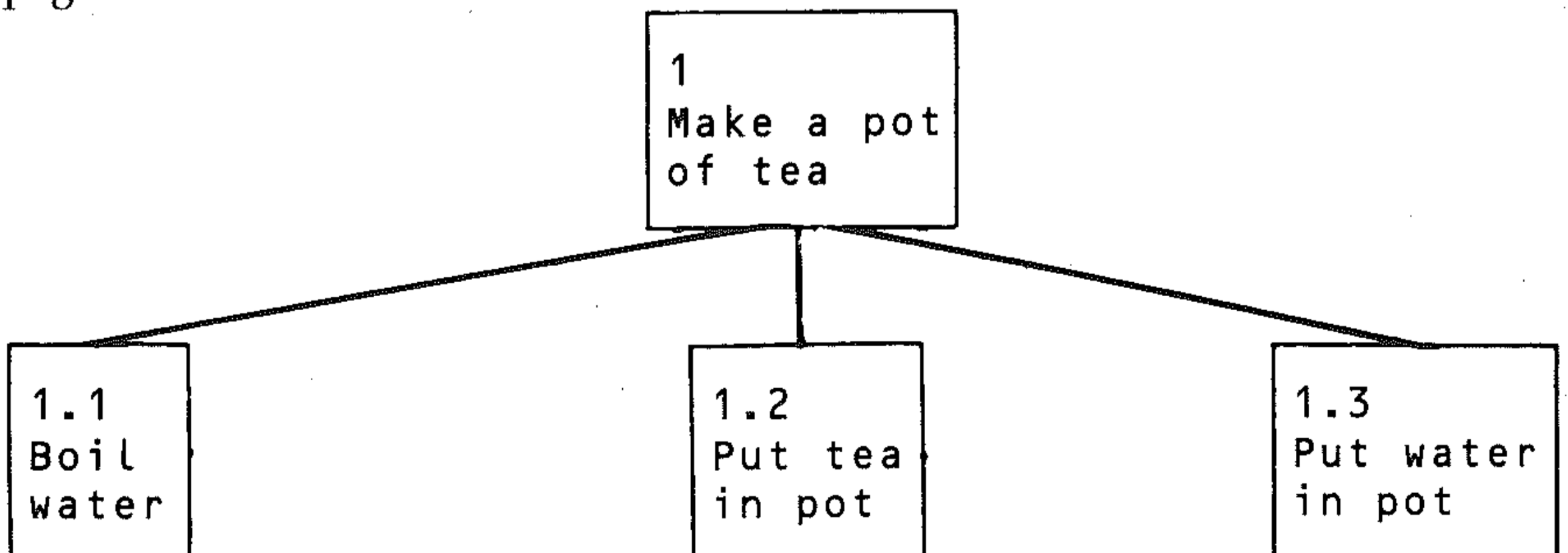


The numbers contained within each box identify where the box is placed on the tree. Take 1.2.3 for example: the first digit shows it comes from the first level 1 'What is to be done'. The second digit '2' shows it has come from the second level box 1.2 'Second thing to be done'. The third digit '3' shows this box 1.2.3 is the third sub-task in the sequence derived from 1.2 which in turn is derived from 1.

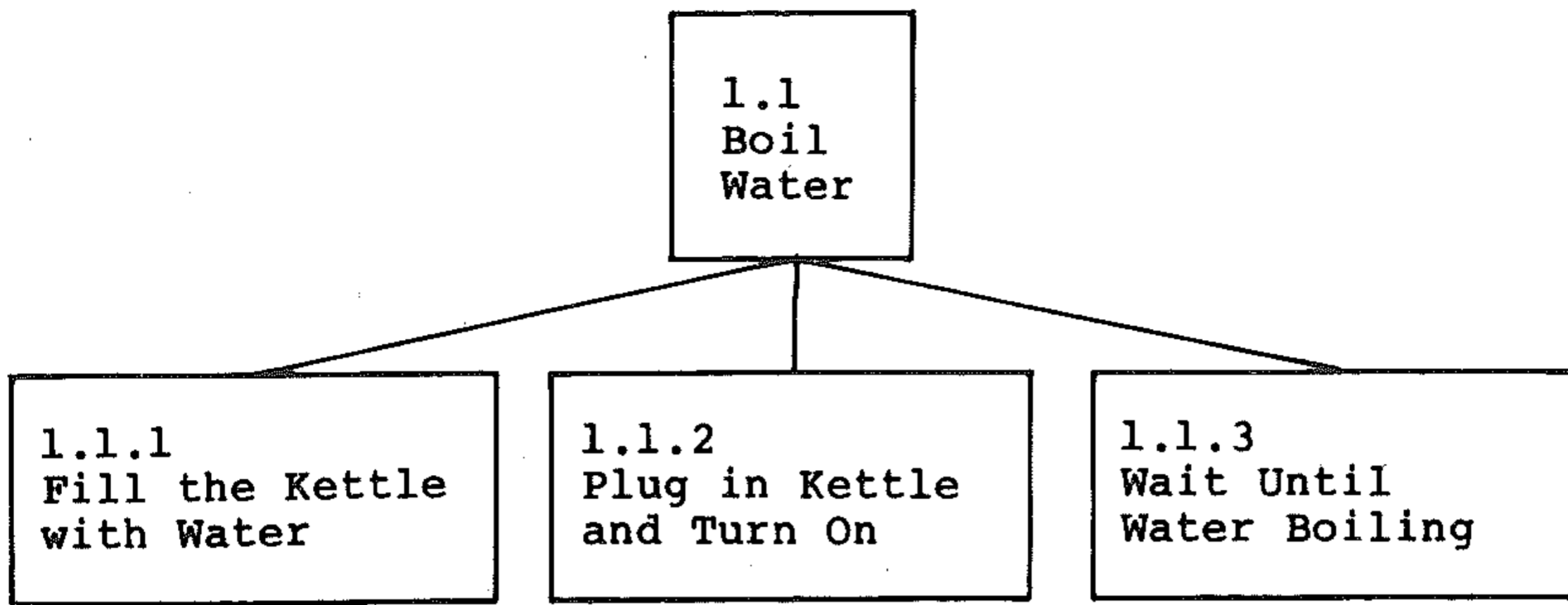
Into the boxes go brief statements of the actions needing to be performed. These are general statements at the top of the tree, eg. 'Get Sum of numbers', but become more specific at each lower level, so that 'Get Sum' is broken down into 'Input first number', 'Input second number', 'Add the two numbers'. Finally the instructions become detailed enough to form our English language *pseudocode* which can be written out, ready to be translated into BASIC instructions.

An example of a tree diagram

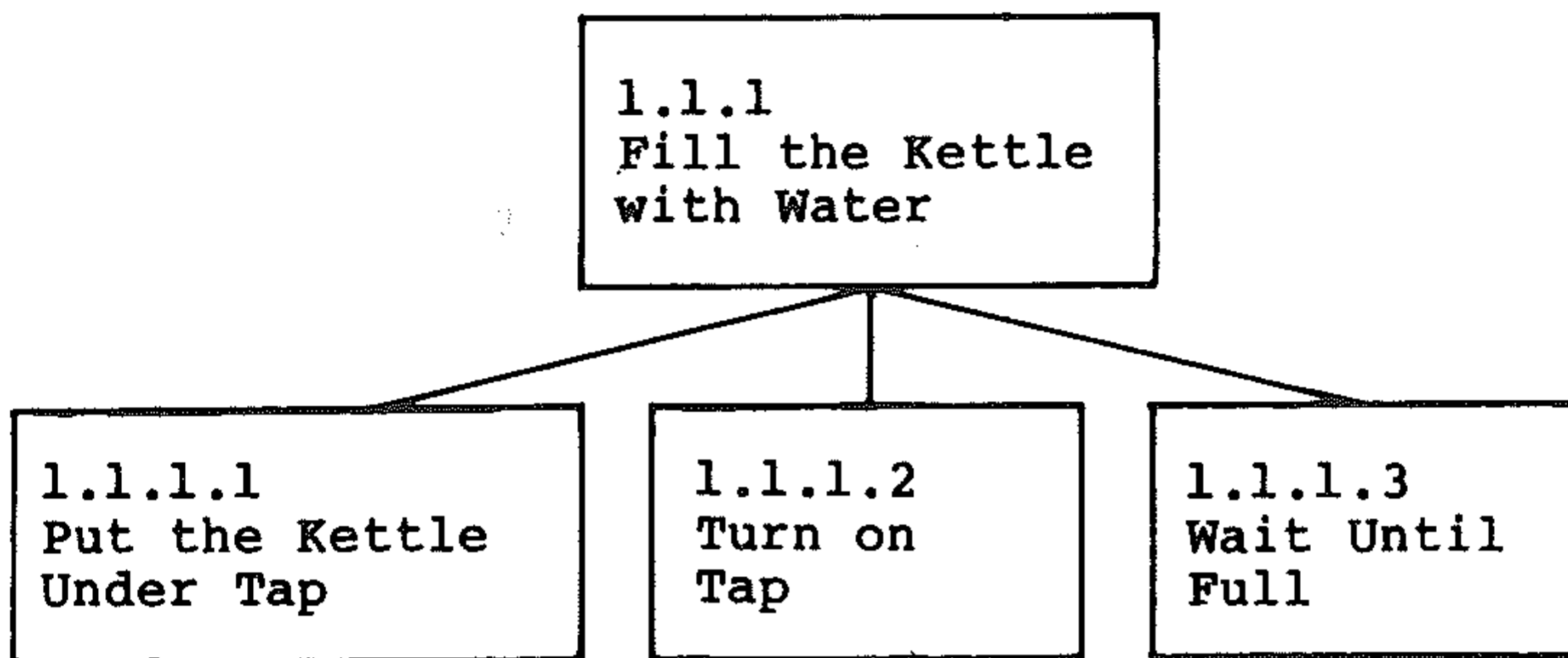
Here is an example. Suppose a robot has to be programmed to make a pot of tea. The major task is broken down into sub-tasks, which are put in order across the page:



Each of these sub-tasks is still far too complicated for the robot; they must be broken down further. Breaking 1.1 into sub-tasks, you get:

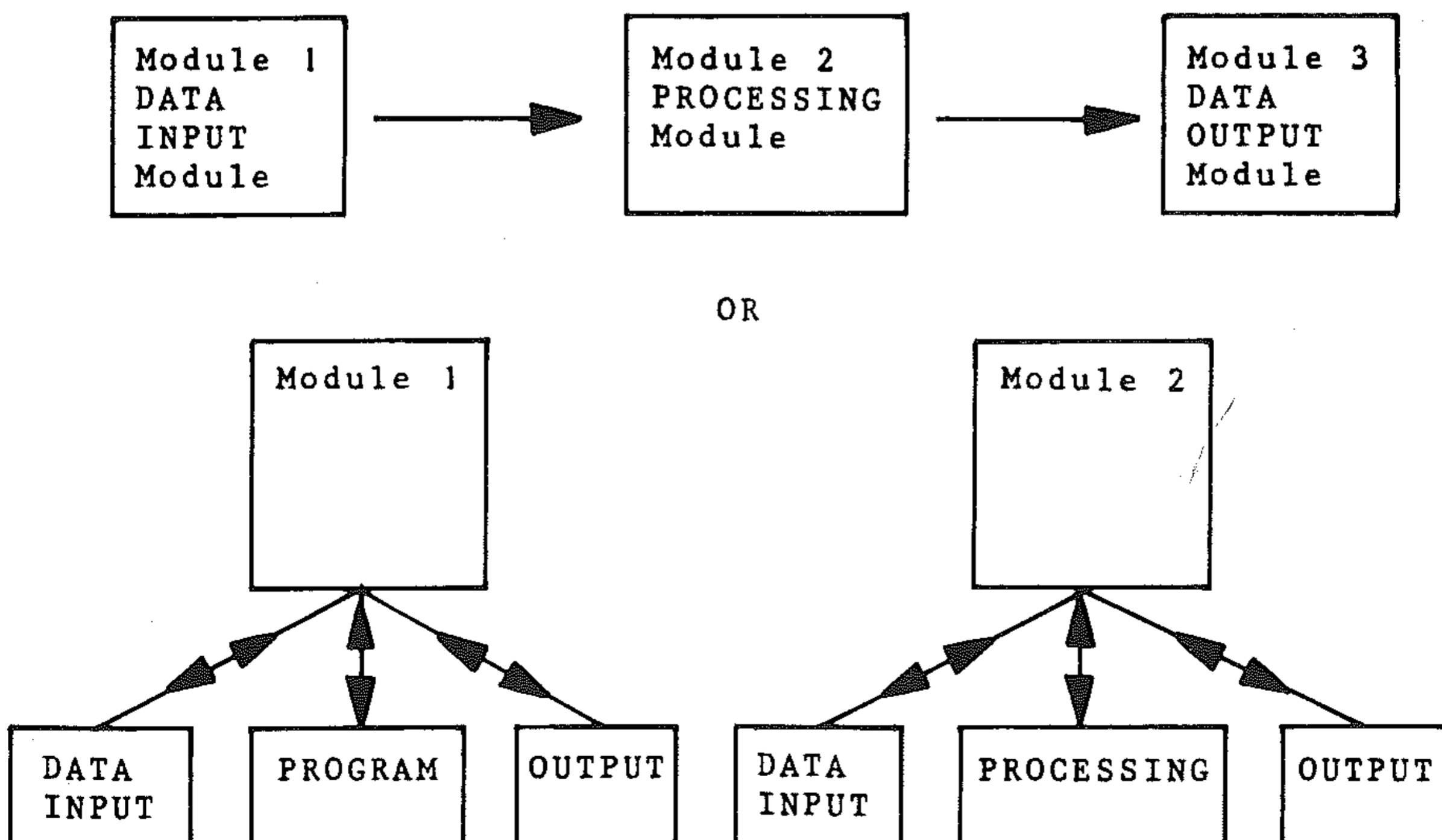


The robot also needs to be told how to fill the kettle; this can be broken down as:

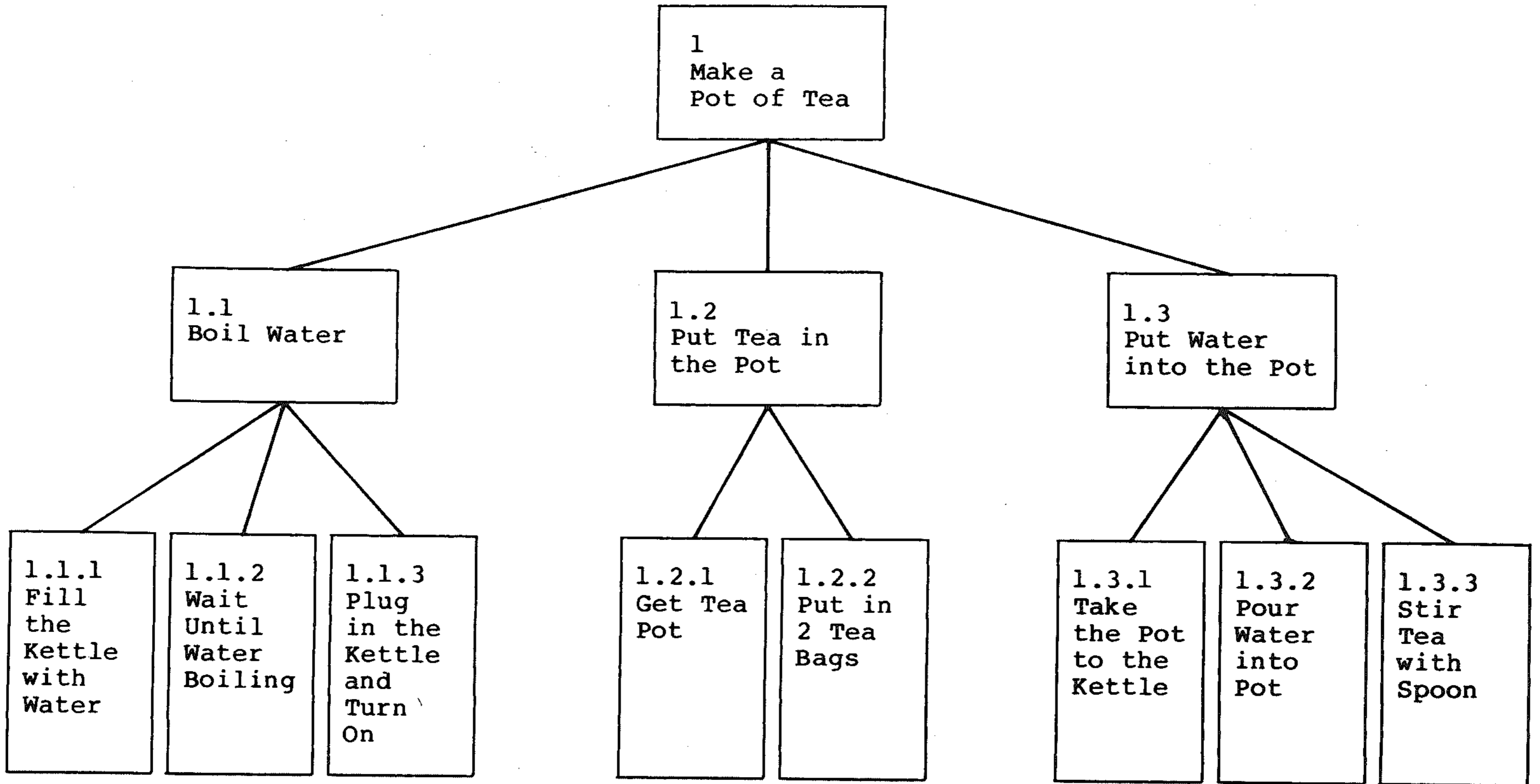


G4: Classifying program modules

Most computer programs involve three activities: input, processing, and output. As programs are designed and modules are formed, it becomes evident from the pseudocode description of the algorithm which of the above functions the modules should have. Depending on the problem and the result of the algorithm design, modules may be separately designated input, processing, and output functions, or may have these functions as sub-modules.



Here is a complete tree diagram. Certain things are still wrong with this algorithm for the robot, but it does show how a problem can be broken down.



G5: Control structures

Control structures are the statements or groups of statements (modules) in a program and algorithm by which the order of processing is controlled. Using them properly is the most important part of programming.

BASIC is a line-numbered language. The order of processing in a program is from the lowest line number in the program sequentially through to the highest, unless this is changed by using a control structure. Control structures link the different modules in a program together, and are themselves modules.

To make algorithms *language independent*, they can be written using a standard notation in pseudocode for the particular control structure, together with its flowchart description. When the structures are coded into the BASIC language, the instructions used and the order of statements in the structure may be slightly different according to the version of BASIC and how 'structured' it is (how easily it accommodates these control structures). The structures to be studied in BASIC are:

- Decision structures
- Transfer structures
- Loops
- Subroutines
- Nested structures
- Sub-programs

Decision structures

Computers make decisions by comparing the value of one variable against another. For example:

```
IF A = 0 THEN      (do something)
IF A$ = "YES" THEN (do something)
```

To make decisions, computers use relational (or conditional) and logical operators, like the equals operator above.

Commodore BASIC uses three decision structures:

- Simple decision
- Double decision
- Multiple decision

As a result of these decisions, control may be transferred to another program module, or local processing within the structure may take place.

Transfer structures

Transfer structures are discussed further in Section H of this book. They include:

- unconditional transfer, a direct transfer of control using a GOTO (line number) statement. Transfer is to another program statement or a module consisting of a group of statements. GOTO is a very powerful structure and must be used with care.
- conditional transfer, in which transfer of control to another segment is made as a result of a decision: for example, IF (condition is true) THEN GOTO (line number).

Loops

The need for repetition of simple tasks is one of the fundamental reasons for building computers. Loop structures are incorporated in most computer prog-

rams. A loop is a sequence of repeated steps in a program; this repetition must be controlled. We shall see in Section L that repetition is controlled by *counting* and by *testing for a condition*. Commodore BASIC uses a convenient and powerful set of statements for controlling repetition by counting called FOR...NEXT statements.

There are three common types of loop structure:

- a Repeat (the process) (forever!)
- b Repeat (the process) until (a condition is met).
- c While (a condition holds) repeat (the process).

Structure (a) is of little use, but a programmer has to be aware of it and make sure it does not occur. In structure (b) the condition is tested after processing; in structure (c) the condition is tested *before* processing.

Subroutines

Structured programming involves breaking down a complicated problem into separated independent program modules, called subroutines, which can be worked on separately.

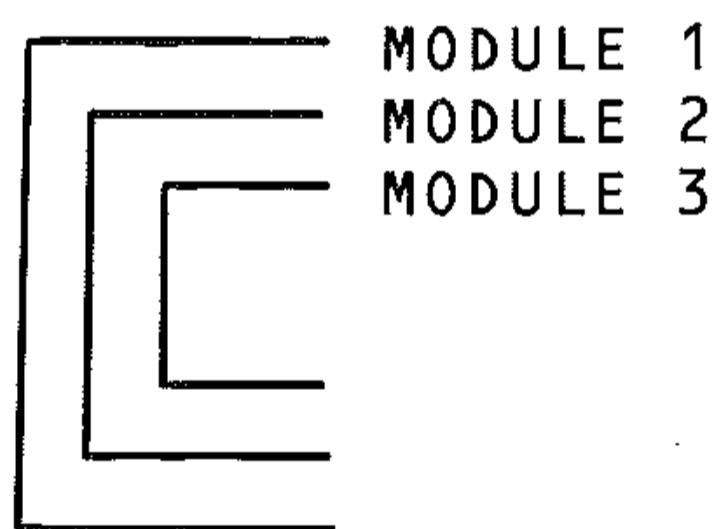
Subroutines are distinct from subprograms, which have similar properties in that they are routines or groups of program statements that are repeated more than once during a program run.

Subroutine modules have a unique address and can have a name (like a person who lives in a house). Transfer of control to the subroutine from the main program when the program runs is by reference to the subroutine address through a *subordinate call instruction*. This is the GOSUB (address of subroutine) statement. A return of control to the main program to carry on processing from where it left off is through a RETURN instruction. Subroutines in Commodore BASIC are explained in Section N.

Nested structures

These are program modules or structures that lie entirely within each other, like a set of Russian dolls.

A simple nested structure looks like this:



In terms of program statements this would look like:

```

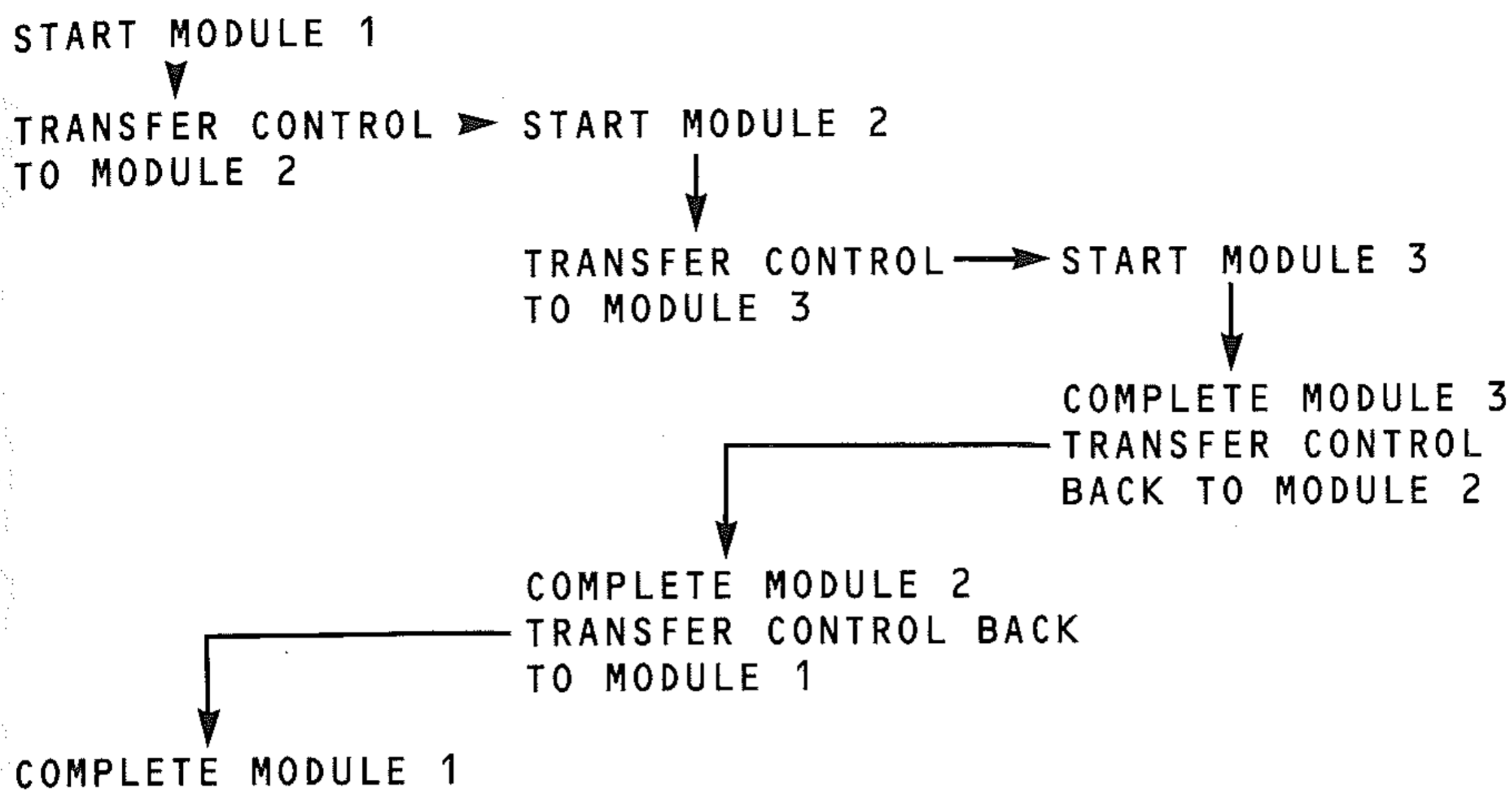
      module 1
10-----
20-----

                module 2
                   40-----
                   50-----
                   60-----

                                module 3
                                   70-----
                                   80-----

                                     100-----
                                     110-----
                                     120-----
                                     130-----
140-----
```

The flow of control is:



Subroutines, subprograms, loops and decisions may be nested in programs. Nesting is fully dealt with in Sections H, L and N.

G6: The data table

When designing a program it is important that knowledge of data and information pertaining to the problem is complete. All data will need to be assigned a variable name, unless it is a numeric constant used in a formula.

The variable type will be either:

- Numeric* – numbers with names, such as A, N1, COUNT, A(IJ)
- String* – characters with addresses, such as A\$, A\$(IJ)
- Logical* – numbers of characters, such as A, A\$, NOT B

Numeric variables will be integers, fractions, real and imaginary numbers. Strings will be names, characters and symbols. Logical variables will be the values TRUE or FALSE, -1 or 0 as appropriate to their use. Logic is dealt with in Section J.

The programmer also needs to know whether the data is *input*, *output* or *intermediate*. Intermediate data is used in the body of the program, for example the value of a loop counter, or the intermediate result of a calculation. Intermediate data is useful for testing and debugging purposes when running the program or algorithm, using machine or hand traces.

The equations, functions and expressions that will use the variables will need to be known. When dealing with equations, functions and expressions, the units of the variables or parameters concerned must be known and should be stated.

The first and simplest data table to construct is a descriptive list of variables to be used in the program. This is important for documentation purposes. For example:

VARIABLE	DESCRIPTION	TYPE
A	First number	Input
B	Second number	Input
SUM	Sum of A and B	Output
A\$	User response to RUN AGAIN?	Output

For program design purposes, the value ascribed to each variable at different points through the programs can be added. This forms a data table that is useful

for checking the algorithm before and after coding it into BASIC, and is also a way of analysing errors in your own program, and understanding how other programs work.

ALGORITHM STEP NUMBER	VARIABLES						
	A	B	S	COUNT	A\$	etc	etc
MODULE 1 1.1 1.2							
MODULE N N.1 . . .							

Loop counters are included in the list of variables. If their values are used for calculation inside the loop, this should be stated. There are some examples of this type of data table in the text.

Refining the algorithm

The tree diagram should be further broken down and refined until the final sub-modules correspond to recognisable BASIC statements and structures. As you get more experienced, you will recognise more complex structures, and the solutions to problems will become apparent at earlier stages.

G7: Describing the algorithm

- 1.4.1 Write out the method of solving the problem (the algorithm) in steps, in a simple English style (pseudocode).
- 1.4.2 Draw a flowchart showing how the program will run from start to finish.
- 1.4.3 Test that the algorithm will work before coding it into BASIC.

Having broken the problem down into subproblems to a stage where a BASIC program can be written, there are more things to do before writing the BASIC code, to ensure that the program will work and that other users can understand it. The documentation of the algorithm description in pseudocode and/or flowchart form is important. This is not part of the program itself, but is a separate document which will also include a listing of the program. This is important for other programmers, who may want to modify your program or use it as part of a larger program, and for you yourself, if you come back to it after a period of time and cannot remember how you designed it or why you did it that way. The program listing alone is often not enough, if the algorithm is complex, to show how the program works.

G8: The pseudocode description

In the structure or tree diagram – which is drawn in rough on a piece of paper as the solution is designed – each block or module down to the lowest level has an English description of the task to be done inside it. (The very lowest level tasks

will be described in sentences that are very similar to the BASIC program statements themselves, as you will see in Section O: Programming Methods II.)

The algorithm will be written out, in a step-by-step fashion, and will include all the descriptions in the boxes. The highest or first level description (simple box) will be the algorithm and program title. The second level will be the titles of the program sections. Each of these major sections will encompass a further group of modules, all of which will be named in our description of the solution.

The lowest level of the tree diagram will be the specific instructions the computer has to perform. These will be translated into the BASIC language on an almost one-to-one basis, and will contain the important and easily recognised language structures for making decisions, branching and jumping, and repetition that have been discussed. (A summary of pseudocode descriptions of some control structures and their flowcharts with BASIC program equivalents is given in Section O: Programming Methods II.) If you imagine turning the tree diagram on its side and taking away the boxes, the descriptions that are left constitute a pseudocode description of the algorithm.

As an example, look at the tree diagram and the algorithm description for the problem of asking the robot to make a pot of tea. Using the tree diagram the algorithm for making a pot of tea is written as a sequence of instructions (to be coded later in a computer language). English is used as a pseudocode and the program is written directly from the sub-tasks in the bottom line of boxes in the tree diagram.

The boxes at higher levels in the tree are used to define distinct modules. Comments or REMARK statements identify each module and explain what is being done in each algorithm section:

```
Remark * * Algorithm for robot to make a pot of tea * *
```

```
Remark * Boil water - task 1.1 *
```

- 1.1.1 Fill the kettle with water
- 1.1.2 Wait until the water is boiling
- 1.1.3 Plug in the kettle and turn it on

```
Remark * End of task 1.1 *
```

```
Remark * Module: Put tea in the pot - task 1.2 *
```

- 1.2.1 Get tea pot
- 1.2.2 Put 200 tea bags in the pot

```
Remark * End of task 1.2 *
```

```
Remark * Module: Put water in the pot - task 1.3 *
```

- 1.3.1 Take pot to kettle
- 1.3.2 Stir tea with spoon
- 1.3.3 Put lid on tea pot

```
Remark * End of task 1.3 *
```

```
Remark * * End of Algorithm - tea is made * *
```

The tree diagram shows why each part of the algorithm is included and why it is in the particular position in which it has been placed on the tree. The tree diagram contains information about three things:

- The problem broken down into different levels of detail, starting from the general concept of what is done down to the specific activities and instructions which will enable the problem to be coded.
- The order in which instructions must be performed.
- The comments which must be included to explain what the program is doing.

EXERCISES

The algorithm has mistakes in it. Some instructions are spelt incorrectly and the robot will not be able to recognise them; some instructions are in the wrong order; some instructions are missing in the algorithm; some instructions are missing in the tree diagram. Find the mistakes and correct them.

Expand the tree diagram and the algorithm to a further sub-task level. For example:

1.1.1 Fill the kettle
becomes

1.1.1.1 Put kettle under the tap

1.1.1.2 Turn on tap

and so on.

Draw a tree diagram and write the algorithm in pseudocode for a robot to set up and switch on your Commodore 64.

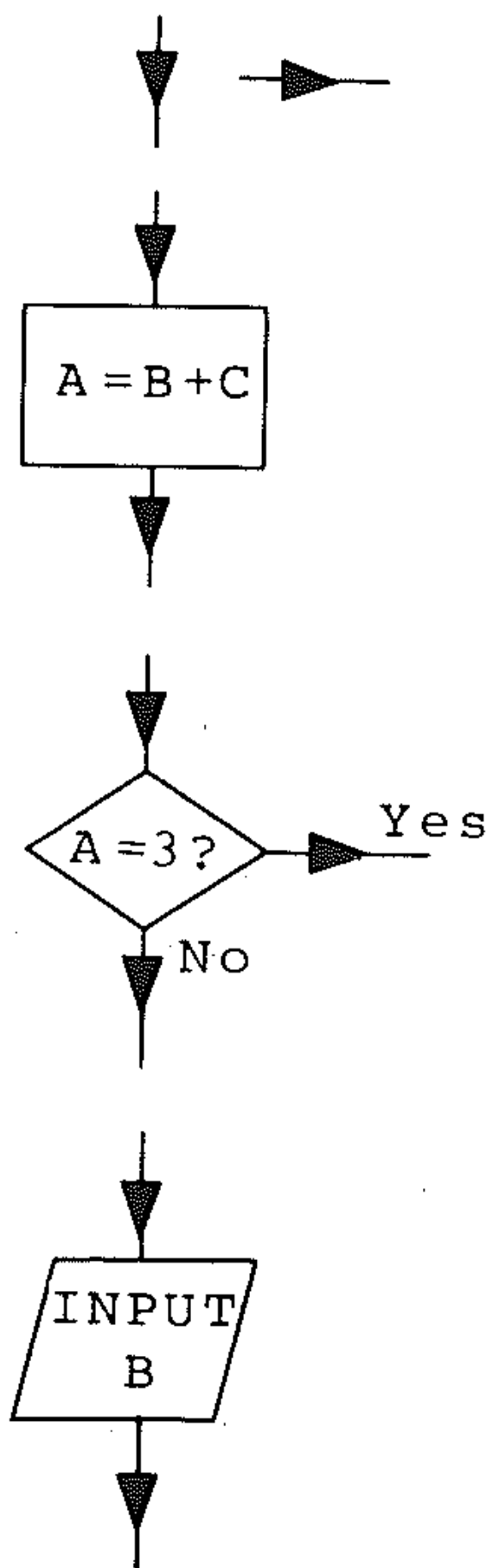
G9: Flowcharts

Flowcharts are a second graphical method used in designing programs. They consist of linked boxes of different shapes. Each shape has a different use and, as with tree diagrams, each contains a brief description of what the program should do at a particular point.

It is harder to design programs using flowcharts than with tree diagrams. Their power comes from using them to help make visible and describe the flow of control in the algorithm and the resulting program. They are used to help code the program into BASIC instructions, and later form an important part of the documentation of a program. Note that flowcharts express the important control structures used in programming in diagram form.

Here is a selection of standard flowchart symbols. There are additional ones, but their usage varies. The conventions of use must be followed if you wish other people to understand your flowcharts. For your own use, in analysing programs, you may be less exact, but not less systematic. Flow in a program can be illustrated by blobs and rectangles, if the lines of flow are correctly shown and the right words are written in the blobs; doing it this careless way is all right for yourself, but not if your flowcharts are to be comprehensible to others.

Flowchart symbols

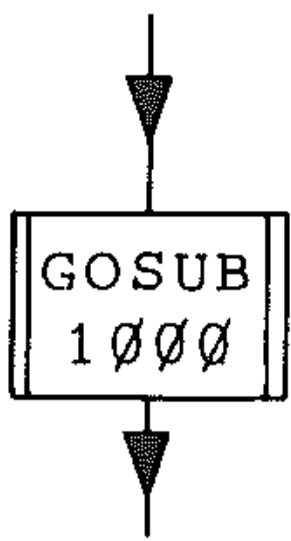


Flow lines. These connect the program blocks. The arrows show the direction of flow, and are very important.

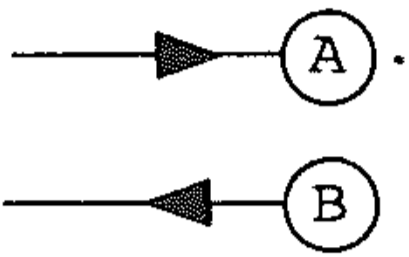
This symbol represents any kind of processing function, or general programming statements, such as 'Purchase tea' or `LET A = B + C`

This represents a decision, with a conditional test, such as 'Is there another shop open' or `IF A = 3 THEN . . .` It has a Yes/No branch, according to whether the condition is True or False, which determines the program flow.

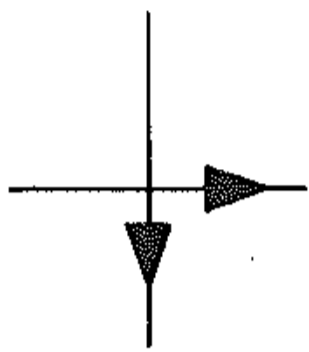
This represents either output in the program to the screen or printer, or input from the keyboard; for example `PRINT "HAVE YOU A PACKET OF TEA?"` or `INPUT B`.



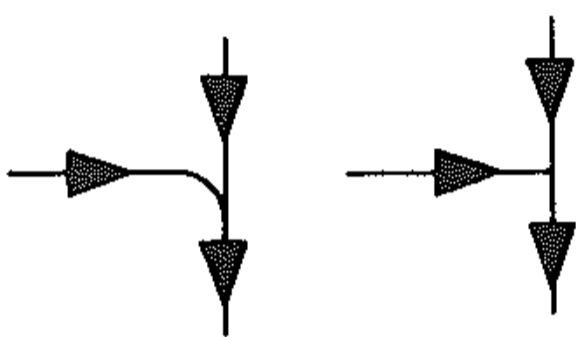
This represents a named process that is specified elsewhere, such as subroutine `GOSUB 1000`. The subroutine would have a separate flowchart.



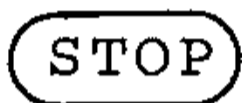
This represents an exit to or entry from another part of the flowchart, allowing one part of the chart to be connected to another part. Used when another direct line link would be confusing, or to connect to a separate page.



This represents the *crossing* of two flow lines. They are not connected.



This represents the *junction* of flow lines. The two lines of flow join.

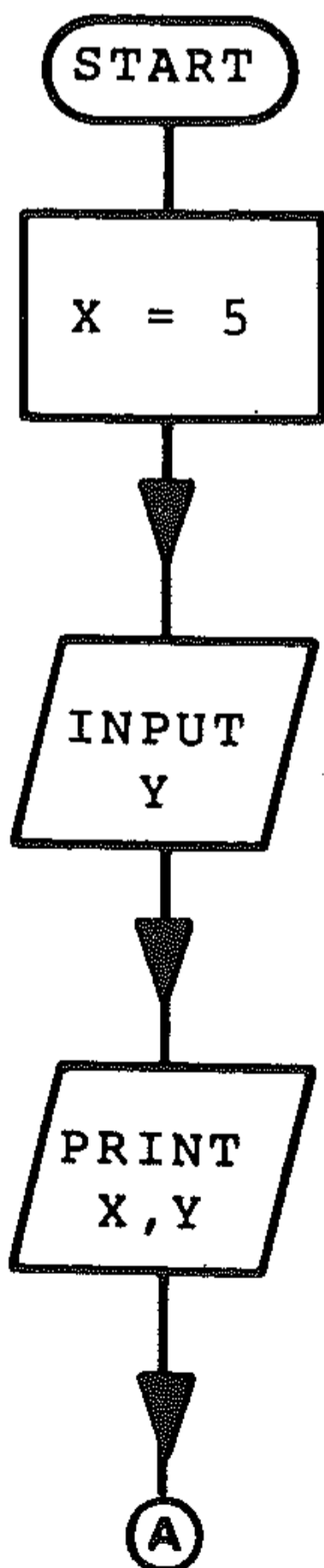


Terminal point, e.g. start, stop, pause.

A flowchart does not branch out like a tree diagram; it always converges to the stop point. It has a direct relationship to the program it describes. Writing down a flowchart is rather like drawing a diagram of the program itself. Below are some examples of simple flow structures, with the program and the flowchart.

Simple sequences flowchart

Flowchart



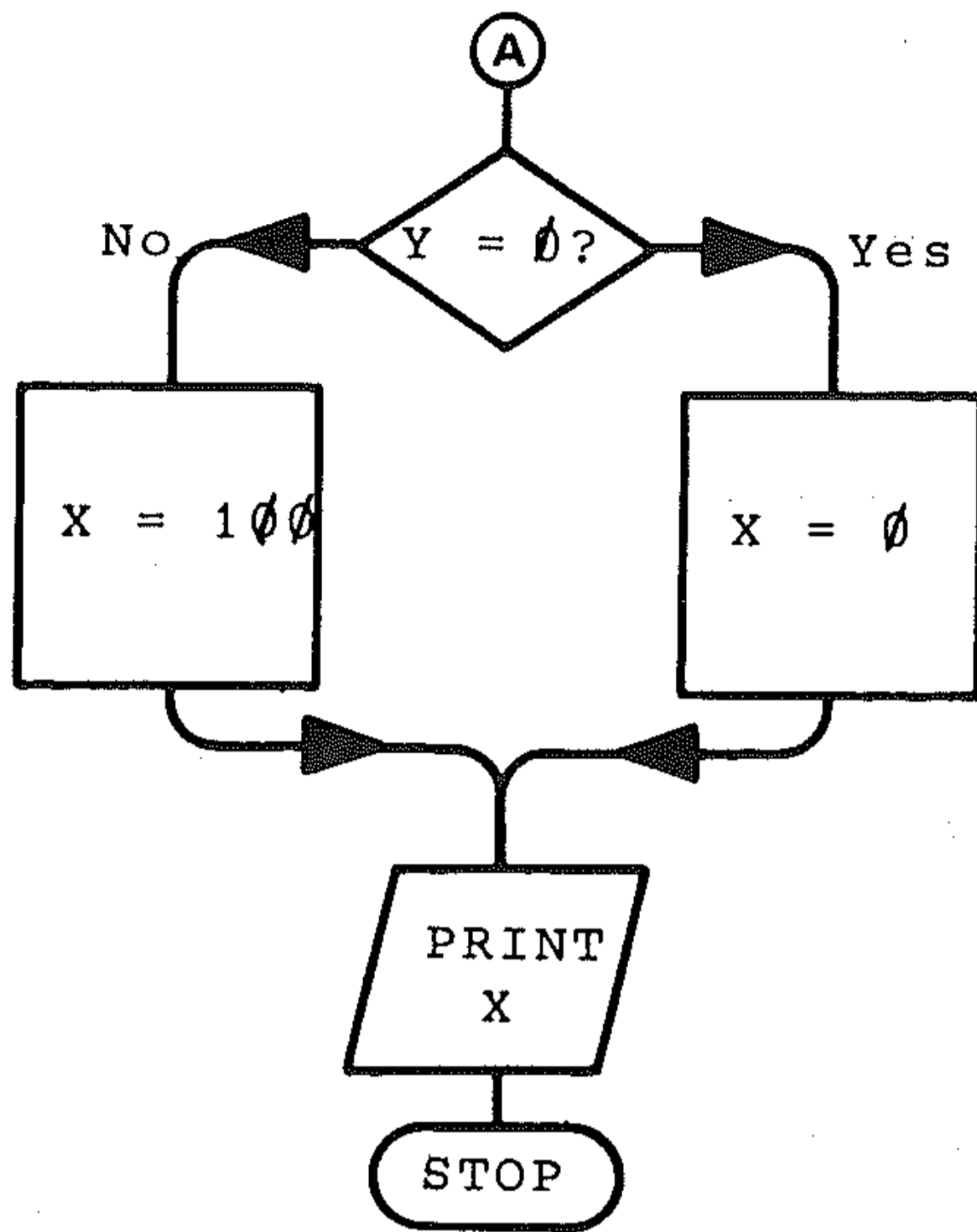
Program

```

10 LET X=5
20 INPUT Y
30 PRINT X,Y
  
```

Decision and program branch flowchart

Flowchart



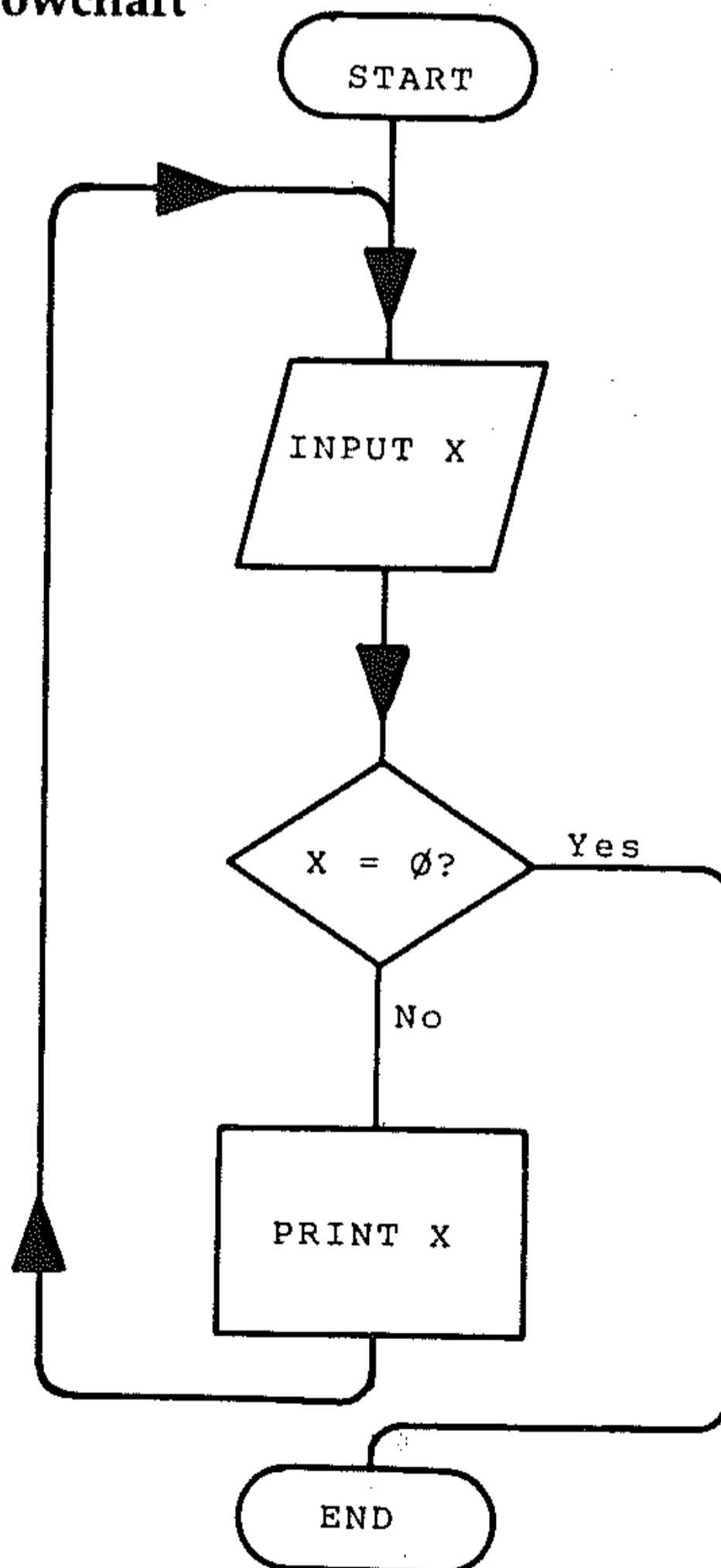
Program

```
40 IF Y=0 THEN GOTO 70
50 LET X=100
60 GOTO 80
70 LET X=0
80 PRINT X
90 STOP
```

Notice that a flowchart symbol is omitted for line 60. This GOTO is indicated by the flow lines. The same is true of the GOTO in the conditional statement of line 40.

Loop flowchart

Flowchart

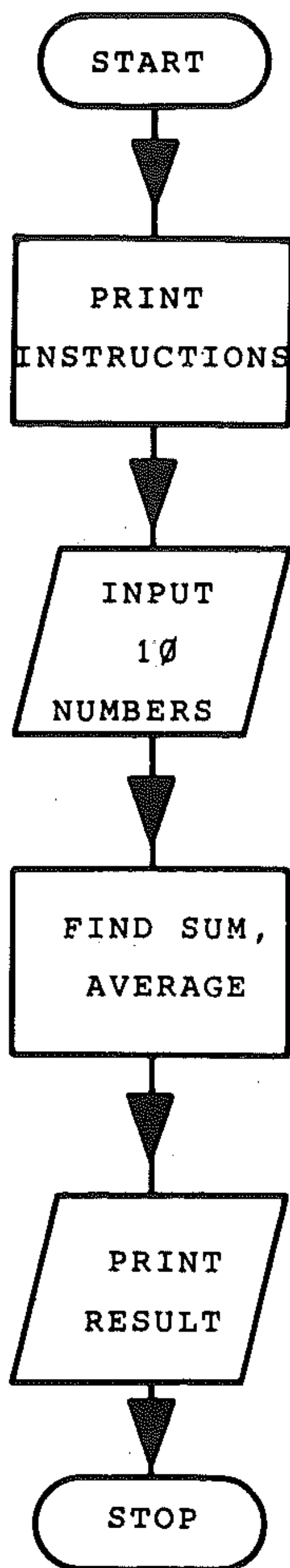


Program

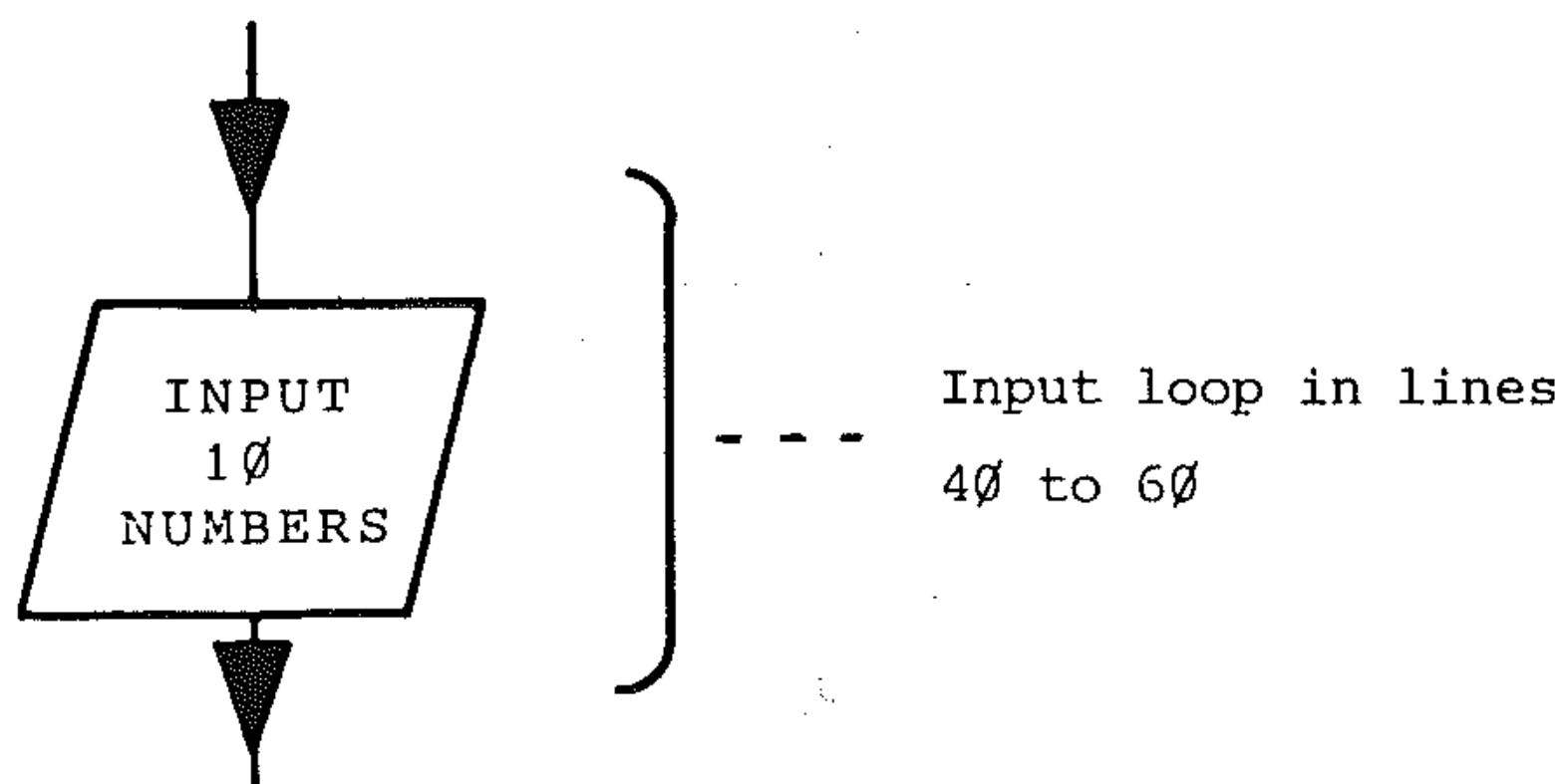
```
10 INPUT X
20 IF X=0 THEN GOTO 50
30 PRINT X
40 GOTO 10
50 REM ** END **
```

Notice that the above flowcharts represent the programs line by line. Flowcharts can also be less detailed, and the flowchart symbols used to represent program

blocks (sequences of program instructions) or modules rather than one or two lines. They then describe a less detailed flow structure. You could have a flow that was represented like this:



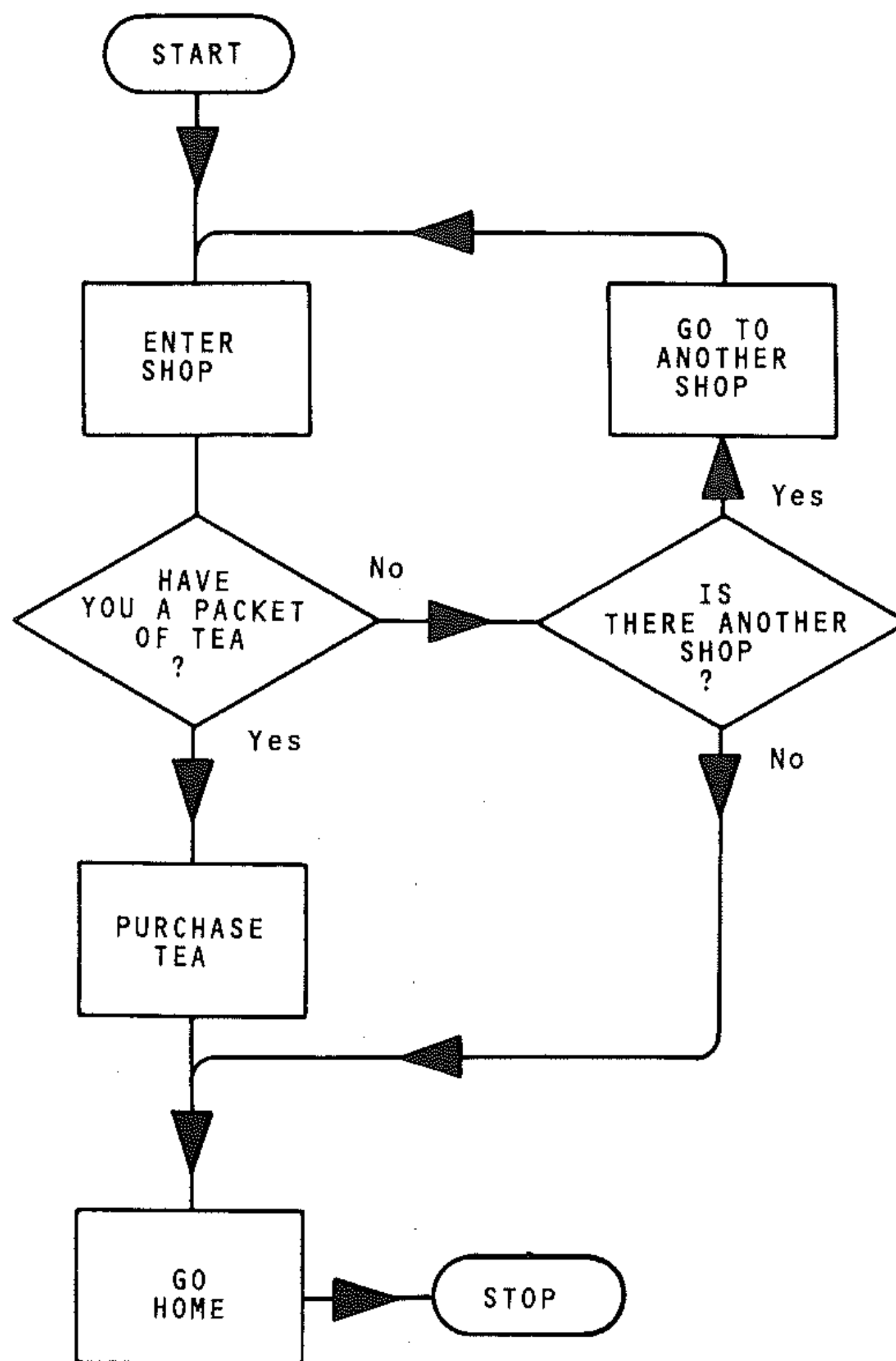
This is like a flowchart of a higher (less specific) level of a tree diagram. Each section could have a more detailed flowchart drawn up to show the individual lines of the program, or comments could be added to the blocks above, relating the program lines to the blocks:



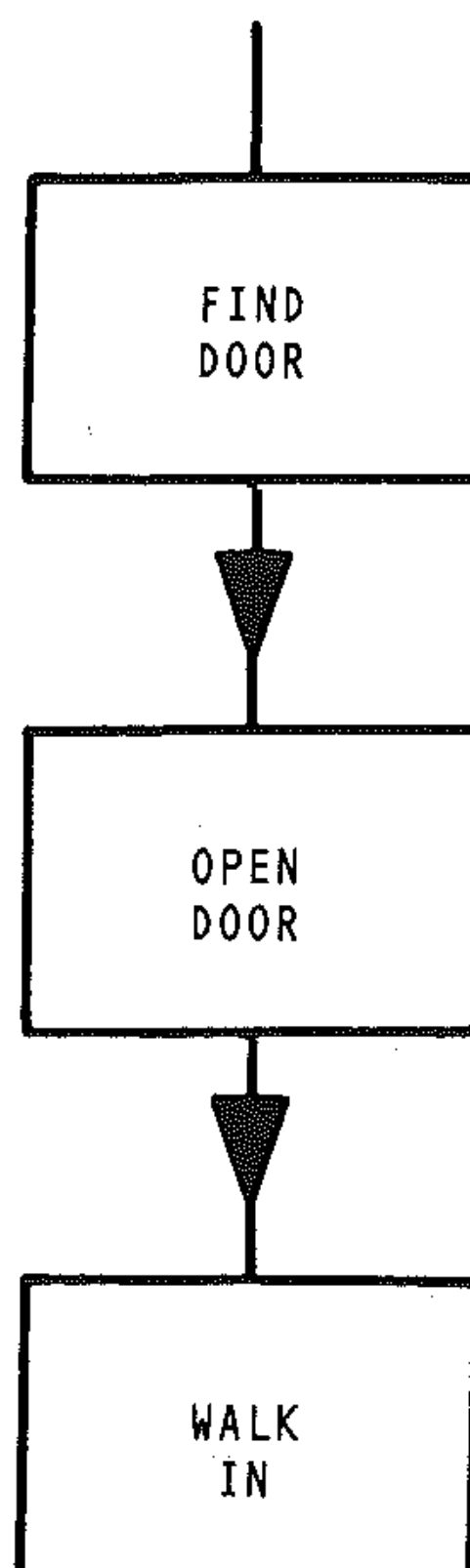
You will soon start to write short programs, and should draw up flowcharts with each program line or instruction indicated separately. Later, for longer programs with large numbers of lines, the flowcharts must be condensed where the sequence is simple to follow in the program, to keep them of manageable size. Any complex manipulations should still be included in full.

Examples

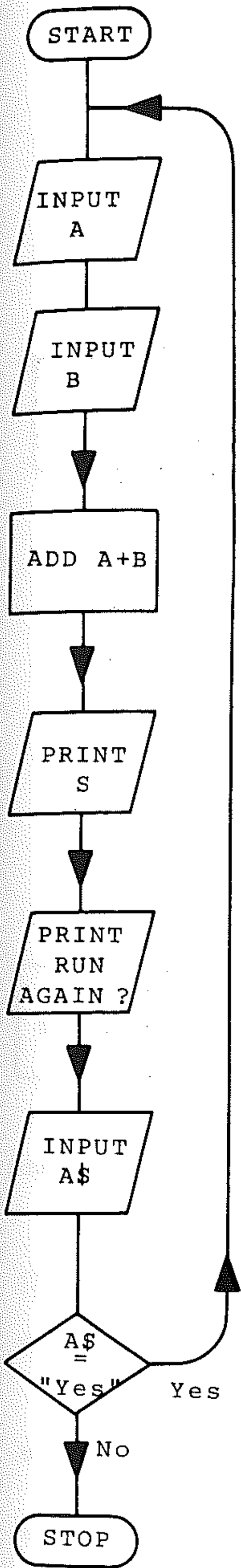
Here is a flowchart for the robot, asking it to buy the tea.



In the same way as the 'making a pot of tea' problem, each of these boxes must be broken down into simpler instructions. On a simple flowchart it may not be possible to see how the problem has been broken down. Either the whole flowchart must be drawn again with more detail, or a new, expanded flowchart drawn at specific points; for example, ENTER SHOP could be replaced with the following:



Here is a flowchart of a program to input two numbers, output the sum, and ask you if you want to run the program again.

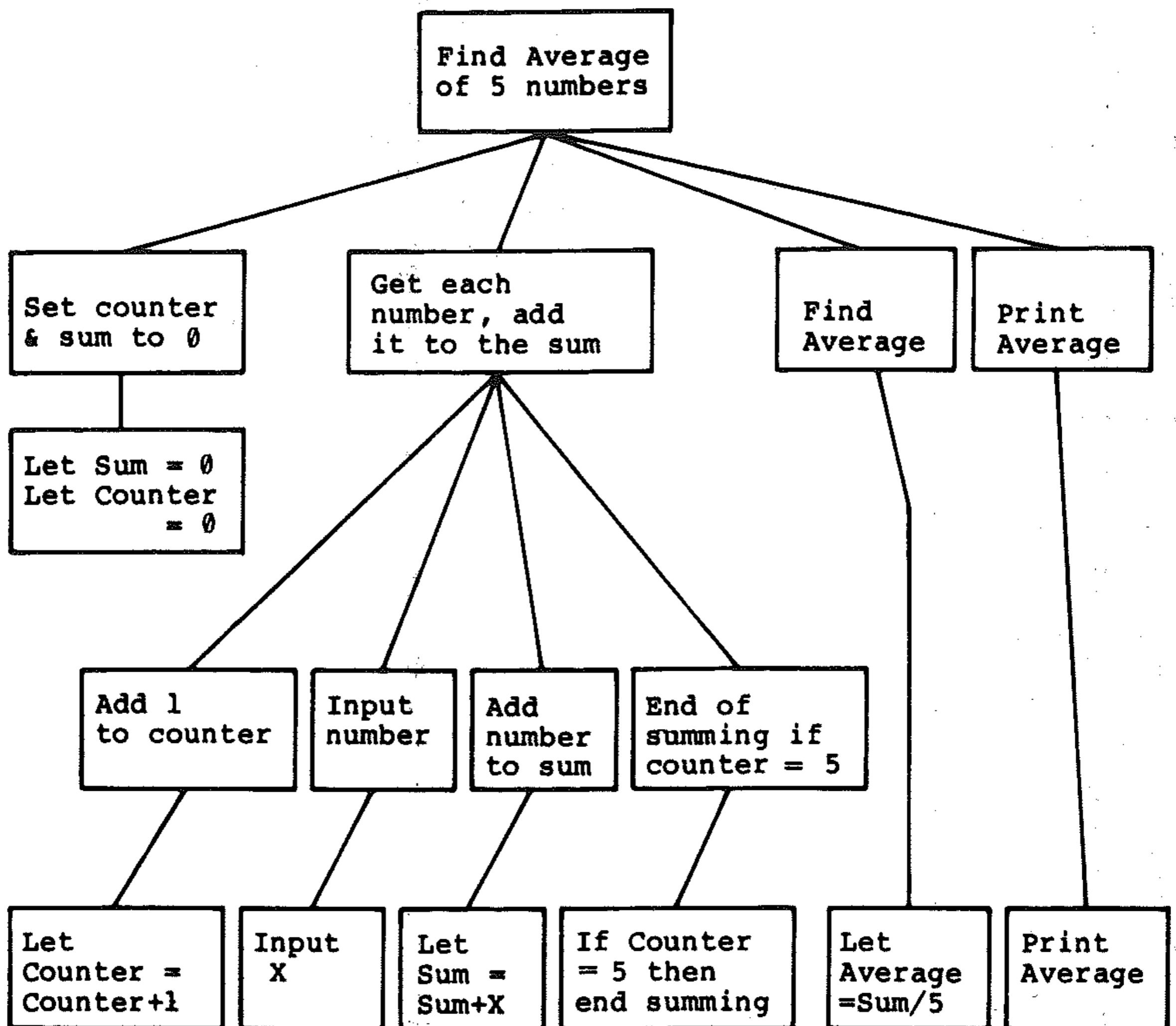


```
10 INPUT A
20 INPUT B
30 LET S = A + B
40 PRINT S
50 PRINT "RUN AGAIN?
(YES/NO)"
60 INPUT A$
70 IF A$ = "YES" THEN
GOTO 10
80 STOP
```

An example of structured design

Problem: find the average of five numbers:

Tree diagram

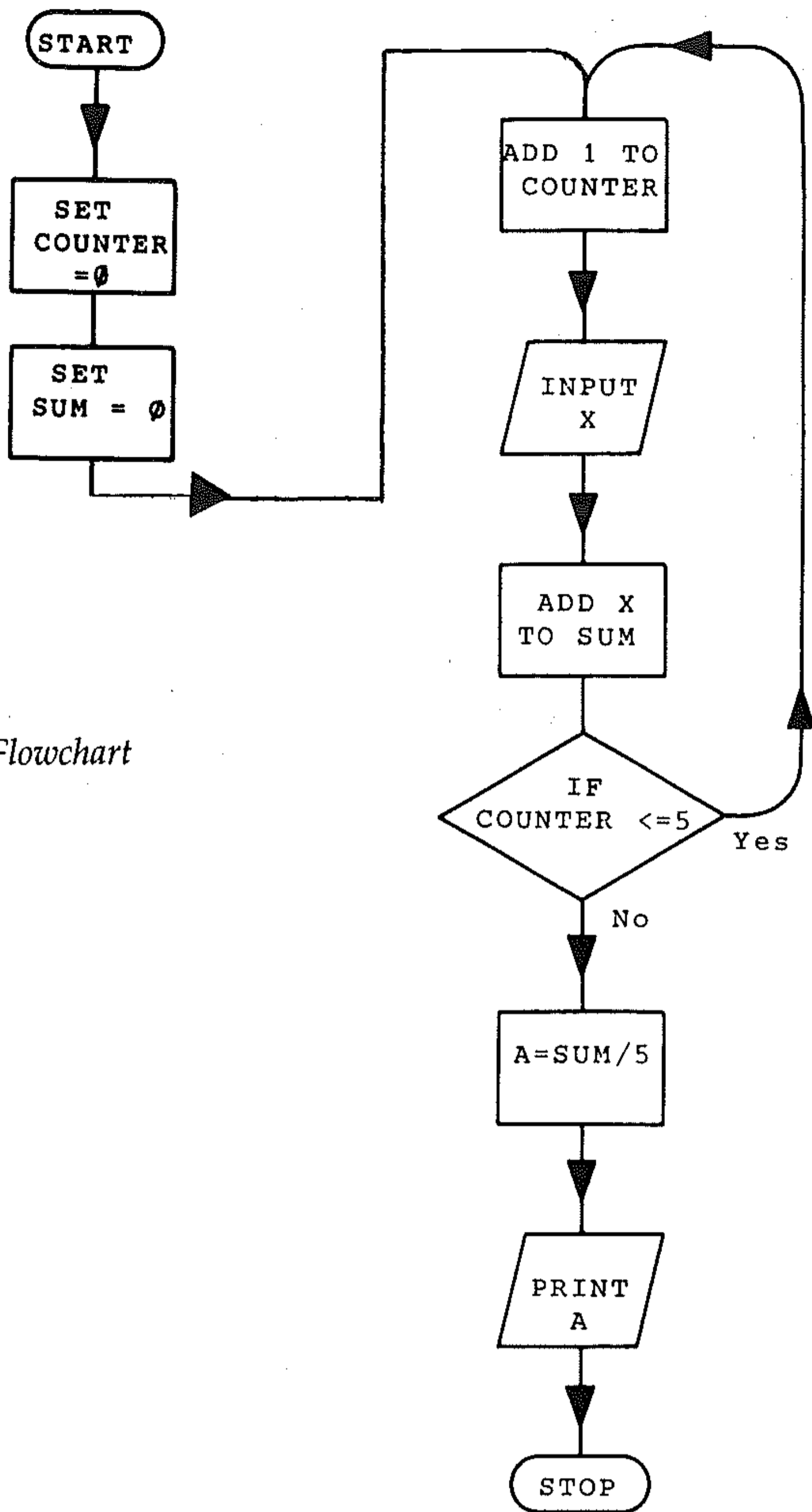


Note that the flowchart and program test whether the counter value is less than 5, using the < symbol.

Program

```
10 REM "AVERAGE"
20 REM * * PROGRAM FINDS AVERAGE OF FIVE
NUMBERS INPUT * *
30 REM * * START * *
40 LET SUM = 0
50 LET COUNTER = 0
60 LET COUNTER = COUNTER + 1
70 INPUT X
80 LET SUM = SUM + X
90 IF COUNTER < 5 THEN GOTO 60
100 LET AVERAGE = SUM/5
110 PRINT AVERAGE
120 REM * * END * *
```

The operand / means *divided by*, and is *equivalent* to the ÷ symbol.



Flowchart

EXERCISES

- Design an algorithm (using tree diagram) and write a BASIC program with a flowchart to find the sum and average of ten numbers to be input at the keyboard.
- Produce the tree diagram, flowchart and program which calculates the area of any rectangle.
- Design the algorithm, BASIC program and flowchart which calculates the total volume and weight of three boxes to be airfreighted from London to New York. Use the following data:

Box	Length cm	Breadth cm	Height cm	Weight Kg
1	20	4	2	2
2	40	3	6	3.5
3	70	10	15	20

Test that it works!

G10: Testing the algorithm

It is always best to make sure your method of solving the problem actually works before coding into BASIC. This pre-coding check is often called a *dry run* or a *walk through*.

Using the data table, the values of all the variables, expressions and counters must be checked through, module by module, through the algorithm. This will uncover errors in the logic and method and will save time when debugging the finished product later on. Professional programmers always do this, because they work to tight time schedules, and by doing things properly at the start they save time later on. Try a few walk-throughs on the simple programs you will be designing at first, just to get the hang of it.

You have now covered the first essential steps in designing a program, and have seen a simple coding process. You have learnt about methods and concepts and introduced some new terminology. After concepts you can go to detail: the algorithm is ready to be coded into a BASIC program. In doing this, you must put into the program the fundamental programming tools, which are language structure and control structures. You have to know what these structures or tools are before you can use them, and this requires a closer look at Commodore BASIC.

Section H: Control

H1: Control in programs

The statements that make up a BASIC program are numbered. BASIC is thus a *line numbered language*. Control in all BASIC programs is carried out by reference to these line or statement numbers. The Commodore 64 will run a program from the lowest numbered statement through to that with the highest number unless instructed to do otherwise. This is exactly what this section is about: that we can control the order in which program statements are executed by using four important instructions in Commodore BASIC:

- GOTO (for direct transfer)
- IF...THEN (for decisions and branching)
- FOR...NEXT (for loops and repetitions)
- GOSUB...RETURN (for accessing program modules, or subroutines)

These instructions are used singly or combined with other instructions to form groups of program statements called *control structures*. There are four principal control structures:

- Decisions and branching
- Loops
- Subroutines
- Nested structures

The most important property of a computer is that it can be programmed to make decisions, by using the relational or conditional operators of BASIC. In this Section you will discover how to take decisions and branch to other parts of the program.

H2: Condition testing

Conditional operators are also called relational operators, as they determine the logical relationship between two expressions, numeric or string. The conditional operators are:

Equality:	=
Inequality:	<>
Greater than:	>
Less than:	<
Greater than or equal to:	>=
Less than or equal to:	<=

Conditional operators are executed in order left to right across a statement, unless they are in brackets.

Notice that three of the operators are complements or opposites of the other three: this is often useful in decision making. The complements are:

Operator	Complement
equality =	inequality <>
greater than >	less than or equal to <=
greater than or equal to >=	less than <

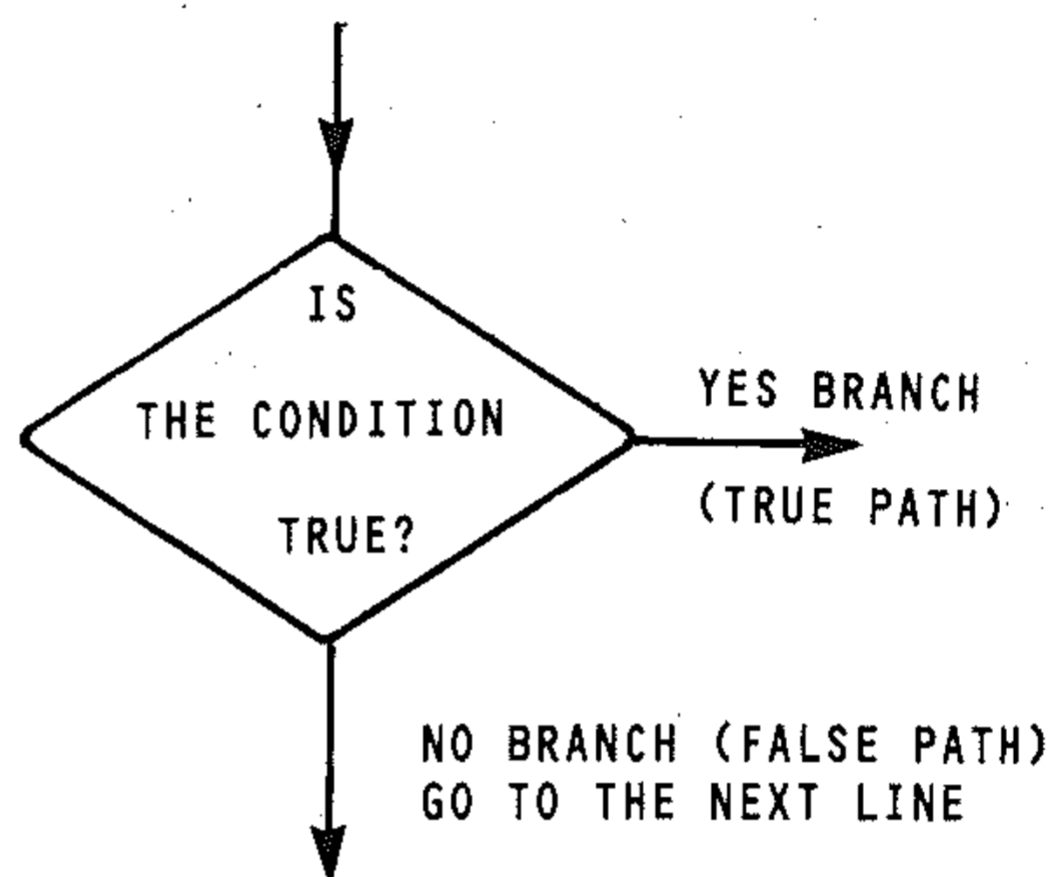
H3: IF...THEN

Conditional operators are used with IF...THEN statements: IF (condition is TRUE) THEN (perform an instruction). For example:

```
40 IF A=B THEN 10
50 IF C <= 6 THEN STOP
60 IF J > K THEN PRINT "J"
```

The format of the statement is: IF (condition) THEN (instruction).

Any BASIC instructions can be used in this kind of statement, although a number of them are unlikely to be useful, such as NEW or CLR. In general, if the condition in the program line is TRUE then the instruction following the condition is obeyed. If the condition is not TRUE (FALSE) then control passes to the next line. This powerful facility enables the programmer to branch and transfer control to another line in the program.



H4: GOTO instructions

The normal control sequence in a program is via numbered statements – from the lowest to the highest. GOTO (line number) switches control to the line number specified:

```
100 GOTO 20
```

As a command GOTO 30 executes a program from line 30. Unlike RUN, with this method variables are not cleared before execution.

EXERCISES

Key in and run this program, which checks that only positive numbers are input and gives BAD DATA ERROR message as well as prompting for the next input. Notice the use of IF...THEN and GOTO. Input both positive and negative numbers.

```
10 REM**INPUT CHECK**
20 INPUT A
30 IF A>0 THEN PRINT A
40 IF A<=0 THEN PRINT"BAD INPUT"
50 INPUT"DO YOU WISH TO RE-ENTER (Y/N)
";A$
70 IF A$="Y" THEN 20
80 STOP
90 REM**END INPUT CHECK**
```


Now try these exercises, which demonstrate the power of GOTO:

```
10 PRINT "CENTURY"  
20 GOTO 10
```

```
10 GOTO 80  
20 PRINT "COMPUTERS ";  
30 GOTO 10  
40 PRINT "PERSONAL ";  
50 GOTO 20  
60 PRINT "COMMODORE ";  
70 GOTO 40  
80 GOTO 60
```

Key them in and study them. The second one is an example of what is called 'spaghetti programming'. Structured programming techniques have been designed to avoid the excessive use of GOTO statements. Now try another one: input some graphics characters and watch the pattern.

```
10 INPUT A$  
20 PRINT A$;  
30 GOTO 10
```

H5: ON GOTO

This instruction can be used to control the flow of a program. Its construction is as follows:

```
ON A GOTO L1,L2,L3,.....,Ln
```

where A is a variable and L1 - Ln are line numbers in program. When A=1 control goes to line L1; when A=2 control goes to line L2; when A=n control goes to line Ln. For example:

```
10 ON A GOTO 10, 20, 30, 40
```

If A=1 control goes to 10; if A=2 control goes to 20... consider what happens if A=5, (or for that matter any number greater than 4). The Commodore will try to find a line number corresponding to A=5. As none exists, your program will ignore the ON GOTO statement, and continue. If the value of the expression is negative then error will result.

Try this:

```
10 INPUT "ENTER A NUMBER BETWEEN 1 AND 3  
";A  
20 REMIF A<=0 OR A>3 THEN 10  
30 ON A GOTO 100,200,300  
40 STOP
```

```
100 PRINT "YOU INPUT 1"  
110 STOP  
200 PRINT "YOU INPUT 2"  
210 STOP  
300 PRINT "YOU INPUT 3"  
310 STOP
```

The ON GOTO must be used carefully. To avoid error resulting, check the value of the variable used before the ON GOTO statement.

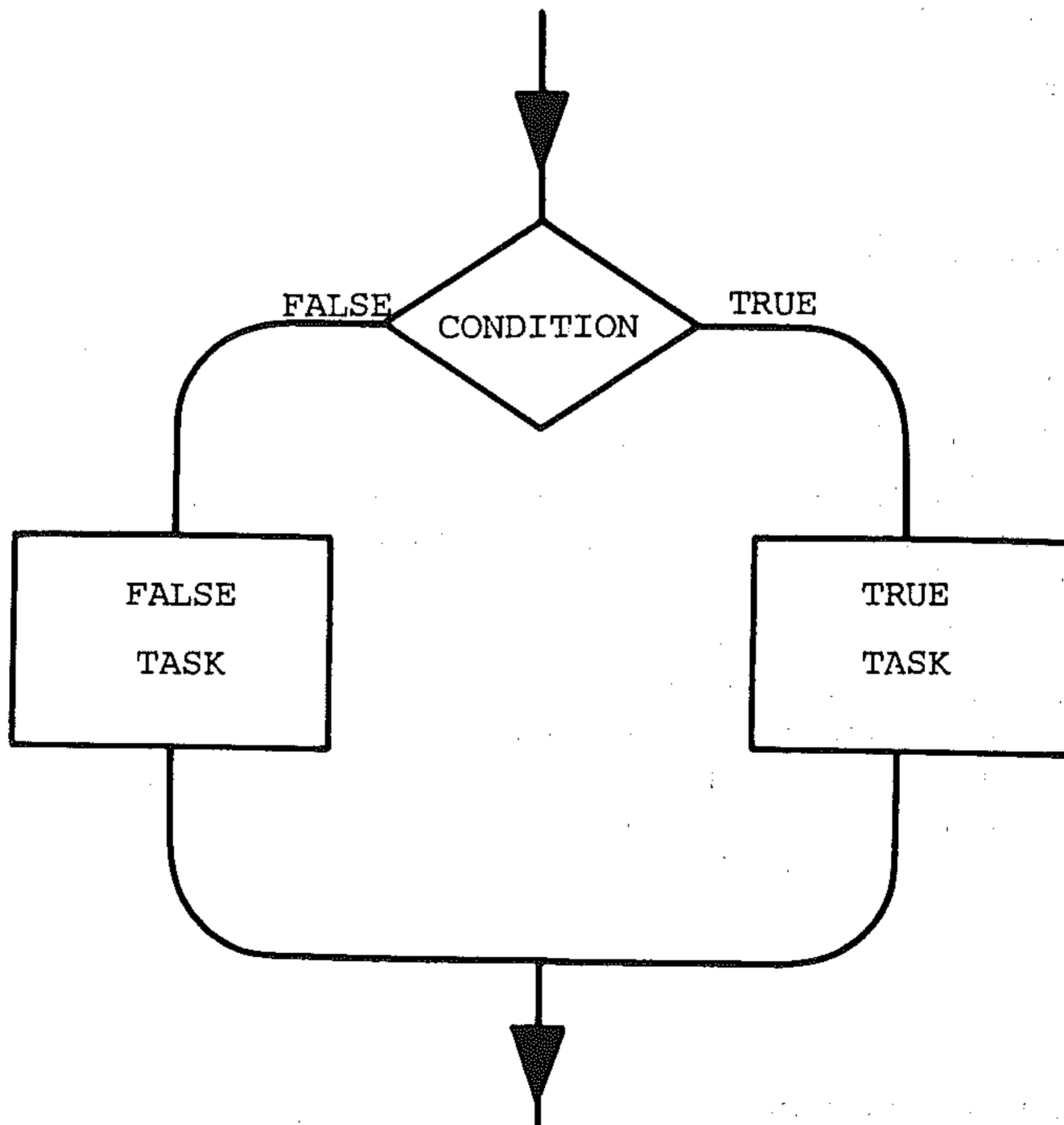
The ON GOTO construction is useful when control needs to be sent to various places in a program (for example, in menu selection – but that's for later in the book).

H6: Decision structures

Double decisions

The simplest decision involves the evaluation of a *logical condition* – that is, a condition that may have the value of true or false. A result of this evaluation decides which part of a program is executed next. These parts of the program are called *true task* and *false task*.

The flowchart for the double decision structure is:



It is called a double decision as there are two alternative modules that can be performed. In the flowchart, if the indicated condition is true, then the program section representing the true task is carried out; otherwise the program section representing the false task is performed. Only one of the paths from the

condition test is taken, and the program will continue at the statement represented by the arrow at the bottom of the flowchart. Each task can be a single instruction, a statement, or a group of instructions.

The double decision structure is known by the general name of the IF...THEN ELSE decision structure. Its general form is: IF (condition) THEN (true) ELSE (false).

This means: IF the condition tested is true THEN perform the true task, and IF the condition is not true perform the false task. The algorithm description of it would look like this:

1. Decision module
 - 1.1 Do the test. If result is true then
 - 1.2 Do true task.
 - 1.3 Otherwise do false task.

This can be written formally in pseudocode as:

```

module - decision
  if condition
  then true task
  else false task
  end if
end module
  
```

End if and end module are bounds to the structure. In Commodore BASIC it is coded as:

```

10 IF A>0 THEN GOTO 100 (or 10 IF A>0 THEN 100)
20 REM *FALSE TASK* (ie. the GOTO is implied)

30 ...
...
90 GOTO 120
100 REM *TRUE TASK*
110 PRINT A
  
```

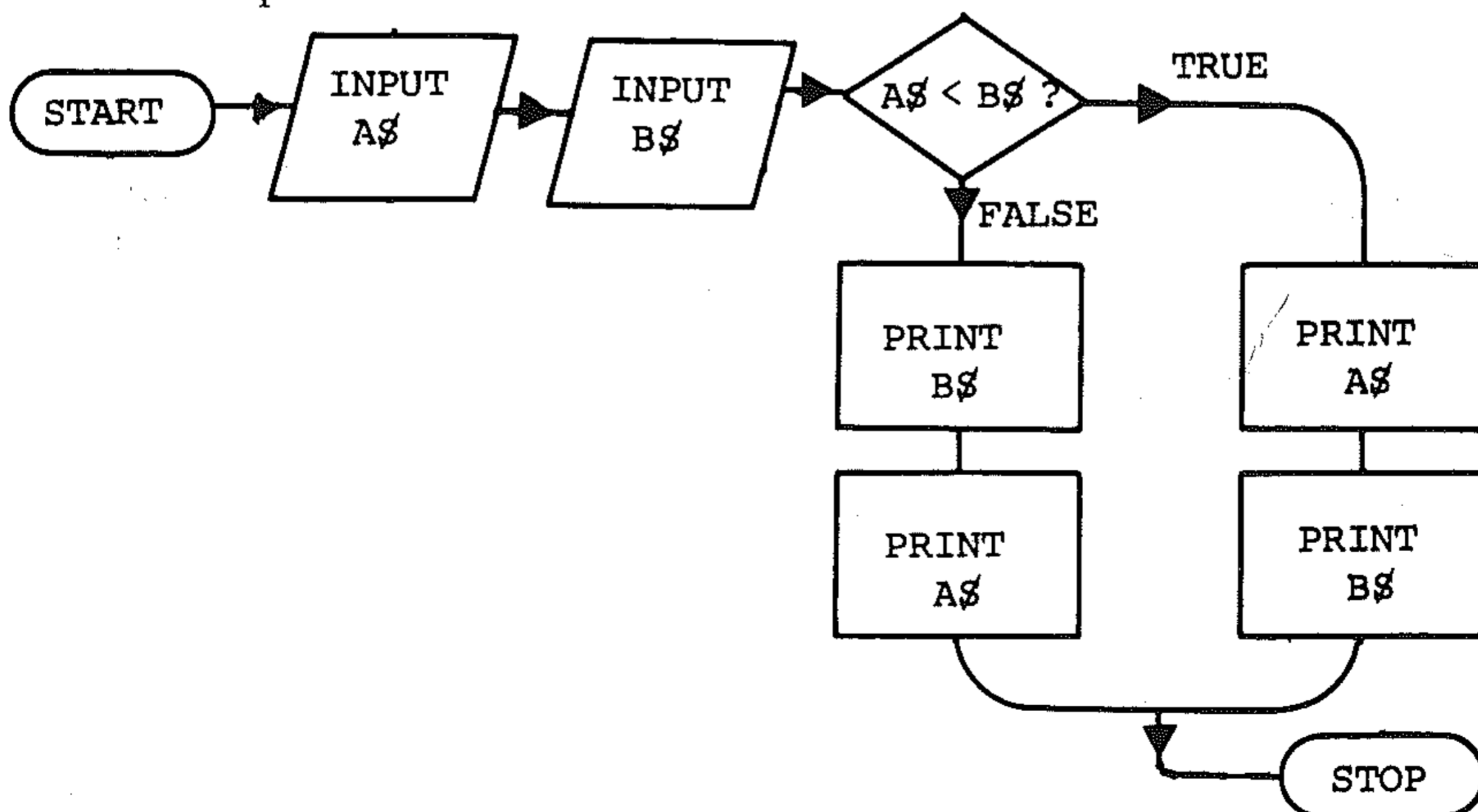
If the program did not branch to the true task starting at 100 and used:

```

10 IF A>0 THEN PRINT A
20 REM FALSE TASK
  
```

in line 10, the true task would be processed and control would then pass to line 20 – the false task. In other words, both tasks would be processed! Watch out for this.

Example: Input two names as strings. The program compares them and prints them out in alphabetical order:



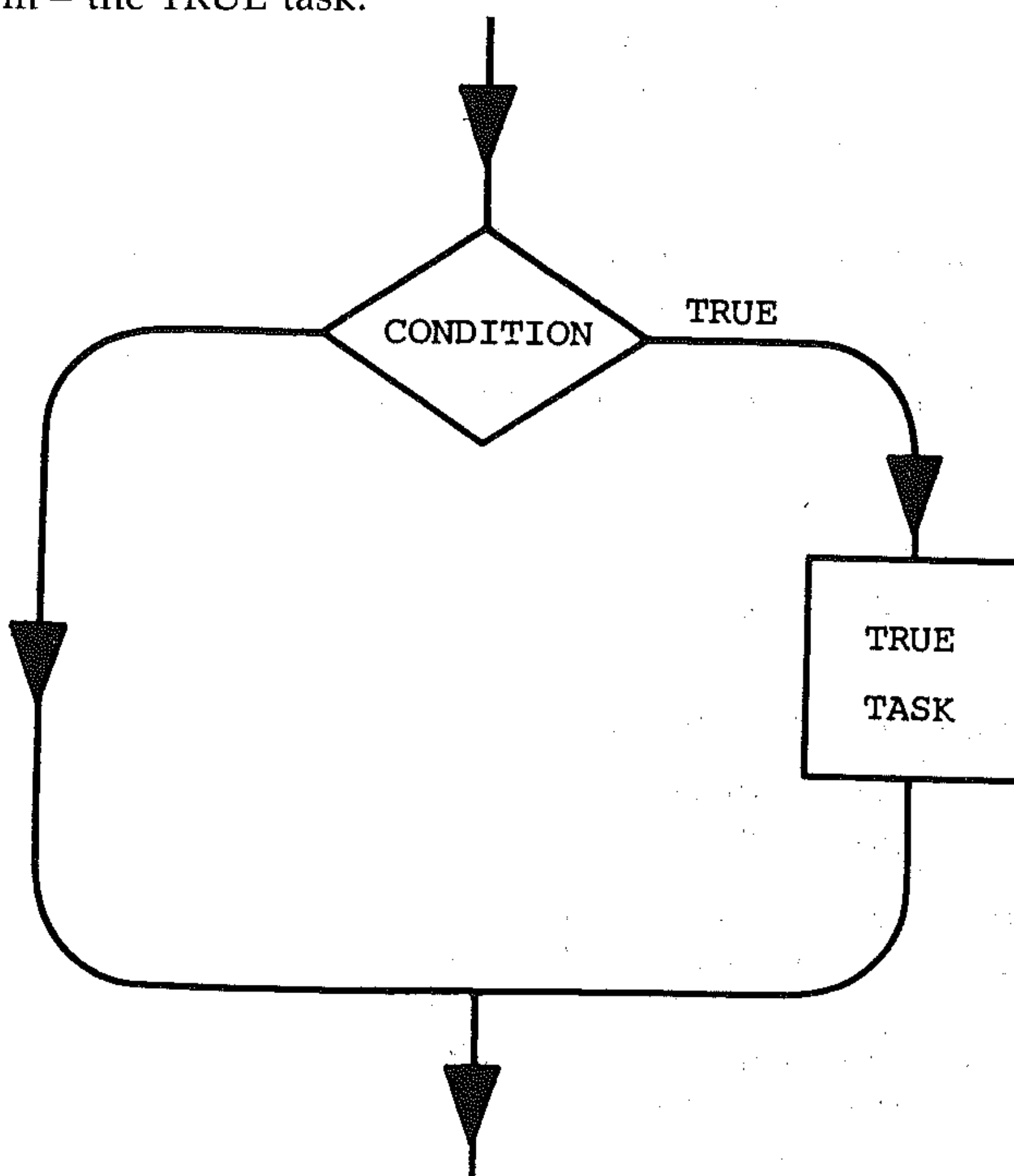
```

10 REM**ALPHA**
20 INPUT A$,B$
30 IF A$ < B$ THEN 70
40 PRINT B$
50 PRINT A$
60 GOTO 90
70 PRINT A$
80 PRINT B$
90 STOP
100 REM** END ALPHA**

```

The single decision

This is a special case of the double decision structure in which there is only one task to perform – the TRUE task.



This is called an IF...THEN decision structure. Its BASIC form is: IF (condition) THEN (true). This means IF (the condition test is true) THEN (perform the true task). The algorithm description would be:

- 1 Decision module
- 1.1 Perform test
- 1.2 If true, process true task

A brief formal pseudocode description is:

```

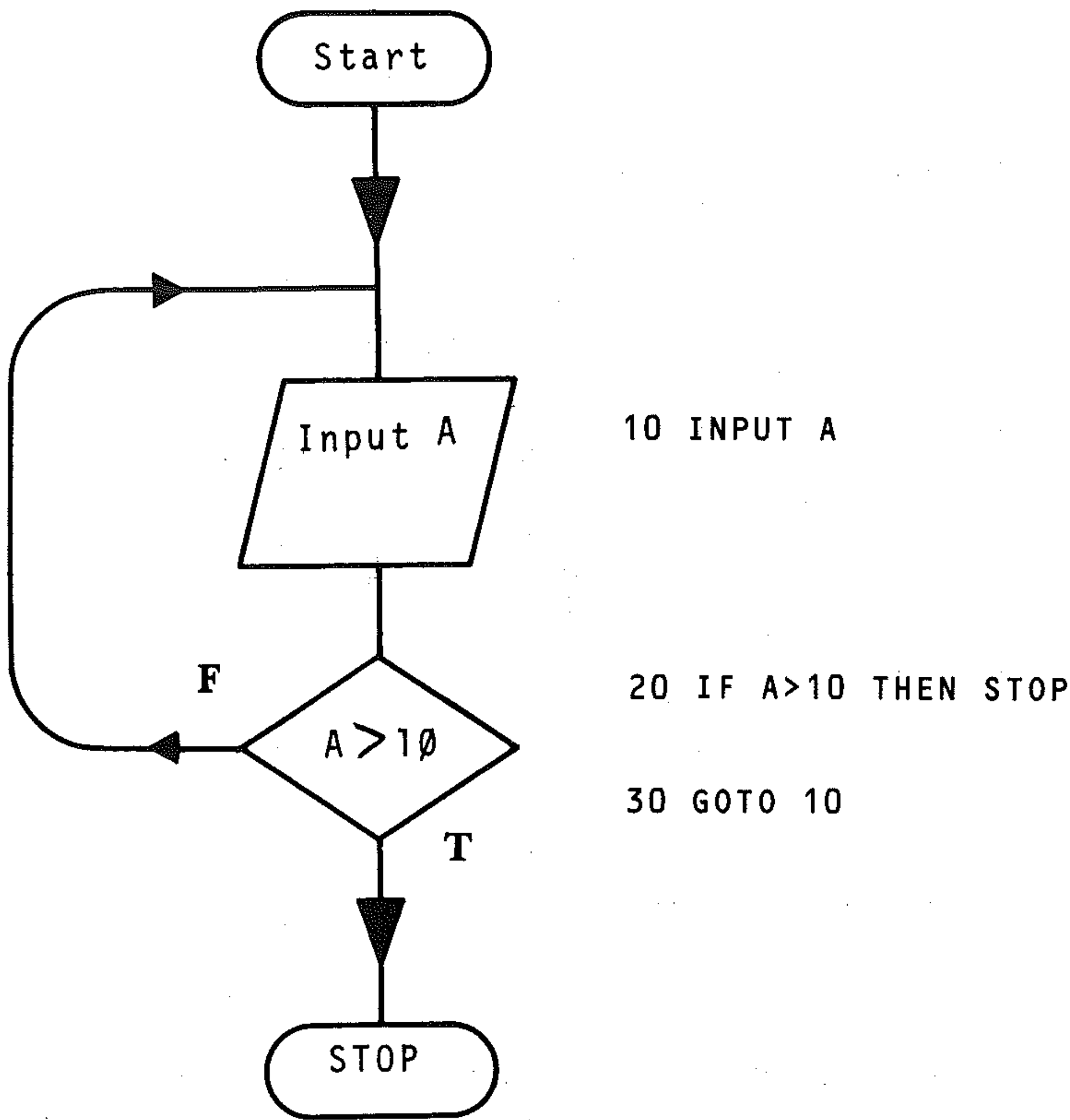
mod - Decision
    if Condition
        Then P
    end if
end mod

```

Our basic statement is:

IF (condition) THEN (TRUE)

Example: Input numbers and stop if a number greater than 10 is input:



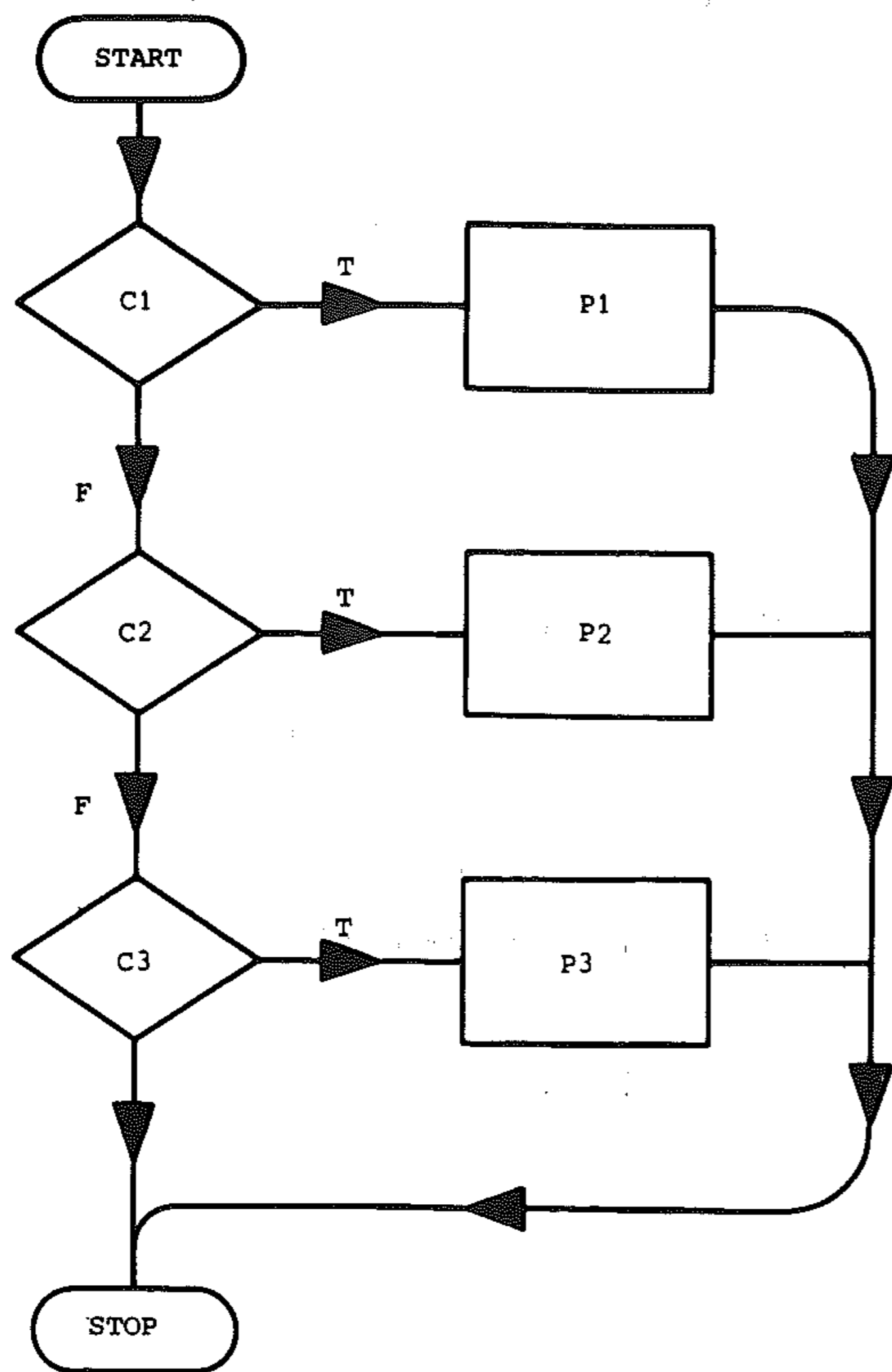
Note the abbreviation of true to T and false to F.

Multiple decisions

There is often the need in programs to perform several tasks based on the result of a set of conditions. To solve these problems a multiple decision structure can be used. This kind of structure is especially useful in breaking up larger tasks into smaller ones. Multiple decisions are most conveniently handled by multiple logical operations. This is covered more fully in Section R, but we will consider the conventional way of handling them.

As an example of multiple decisions, consider a vending machine: you put a coin in and press the appropriate button for the item you want. Another example would be a set of arithmetic testing programs, with questions in each. The computer would ask you which set of tests you required, you would key in the reply and, from several alternatives, the required program would run.

The flowchart for such a structure is:



BASIC

```
10 IF (C1) THEN (P1)
20 IF (C2) THEN (P2)
30 IF (C3) THEN (P3)
```

where C1, C2, C3 are the conditions and P1, P2, P3 are the true tasks.

Example: Input any of three letters A, B, C and print out a corresponding reply.

```
5 PRINT "ENTER A, B OR C"
10 INPUT A$
20 IF A$ = "A" THEN GOTO 60
30 IF A$ = "B" THEN GOTO 80
40 IF A$ = "C" THEN GOTO 100
50 STOP
60 PRINT "YOU INPUT A"
70 STOP
80 PRINT "YOU INPUT B"
90 STOP
100 PRINT "YOU INPUT C"
110 STOP
```

The pseudocode description of this structure is:

```
mod
  case
    if C1
      then P1
    if C2
      then P2
    if C3
      then P3
  endcase
endmode
```

Programming with GOTO

When programming in BASIC, take great care in how you use the GOTO statement. It takes two main forms. Used on its own it is called an unconditional GOTO; when used with IF...THEN it is called a conditional GOTO.

GOTO enables you to jump around in a program like a flea on a blanket – don't do it! Try to code your program to execute in sequence and avoid letting it become a bowl of spaghetti. Excessive use of GOTO makes programs difficult to refine and debug, and relationships between the program paths become difficult to follow. However – do not take the other extreme and write awkward complicated code to try and avoid GOTOs.

Ideally, unconditional GOTO statements should only be used to skip over code and not to repeat code sections (i.e. they should only be used to transfer control forward in a program). Do not put an unconditional GOTO inside a loop or subroutine to jump out of it. Do not jump inside a loop or subroutine, because you'll find that jumping in and out of loops can cause unpredictable results.

Do not jump to another GOTO. For example:

```
100 GOTO 200
200 GOTO 300
```

EXERCISES

- Write a program to input integer numbers and stop if zero is input.
- Write a program to input integers and count the number of times zero is input.
- Write a program to input integers and calculate the percentage of zeros input.
- Write a program which prints out the result of dividing any two numbers' input and gives a **BAD DATA - TRY AGAIN** message if any of the input values is zero.
- Write a program which will print out on request a lunch menu for the different days of the week.

H7: Logical operators: AND/OR

You will be introduced to simple logical operations here, but logic is dealt with more fully in Section J. Use of the AND and OR instructions enable you to combine conditions together to make efficient multiple decision structures in programs.

AND combines relations so that the result of the logical operation (condition 1) AND (condition 2) – for example:

$(A = B) \text{ AND } (B < 1)$

is true when both conditions are true, and false when one or both conditions are false.

OR combines relations so that the result of the logical operation (condition 1) OR (condition 2) – for example:

$(A = B) \text{ OR } (B < > 1)$

is true when *either* condition is true, and false when *both* conditions are false.

The expressions formed by the use of AND and OR are used with IF...THEN statements. For example:

```
20 IF X >= 1 AND X <= 10 THEN PRINT  
  "BETWEEN 1 AND 10"  
50 IF X < > 2 AND X < > 3 THEN PRINT  
  "X NOT EQUAL TO 2 OR 3"  
40 IF A = B OR B = C THEN LET F = F + 1
```

We can also combine more than two conditions:

```
20 IF A = B AND B = C AND C = 20 THEN STOP
```

will stop if all three conditions are true. If one or more is false then the whole expression is false. Similarly:

```
20 IF B = 2 OR B = 3 OR B = 4 THEN LET B = 1
```

will make B = 1 if B is equal to 2 or 3 or 4.

It is possible to use combinations of AND and OR.

```
30 IF (A = B AND B > 2) OR (A = 2 AND B = 3) THEN GOTO 60
```

The expressions in brackets are evaluated first. The first expression in brackets will be true if B is greater than 2 and equal to A. The second expression will be true if A is 2 and B is 3. The program will pass control to line 60 if either expression in brackets is true.

Section I: Arithmetic and Functions

I1: Arithmetic operations

An important function of the computer is to evaluate formulae and expressions similar to those used in standard mathematical calculation. Algebraic *expressions* are written in BASIC using the following *operators*, with a set of variables or numbers as the *operands*.

ARITHMETIC OPERATOR		EXAMPLE	
Symbol name	Priority	BASIC	Maths
↑ exponentiation (raising to a power)	10	$A \uparrow 3$	A^3
- negation	9	$-A$	$-A$
* multiplication	8	$A * B$	$A \times B$ or $a.b$
/ division	8	A / B	$A \div B$ or $\frac{a}{b}$
+ addition	6	$A + B$	$A + B$
- subtraction	6	$A - B$	$A - B$

Note that negation operates on one operand – a *unary* operation (makes a variable negative; for example, minus A) and that the subtraction operator uses two operands – A minus B, a binary operation.

I2: Priority

- All arithmetic, conditional and logical operations are assigned a priority number from 10 to 1. High priority is 10, low priority is 1.
- The priority of an operation determines the order in which it is evaluated in a *complex* statement (in which more than one operation is to be performed). High priority operations are performed earlier.
- Brackets (parentheses) are used in BASIC algebraic expressions. Brackets clarify which expressions constitute separate values to be operated on. Expressions inside brackets are evaluated before the quantity is used in further computation. With multiple (nested) brackets the evaluation proceeds from the innermost bracketed expression to the outermost.
- For operations of equal priority in the same statement, evaluation is from left to right.

Brackets can often be omitted when the sequence of evaluation is understood, but there is never any harm in using them to ensure correct evaluation, or just for clarity. Expressions may be tested by using PRINT as a direct command, for example

```
PRINT 3*4/7↑2
```

or by including them together with test values for variables in short programs which can be edited and re-run:

```
10 A=1:B=2:C=3
20 PRINT (-B+(B↑2-4*9+C)↑0.5)/2*A
```

Examples: study the following, or better still check them on the Commodore using test values.

- Evaluation of $A+B-C$
In BASIC $A + B - C$ or $A+B-C$ operators have equal priority:
 - a) left to right: $A + B$
 - b) left to right: $(A + B) - C$
- Evaluation of $\frac{a \cdot b}{c}$, $(a \times b) \div c$
In BASIC $A*B/C$ or $A * B/C$, $*$ has same priority as $/$
 - a) left to right: $A*B$
 - b) left to right: $(A*B)/C$
- Evaluation of $(b^2-6c)^2 + 5$
In BASIC $(B\uparrow 2 - 6*C)\uparrow 2 + 5$
 - a) Inside bracket: exponentiation $B\uparrow 2$
 - b) Inside bracket: multiplication $6*C$
 - c) Inside bracket: subtraction $(B\uparrow 2)-(6*C)$
 - d) Exponentiation $((B\uparrow 2)-(6*C)\uparrow 2$
 - e) Addition $((B\uparrow 2)-(6*C)\uparrow 2)+5$
- Evaluation of $-70+2 \times 4^2 \times 3 - 3 \times 7$
In BASIC $-70 + 2 * 4\uparrow 2 * 3 - 3 * 7$
 - a) Priority 10: $4\uparrow 2$
 - b) Priority 9: -70 (negation)
 - c) Priority 8, left to right: $2 * 16 * 3$; $3 * 7$
 - d) Priority 6, left to right: $-70 + 96 - 21$
 Result: 5

EXERCISES

- Write the order in which the following BASIC expression is evaluated:
 $-A + ((B\uparrow 3/C) - (A\uparrow 2/D))* (E + F)/G$
- Write down the BASIC expression for:
 - a) $(u^2 + 2as)^{1/2}$
 - b) $ut + 1/2 at^2$
 - c) $\frac{-b + (b^2 - 4ac)^{1/2}}{2a}$
 - d) $(X^a)^{1/6}$

Work out the order in which each of the expressions is evaluated. Test your results on the computer by writing short programs and include test values for the variables. In (c) test what happens when $a=0$ and when $b^2 < 4ac$.

I3: Number

For calculations on the Commodore 64, a 'number' is a positive or negative decimal number whose magnitude is between an approximate minimum of

$$\pm 3 \times 10^{-39}$$

and an approximate maximum of

$$\pm 2 \times 10^{38}$$

Zero is included in this range. The smallest number the computer can handle is

$$2.9387359 \times 10^{-39}$$

The largest the computer can handle is

$$1.7014118 \times 10^{38}$$

The computer stores and calculates numbers internally to an accuracy of nine or ten digits, but prints out the results of calculations to eight significant figures only, rounding where necessary.

I4: The E notation

The E or *exponent* or scientific notation is the notation computers use for input and output of numbers having a large number of decimal digits. E should be taken to read: 'times ten to the power of'. For example, 1.73 E5 is:

$$1.73 \text{ times } 10 \text{ to the power of } 5 = .73 * 10^{\uparrow 5} = 173000$$

Similarly, 3.8 E-7 is:

$$3.8 \text{ times } 10 \text{ to the power of } -7 = 3.8 * 10^{\uparrow -7} = .00000038$$

The computer will accept any number keyed in in this form, and will print out numbers in this notation when their values are outside a certain range. For large positive and negative numbers the E notation is automatically used by the computer for numbers $\geq 10^9$. Numbers up to this figure are first rounded to 8 significant figures; trailing zeros are added until 10^{13} is reached.

Key in and run this program:

```
10 A = 9.9999993E12
20 PRINT A
30 A = A + 1E 5
40 GOTO 20
```

EXERCISES

- Key in and run the following simple program, which illustrates how numbers are printed, the E notation, and the largest number which may be obtained:

```
10 A = 1
20 A = A * 10
30 PRINT A
40 GOTO 20
```

- Change line 10 of the program to each of the following and run the program each time.

```
10 A = 1.00000000
10 A = 1.11111111
10 A = 1.7
10 A = 1.7014118
10 A = 1.71
```

What conclusion do you draw?

- To show that negative numbers behave in the same way, change line 10 to each of the following and run the program.

```
10 A = -1
10 A = -1.7
10 A = -1.7014118
10 A = -1.71
```

- To show how small numbers are handled by the computer, a similar program divides a number (A) by increasing powers of 10. Key this in and run it:

```
10 A = 1
20 A = A / 10
30 PRINT A
40 GOTO 20
```

- Change line 10 to each of the following and re-run the program:

```
10 A = 3
10 A = 2.9
```

Notice that 2.9387359E -39 is the smallest number before zero. Write this number out in full. Can you think of any application for very large and very small numbers?

I5: Rounding

Rounding up

The computer will print out computed values to an accuracy of 9 significant figures, ignoring leading zeros. Digits after the 9th significant one will be rounded up. For example if we key in

```
PRINT 0.1111111111 + 0.8888888888
```

the answer on the screen is 1. Try this:

```
PRINT 0.001
```

Adding two zeros after the decimal point forces the use of the E notation.

Rounding down

The INT function returns the nearest integer of the expression X, which is $\leq X$; that is, it rounds down. For example:

```
INT (3.9)           = 3
INT (-2.8)          = -3
INT (4-8.7 + 0.8) = -4
```

Try printing these functions. Notice that for negative numbers -6 is less than -5, and so on. To round to the nearest integer, add 0.5 to the number first:

```
INT (3.9 + 0.5)    = 4
INT (2.4 + 0.5)    = 2
INT (-1.7 + 0.5)   = -2
INT (-2.3 + 0.5)   = -2
```

Notice this assumes that 0.5 rounds to 1.

```
INT (1.5 + 0.5)    = 2
INT (-1.5 + 0.5)   = -1
```

Entering the zero before the decimal point is optional.

I6: How numbers are handled

All computers perform their arithmetic and processing using the *binary number system*.

In the binary (base two) system only two digits are used, 1 and 0. A group of 8 binary digits (bits) is called a *byte*, such as 10101101. This binary number means

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

and equals $128 + 0 + 32 + 0 + 8 + 4 + 0 + 1 = 173$ in decimal.

Users and programmers, however, communicate with the computer in decimal (base ten) notation. This is rather more convenient than using binary, but it means that conversion from decimal to binary and vice versa is necessary, and occurs inside the machine.

One byte is equivalent to a single character of the computer's character set. A byte represents a number between 0 and 255 (decimal). (This is why the character codes are in this range; a group of 8 zeros and ones can have 256 different states.) A digit or number is represented by one or several bytes, according to its context in the computer.

A character input to the computer or output to the screen or printer is held in one byte. Program line numbers, which are whole numbers 1 to 32999, are held in two bytes.

Numbers are held in a form which occupies five bytes. The point to be noted here is that conversion from decimal to binary and back is involved in the operation of the computer, and this conversion is not always exact. This must be allowed for in certain circumstances, especially where the computer is asked to check whether two numbers are equal. A difference in the binary form of the number, however small, will cause the computer to decide they are not equal. In testing two numbers for equality, therefore, if non-integer values have been utilised, and the value of one number arrived at by calculation, the equivalence check should be replaced by assessing the difference. A statement such as:

IF ABS (A-B) < 1E -4 THEN...

which checks that the difference between the numbers is less than .0001, can be used instead of IF A = B THEN..., if either A or B has been calculated.

The 64 stores real numbers (decimal) in memory in the form

$$\pm M * 2^e$$

where M is called the *mantissa* and $0.5 < M < 1$ (It can never be one) and e is the *exponent*, where $1 \leq e \leq 127$.

When you key in a number in a program, the 64 converts it into the form above, and stores it as a sequence of binary numbers which take up 5 bytes. For example, if you enter 4.2:

$$4.2 = 4.2 \times 10^0 = 4.2 \times 2^0 = 2.1 \times 2^1 = 1.05 \times 2^2$$

which is in the form $\pm M * 2^e$ so $M = .525$ $e = 3$

The 64 then does a decimal-to-binary conversion and stores e in the first byte and M in the next four. It is evident from this representation that the maximum number which can be stored is

$$0.99999999 * 2^{127} = 1.7014118 * 10^{38}$$

and the smallest is

$$0.5 * 2^{-127} = 2.9387359 * 10^{-38}$$

The forms in which numbers are held in the computer are considered in more detail in Section U: The computer memory.

I7: Functions

A *function* is defined as $Y = F(X)$, or

'Y equals some function of X, F(X)'

A function is the mathematical relationship between two variables X and Y such that for each value of X there is a unique value of Y. In other words, it is a mathematical operation which gives a number.

Y takes the function value and is the *dependent variable*. X is the *argument* and is called the *independent variable*.

F is the function name – square root, sine, natural logarithm, etc. In a program statement we write, for example:

100 Y = SQR(X)

The argument X can be a variable, a number or an expression. For example:

```
100 Y=SQR(9)
```

```
100 Y=SQR(B↑2 -4*A*C)
```

The function – a mathematical operation that gives a number – is treated in BASIC as a numeric expression, with priority 11.

The standard mathematical and trigonometric functions are important time-savers for programmers. They are the same as the function keys on scientific calculators. (Other *functions* control or monitor the handling of data by BASIC, rather than performing mathematics.) If the mathematical functions were not available in BASIC, a programmer would need to write separate programs to undertake their tasks every time they needed to be used!

I8: List of functions used in Commodore BASIC

In this list of functions X is the argument: a variable, a number or an expression. The first three sets of functions will be discussed in more detail later in this section. Logic functions will be investigated in Section J, character and string functions in Section K, printing functions in Section M and special functions in Section U.

Standard mathematical functions

- ABS(X) gives the absolute value of X
- EXP(X) gives e^x , value of e raised to the power of X
- INT(X) gives the largest integer $\leq X$ (rounds down)
- LOG(X) gives natural logarithm (value of $\log_e X$)
- SQR(X) gives the value of \sqrt{X} or $X^{1/2}$ (X positive)
- SGN(X) gives sign of X (whether X is negative, positive or zero).

Trigonometric functions

- SIN(X) gives the value of sine X (X in radians)
- COS(X) gives the value of cosine X (X in radians)
- TAN(X) gives the value of tangent X (X in radians)
- ATN(X) gives the angle in radians whose tangent is X, arctangent X

Special mathematical functions

- RND(A) random number generating function; gives the next pseudo-random number N from a fixed series of random numbers ($0 \leq N < 1$). Variable A starts the series. If A is greater than 0, a new random number is returned. If $A = 1$ then the same number is returned. To start to generate random numbers from the same place, use RND (-1).
- FN calls up a previously defined user function specified by a variable. Variables must be enclosed in brackets. For example: FNA(X), FNB(Y)

Logical functions

- AND logical operation between two operands. The left operand is a number. The right is a number or a condition (result of another logical operation). The result is a Boolean data type and is TRUE or FALSE. This is output by the Commodore as the left operand or -1

indicating true, or a 0 indicating false. Operations are performed in binary in the Commodore and AND has a priority of 3.

OR a logical operation between 2 numbers. OR has priority two.

NOT a logical function with a single numeric operand.

NOT A = -1 if A = 0 - false

NOT A = 0 if A = -1 - true

The numeric operands are considered as FALSE if 0 and TRUE if any other number.

Character and string functions

CHR\$(X) $0 \leq X \leq 255$ returns the single character whose code is X.

ASC(A\$) when applied to the string A\$, this function returns the code of the first character in the string or 0 if the string is empty (null string).

LEN(A\$) returns the number of characters in the string.

VAL(A\$) turns a string in number representation into the number for calculation: A\$="12.4", VAL(A\$)=12.4

STR\$(N) turns the number in N into the string form.

Printing functions

POS(A) returns in A the next position on the screen where printing will occur. For example:

```
10 PRINT 1234;
```

```
20 B = POS(A)
```

```
30 PRINT B
```

Try this example and check it. B should return 6. This is because 1234 requires five print positions (including a trailing space) and thus the next printing position is 6.

SPC(A) where A is a variable; tells the computer how many spaces to print. For example:

```
10 PRINT 1234;SPC(4);1234
```

will print

```
1234      1234
```

TAB(X) places the print position in column X. If $X > 32$ then column number is the remainder when X is divided by 32. If it involves back spacing, goes on to next line. Rounds X to nearest integer.

Special functions

FRE(N) returns the number of free bytes available, regardless of what value N has.

PEEK(X) $0 \leq X \leq 65535$ returns the value of the byte at address of X in RAM or ROM memory.

SYS(X) as in USR, but no parameter is passed between BASIC and machine code routine. X is the starting address of the routine.

USR(A) transfers control from BASIC to machine language routine whose address is given by the contents of memory locations 784 and 785 (user function jump). The parameter A is passed to the machine language program and returns a value back to BASIC.

TI & TIS return the contents of the timer which is updated every 1/60 of a second.

I9: The standard mathematical functions

ABS(X) returns the *absolute* or value or modulus of X, which may be a number, a variable or an expression. It returns the *positive* value of X. For example:

```
10 PRINT ABS(-3.7) gives 3.7
10 PRINT ABS(4) gives 4
```

EXERCISE

Key in and run this program:

```
10 INPUT A,B
20 PRINT TAB(3);A;TAB(10);B
30 PRINT TAB(3);ABS(A-B)
40 GOTO 10
```

Input positive and negative values for A and B.

Now change line 30 to:

```
30 PRINT TAB(3);ABS(A*B)
```

and input some more values. Try replacing the * with ↑ or using ABS(SQR(A))

EXP(X) gives the value of the constant e raised to the power of the value of X where X is a number or an expression. $e = 2.7183$

For example:

```
10 PRINT EXP(3.4) or
10 PRINT 2.7183↑3.4
```

The function EXP is the inverse of LOG.

EXERCISES

- Using log tables, write a program to check the values of e^x given in the log tables.
- Write a program which will calculate Q from the expression:

$$Q = Q_0 \cdot e^{-t/rc} \quad (\text{In BASIC } Q = Q_0 * \text{EXP}(-T/R * C))$$

If you know anything about electricity, you might recognise this expression.

- Key in this program. It calculates a value for e from the formula:

$$e = (1 + 1/N)^N$$

where N is very large.

```
10 REM**VALUE OF E**
20 I=1
30 N=10↑I
40 E=(1+1/N)↑N
50 PRINT TAB(1);N;TAB(12);E
60 I=I+1
70 IF I<5 THEN 30
```


LOG(X) gives the value of the natural logarithm:

$$\text{LOG}(X) = \text{Log}_e(x)$$

Note that $\log_{10}(X)$ (common logarithm) = $\text{LOG}(x)/\text{LOG}(10)$

The LOG function is the inverse of EXP, so if $\text{EXP}(X) = y$ then $(X) = \text{LOG}(Y)$. LOG(Y) is the natural logarithm of Y. The antilog is $\text{EXP}(\text{LOG}(Y))$. The normal log operations can be used if appropriate, as with common logs. For example $\text{EXP}(\text{LOG}(X) + \text{LOG}(Y))$ gives the product of X and Y.

EXERCISE

```
10 Y=1
20 PRINT TAB(3);Y;
   TAB(10);EXP(LOG(Y))
30 Y=Y+1
40 GOTO 20
```

SQR(X) returns the square root of (X), (X) or $X^{0.5}$. For example:

```
PRINT SQR(9)           gives 3
PRINT SQR(23)          gives 4.7958315
PRINT SQR(19 + 17)     gives 6
```

SGN(X) returns + 1 if (X) is positive, 0 if (X) is zero, -1 if (X) is negative.

SGN is short for Sign or Signum (Signum doesn't sound like Sine). For example:

```
PRINT SGN(23)           gives 1
PRINT SGN(-5)           gives -1
PRINT SGN(3-3)          gives 0
PRINT SGN(1)            gives 1
PRINT SGN(25-(2*23))    gives -1
```

PI appears on the keyboard and screen as π . It is a function which has no argument. It returns the value of π as 3.14159265.

Try these:

```
PRINT 3- $\pi$ 
PRINT 45* $\pi$ /180
```

I10: Trigonometric functions

SIN, COS, TAN

SIN(X), COS(X), TAN(X) give the value of the sine, cosine, and tangent of the number or expression X, which is an angular measure. X must be in radians.

Angles are normally expressed in degrees:

$$1 \text{ degree} = \pi/180 \text{ RADIANS } (1^\circ = \pi/180 \text{ radians})$$

To convert degrees to radians multiply by $\pi/180$. For example, if Y is our measure of angle in degrees then:

```
SIN(Y* $\pi$ /180)
```

gives the correct value of Sine Y.

EXERCISES

- Generate a table of values for SIN (X), COS (X) and TAN (X) for every 20 degrees in the range 0 – 360 degrees.
- Write a program to verify the trigonometric formula:

$$\begin{aligned}\text{SIN}^2(X) + \text{COS}^2(X) &= 1 \\ 1 + \text{TAN}^2(X) &= \text{SEC}^2(X)\end{aligned}$$

- Write a program to calculate the area of a triangle from a knowledge of the length of three sides and an angle.

ATN(X) gives the arc tangent of (X). The returned value is the angle in radians for which the tangent would be given by the value of (X). To get the angle in degrees multiply by $180/\pi$; that is, $Y=180/\pi*\text{ATN}$ gives arc TAN(X) in degrees.

I11: Random numbers

Random number generators are useful for games and simulation in statistics. The numbers generated are part of a very long sequence of numbers (there are 65536 of them) and are in fact only *pseudo-random*, but good enough for most purposes.

RND gives a random number greater or equal to zero but less than one.

```
10 A = RND(B)
```

where B is any positive or negative number. If a negative number is used, the value returned will always be the same for the same negative number.

```
10 A=-1
20 PRINT RND(A)
30 A=A-1
40 GOTO 20
```

If you key in `PRINT RND(1)` you get a number like 0.112519041 or 0.437156825 – for `RND(1)` always between 0 and 1 – which is not much use in that form. You need to be able to generate random numbers within a useful range according to a specific purpose.

To obtain a random number 0 – 9

To obtain a random number 0 – 9, multiply the function by 10 and take the integer value.

```
PRINT INT(RND(1)*10)
```

`RND(1)*10` gives random numbers between 0.000000000 and 9.999999999. `INT` will round these values down to integers 0 to 9.

Numbers 1 – 10

Although 0 to 9 gives ten values, the range 1 to 10 would be more useful. This is obtained by adding one to the `RND` function:

```
PRINT INT(RND(1)*10+1)
```

Suppose you wanted random numbers generated for simulating a dice roll. You would use:

```
PRINT INT(RND(1)*6+1)
```

Random numbers for a card game

There are 4 suits, with 13 cards per suit = 52 cards. So if you used:

```
PRINT INT(RND(1)*52+1)
```

you could select cards at random. Think about how you could identify the suits and not deal the same card twice.

A final test simulates the tossing of a coin. There are two choices, heads or tails. Taking the integer value of $RND(1)*2$ will generate 0 or 1.

Run the program, which keeps a continuous count of 1000 throws.

```
10 PRINT "*****THROWS*****HEADS*****  
*****TAILS*****"  
20 H=T=0  
30 FOR N= 1 TO 1000  
40 S=INT(RND(1)*2)  
50 IF S=1 THEN H=H+1  
60 IF S=0 THEN T=T+1  
70 PRINT " "; N; " "; H; " "; T  
80 FOR M=1 TO 200:NEXT M  
90 NEXT N
```

I12: User-defined functions

Although Commodore BASIC is rich in functions, there is always a requirement to define your own. This facility saves programming time and improves program structure and compactness. Very complex functions can be defined and called with a single letter name. The function is established using the DEFine instruction. FN is printed by the Commodore automatically.

```
DEF FNvariable(argument) = expression
```

The naming variable is any legal variable. The argument is called a dummy variable. The expression can contain variables other than those specified in the argument. The function is called using

```
FNvariable(argument)
```

To illustrate, run this program.

```
5 LET Y=2  
10 DEF FN A(X)=X+Y  
20 PRINT FN A(1)
```

Here the function is called A. It is a function of variable X and is the sum of X and Y. Various values can be assigned to X as in 20, or assign it previously.

Use DEF FN to define complex functions, as seen by obtaining values of $SINH(X)$ in this program.

```

10 DEF FN Y(X)=(EXP(PI*X180)-EXP(PI*(X-X-X
)/180))/2
20 FOR X=-360 TO 360 STEP 45
30 PRINT X, FN Y(X)
40 NEXT X

```

Check the table generated with your book of tables.

TI, TI\$ The timer function functions TI and TI\$ relate to the real time clock and return the contents of the timer. This value is updated at every 1/60 of a second. To reset the timer, use:

```
TI$="000000"
```

The first two digits represent the hour, the next two minutes and the third two digits represent seconds. For example:

```
TI$="013000"
```

will set the timer to 1 hour and 30 minutes.

Examples:

```

10 TI$="000000"
20 PRINT TI$
30 GOTO 20

```

```

10 TI$="000000"
20 PRINT TI/60
30 GOTO 20

```

Section J: Logical Operations

J1: Logic values and numeric values

When using the logical capability of Commodore BASIC, you must distinguish between logical values and the numeric values produced by logical evaluation. *Logical value* is the value of an expression using the criteria:

- any non-zero value of the expression = TRUE
- a zero value of the expression = FALSE

When an expression is logically evaluated, it is assigned one of two numeric values: true = -1, false = 0.

J2: Boolean operators: the AND operator

The AND operator forms a logical conjunction between two expressions involving conditional operators:

- If both expressions are TRUE the conjunction is TRUE
- If one or both are FALSE the conjunction is FALSE
- The numeric value of TRUE is -1
- The numeric value of FALSE is 0
- All non-zero values are TRUE

For example:

```
100 IF (A = 10) AND (B<>3) THEN 60
200 PRINT (A AND B)
```

In line 100, if the relation $A = 10$ is TRUE and the relation $B \neq 3$ is TRUE then control will pass to line 60. If either or both of the relations is FALSE then control passes to the next line. In line 200 the computer will not print $A + B$; it will print the value A and B ANDed bitwise. So if

$A = 15$ and $B = 6$

The relation $(A \text{ AND } B)$ will print 6. Since A is 00001111 in binary and B is 00000110, $A \text{ AND } B$ will give

```
00001111 = A
00000110 = B
-----
00000110 = 6
```

Truth table for AND

A	B	A AND B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

A and B are conditional expressions.

J3: The OR operator

The Boolean operator OR forms the logical disjunction of two expressions involving conditional operations:

- If either or both of the expressions is TRUE the OR disjunction is TRUE
- If both expressions are FALSE the OR disjunction is FALSE

For example:

```
100 IF (A>1) OR (B = 0) THEN STOP
200 PRINT (C OR D)
```

In line 100 if either of the expressions (A>1) and (B = 0) are TRUE the program will stop. If both are FALSE control passes to the next line. In line 200 if C=10 and D=1, C OR D will give 11 as follows:

```
00001010 (10 in binary)
00000001 (1 in binary)
-----
00001011 = 11
```

Truth table for OR

A	B	A OR B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

A and B are conditional expressions.

J4: The NOT operator

NOT logically evaluates the complement (reverse) of a given expression. For example:

```
30 IF NOT (A = B) THEN STOP
100 PRINT NOT A
```

In line 30, if (A=B) is FALSE, then NOT (A=B) = NOT FALSE = TRUE, and the program stops. In line 100, if A=0 = FALSE, the NOT A = TRUE = -1. So -1 is printed. If A=29 then NOT A = -30 is printed.

```
29 = 00011101 in binary
NOT 29 = 11100010 +
        00000001
-----
        11100011 = -30
```

Truth table for NOT

A	NOT A
TRUE	FALSE
FALSE	TRUE

A is a conditional expression.

J5: Conditional operators

There are two ways to use conditional operators in logical evaluations.

To check the numeric value of an expression

```
100 IF A = 3 THEN 60
200 IF B<>C THEN STOP
```

The numeric value produced by the logical operation is not important. The computer is concerned only with the truth or falsity of the condition indicated in the IF...THEN statement, which determines whether the instruction is executed. When the specified condition is present (TRUE), the statement after THEN will be carried out.

Checking an expression

In this case the numeric values are required, where TRUE = -1 and FALSE = 0.

```
200 PRINT A < B
300 PRINT A = 3
```

The PRINT statement used as above will give the numeric values produced by logical evaluation. Line 200 is evaluated as a logical expression, so that if it is TRUE that A is less than B, -1 will be printed, and if A is equal to or bigger than B, the expression is false and 0 will be printed.

Line 300 is interpreted by the BASIC as: 'Print -1 if A=3 and 0 if A does not equal 3'. The numeric value of the logical evaluation is distinct from the logical value of the expression.

J6: Logic operations on conditional expressions

IF (condition) AND (condition) THEN...

or

IF NOT (condition) THEN...

The effect of the logical operators AND, OR and NOT on conditions which are TRUE or FALSE gives a result which is TRUE or FALSE and on which the IF...THEN instruction acts accordingly.

For example:

```
100 IF (A>10) AND (B = 0) THEN 20
200 IF (A = 0) OR (B =0) THEN STOP
300 IF NOT (A = B) THEN PRINT "A<>B"
```

These all mean: IF (combined result is TRUE) THEN (do it); IF (combined result is FALSE) go to the next line of the program. Using logical operations is a way of combining conditional operators in a statement. If (condition1) AND (condition 2) AND (condition 3) evaluates as TRUE or FALSE THEN...act accordingly. For example (note brackets):

```
60 IF ((A>B) AND (C>A) AND (D>C)) THEN STOP
AND
```

IF (condition 1)	AND (condition 2)	THEN (result)
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

For example:

- Condition 1 - My age is > 12 years
- Condition 2 - My age is < 20 years
- Result - I am a teenager.

IF (my age > 12 years) AND (my age < 20 years) THEN (I am a teenager). Run the program:

```
10 INPUT " INPUT AGE"; A
20 IF A>12 AND A<20 THEN 50
30 PRINT "YOU ARE A TEENAGER"
40 STOP
50 PRINT "YOU ARE NOT A TEENAGER"
```

OR

IF (condition 1)	OR (condition 2)	THEN (result)
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

For example:

- Condition 1 - I earn wages
- Condition 2 - I get pocket money
- Result - I have money

IF (I earn wages) OR (I get pocket money) THEN (I have money). Here is a program.

```
10 PRINT "INPUT AMOUNT OF WAGES AND POCCKET MONEY YOU GET"
20 INPUT W,P
30 IF W>0 OR P>0 THEN 60
40 PRINT "YOU HAVE NO MONEY"
50 STOP
60 PRINT "YOU HAVE"; W+P; "POUNDS"
```

NOT

IF (NOT (condition))	THEN (result)
TRUE	FALSE
FALSE	TRUE

For example:

- Condition - I have money
- Result - I do not have money

IF (NOT (I have money)) THEN (I don't have money). Here's a program.

```
5 PRINT "ARE YOU A LIAR"
10 INPUT "AMOUNT OF MONEY"; M
20 IF NOT M>0 THEN 50
30 PRINT "APPARENTLY YOU DO NOT HAVE MONEY"
40 STOP
50 PRINT "YOU DO HAVE MONEY REALLY ?"
```


J7: Multiple logic on conditions

Multiple logical operations on conditions are often useful. They take the form:

```
IF ((C1) AND (C2)) AND ((C3) AND (C4)) THEN...
```

C1 = Condition 1

C2 = Condition 2

C3 = Condition 3

C4 = Condition 4

```
IF ((C1 AND C2) OR (C3 AND C4)) THEN...
```

The above statement means that IF conditions 1 AND 2 are obeyed, OR conditions 3 AND 4 are obeyed, then the combined expression is TRUE, and the instructions will be executed; that is, either pair of conditions being both TRUE will give the result.

```
IF ((C1 AND C2) AND (C3 AND C4)) THEN
```

In this statement all four conditions must be true to give the result.

What do the following imply?

```
IF ((C1 OR C2) AND (C3 OR C4)) THEN...
```

```
IF ((C1 OR C2) OR (C3 OR C4)) THEN...
```

Notice the importance of brackets in the statements. Their placing gives a clear logical meaning to an expression. Any bracketed expression will be evaluated first. The result (TRUE or FALSE) obtained from the bracketed expression will be used in evaluating the whole expression.

A practical example of multiple logical operations on conditions would be obtaining a loan. The relevant conditions could be:

C1 = Husband is over 21 years old

C2 = Husband's salary is over 5,000 per year

C3 = Wife is over 21 years old

C4 = Wife's salary is over 5,000 per year

A statement can be written which indicates whether the bank will grant the family a loan to buy a car:

```
IF ((C1 AND C2) OR (C3 AND C4)) THEN loan granted.
```

EXERCISES

Key in and run the program which illustrates this:

```
10 INPUT "AGE OF HUSBAND"; HA: INPUT "AGE OF  
WIFE"; HW: INPUT "HUSBANDS SALARY"; SH  
15 INPUT "WIFE SALARY"; SW  
20 IF (HA > 21 AND SH > 5000) OR (HW > 21 AND  
SW > 5000) THEN 40  
30 PRINT "SORRY NOT ELIGIBLE FOR LOAN": ST  
OP  
40 PRINT "LOAN AVAILABLE"
```

- Write a program which inputs four numbers and outputs a message if any of them are zero.

J8: Logical operations on numbers

The logical operations AND, OR, NOT, when applied to numbers, return a number as the result. The rules for the operations on two numbers X and Y are given in the following truth tables. Non-zero values may be either positive or negative. Take for example:

7 AND 3

The Commodore converts both numbers to binary, then performs the *binary logical operator* on the numbers, and then converts to decimal and returns the result to BASIC.

Thus 7 AND 3:

7 is 00000111 in binary
3 is 00000011 in binary

Now AND each byte:

1 0 gives 0
1 1 gives 1
1 1 gives 1

Thus the result is 3. Now consider 7 OR 3.

7 = 00000111
3 = 00000011
1 V 0 gives 1
1 V 1 gives 1
1 V 1 gives 1

Thus the result is 7.

The NOT operator is slightly different. It returns a negative number 1 larger than the original number. Consider:

B = 2
NOT B = -3
B = 4
NOT B = -5

or

B = -2
NOT B = 1
B = -5
NOT B = 4

AND

X	Y	X AND Y
X	1	X
X	0	0

that is, X and Y returns X if Y is 1; 0 if Y is zero.

OR

X	Y	X OR Y
X	1	1
X	0	X

That is, X OR Y returns 1 if Y is -1; X if Y is zero.

NOT

Y	NOT Y
-1	0
0	-1

That is, NOT Y returns 0 if Y is -1; -1 if Y is zero.

Examples:

```
7 AND 3 = 3
7 AND 0 = 0
5 OR 2 = 7
5 OR 0 = 5
NOT 8 = -9
NOT 0 = -1
```

EXERCISES

- Key in the examples given above as direct commands, and verify the rules of logical operations on numbers.
- Key in and run the following programs:

```
10 REM**LOGIC 1**
20 INPUT A,B
30 PRINT "A=";A;"B=";B
40 PRINT"A AND B =" ;A AND B
50 PRINT"A OR B =" ;A OR B
60 PRINT"NOT B =" ;NOT B
70 GOTO 10
```

```
5 REM**LOGIC 2**
10 REM**THIS PROGRAM TESTS THE LOGICAL O
R OPERATOR ACTING ON A NUMBER AND A
15 REM CONDITION TOGETHER **
20 PRINT"Y=10*(7 OR A=3)"
30 INPUT"ENTER A VALUE FOR A PLEASE ";A
40 Y=10*(7 OR A=3)
50 PRINT"IF A=";A;"THEN Y=";Y
60 PRINT
70 PRINT"WHAT ARE YOUR CONCLUSIONS ??"
80 GOTO 30
```

```
5 REM**LOGIC 3**
10 INPUT A
20 PRINT 77+(10 AND A=3)
30 GOTO 10
```

J9: Priority

OPERATOR	PRIORITY
=, <>, <, <=, >, >=	5
NOT	4
AND	3
OR	2

Priority rules are strictly obeyed. If brackets are not used properly when logical operators act on conditions the desired result will not be achieved. For example:

NOT (FALSE AND FALSE) gives NOT FALSE = TRUE

but

NOT FALSE AND FALSE gives TRUE AND FALSE = FALSE

Exactly the opposite.

EXERCISES

- Key and run this program, which checks priority.

```
10 A=1
20 B=1
30 PRINT NOT (A=0 AND B=0)
40 PRINT NOT A=0 AND B=0
```

- What result would the following give?

```
PRINT 5 AND 3 OR 0 OR NOT 7 AND 4
```

- Key in and run program LOGIC 4, which tests priorities.

```
5 REM**LOGIC 4**
10 REM** THIS PROGRAM TESTS MULTIPLE LOG
IC OPERATORS **
20 A=5 AND 3 OR 0 OR NOT 7 AND 4
30 PRINT"5 AND 3 OR 0 OR NOT 7 AND 4=";A

40 PRINT:PRINT
50 B=((4 AND 2) AND NOT(0 AND 3)) OR ((3
OR 0) AND(4 OR 0))
60 PRINT"((4 AND 2) AND NOT(0 AND 3))OR(
(3 OR 0) AND(4 OR 0))=";B
```

J10: Logical operations with strings

Logical operations using AND, OR and NOT may be performed on conditional string expressions. For example:

```
10 IF (A$ = B$) AND (C$ = D$) OR (D$ = E$) THEN...
50 PRINT NOT A$ = B$.
```

Two strings cannot be directly operated on by any logical operator because string cannot have logical values. For example, A\$ AND B\$, A\$ OR B\$, NOT A\$ are meaningless expressions.

EXERCISES

- Key in
PRINT NOT "A" = "B"
and
PRINT "A" = "B" AND "B" = "C" OR "G" = "E"
to test the rules of logical string operation. Try other combinations.
- Write a program which requests a name and then checks to see if it corresponds to several strings stored in the program, printing out a message to say if the word was found.
- Write a program to test the truth table for AND, OR and NOT.

J11: Applications of logical operators

- Simple conditional tests
- Multiple conditional tests
- Multibranch GOTO and GOSUB
- Finding the maximum and minimum values
- Checking characters input
- Checking input values
- Testing for zero
- Default values

Simple conditional tests

IF (logical operation) THEN (statement). If the logical operation is TRUE the statement is executed. AND, OR and NOT operators are used.

Multiple conditional tests

IF ((condition 1) AND (condition 2) OR (condition 3) THEN (statement). If the multiple logical operations are TRUE the statement is executed.

Finding maximum and minimum values

The AND operator can be used to find the maxima and minima of two numbers X and Y.

```
10 INPUT X,Y
20 PRINT"MAX IS"; (X AND X>=Y)+(Y AND Y>X
)
30 PRINT"MIN IS"; (X AND X<=Y)+(Y AND Y<X
)
```

or we could program this as

```
10 INPUT X,Y
20 IF X>=Y THEN 50
30 PRINT"MAX=";Y;"MIN=";X
40 STOP
50 PRINT"MAX=";X;"MIN=";Y
```

Which do you think is the best method?

Finding the largest number in a list is another application. If you have a list of numbers $A(1)$ to $A(N)$, you can compare the first two, $A(1)$ and $A(2)$, and put the largest of these into a variable L by the statement:

```
LET L = (A(1) AND A(1) >= A(2) + A(2) AND A(2) > A(1))
```

You then compare this value of L with the next number $A(3)$ and make L take the larger value of the two, and so on through the list.

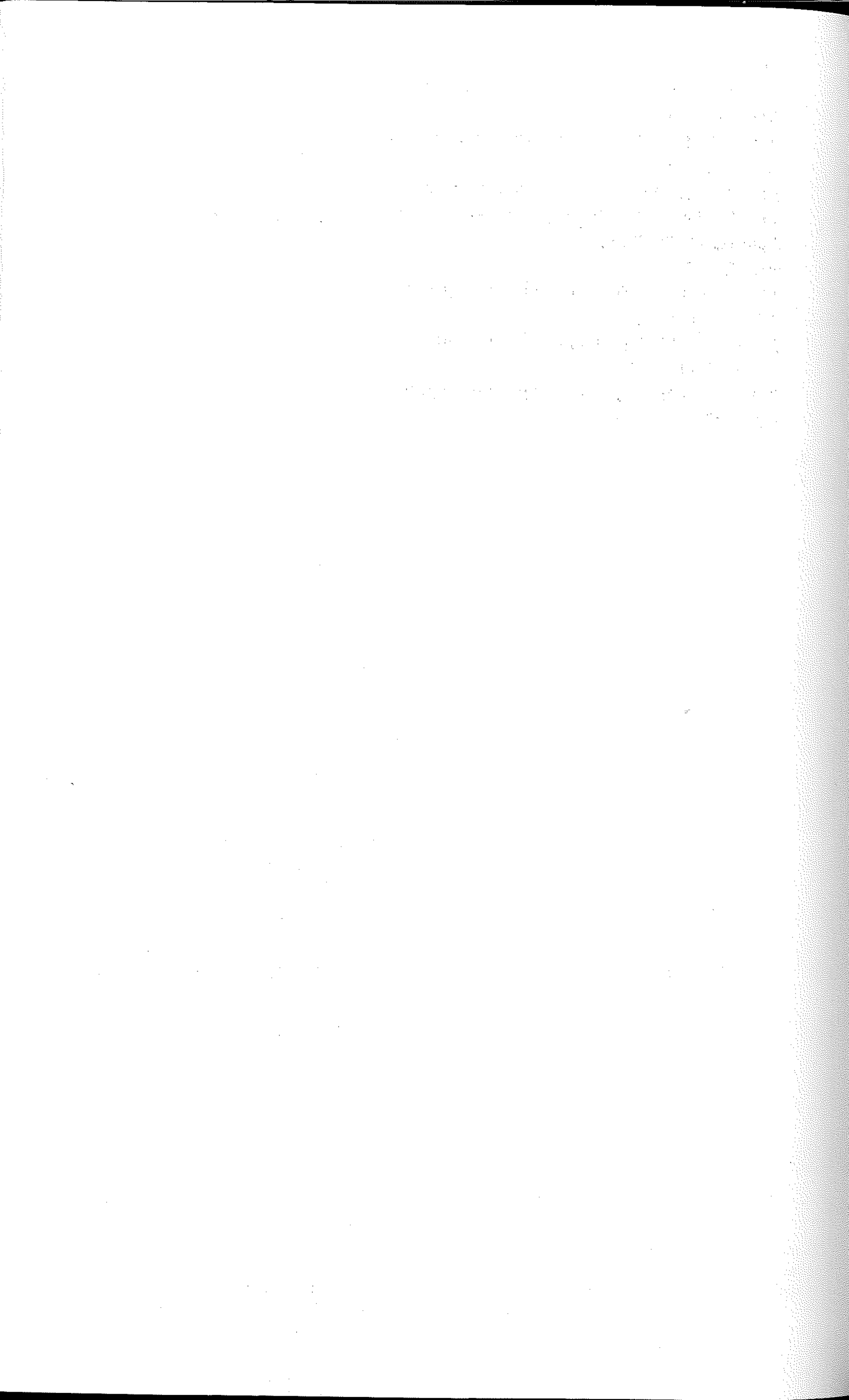
```
LET L = (L AND L >= A(3)) + (A(3) AND A(3) > L)
```

The program asks you to input how many numbers will be in a list A . You then input the numbers $A(I)$. These are printed on the screen together with the largest value. Two loops are used, the first to input the numbers and the second to perform the comparisons. Key in and run the program.

```
10 REM**LARGEST**
20 INPUT"HOW MANY NUMBERS";N
30 DIM A(N)
40 FOR I=1 TO N
50 INPUT A(I)
60 NEXT I
70 L=(A(1) AND A(1) >=A(2))+(A(2) AND A(2)
) >A(1))
80 FOR I=3 TO N
90 L=(L AND L >=A(I))+(A(I) AND A(I) >L)
100 NEXT I
110 PRINT"LARGEST NUMBER IS";L
```

This is an appropriate place to emphasise the care needed in programming logical operations. There are two areas where the most problems arise: the setting of conditions, and the grouping of these conditions in a logical sequence that will produce the required result.

```
10 INPUT A
20 IF A<0 OR A>9 THEN 100
30 INPUT B
40 IF B<10 OR B>99 THEN 200
50 INPUT C
60 IF C<100 OR C>999 THEN 300
70 PRINT "*****"; A; "*****"; B; "*****
*****"; C
80 STOP
100 PRINT "A OUT OF RANGE"
110 GOTO 10
200 PRINT "B OUT OF RANGE"
210 GOTO 30
300 PRINT "C OUT OF RANGE"
310 GOTO 50
```



Section K: Strings

K1: Strings

A string is a set of characters enclosed by quotation marks. For example: "THIS IS A STRING", or the *null string* "" (no characters). These are strings:

```
"ALL OF STRING"  
"JANUARY 1ST 1982"  
"URGHH!"  
"FAB * * - + / ! 3"  
"          "(String of spaces)  
"1234"  
"" (Null string)
```

Two kinds of data which computers handle are:

numeric – numbers
alphanumeric – names or text

The way a computer deals with text is called *string handling*. Strings deal with *alphanumeric* information. The sequence of alphanumeric characters is handled in a string as a single unit of data.

Characters are defined as *literals* when placed inside quotes. They are taken literally to represent themselves. Strings are therefore literals. Characters are *identifiers* where they are not enclosed in quotes. Thus, for example, A represents or identifies a numeric variable and A\$ identifies a string variable.

```
DIM A$(N)
```

where N is the number of elements in the string A\$.

All characters in your computer's character set can be used in strings. Run this program; it illustrates that most of the character set can be included in strings.

```
10 FOR A=1 TO 255  
11 IF A=5 OR A=13 OR A=17 OR A=20 OR A=2  
8 OR A=29 OR A=30 OR A=31 THEN 30  
12 IF A=141 OR A=144 OR A=145 OR A=148 O  
R A=156 OR A=157 OR A=158 THEN 30  
13 IF A=159 THEN 30  
15 IF A=19 OR A=147 THEN 30  
20 PRINT CHR$(A),A  
30 NEXT A
```

This program traps certain codes (all the colour codes, and cursor movement ones).

Examples of the sort of text we may want the computer to handle are:

- a telephone directory
- names and addresses
- a timetable
- expenses and details

Computers store all this textual information as strings. String manipulation by the computer would, for the telephone directory, need to deal with:

<i>Creating</i>	the telephone directory
<i>Sorting</i>	the names and numbers into the correct order
<i>Searching</i>	the directory for somebody's number
<i>Revising</i>	the directory (updating or adding an entry)
<i>Printing</i>	the directory in whole or in part.

K2: Quotes

All strings are enclosed in quotes " " when:

- used in programs with the PRINT instruction, as in
20 PRINT "STRING"
- assigned in a program to a string variable, for example
30 A\$ = "STRING"
- in a data statement when the string contains (,),(:), or (;).

Note: you cannot enter quotes from the keyboard (using INPUT A\$, for example, or using quotes as part of the string). This can only be done by adding the quote character (CHR\$(34)) to the string. Try the following example:

```
10 INPUT A$
20 A$=CHR$(34)+A$+CHR$(34)
30 PRINT A$
```

Now A\$ will have as its first character a quote " .

K3: String input

As can be seen from the example above, string input is exactly the same as numeric input. The only major difference is that, if you just press the key marked return, A\$ will contain nothing; whereas on numeric input (INPUT A) a zero is returned. When the computer is waiting for INPUT, pressing the **RUN/STOP** key has no effect. The only way to break out is to hit the **RUN/STOP** and the **RETURN** keys together.

There is a second way to input string data and that is using the GET A\$ statement. This statement does not wait until a key is pressed; if none is pressed, it returns with an empty string. Enter this program and run it.

```
10 GET A$
20 PRINT A$; " ";
30 GOTO 10
```

Press the **RUN/STOP** key to break execution of the program; enter the following line and re-run the program.

```
10 GET A$:IF A$="" THEN 10
```

Now whenever a key is pressed it is printed on the screen with one space in between each character, not as before, when if no keys were pressed spaces

were printed. This shows that the statement GET A\$ scans the keyboard once; if no key is pressed it returns with a null string (empty). The use of GET A\$ has many applications. Enter this program and run it:

```
10 PRINT "PRESS A KEY WHEN READY"  
20 GET A$: IF A$="" THEN 20  
30 PRINT "YOU PRESSED "; A$
```

Line 20 sends the program back to the beginning of the line as long as no key has been pressed. Now enter and run this program:

```
10 PRINT "PRESS 6"  
20 GET A$: IF A$="" THEN 20  
30 IF A$="6" THEN 60  
40 PRINT "OBEY INSTRUCTIONS!!"  
50 GOTO 20  
60 PRINT "THANK YOU"  
70 STOP
```

Line 20 does the same as before, but line 30 now checks that the right key has been pressed. If 6 was pressed, the program goes to line 60. If any other key was pressed, it goes to 40, prints the message, and then is sent back by line 50 to line 20, which waits for another key to be pressed.

Games programs, which require interaction, often use GET in a loop, so that every time the program loops, it checks which key, if any, is being pressed.

K4: Length of a string

LEN(A\$)

The length of a specified string A\$ is obtained by using the function LEN(A\$). The length is given as the number of characters and is the current length of a string. Spaces are included in the length of a string.

Run these programs:

```
10 A$="COMMODORE 64"  
20 PRINT LEN(A$)
```

Check that the result is 12.

```
10 A$="A           B"  
20 PRINT TAB(10); A$  
30 PRINT "LENGTH="; LEN(A$)
```

Now run this program.

```
20 FOR N=1 TO 255
30 A$=A$+CHR$(96+INT(RND(1)*16))
40 NEXT N
50 PRINT A$
```

The string A\$ is filled with graphics characters of codes 96 – 111, and then printed. The maximum length of a string is 255 characters long. Alter line 20 in the above program to read

```
20 FOR N=0 TO 255
```

to show this.

K5: Null strings

A string with no characters is called a null string. For example:

```
LET A$ = ""
```

The length of the string is 0.

A string which contains spaces is not a null string. A space is the character obtained by pressing the space bar. The null string is returned by GET A\$ if no key is being pressed.

EXERCISES

- Key in and run the following program:

```
10 A$=""
20 PRINT A$
30 PRINT LEN(A$)
```

- Key in and run this program:

```
10 A$=" "
20 PRINT A$
30 PRINT LEN(A$)
```

K6: String variables and dimensions

The statement

```
DIM A$(N)
```

where N is any number and A\$ is any legal string variable, is a *dimension* statement that allows you to store string data. Thus for example:

```
DIM A$(20)
```

would reserve space for 20 strings, of any length (up to 255 characters).

Consider this program:

```
10 DIM A$(3)
20 FOR N=1 TO 3
30 READ A$(N)
40 PRINT A$(N):NEXT N
50 DATA COMPUTERS,ARE,PHENOMENAL"
```

K7: Multi-dimensional string variables

These are dealt with in greater depth in Section S, on arrays. They are created and referenced in terms of their position in the multidimensional array. Think of how a book is built up. The individual strings or words on a page are created and referenced as:

- a string (a word)
- a list, row, line or column (a one-dimensional array)
- a page or table of rows & columns (a two-dimensional array)
- a book of many pages (a three-dimensional array)

A two-dimensional string array is like a single page of a book, where the words are referenced in row and column position.

A\$ (row no., column no.) eg. A\$(4,6)

and a three-dimensional array...

A\$ (page number, row number, column number)

and a four-dimensional array...

A\$ (book number, page number, row number, column number)

What would be the format for the next two dimensions up? Try setting up and reading a three-dimensional string array; use four pages as an example.

- Page 1 will contain names of fish
- Page 2 will contain names of mammals
- Page 3 will contain names of fruit
- Page 4 will contain names of plants

The arrays or tables on each page will be ordered according to size.

```
5 REM 4 PAGES, 3 ROWS, AND 2 COLUMNS
10 DIM A$(4,3,2)
20 FOR P=1 TO 4
30 FOR R=1 TO 3
40 FOR C=1 TO 2
50 READ A$(P,R,C):NEXT C,R,P
60 DATA MINNOW,PERCH,BARBEL,PIKE,COD,SHARK
70 DATA MOUSE,RAT,CAT,DOG,PIG,HORSE
80 DATA NUT,GRAPE,APPLE,ORANGE,BANANA,MELON
90 DATA LOBELIA,DAISY,ROSE,SUNFLOWER,WILLOW,CORNATION
```

```

100 REM PRINTING ARRAY
110 FOR P=1 TO 4
120 PRINT "PAGE";P:PRINT
130 FOR R=1 TO 3
140 FOR C=1 TO 2
150 PRINT A$(P,R,C):NEXT C,R
160 PRINT:NEXT P

```

EXERCISES

- Key in and run the program. Draw on paper the organisation or structure of the data with its coordinates. Here is Page 1 with strings and coordinates.

```

Minnow (1,1,1)
Perch (1,1,2,)
Barbel (1,2,1)
Pike (1,2,2)
Cod (1,3,1)
Shark (1,3,2)

```

Use a command mode to access single strings: for example,

```
PRINT A$(2,1,2)
```

K8: String and string array assignment

Strings are assigned to string variables using the LET, READ and INPUT instructions. For example:

```

A$ = "A STRING" or
INPUT A$ or
READ B$

```

This establishes a value for the string. The value may be a literal value in quotation marks, or a string or substring value.

Here is a program which assigns 3 strings to the string array variable A\$(3).

```

10 DIM A$(3)
20 A$(1)="COMMODORE"
30 A$(2)="COMPUTING"
40 A$(3)="COURSE"

```

Using the INPUT instruction and a loop:

```
FOR I =1 TO N:INPUT A$(I):NEXT I
```

allows you to enter and assign N strings when dimensioned with DIM A\$(N).

Using READ and DATA:

```

FOR I=1 TO N:READ A$(I):NEXT
DATA D,.....N items

```

EXERCISE

- Write programs to create a 3x3 table or array of names, occupations and salaries using INPUT, READ, and LET assignment techniques with loops.

- Write a program which will update your table as occupations and salaries change.

K9: Substrings and string slices

A *substring* or a *string slice* is any set of consecutive characters taken in sequence from the parent string. The Commodore has several functions which allow strings to be subdivided:

```
LEFT$(A$,N)
RIGHT$(A$,N)
```

Where N is the number of characters. Thus if

```
A$ = "ABCDEFGH"
LEFT$(A$,2) returns AB
RIGHT$(A$,2) returns GH
```

There is a third string function:

```
MID$(A$,M,N)
```

where A\$ is the string, M is the starting position and N is the number of characters. Thus if A\$ = "ABCDEFGH" then MID\$(A\$,2,2) would return CD. Try this program:

```
10 A$="CENTURY COMPUTING COURSE"
20 PRINT LEFT$(A$,8)
30 PRINT MID$(A$,9,10)
40 PRINT RIGHT$(A$,6)
```

For example, for the string "ABCDEFGH", a substring is "CDEF", or "ABC", or "G". A substring can be a single character. For example, try this program:

```
10 A$="NAME AGE"
20 B$="TOM 16"
30 C$="BILL 14"
40 D$="JANE 17"
50 PRINT " "; LEFT$(A$,4); " "; RIGHT$(A$,3)
60 PRINT " "; LEFT$(B$,4); " "; RIGHT$(B$,3)
70 PRINT " "; LEFT$(C$,4); " "; RIGHT$(C$,3)
80 PRINT " "; LEFT$(D$,4); " "; RIGHT$(D$,3)
```

K10: String concatenation

A\$ + B\$

Concatenation means chaining strings together. It is derived from the word *catenary*, meaning chain. What the computer does is to 'add' them together to form a new string.

```
"COM"+"PU"+"TER"="COMPUTER"
```

```
10 A$="COM"  
20 B$="PU"  
30 C$="TER"  
40 T$=A$+B$+C$  
50 PRINT T$
```

You cannot subtract, multiply, or divide strings, or raise them to powers, because they are not numbers. Although the 'adding' of concatenation uses the same symbol, it is not an arithmetic operation. Key in and run the example program given above.

Now try this program:

```
10 INPUT A$,B$  
20 PRINT A$,B$,A$+B$  
30 A$=A$+B$  
40 PRINT A$  
50 A$=A$+A$  
60 PRINT A$
```

Notice in line 30 the string has been incremented by adding B\$ on to A\$. This gives a new A\$ made up of the old A\$ plus B\$. The statement in line 50 is equivalent, in string terms, to having a line which for numeric variables says LET A = A + A.

K11: Comparing strings

The conditional operators =, <>, <, <=, >, >= may be used between strings and string variables using the IF...THEN instructions. For example:

```
IF A$ = "YES" THEN GOTO.....  
IF N$ = B$ THEN PRINT.....  
IF A$ = B$ THEN GOTO.....
```

When the computer compares strings of characters, it does so by comparing the ASCII code of each of the characters in sequence. A string is found to be less than another if it comes first in alphabetic order. If the strings contain numbers you must remember that numeric codes are less than alphabetic codes; this will affect comparisons. (See appendix for the character codes.) Strings are compared in order of characters from left to right. For example:

"Aa"	>	"AA"	"A"	<	"B"
"a"	>	"Z"	"AB"	<	"AZ"
"b"	>	"a"	"A"	<	"AA"
"z1"	>	"Z1"	"2"	<	"5"
"Smith"	>	"SMITH"	"6"	<	"Q"

Key in and run the next program. Input the strings above plus others you want to try and it will print out their relative alphabetic orders.


```

10 INPUT A$,B$
20 IF A$>B$ THEN 60
30 IF A$=B$ THEN 80
40 PRINT A$;" < ";B$
50 STOP
60 PRINT A$;" > ";B$
70 STOP
80 PRINT A$;" = ";B$
90 STOP

```

This reveals a method for putting names into alphabetic order, as in a telephone directory. There is also a method of searching it, since you can check whether any name in the list is equal to the desired name. Here's another example of string comparison:

```

10 PRINT"DO YOU UNDERSTAND STRINGS"
20 INPUT"ANSWER YES OR NO";A$
30 IF A$="YES" THEN 60
40 PRINT"THEN READ THIS SECTION AGAIN!!"

50 STOP
60 PRINT"YOU ARE A GENIUS!!"
70 STOP

```

EXERCISES

- The telephone program sets up a telephone directory with names and telephone numbers. It will search through its lists to find the telephone number corresponding to a given name. Run and analyse the program to find out how it works.

```

10 REM**TELEPHONE**
20 REM**PROGRAM SETS UP A TELEPHONE**
21 REM**DIRECTORY AND USES IT      **
30 PRINT"HOW MANY NAMES DO YOU WISH TO E
NTER INTO THE DIRECTORY";
40 INPUT N
50 PRINT
60 PRINT"INPUT ";N;" NAMES (UPTO 20 LETT
ERS) AND NUMBERS(8 FIGS) PAIRS"
70 DIM A$(N)
80 DIM B$(N)
90 DIM D$(20)
100 PRINT
110 PRINT
120 PRINT"NAME";TAB(22);"NUMBER"
130 PRINT
140 FOR F=1 TO N

```

```

150 INPUT A$(F),B$(F)
160 NEXT F
170 PRINT""
175 PRINT"NAME";TAB(22);"NUMBER"
180 FOR F= 1 TO N
190 FOR F= 1 TO N
200 PRINT A$(F);TAB(22);B$(F)
210 NEXT F
220 PRINT"PRESS ANY KEY TO CONTINUE"
230 GET A$:IF A$="" THEN 230
240 PRINT""
250 PRINT
260 INPUT"WHAT NAME";D$
270 REM**NEXT PART OF THE PROGRAM**
280 REM**SEARCHES FOR THE NAME **
290 PRINT
300 PRINT D$;
310 FOR F=1 TO N
320 IF A$(F)=D$ THEN 370
330 NEXT F
340 PRINT
350 PRINT"NAME NOT FOUND"
360 GOTO 260
370 PRINT TAB(22);B$(F)
380 PRINT
390 PRINT
400 INPUT"ANOTHER NAME (Y/N) ";Q$
410 IF Q$="?" THEN 400
420 IF Q$="Y" THEN 240
430 PRINT

```

- Modify the program to create your own directory with your friends' names and addresses or birthdays or telephone numbers.
 - a) Redesign the program
 - b) Document it
 - c) Key it in
 - d) SAVE it
 - e) Debug it
 - f) SAVE the working version
 - g) Put it in your personal tape library
 - h) Enter details in your notebook

K12:VAL and STR\$

The function VAL converts a string (only containing numeric data) to a numeric variable.

Enter and run this program:

```
10 A$="1234"  
20 A=VAL(A$)  
30 PRINT A$,A  
40 A=A+4  
50 PRINT A
```

STR\$(N) returns the value of (N), a numeric expression, as a string. For example:

```
STR$(3.4) gives "3.4"  
STR$(3 * 31) gives "93"  
STR$(SQR(4)) gives "2"
```

STR\$ is the complementary or opposite function to VAL.

To see STR\$ in operation, and the complementary functions of VAL and STR\$, try this program:

```
10 X=3  
20 Y=0.5  
30 A$=STR$(X/Y)  
40 PRINT A$,VAL(A$)  
50 B$=A$+STR$(X)  
60 PRINT B$,VAL(B$)  
70 C=VAL(STR$(VAL(A$)+VAL(B$)))  
80 PRINT C
```

K13: ASC and CHR\$

The purpose of the instructions ASC and CHR\$ is to convert from the code to character and vice versa.

ASC is a function which takes a character or a string and gives as a result the numeric (ASCII) to which the character corresponds. For example:

```
ASC("S") gives 83.  
ASC("ABCD") gives the ASCII code for A, which is 65.  
ASC(X$) gives the ASCII code for the first character in X$.
```

CHR\$ is a function which gives as a result the single character whose ASCII code is given by N. CHR\$ does the opposite of ASC. For example:

```
CHR$(A+B+C)  
CHR$(X/Z)  
CHR$(INT(RND(1)*255)) gives a random number in the range 0 to 255  
CHR$(36) gives "$"  
CHR$(83) gives "S"
```

Try this program, which prints all the characters used on the Commodore.

```
10 FOR I=32 TO 255  
20 PRINT CHR$(I)"";  
30 NEXT
```

EXERCISES

- Write a program which inputs a number of strings and calculates the total number of characters in each, and the total number of characters in all the strings.
- Write a program which calculates the total price of items in a shopping list, after receiving and printing out the string inputs of each item and its cost.
- Write a program which will print a calendar for any month of next year. Key in the month names and lengths as a string in the program.

Section L: Loops

You have already used loops in the course. The simple GOTO loop was introduced in Section E as a method of jumping back to a previous program line to repeat a program operation. The classical FOR...NEXT loop was defined and used in Section F, which considered data INPUT. Multiple loops were used in the last Section to create and print out a string array. Loops are so important that they deserve a Section on their own.

L1: Loops

A loop is a block of instructions that the computer executes repeatedly until a terminating condition is met. The usefulness of loops can be seen by considering three forms of a program to print out the first one hundred positive integers.

```
10 PRINT 1
20 PRINT 2
30 PRINT 3
.....
.....
.....
100 PRINT 100
```

This program, which does not use a loop, is 100 statements long. This next program uses a conditional jump loop which does the same thing and uses only five statements.

```
10 C=1
20 PRINT C
30 C=C+1
40 IF C<=100 THEN 20
50 STOP
```

The third program uses a FOR...NEXT, loop which is the commonest method of looping in BASIC, the most economical in program lines, and the most versatile and powerful.

```
10 FOR F=1 TO 100
20 PRINT F
30 NEXT F
40 STOP
```

All loops have these four characteristics:

- Initialisation (start value counter)
- BODY of loop
- Modification of counter
- Exit condition or repeat condition

There are two types of *loop structure* in programming.

- The **repeat-until** structure, which means *repeat* the task *until* the *exit* condition is true.
- The **while-do** structure, which means *while* the *repeat* condition is true, *do* the task continuously.

The differences in the structures are, firstly, the order in which the loop characteristics are placed in the program.

Repeat-until

- Initialise counter
- Task
- Modify counter
- Test EXIT condition

While-do

- Initialise counter
- Test REPEAT condition
- Task
- Modify counter

The essential difference is where the condition is tested and whether it is for exit or repeat. The other important difference is that the consequence of the order is that the task is processed at least once in the repeat-until structure, whereas it need not be processed at all in while-do.

All programming languages use these structures, and some include the instructions REPEAT, UNTIL, WHILE, DO. Commodore BASIC does not, and so the structures are formed using *conditional GOTO statements* and *FOR...NEXT statements*.

In the case of repeat-until loops, the complement of the exit test condition is also used; this allows a more elegant program. FOR...NEXT loops are a special case of the while-do structure.

To compare the three looping methods, a common example will be studied: a program that inputs and prints ten numbers.

L2: Counters

A counter is a *variable* used to count the number of times an event takes place during a program. For example, a *loop counter* will count how many times a letter, number or string occurs. Counters are used to provide information and to control processing.

Variables used for counters are usually I, J, K, L, M, N, COUNT, NUM. Upper and lower case letters are treated identically.

Initialisation

When counters are used in programs they must be assigned a start or *initial* value. This will usually be

- LET C=0 when counting events
- LET C=1 for loops

Incrementing

Each time the event occurs or the program loops, the counter value is increased by one:

```
LET C=C+1
```

Normally the counting is done in ones, but the start and stop values of counters can be set to any positive, negative, integer, real number, or value which is the result of calculating an expression. This will be illustrated in FOR...NEXT loops.

L3: Counting events

To count events in a program, the following steps must be taken:

- 1 Set the counter variable C to 0: LET C=0
- 2 Set up a conditional test for the event: IF (condition of event occurring is true) THEN...
- 3 Increment the counter when the event occurs: ...THEN LET C=C+1.
- 4 Print out the number of times it has occurred: PRINT C.

For example, count the number of times the vowel E occurs in a sentence.

```
10 INPUT"KEY IN A SENTENCE";A$:PRINT A$
20 C=0:REM INITIALISE COUNTER
30 FOR N=1 TO LEN(A$):REM SEARCH THE COMPLETE STRING
40 IF MID$(A$,N,1)="E" THEN C=C+1
50 NEXT
60 PRINT"E ENCOUNTERED";C;"TIMES"
```

EXERCISES

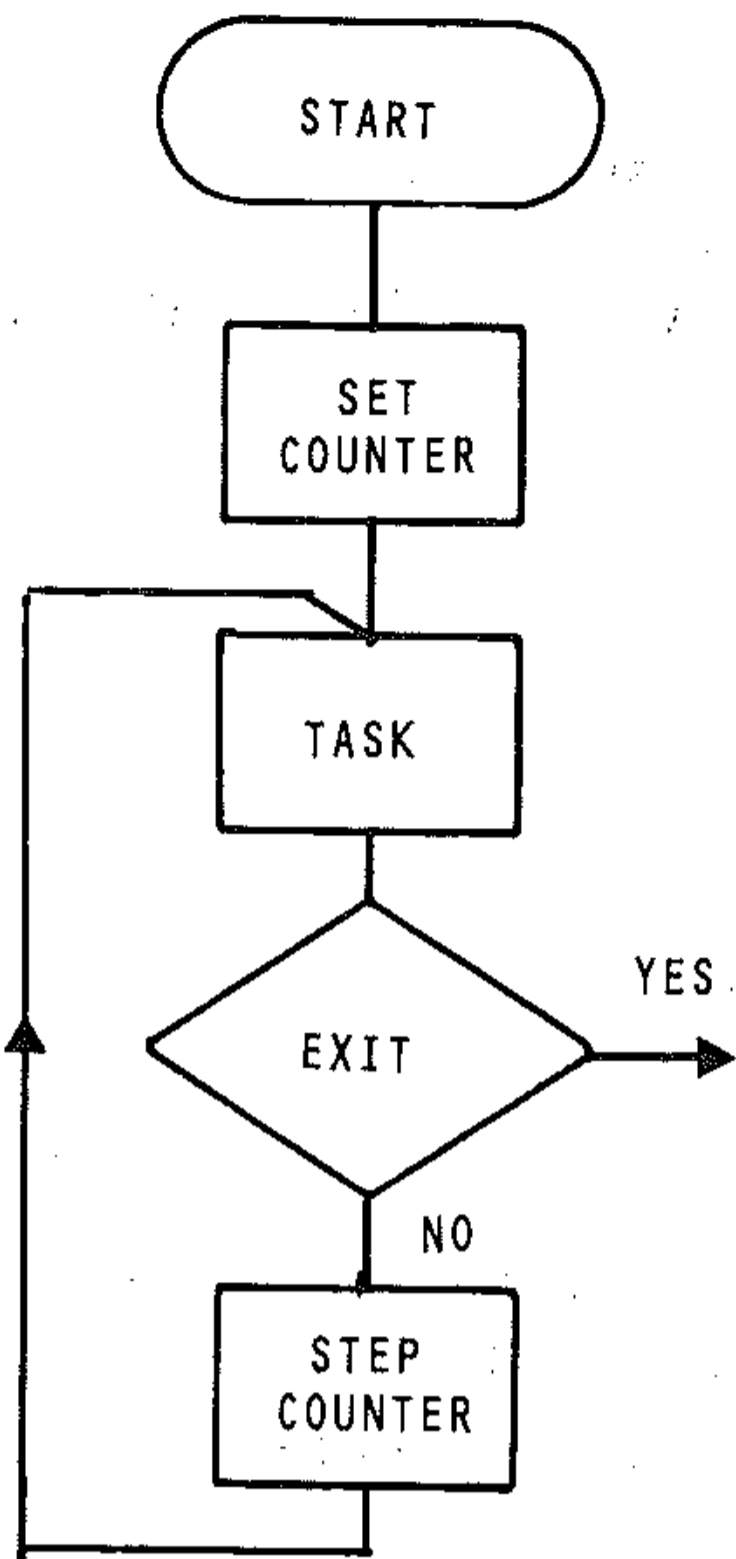
- Write a program which generates 100 random integers between 1 and 10 and counts the number of times 7 is generated.
- Change the above program to count how many numbers of 5 or less are generated.

L4: Repeat-until loops

In repeat-until loops:

- the program will loop until the exit condition is true.
- the exit test is below the task.
- the task is executed at least once.

Structure



BASIC example

```

10 C=1
20 INPUT A:PRINT A
30 IF C=10 THEN 60
40 C=C+1
50 GOTO 20
60 REM EXIT
  
```

Complement version

```

10 C=1
20 INPUT A: PRINT A
30 IF C<10 THEN LET C=C+1: GOTO 20
40 REM EXIT
  
```

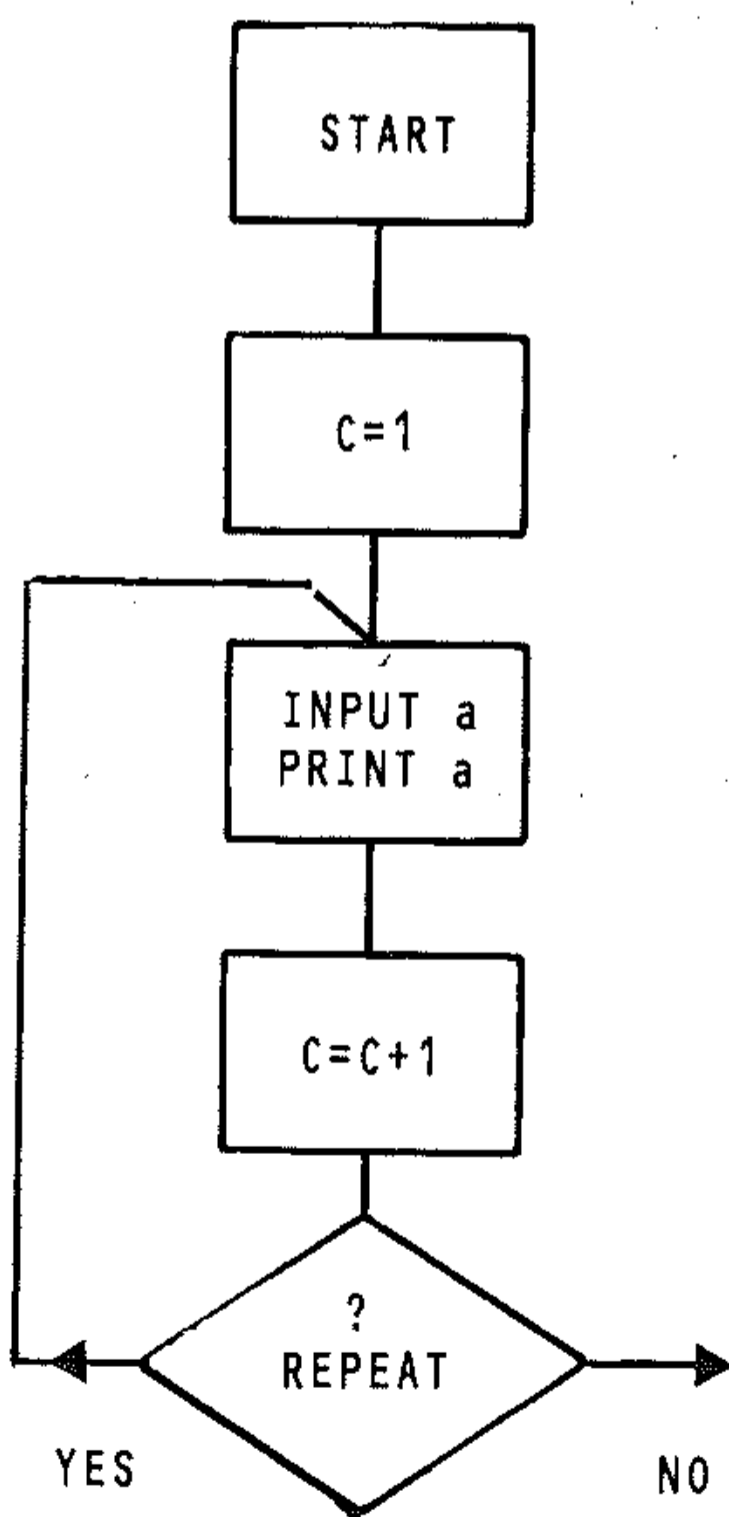
Using the complement of the EXIT test condition gives a more elegant program. It is a repeat test:

- C=10 exit condition
- C<10 repeat condition

Stepping the counter before the exit test is an alternative.

The test condition changes. The complement version is

Structure



Complement BASIC

```

10 C=1
20 INPUT A: PRINT A
30 C=C+1
40 IF C<=10 THEN 20
50 REM EXIT
  
```

In the complement version a repeat condition is tested. For example:

- C<=10 - repeat condition
- C>=10 - exit condition

When using counters for repeat-until loops, be careful to set the exit conditions properly (or their complement) to achieve the correct number or passes through the loop.

EXERCISES

- Study this program, which prints the seven times table. Notice the use of the AND operator in line 50 to line up the numbers printed.

```

10 PRINT "SEVEN TIMES TABLE"
20 N=1
30 PRINT N; "*7= "; N*7
40 IF N<20 THEN N=N+1:GOTO 30
50 STOP

```

READY.

- Write a program which calculates and prints the squares and cubes of powers of even numbers between 10 and 30. The counter will need to be incremented by 2 each time.
- Write a program which uses two counters to print the squares of numbers 5.00, 4.75, 4.50 ... 3.00 in that order.

L5: While-do loops

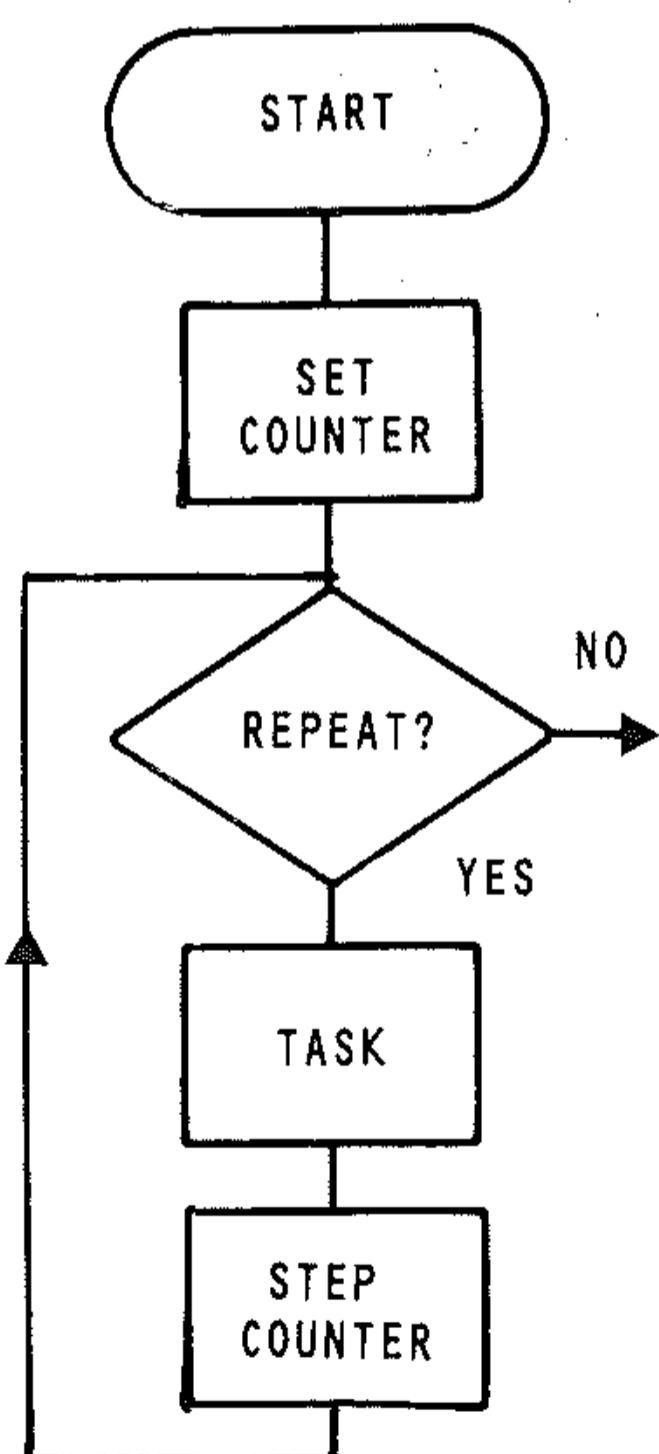
In these structures:

- the program will not loop until a repeat condition is true;
- the repeat test is above the task, and
- the task need not be performed.

Structure

BASIC Example

Complement



```

10 C=1
20 IF C<=10 THEN
GOTO 40
30 GOTO 70
40 INPUT A:
PRINT A
50 C=C+1
60 GOTO 20
70 REM EXIT

```

```

10 C=1
20 IF C>10 THEN
GOTO 60
30 INPUT A:
PRINT A
40 C=C+1
50 GOTO 20
60 REM EXIT

```

Again, the complement version is neater. The complement of the repeat test is an exit test. Counters are rarely used to perform this kind of loop. Instead we use the more powerful FOR...NEXT structure.

L6: FOR...NEXT loops

FOR...NEXT loops are a special BASIC language implementation of the while-do loop structure. They do not need a repeat condition test statement; the Commodore does this automatically when it executes the FOR...NEXT loop.

To be able to leave out this condition test (and the GOTO), certain things have to be specified:

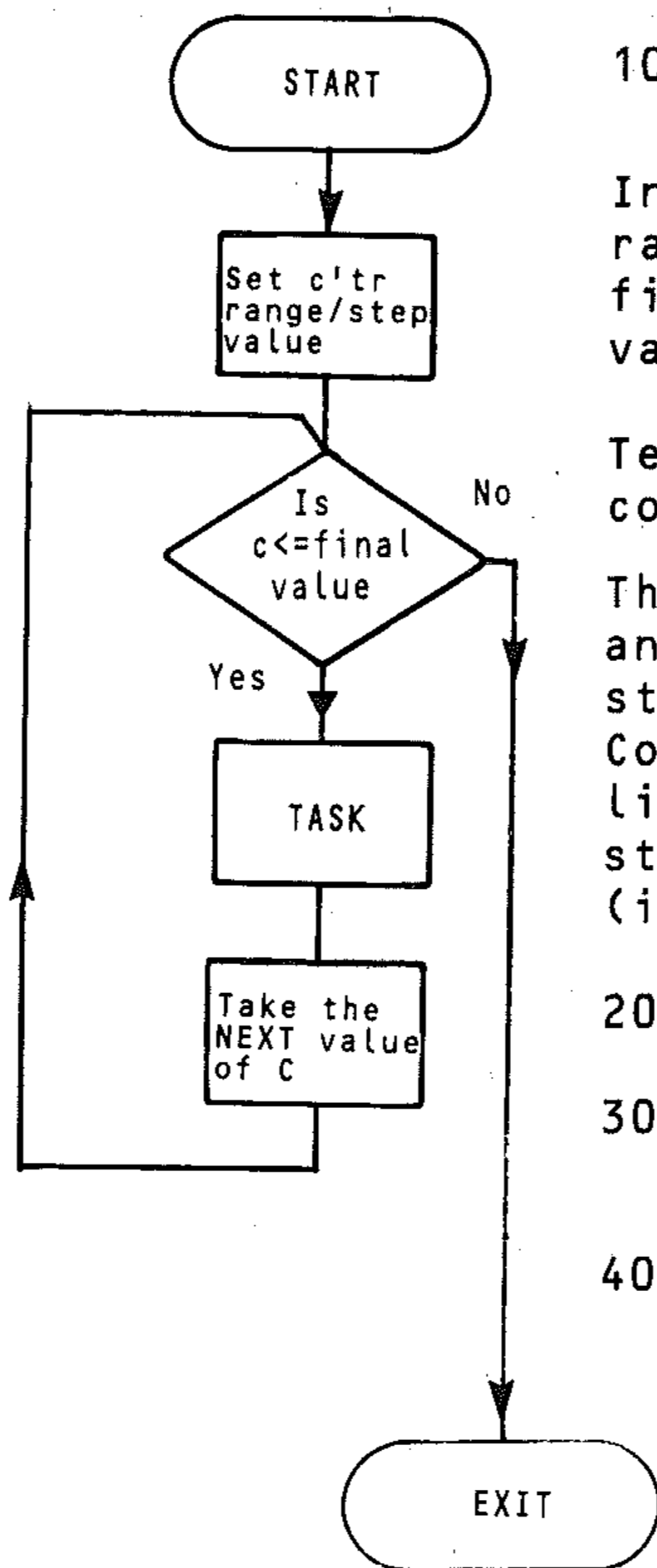
- the final value of the counter;
- the STEP value of the counter; and
- a statement which steps the counter to the NEXT value.

The FOR...NEXT loop is set up using four instructions: FOR, TO, STEP and NEXT. Using the While-do structure, consider the FOR...NEXT form in the example.

*While-do characteristics
for FOR...NEXT*

Structure

Program



```
10 FOR C=1 TO 10 STEP 2
```

Initialise the counter range to start value TO find value, and STEP value.

Test the repeat condition.

This test is built-in and needs no program statement. The Commodore jumps to line 40 when the statement is false (ie. when $C > 10$).

```
20 INPUT A: PRINT A
```

```
30 NEXT C
```

```
40 REM: EXIT
```

This is obviously a more convenient and powerful way of looping. The loop goes from the first value to the last value of the counter (or covers the range) by adding the defined STEP value to the previous value of the counter each time it loops until the repeat condition is false.

FOR...NEXT definition

FOR (variable) = (first value) TO (last value) STEP (step value)

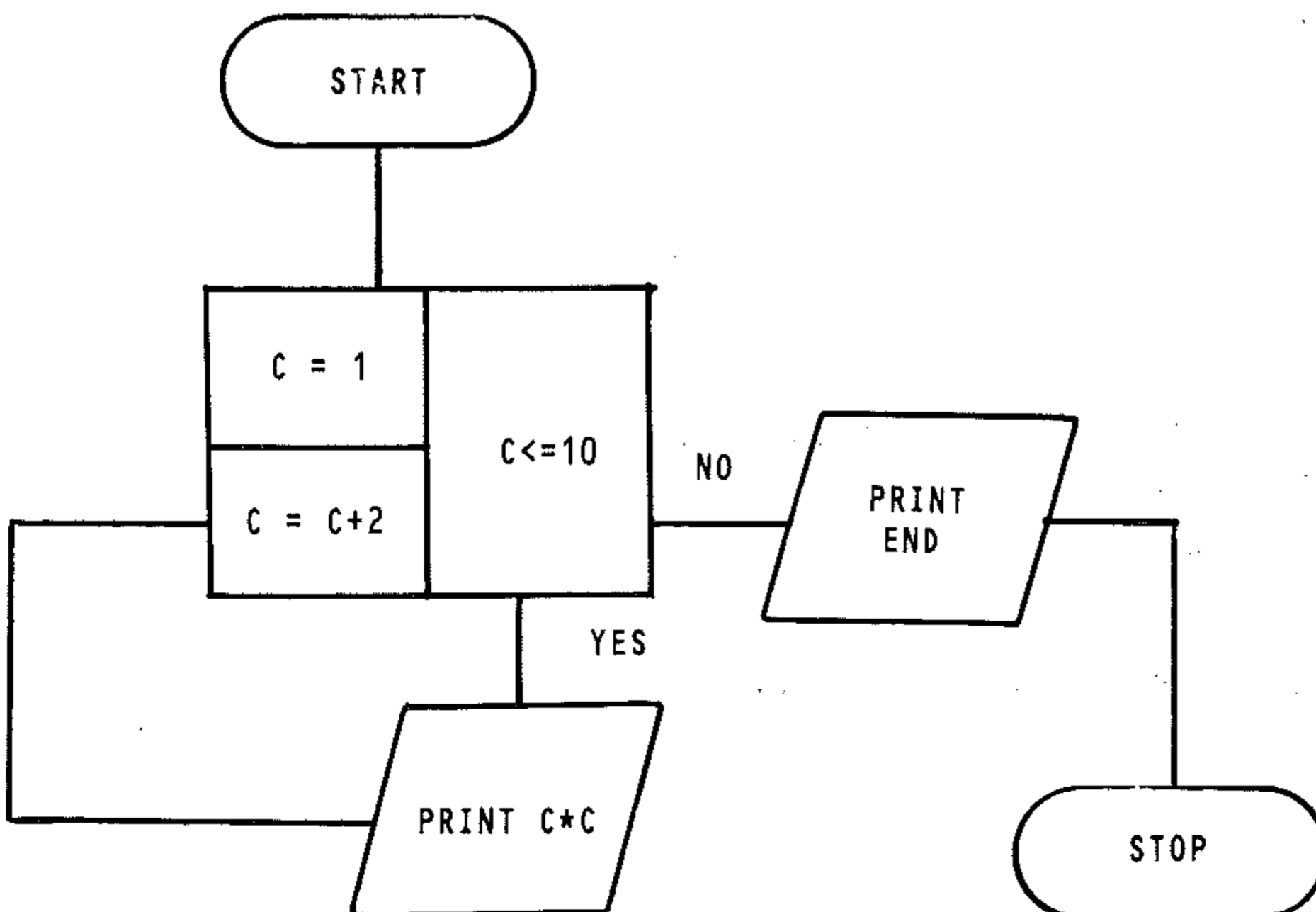
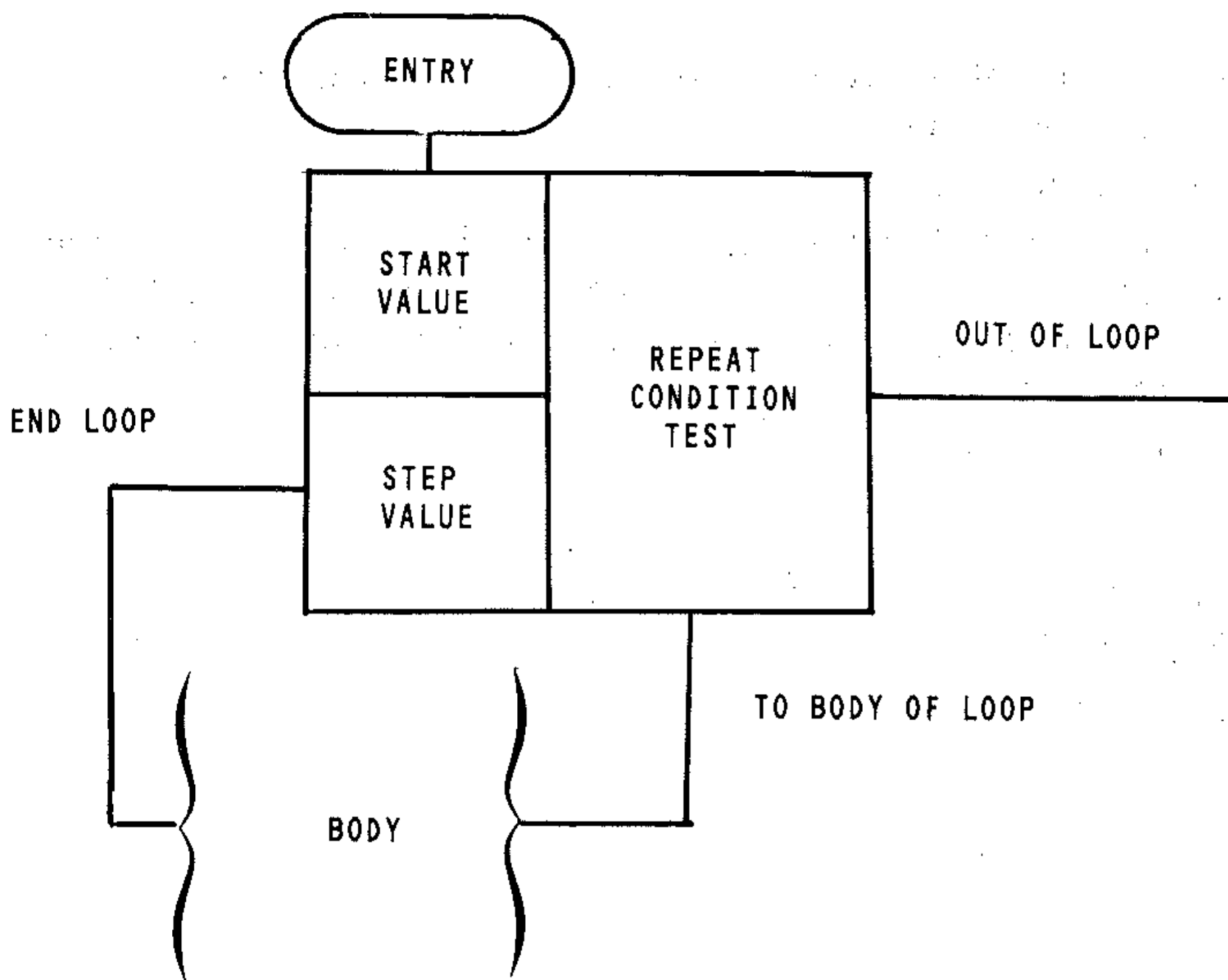
FOR C=N TO M STEP X

where C is the counter variable or control variable of the loop and N, M and X are numeric expressions.

C, N, M and X can be any legal numeric variable. It is initialised at value N. N, M and X may take any values, positive or negative, as long as repeated additions of X to N will reach M. If STEP is omitted, +1 is assumed. NEXT C indicates last line of the loop. It adds X to C and repeats the task if the total is less than M. The program loops back to the line after the line with the FOR...TO...STEP instruction.

L7: FOR...NEXT flowcharts

FOR...NEXT loops are used so frequently that a condensed version of the standard flowchart is used.



L8: FOR...NEXT examples

The FOR...NEXT loop has a fixed procedure, unlike loops formed with conditional GOTO instructions. A FOR...NEXT loop is formed in a program like this:

```
10 FOR F=0 TO 100 STEP 2
.....
..... (body of loop)
40 NEXT F
```

- The FOR statement initialises the loop.
- 0 is the start value.
- 100 is the stop value.
- F is the counter variable and is initialised as 0.
- STEP 2 is the increment.
- NEXT F is the last line of the loop and increments the counter F by the STEP value.

You can also decrement the counter (decrease it). For example:

```
10 FOR F=100 TO 0 STEP -2
(where the decrement is 2).
```

The loop will be exited in the first example when $F > 100$ and in the second when $F < 0$. F will take values 0, 2, 4 ... 98, 100 in the first case, and 100, 98 ... 4, 2, 0 in the second. Any program lines in the body of the loop will be repeated each time the program loops.

Try these simple examples:

```
10 FOR F=2 TO 4 STEP 1.3
20 PRINT F
30 NEXT F
```

```
10 FOR F=4 TO -1 STEP -1
20 PRINT F
30 NEXT F
```

```
10 FOR F=-2 TO 4 STEP 2
20 PRINT F
30 NEXT F
40 PRINT
50 PRINT "F EQUALS ";F;" ON EXIT"
```

Convince yourself that this does not work:

```

10 FOR F=2 TO 4 STEP -1
20 PRINT F
30 NEXT F

```

The next one is an interesting example of the inaccuracies in the computer's arithmetic:

```

10 FOR F=1.2 TO -0.3 STEP -0.2
20 PRINT F
30 NEXT F

```

```

10 FOR N=1 TO 15 STEP 1
20 PRINT N,N*N
30 NEXT N

```

You can use the value of the control variable in calculation within the loop. Edit STEP 1 so that line 10 reads:

```

10 FOR N=1 TO 15

```

and run it again. STEP may only be omitted for a step of +1. In the program, line 10 allows N to go from 1 to 15 with a step value of 1. That is to say, N takes the values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 each time performing the calculations within the loop.

The next program illustrates the use of different values for the step. The value can be positive or negative, integer or non-integer. In the case of decimal increments or decrements there is the possibility of rounding errors if the loop is executed many times – it is therefore advisable to use integer values for the step and divide by the appropriate power of ten, if the loop variable is to be used in calculations. If this were done in the program below, lines 130 and 140 would read:

```

130 FOR N= 10 TO 56 STEP 7
140 PRINT N/10; TAB(8);(N/10)↑2

```

The program calculates squares and cubes for:

```

1, 4, 7,.....31 (line 20)
120, 115, 110,.....60 (line 70)
1, 1.7, 2.4, .....5.6 (line 130)

```

```

5 REM**MULTILOOP**
10 PRINT "NUMBER"; TAB(8); "SQUARE"; TAB(20); "CUBE"
20 FOR N=1 TO 31 STEP 3
30 PRINT N; TAB(8); N↑2; TAB(20); N↑3
40 NEXT N
50 PRINT "PRESS ANY KEY TO CONTINUE"
55 GET A$: IF A$="" THEN 55
57 PRINT ""

```

```

60 PRINT "NUMBER"; TAB(8); "SQUARE"; TAB(20)
; "CUBE"
70 FOR N=120 TO 60 STEP -5
80 PRINT N; TAB(8); N↑2; TAB(20); N↑3
90 NEXT N
100 PRINT "PRESS ANY KEY TO CONTINUE"
105 GET A$: IF A$="" THEN 105
110 PRINT ""
120 PRINT "NUMBER"; TAB(8); "SQUARE"; TAB(20)
); "CUBE"
130 FOR N=1 TO 5.6 STEP 0.7
140 PRINT N; TAB(8); N↑2; TAB(20); N↑3
150 NEXT N

```

In the next program, the total is represented by T, which is initialised equal to zero (line 10). Each time the program goes through the loop the INPUT number is added to T (line 40) so that when the loop (lines 20 to 50) is exited, T represents the sum of the ten numbers input. The program evaluates the average by dividing the total by the number of numbers input.

```

5 REM**AVERAGE**
10 T=0
20 FOR N=1 TO 10
30 INPUT X
40 T=T+X
50 NEXT N
60 PRINT "AVERAGE = "; T/10

```

The next program illustrates a loop used to print a table. In this case a heading is given (line 70) and this must be outside the loop, as it is only required at the beginning. All names and ages are required to be tabulated, so the print statement doing this (lines 140, 150) must be within the loop. Finally, the average age, to be printed underneath, is required, so the print statement (lines 170, 180) is inserted after the loop has been completed.

```

10 REM**LOOPS3**
20 PRINT ""THIS PROGRAM PRINTS OUT THE NA
ME AND AGE OF A GROUP OF PEOPLE AND ";
25 PRINT "WORKS OUT THE AVERAGE AGE"
30 PRINT
40 INPUT "ENTER NUMBER IN GROUP"; X
50 T=0
60 PRINT "NAME", "AGE"
70 FOR N=1 TO X
80 INPUT "ENTER NAME "; N$
90 INPUT "ENTER AGE "; A
100 T=T+A
110 PRINT N$, A
120 NEXT N
130 PRINT "AVERAGE AGE = "; T/X; " YEARS"

```

L9: Loops of variable length

The first value, final value and step of a loop may have any values (including variables, which may be specified using INPUT). The first example shows a simple program which allows all conditions in the FOR statement to be specified using the INPUT statement.

```
10 REM**VARLOOP**
20 INPUT"TYPE INITIAL VALUE";I
30 INPUT"TYPE FINAL VALUE";F
40 INPUT"TYPE STEP";S
50 PRINT"X", "X2+4*X-3"
60 FOR N=I TO F STEP S
70 Y=N2+4*N-3
80 PRINT N,Y
90 NEXT N
```

It is important in such calculations to avoid the case where division by zero occurs. A simple example of how this may be done (line 40) is shown below.

```
5 REM**DIVZER**
10 PRINT"X", "1/(X-3)
20 PRINT
30 FOR N=-9 TO 15 STEP 3
40 IF N-3=0 THEN 80
50 Y=1/(N-3)
60 PRINT N,Y
70 GOTO 90
80 PRINT N,"INFINITY"
90 NEXT N
```

The final program in this section illustrates another way of having a variable loop size. The operator may use this program for any number of numbers between 1 and 100. A marker (in this case -1) is set to indicate when the input is complete, allowing a jump out of the loop (line 69). This is a *dummy value* - a value not normally entered.

Remember: do not jump into the middle of a loop. A loop must always be entered from the FOR statement.

```
5 REM**STDDEV**
10 T=0:S=0:C=0
20 PRINT"THIS PROGRAM WORKS OUT AVERAGE
AND STANDARD DEVIATION OF A SET OF";
30 PRINT" NUMBERS"
40 PRINT"THIS PROGRAM WORKS OUT AVERAGE
AND STANDARD DEVIATION OF A SET OF";
45 PRINT" NUMBERS"
50 PRINT
60 PRINT"ENTER NUMBERS ONE AT A TIME, TY
PE -1 TO FINISH"
70 FOR N=1 TO 100
```

```

80 INPUT X
90 IF X=-1 THEN 140
100 T=T+X
110 S=S+X↑2
120 C=C+1
130 NEXT N
140 PRINT
150 PRINT "AVERAGE IS "; T/C
160 PRINT "STANDARD DEVIATION IS "; SQR(S/
C-(T/C)↑2)

```

The procedure used in this program can confuse the flow of a program and must be used with care. It is useful on occasion, but it is preferable to have only one entry and one exit from a loop. In this program, the loop may be exited from line 90 in addition to the normal termination, when $N > 100$.

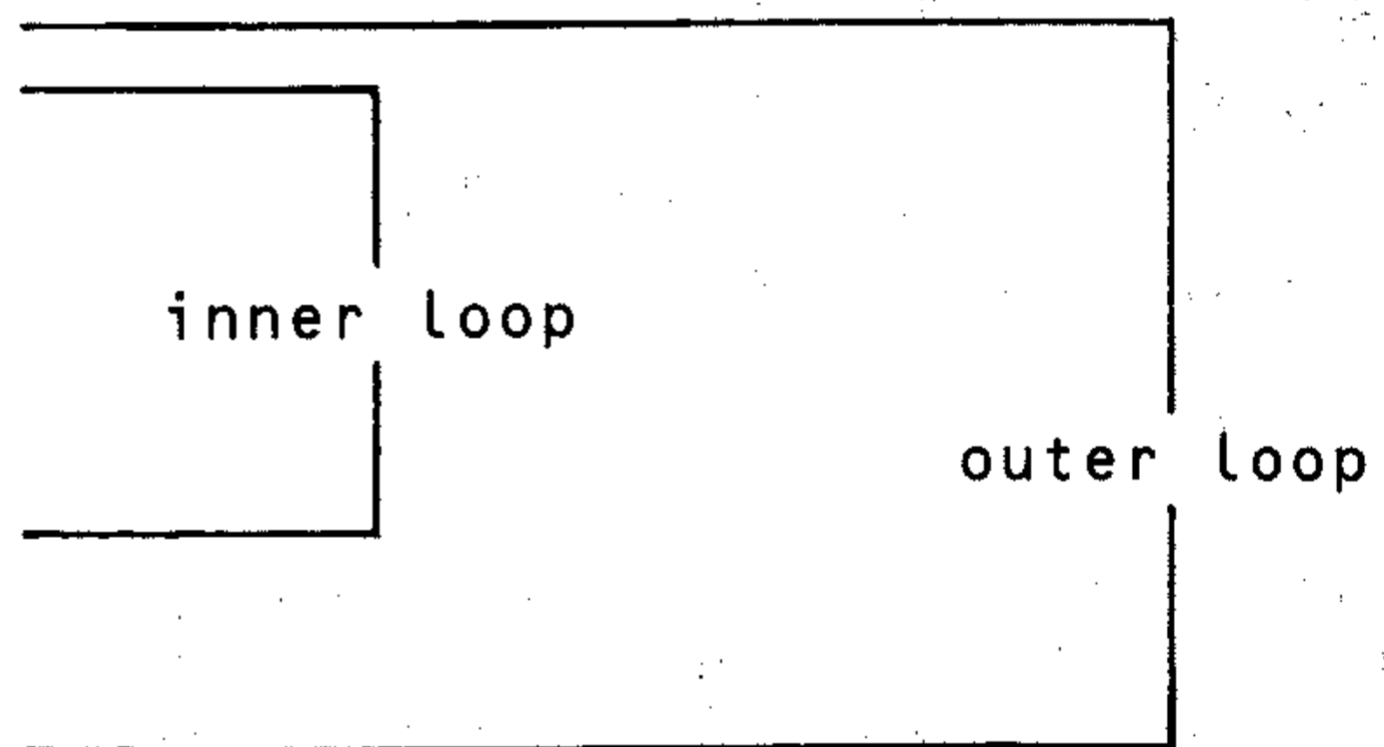
L10: Nested loops

You can place one loop inside another loop, so that every time the program goes through the outside loop, it will perform the inner loop sequence. The inner loop must be entirely within the outer loop. Loops are said to be NESTED one inside the other. Loops can be nested to any depth; that is, you can have as many loops as you wish, as long as they're correctly arranged.

```

30 FOR A = 1 TO 6
40 FOR B = 1 TO 3
.....
.....
80 NEXT B
.....
120 NEXT A

```



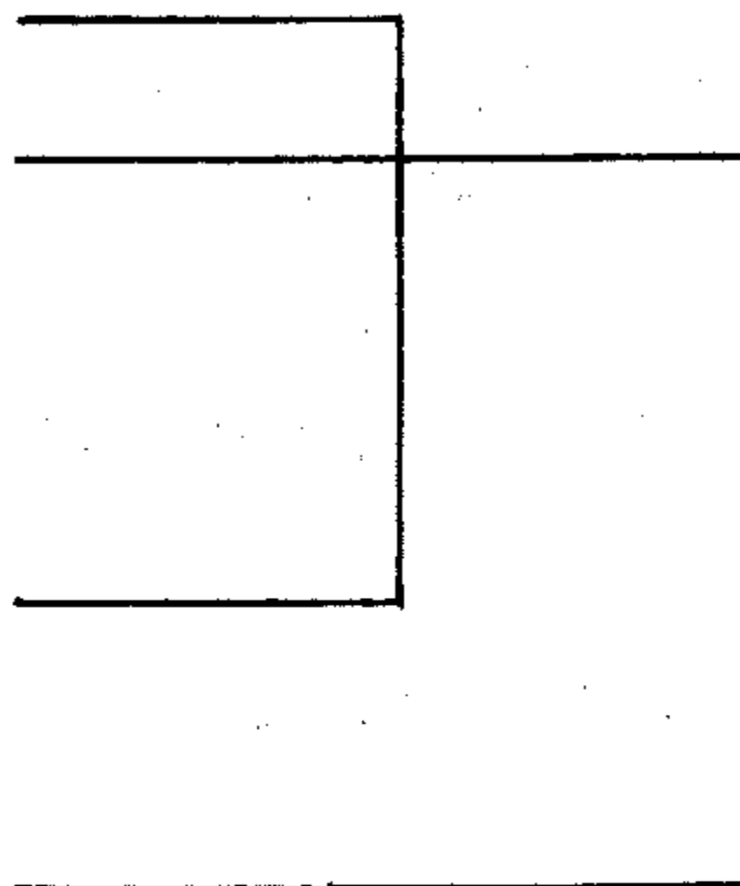
To have a third loop correctly placed, it would have to be inside the B (inner) loop, or outside the A (outer) loop. Crossing the loops must be avoided.

A program like this would run without giving you an error message, but it will not give you the correct answers:

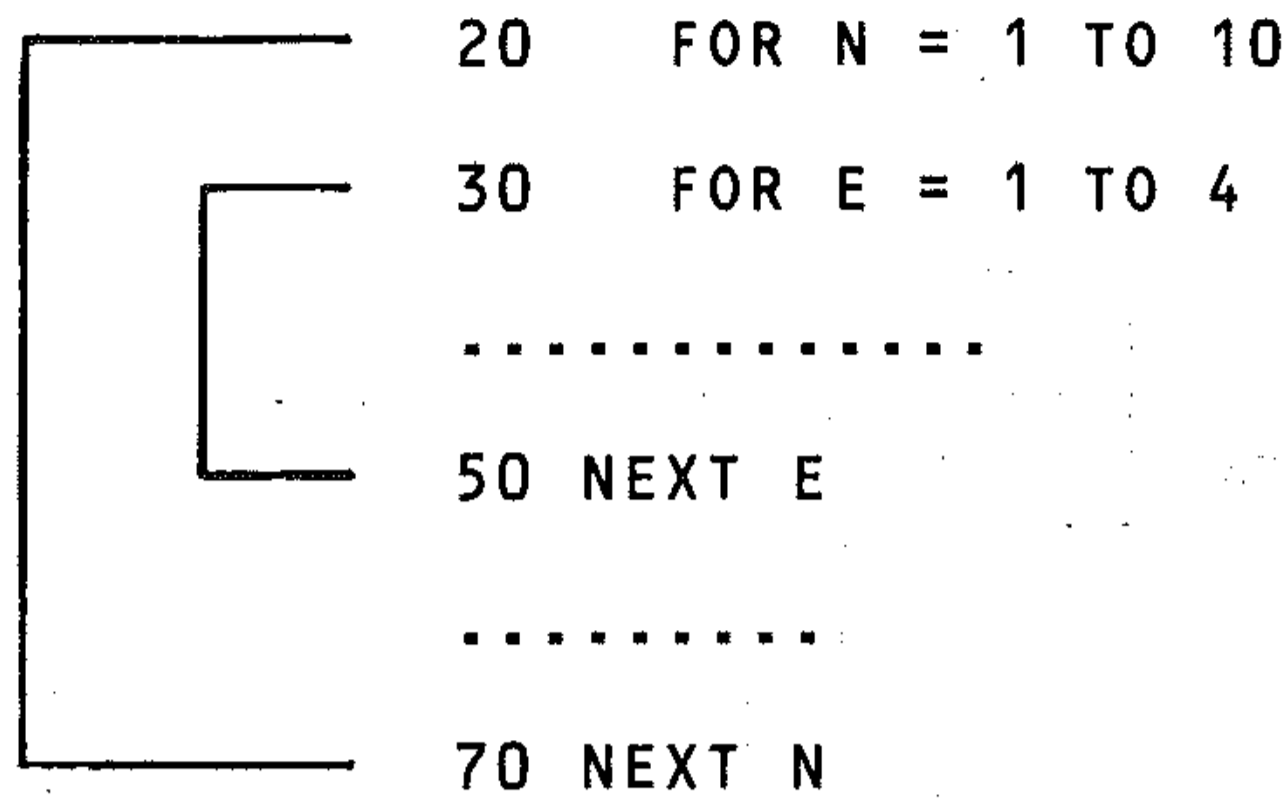
```

10 FOR A = 1 TO 6
20 FOR B = 1 TO 3
.....
.....
60 NEXT A
.....
80 NEXT B

```



To illustrate the use of nested loops, here are two programs. The first evaluates and prints out the squares, cubes and fourth powers of the first ten integers. Each number ($N = 1$ TO 10) is to be raised to the appropriate power ($E = 1$ TO 4). Note that the loops are correctly nested.



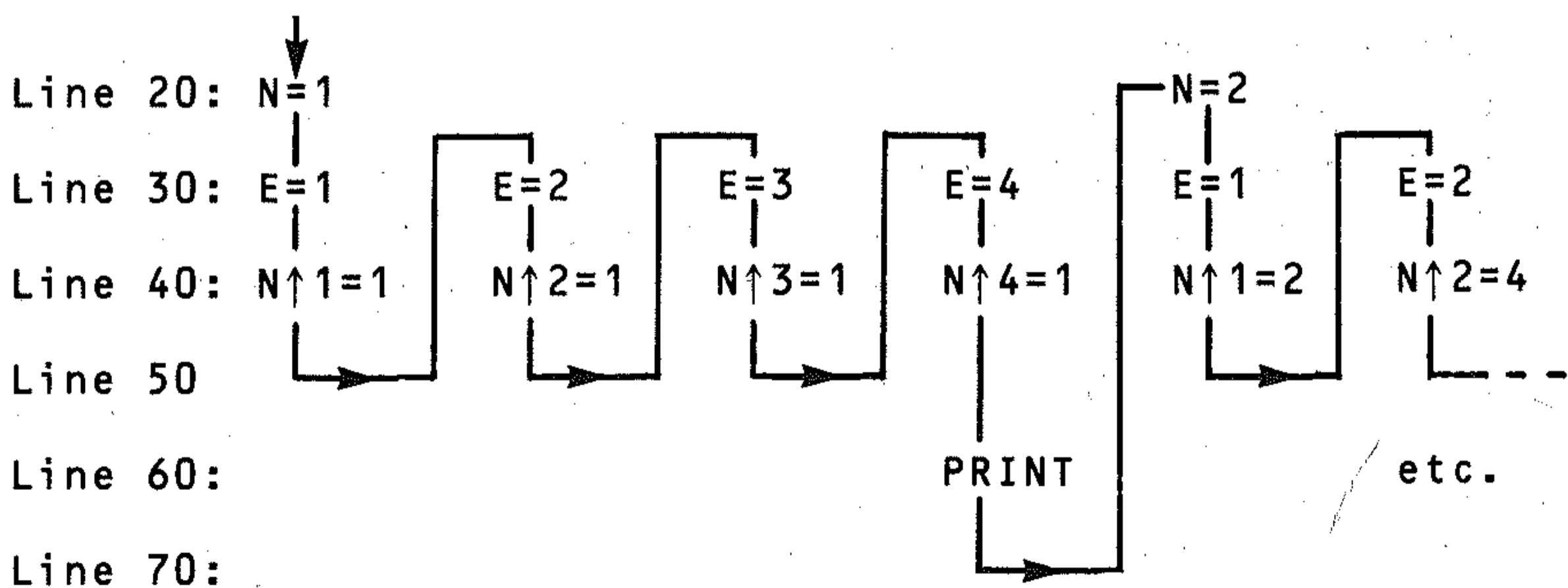
```

5  REM**NEST1**
10 PRINT "NUMBER"; TAB(7); "SQUARE"; TAB(14)
   ); "CUBE"; TAB(21); "4TH POWER"
20 FOR N=1 TO 10
30 FOR E=1 TO 4
40 PRINT TAB((E-1)*7); INT(N↑E);
50 NEXT E
60 PRINT
70 NEXT N
  
```

You will get a printout that starts off like this:

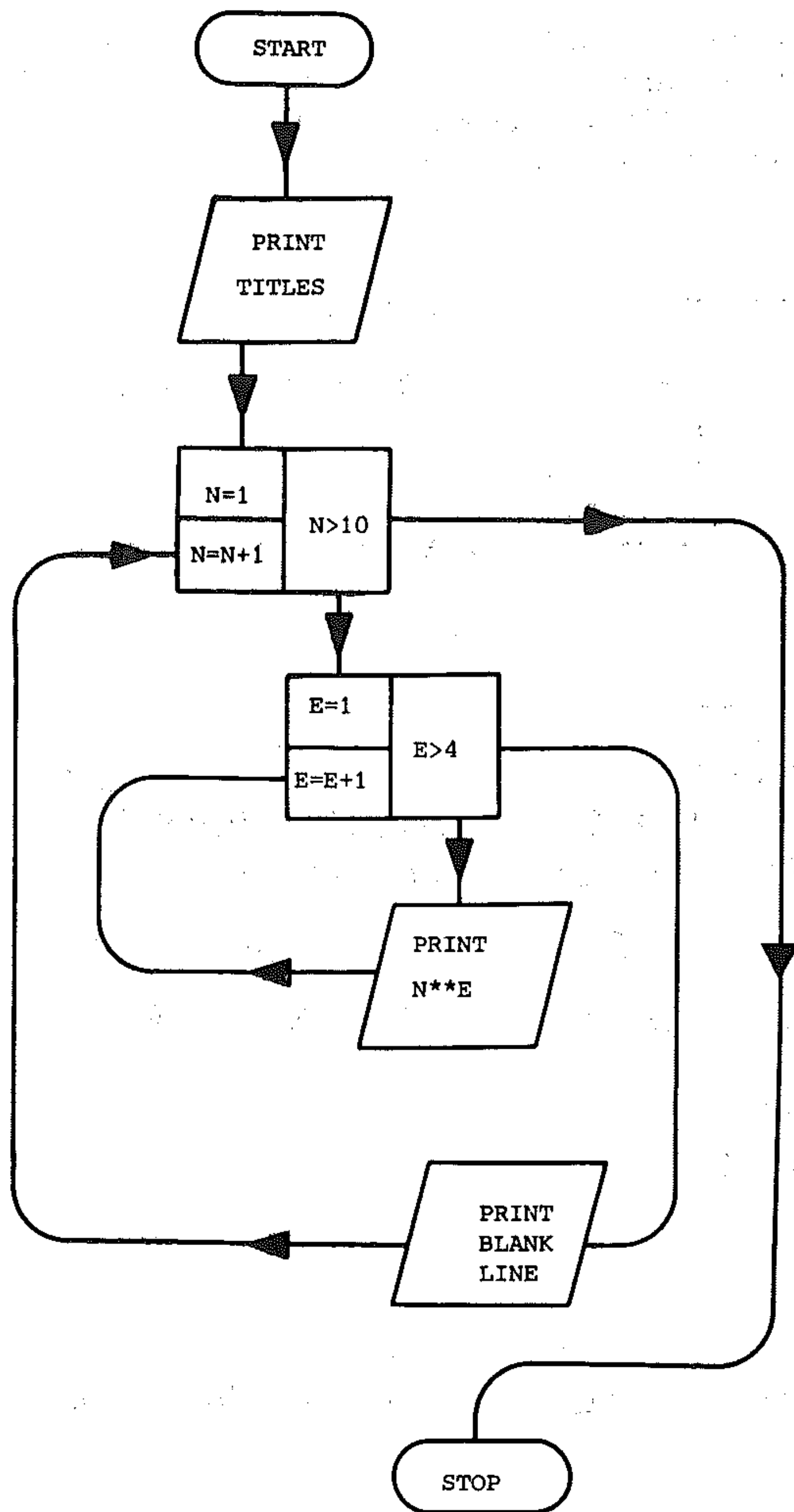
NUMBER	SQUARE	CUBE	4th POWER
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256

You can see the sequence of operations by tracing the program.



Notice how line 40 uses the value of E to format the output.

The flowchart for this program, using the flowchart symbol already introduced for FOR...NEXT loops, will look like this:



EXERCISES

Write programs using loops to perform these operations:

- Calculate the reciprocals ($1/N$), logarithms $\text{LOG}(N)$ and cubes (L^3) of even numbers between 20 and 36 and print them out in a table.
- An object is dropped, and the variation of distance s with time t is given by $s = 4.9t^2$. Print a table of the distances fallen for each second from 1 to 15 seconds.
- Evaluate and print the values of $3X^2 + 4X - 7$ for values of X between 6 and 8 in steps of 0.25.
- Evaluate $\text{SIN}(X)$ for values of X from 0° to 360° in intervals of 10° . Remember you must convert from degrees to radians. Print out the results in two columns.
- Print a table of the discount at 5%, 10%, 15%, 20% on articles from \$100 to \$200 in steps of \$10.
- Find the sum of all odd numbers between 39 and 75.
- Find the sum of the series 1, 1, 2, 3, 5, 8 ... (the Fibonacci sequence) to 20 terms. (Each term is the sum of the previous two terms.)

- Find all numbers less than 50 which can be written as the sum of two squares.
(for example, $13 = 2^2 + 3^2$)
- A ball is dropped from a height of twenty metres and rebounds one-half the height on each bounce. What is the total distance it travels? Assume the ball stops bouncing on its hundredth bounce.



Section M: Printing and plotting

M1: The print screen

The print screen is a 25 row by 40 column character grid. Each element of the grid is a character cell. The rows are numbered 0 – 24 vertically from the top, and the columns 0 – 39 from the left horizontally. Each character cell has a row column coordinate. Characters are printed on the screen using the PRINT instruction. The print screen and the plot screen (which is used for high resolution graphics) are the same areas of the screen, but are accessed through different coordinate systems. The plot screen has a finer gridmesh than the print screen.

M2: The plot screen

The Commodore 64 has available a *high resolution* plot screen. This is composed of 40 bytes across by 25 bytes down the screen. As each byte holds 8 bits, 64,000 bits of information can be used for graphics in the high resolution mode. Each bit makes a *pixel* (or *picture cell*); each pixel in the 320 by 200 pixel screen can be altered by POKEing a 0 (to turn off the pixel) or a 1 (to turn it on) into the respective memory location that is allocated to that pixel. The screen will occupy 8K of memory, so it is necessary to move the screen upwards in memory to display the full graphics capability.

To locate any pixel on the plot screen at point (X,Y), the following formula is used:

ROW=INT(Y/8) finds row
COL=INT(X/8) finds character position
LIN=(Y AND 7) finds line of character position
BIT=7-(X AND 7) finds bit of the character byte
BYT= start of screen memory address + ROW*320 + COL*8 + LIN

The following statement will set the point (X,Y):

```
POKE BYT,PEEK(BYT) OR 2↑BIT
```

Enter the following program and run it:

```
10 XC=150
20 YC=100
30 RD=90
40 POKE56576,(PEEK(56576)AND 252)OR 2
50 POKE53272,8
60 POKE53265,PEEK(53265) OR 32
70 FOR I=0 TO 999
80 POKE 16384+I,230
90 NEXT I
100 FOR I=1 TO 8000
110 POKE 24576+I,0
120 NEXT I
130 GOSUB 1000
140 GET A$:IF A$="" THEN 140
150 GOSUB 2000
160 STOP
```

```

1000 FOR D=0 TO 360 STEP 0.5
1010 RA=D*PI/180
1020 X=XC+RD*COS(RA)
1030 Y=YC+RD*SIN(RA)
1040 X1=INT(X/8):B=7-(X AND 7)
1050 Y1=INT(Y/8):L=Y AND 7
1060 C=Y1*320+X1*8+L
1070 POKE24576+C,PEEK(24576+C) OR 2+B
1080 NEXT
1090 RETURN
2000 POKE 56576,151
2010 POKE 53272,21
2020 POKE 53265,155
2030 RETURN

```

The screen will first show a random picture of coloured objects which will clear. This is done in lines 70 to 120, which clears the screen and colour memory. In line 130 a subroutine is called (line 1000) to plot a circle on the screen. In line 140 the program waits until a key is pressed then goes to 2000 to return to a normal screen. To return to normal screen, hold the **RUN/STOP** key, then depress the **RESTORE** key.

Note: As you will see, it takes quite a long time to clear the screen and do any plotting. In the machine code section of this book, routines will be given to enable you to execute clearing the screen much more quickly.

M3: PRINT definitions

PRINT ITEM (separator) ITEM (separator)... prints ITEMS on the screen. Items are separated by commas or semicolons; these are called *separators*. ITEMS are written to the display file for output to the screen.

; ITEM is printed in the next character position on the screen, for example
PRINT A\$;B

, ITEM is printed at the next 10th column position.

If there is no separator at the end of the **PRINT** statement, the carriage return character is output, and the next printing will occur on the next line.

ITEM

ITEMs can be

- null, ie. nothing
- a number or numerical expression
- a string or string expression
- TAB(N)
- Colour items: using the control key, eg. **PRINT "[CTRL][1]"**

PRINT

PRINT	prints a blank line
PRINT "STRING"	prints a string expression
PRINT A\$	prints a string expression
PRINT CHR\$(A)	prints the ASCII character represented by the numeric variable A.
PRINT 1234	prints number 1234.

PRINT N	prints a number N.
PRINT X 2+Y/3	prints a number.
PRINT "<cursor control keys specifying row and column>"	moves the cursor to the row and column specified, using the cursor commands up, down, left, right and home. See later in this Section on various supplementary methods of formatting output.
PRINT TAB(N)	moves the print position to column N. For example (and remember that semicolons are optional here): <pre>PRINT TAB(5);"NAME";TAB(15);A\$</pre>
PRINT "<colour control keys>"	changes the colour of characters printed on the screen. The effect lasts until the colour is changed. Colour items are set up using the CTRL and number keys.
PRINT SPC(N)	prints N number of spaces.
PRINT POS(N)	returns the current print position.
PRINT "CLR/HOME"	(press HOME/CLR key); clears the screen
LIST	lists a program in memory on the screen starting from the lowest line number onwards.
LIST L1 - L2	lists program lines L1 to L2
LIST - L	lists program lines from start to line L.
LIST L -	lists program lines from line L onwards.

M4: Formatting numbers

Consider a table of numbers containing integer and decimal parts in which the decimal points are aligned and all numbers are printed to a given number of decimal places. Zeros are to be added where required or the number truncated. Signs will be included.

<i>Unformatted</i>	<i>Formatted</i>
3.61	3.610
-21.4	-21.400
2	2.000
.36428	.364

The first method is to set up a *print image*, a standard number format to which all numbers in the table will be printed. In the program below, the print image is created with integer characters *r* and decimal characters *d* (lines 10, 20). The print position for the table starts at *t* (line 40). For each number input (lines 50,60) the number of characters *d* are counted (line 80) to the decimal (lines 100 – 110). Consider 3 conditions:

- the number is integer only (line 130). The required number of zeros to make our number fit the image are added (line 130) using a string of zeros *Z\$* (line 6) and then print the number (line 150) at the top position by adding spaces to fit the image.
- where there are more decimal digits than required (line 135). In this case the length of the number string is reduced and then it is printed.
- trailing zeros need to be added (line 140).

If a number that is out of range is entered the program provides a prompt.

Numbers less than 1

These are printed with no zero before the decimal point: .0N, or .0123. Try this:

```
PRINT .0123
PRINT 0.0123
```

The integer value of .0123 is 0. As a string, 0 has the length 1. The printing of the number .0123 will start one place to the left of the decimal point column, which is incorrect. The function must be corrected for this.

For numbers less than 1 a column must be added to the decimal point position P. This is done by testing if the number is < 1 and subtracting one from the value of the function if it is. This is best done logically:

$+(n < 1)$

$n < 1$ will give the value 1 if TRUE, 0 if FALSE. Try this:

```
PRINT .03 < 1
```

and

```
PRINT 3 < 1).
```

So our function now becomes:

```
(LEN(STR$(INT(n))))+(n < 1)
```

Negative numbers

Negative numbers are <0 but also <1 which has been corrected in the function. It is recorrected by subtracting one from the function if the number is negative, using

$-(n < 0)$

Negative numbers are printed with the sign first. The integer value of a negative number is given as the integer part of a rounded up number:

```
INT(-30.1) = -31
```

The length of the string of this integer value will be 3; that is, it includes the negative sign.

So our function now is:

```
DEF FNj(n) = (LEN(STR$(INT(n)))) + (n < 1) - (n < 0)
```

Zero

The number zero is less than .1, which has been corrected for, so one must be subtracted from our function with:

$-(n = 0)$

The complete justifying function is:

```
DEF FNj(n) = (LEN(STR$(INT(n)))) + (n < 1) - (n < 0) - (n = 0)
```

Choose the decimal point position as column p and print number n as:

```
PRINT N;TAB(p-FNJ(n));n
```

EXERCISES

- Key in and run the example program, then save it.

```
10 DEF FNJ(N) = (LEN(STR$(INT(N)))) + (N < 1) -
(N < 0) - (N = 0)
20 INPUT "INPUT DECIMAL POINT COLUMN >=13
";P:PRINT ""
30 READ N
40 PRINT N;TAB(P-FNJ(N));N
```

```

50 GOTO 30
60 DATA 100.6,0.3,0.5,10.6,-35.9,0.222,1
2345.00,0.1231,0.00

```

Rounding

$DEF FNR(n) = (INT(n*10^{\uparrow d} + .5)) / 10^{\uparrow d}$

where $r(n)$ is the function, n is the number, and d is the number of decimal places. This function rounds numbers to d places of decimals, for example $n=3.4372$ $d=2$.

```

n*10^{\uparrow d} = 343.72
n*10^{\uparrow d} + .5 = 344.22
INT(n*10^{\uparrow d} + .5) = 344
INT(n*10^{\uparrow d} + .5) / 10^{\uparrow d} = 3.44

```

Key in, run and save the following program.

```

10 DEF FNR(N)=(INT(N*10^{\uparrow D}+.5))/10^{\uparrow D}
20 INPUT"INPUT NUMBER OF DECIMAL PLACES"
;D:PRINT" "
30 READ N
40 PRINT TAB(3);N;TAB(15);FNR(N)
50 GOTO 30
60 DATA 3.6322,.0355,25,-23.46,789,-.123
45

```

Using the functions together

Care must be taken here, as the program must justify with the rounded number and print as:

```
PRINT TAB(p-FNJ(FNR(n)));FNR(n)
```

EXERCISE

Produce the following program by merging and editing the two previous programs. Run it and save it.

```

10 DEF FNJ(N)=(LEN(STR$(INT(N))))+(N<1)-(N<0)-(N=0)
20 DEF FNR(N)=(INT(N*10^{\uparrow D}+.5))/10^{\uparrow D}
30 INPUT"INPUT DECIMAL POINT COLUMN >=13"
;P
40 INPUT"INPUT NUMBER OF DECIMAL PLACES"
;D:PRINT" "
50 READ N
60 PRINT N;TAB(P-FNJ(N));FNR(N)
70 GOTO 50
80 DATA 100.6,0.3,0.5,10.6,-35.9,0.222,1
2345.00,0.1231,0.00

```

M5: Word processing

In word processing and computer typesetting, the text to be printed can be formatted or adjusted for line width and the margin space on either side. Additional words and characters can be inserted. Real formatting consists of placing spaces of different sizes in between words to make the words exactly fit the line.

In the word processing program below, once you have entered the left and right margins (the numbers of the columns where printing is to start and stop on each line) you can enter the text that you want printed. The main body of the program starts at line 80, which checks to see if the control keys have been pressed. The next part of the program from line 1000 calculates the number of words that will fit on to the line length specified. This is done as follows:

- Line 1000 checks to see if the number of characters remaining in the original text is less than the length. If it is, the line is printed on the screen; if it isn't, spaces are inserted between the words in S\$.
- Line 1030 tests to see if the last character minus one is a space. If it is, the original text is altered so that S\$ contains the text up to this character, and B\$ will contain that part of S\$ after the space. Line 1090 then inserts spaces into the text in S\$ so that it can be printed in the specified form.

The POKE and SYS commands in lines 170 and 180 are to move the cursor to the right place on the screen.

```
10 INPUT "SPECIFY LEFT MARGINE: "; L
20 INPUT "SPECIFY RIGHT MARGINE: "; R
30 PRINT ""
35 LI=0
40 IF L>R THEN 20
50 PRINT TAB(L-1); L
60 PRINT TAB(R-3); R
70 PRINT TAB(L); : FOR I=1 TO R-L: PRINT "-"
  ; : NEXT I
80 S$="": B$="": PRINT: PRINT TAB(L);
90 GET A$: IF A$="" THEN 90
100 IF ASC(A$)=13 THEN PRINT: PRINT TAB(L
); : S$="": LI=LI+1: GOTO 90
110 X$=A$
120 IF ASC(A$)<> 20 THEN 150
130 IF LEN(S$)>0 THEN S$=LEFT$(S$,LEN(S$
)-1): GOTO 160
140 GOTO 90
150 S$=S$+X$
160 IF LEN(S$)<(R-L+1) THEN PRINT X$; : GOTO
  90
170 GOSUB 1000: POKE 781,LI+2: POKE 782,L:
  SYS65520: PRINT TAB(L); S$: S$=B$: B$=""
180 LI=LI+1: POKE 781,LI+2: POKE 782,L: SYS
  65520: PRINT TAB(L); S$; : GOTO 90
1000 IF RIGHT$(S$,1)=" " THEN S$=LEFT$(S$
,LEN(S$)-1): RETURN
```

```

1010 I=R-L+1
1020 I=I-1
1030 IF MID$(S$,I,1)=" " THEN 1040
1035 GOTO 1020
1040 SP=R-L+1-I
1050 B$=RIGHT$(S$,SP)
1060 I=I-1
1070 S$=LEFT$(S$,I)
1080 I=I-1
1090 IF MID$(S$,I,1)<>" " THEN 1110
1100 IF MID$(S$,I-1,1)<>" " THEN S$=LEFT$(S$,I)+" "+RIGHT$(S$,LEN(S$)-I):SP=SP-1
1110 IF SP=0 OR I=1 THEN 1120
1115 GOTO 1080
1120 IF SP>0 THEN I=LEN(S$):GOTO 1080
1130 RETURN

```

EXERCISES

- Key in and save the program.
- Set different margins, and study the method using ragged and formatted text.
- Think about how you might modify the program to delete and insert words or characters.

Section N: Subroutines

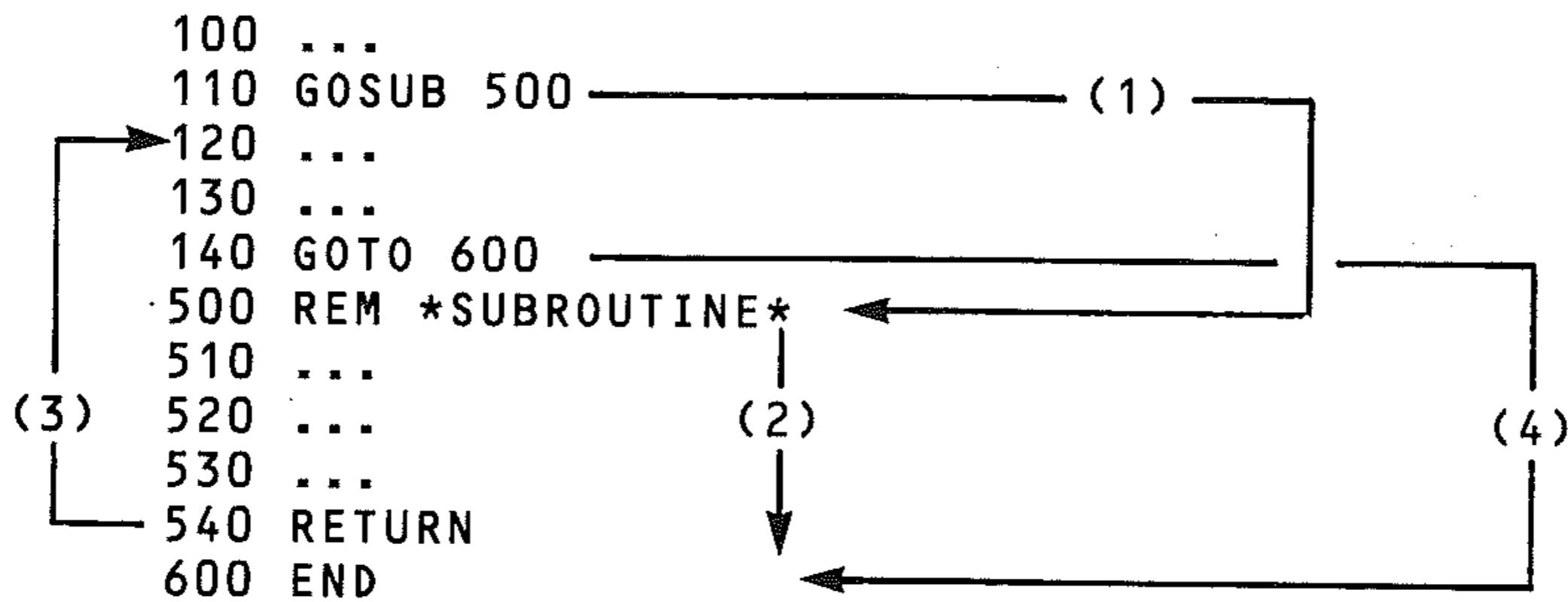
N1: Subroutines

A subroutine in BASIC is a program module performing an allotted task and is entered using a GOSUB statement. The section of the program is exited by a RETURN statement, which sends the computer back to the line following the GOSUB statement. A subroutine must *only* be entered via a GOSUB statement and exited by a RETURN statement.

So these two instructions are used to create subroutines:

GOSUB (line number) transfers control to the specified line number
RETURN leaves subroutine and returns control to the line immediately after the GOSUB instruction which transferred control to the subroutine.

Here is an example of the program structure for these instructions:



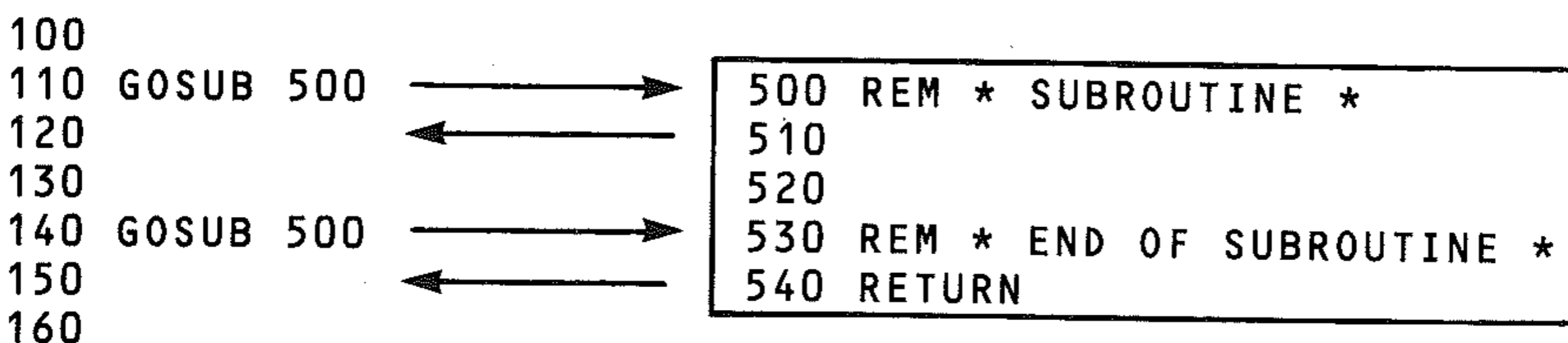
- GOSUB goes immediately to line indicated (500)
- continues program (lines 500 – 540) until RETURN reached
- program returns to line 120 (the line immediately after the GOSUB statement).

It is vital to ensure that a subroutine is not entered accidentally when writing a program. Note that line 140 does this by using a GOTO statement to bypass the subroutine. Line 600 could continue the program or end it; it is good practice to end a program with the highest number, but in this case 140 STOP could also be used.

Subroutines are often used for repeated procedures and are unique program structures:

Main routine

Subroutine



The computer stores the line number of the GOSUB instruction (whereas it doesn't with a GOTO). The RETURN instruction transfers control back to the line number after the latest GOSUB. As shown above, this means that you can enter a subroutine repeatedly in the course of a program.

N2: Subroutine example

The example program given below evaluates the circumference and area of a circle, and has a subroutine to round the results to two decimal places. The program works as follows:

- calculates the circumference (line 40), and makes this figure equal to variable Z (line 45), which is the variable the subroutine will round.
- enters subroutine (line 50).
- corrects answer to 2 significant figures (subroutine lines 200 – 230), and returns with rounded value of Z to line 60, which
- prints out circumference (line 60).

The same procedure is then repeated for the area, the subroutine being entered (called) again in line 90. Lines 200 to 230 of the program are then executed again, but the RETURN statement this time returns control line to 100 (the next line after the last GOSUB statement).

It is essential to have line 110, which prevents the subroutine being entered accidentally when the calculation is complete.

```

10 REM**CIRCLE**
20 INPUT"ENTER RADIUS";R
30 PRINT""RADIUS IS ";R
40 C=2*PI*R
45 Z=C
50 GOSUB 200
60 PRINT"CIRCUMFERENCE IS ";Z
70 A=PI*R^2
80 Z=A
90 GOSUB 200
100 PRINT"AREA IS ";Z
110 GOTO 300
120 REM**MUST NOT ENTER A SUBROUTINE EXC
EPT BY A GOSUB**
200 REM**SUBROUTINE TO CORRECT TO TWO DE
CIMAL PLACES**
210 Z=INT(100*(Z+.005))
220 Z=Z/100
230 RETURN
240 REM**END OF SUBROUTINE**
300 REM**END OF PROGRAM**

```

The second example is a program to evaluate the sum of the series $1 + 1/2! + 1/3! + \dots + 1/10!$ to 6 decimal places. (The exclamation mark (!) means *factorial*. Factorial 5 (5!) for example is $5 \times 4 \times 3 \times 2 \times 1$.)

In this program there are two separate subroutines, both entered repeatedly. The first is to evaluate the fractions and the second corrects the answer to 6 decimal places. Although it is not essential to use subroutines in such a program it does improve the structure and make it considerably easier to follow the sequence of operations.

```
10 REM**FACTORS**
40 S=0
50 FOR Z=1 TO 10
60 GOSUB 200
70 T=1/X
90 S=S+T
100 PRINTZ;" TH TERM IS ";T
110 NEXT Z
120 GOSUB 300
130 PRINT
140 PRINT"SUM OF SERIES ";V
150 GOTO 400
200 REM**SUBROUTINE FACTORIAL**
210 X=1
220 FOR N=1 TO Z
230 X=X*N
240 NEXT N
250 RETURN
300 REM**SUBROUTINE TO 6 D.P.**
310 V=INT(1E6*(S+5E-7))
320 V=V*1E-6
330 RETURN
400 REM**END**
```

Results on screen:

```
1 TH TERM IS 1
2 TH TERM IS 0.5
3 TH TERM IS 0.166666667
4 TH TERM IS 0.0416666667
5 TH TERM IS 8.333333334 E -03
6 TH TERM IS 1.38888889 E -03
7 TH TERM IS 1.98412698 E -04
8 TH TERM IS 2.48015873 E -05
9 TH TERM IS 2.75573192 E -06
10 TH TERM IS 2.75573192 E -07
```

```
SUM OF SERIES 1.7182862
```

Trace the program through for the first two terms to ensure that you can follow the flow.

N3: Nested subroutines

This technique is similar to nested loops in that a subroutine is entered from another subroutine. In the simple example given below, the program enters the first subroutine (line 300) and from within this calls up the second subroutine

(line 320 calls up a subroutine at line 400), which is completed and returns (line 420) to the first subroutine which is then completed. See the diagram below of the program flow.

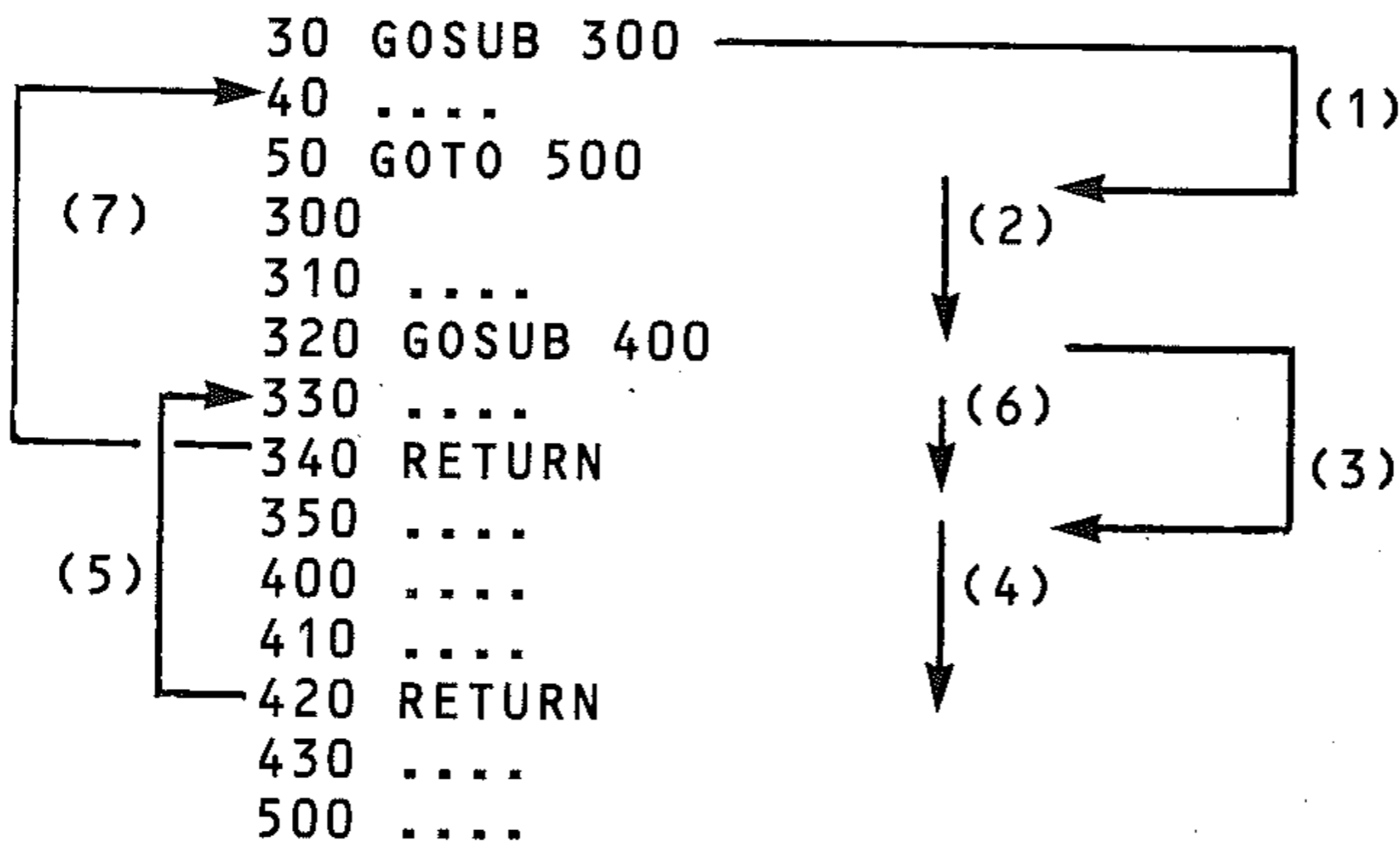
Hand trace this program to discover the result of running it. (*Note:* this program is used only to illustrate nested subroutines; the calculation carried out clearly could be done more easily without them.)

```

10 REM**SUBROUTINE1**
20 M=5
30 GOSUB 300
40 PRINT M
50 GOTO 500
300 REM***1ST SUBROUTINE***
310 M=M+1
320 GOSUB 400
330 REM**RETURN TO MAIN PROGRAM**
340 RETURN
350 REM*****
400 REM***2ND SUBROUTINE***
410 M=M*(M+1)+2
420 RETURN
430 REM*****
500 REM***END OF PROGRAM***

```

The diagram illustrates the procedure in the above program for two nested subroutines.



- 1 Subroutine 1 is called at 30. Enter first subroutine at 300.
- 2 Start executing first subroutine.
- 3 Subroutine 2 is called at 320. Enter second subroutine at 400.
- 4 Execute second subroutine.
- 5 RETURN at 420 returns program to 330.
- 6 Continue execution of first subroutine.
- 7 RETURN at 340 returns program to 40.
- 8 Statement to avoid entering subroutines accidentally.

The second example is typical of a computer games program. The nested subroutine ensures that the computer's move is printed out each time before the player makes his move.


```

10 REM**NESTSUB**
20 REM**PART OF GAMES PROG**
30 X=3
40 GOSUB 600
50 PRINT"YOUR MOVE WAS ";M
60 PRINT"COMPUTER MOVE ";X
70 STOP
600 REM**SUBROUTINE PLAYER**
610 GOSUB 700
620 INPUT"YOUR MOVE";M
630 RETURN
700 REM**SUBROUTINE COMPUTER**
710 PRINT"COMPUTER MOVE ";X
720 RETURN

```

Note that it would make no difference if the nested (called from a subroutine) subroutine were to start at a lower line number than the subroutine which called it. Subroutines are always discrete program modules, wherever they are located in a program.

N4: Recursive subroutines

A recursive subroutine is a subroutine that calls itself. This facility is not available in some versions of BASIC used on other computers. For some purposes it can be a very useful program structure. From within a subroutine, a GOSUB instruction is used to transfer control so that the program re-enters the subroutine. The computer stores each GOSUB call, with the line number to RETURN to, just as if the GOSUB call had been made to a different subroutine. The RETURN instructions are executed in reverse sequence to the order in which the GOSUB instructions were encountered.

The example program below evaluates the factorial of any number N, input as an integer less than 30. First the program, then the explanation:

```

10 REM**RECSUB**
20 INPUT"TYPE A NUMBER LESS THAN 20";N
30 IF N>20 THEN 200
40 GOSUB 100
50 PRINT F
60 GOTO 220
100 REM**SUBROUTINE**
110 IF N<>1 THEN 140
120 F=1
130 GOTO 180
140 N=N-1
150 GOSUB 100
160 F=F*(N+1)
170 N=N+1
180 RETURN
200 REM**END SUB**
210 PRINT"OBEY INSTRUCTIONS":GOTO 20
220 REM***END**

```

To help decipher the program flow, you can insert PRINT statements and add a counter, in order to code the GOSUB and RETURN instructions with a number

to indicate the sequence in which the recursive calls are performed. Add the following lines to the program:

5 C=0	(sets counter to count GOSUB calls)
35 PRINT N	(prints first value of N)
45 C=C+1	(first GOSUB call from main program)
55 PRINT"RETURN TO MAIN PROGRAM"	(final RETURN executed)
145 C=C+1	(increments counter each time GOSUB is used recursively)
146 PRINT"GOSUB CALL";C	(prints each time GOSUB is used recursively)
147 PRINT"N = ";N	(value of N before each recursive GOSUB call)
155 PRINT"RETURN CALL";C	(prints each RETURN call as made, corresponding to the GOSUB call of the same number)
156 C=C-1	(decrements counter as each RETURN is executed)
165 PRINT"F = ";F	(value of F at each stage)
175 PRINT"N = ";N	(value of N at each stage)

Then run the program for N=3. The resulting 'machine trace' screen display is:

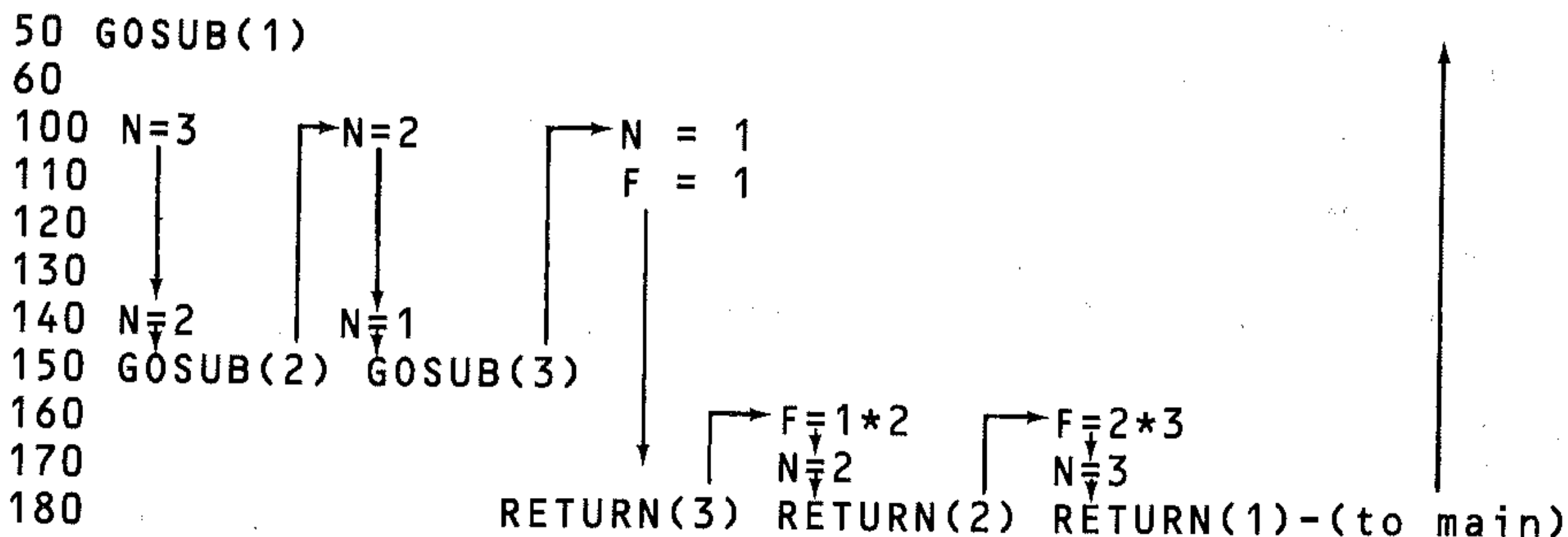
```

3
GOSUB CALL 2
N=2
GOSUB CALL 3
N=1
RETURN CALL 3
F=2
N=2
RETURN CALL 2
F=6
N=3
RETURN TO MAIN PROGRAM
6

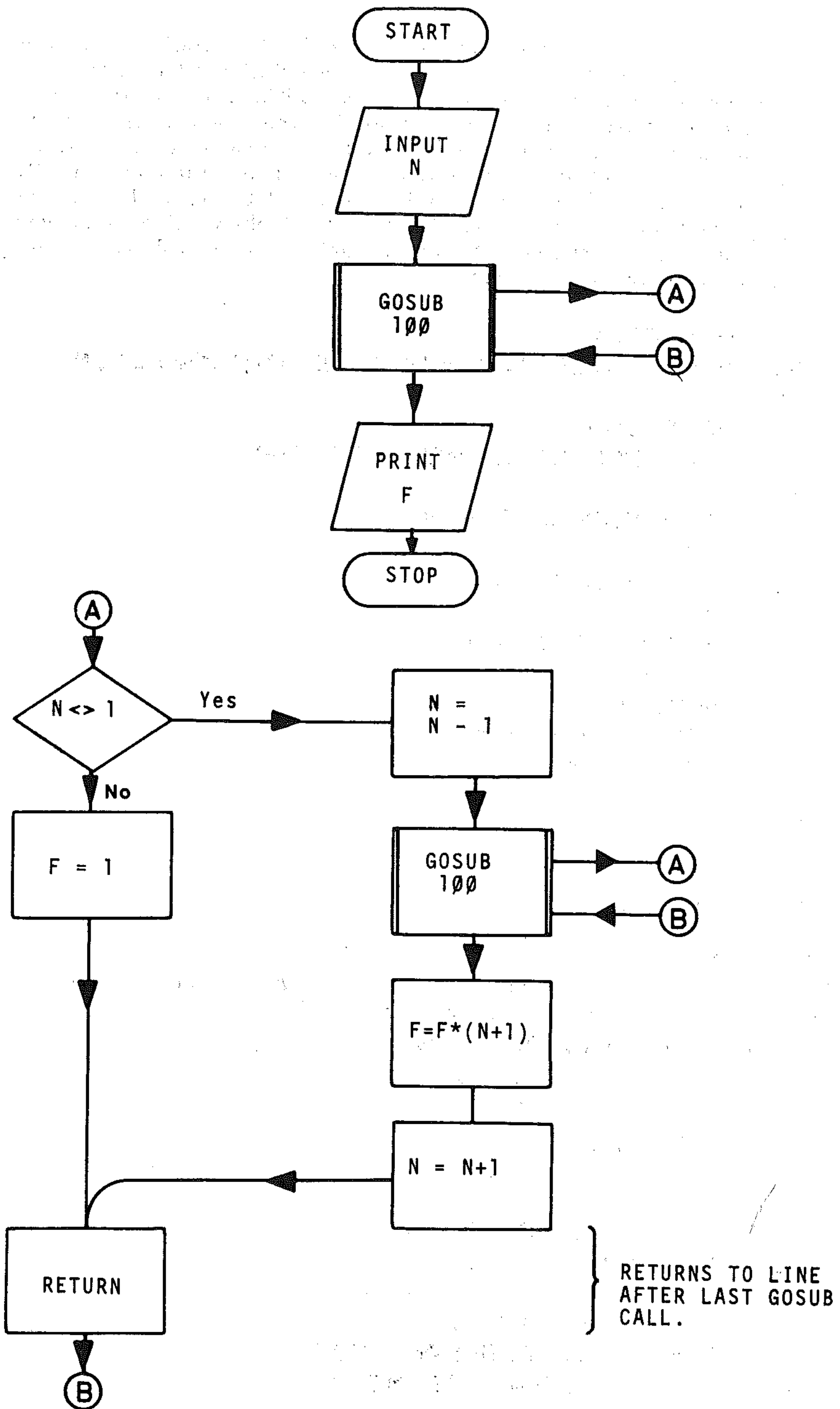
```

If you draw up a trace using the data from this display (as below), you'll see that the GOSUB at line 150 is executed for each value of N from 1 to N. The RETURN calls are then made for each value from 1 to N, calculating F each time (line 160) and incrementing N (line 170), so that the value of factorial N is calculated as $1 \times 2 \times 3 \dots \times N$. The flowchart of this program is quite simple, but the algorithm is not clear unless the sequence of GOSUB and RETURN calls is understood.

The computer stores each GOSUB call in sequence in a portion of memory called the GOSUB stack, and each RETURN instruction removes one of these stored GOSUBs, passing control to the line after the GOSUB call. Confusion is possible with recursive subroutines because the RETURNS are made to the same program line each time (line 160 in this case).



Flowchart: RECSUB



Notice that line 130 passes control to the RETURN statement of line 180. Check for yourself that line 130 could be a RETURN instruction and the program would still run correctly. This is likely to result in a less visible flow in the program, however.

The next program has a subroutine (starting at line 100) which calls itself in line 150. As for the previous program, insert suitable PRINT statements to print out the values of the variables and the number of GOSUB calls made. Hand trace the program for suitable integers, e.g. 15 and 25. The program evaluates the highest common factor of the two numbers input. Note that in this case there are no processing statements between the GOSUB call in line 150 and the RETURN instruction of line 160. The sequence of RETURNS will be executed by control going repeatedly to line 160 (the line after the GOSUB call), which does the next RETURN, until the last stored GOSUB is encountered, which will pass control back to line 50 of the main program.

```
5 REM**HCF**
10 INPUT"ENTER TWO POSITIVE INTERGERS";M
,N
20 GOSUB 100
30 PRINT"HCF OF M AND N IS ";P
40 END
100 REM**SUBROUTINE**
110 P=N
120 N=M-N*INT(M/N)
130 M=P
140 IF N=0 THEN 160
150 GOSUB 100
160 RETURN
200 REM**END SUB**
210 REM**END**
```

N5: ON GOSUB

```
ON nv GOSUB L1, L2, L3, .....
```

This allows several possible transfers of control to a subroutine, depending on the value of the numerical variable nv. For example:

```
10 ON X GOSUB 1000, 3000, 1500
```

transfers control to subroutine:

```
at line 1000 IF X = 1
at line 3000 IF X = 2
at line 1500 IF X = 3
```

```
10 PRINT"MAIN PROGRAM"
20 INPUT"ENTER 1 OR 2 ";X
30 IF X<1 OR X>2 THEN 20
40 ON X GOSUB 100,200
50 PRINT"MAIN PROGRAM ENDS"
60 PRINT"NOW BACK TO MENU"
70 PRINT
80 GOTO 10
```

```

100 PRINT "FIRST SUBROUTINE"
110 RETURN
200 PRINT "SECOND SUBROUTINE"
210 RETURN

```

This technique can be combined with the use of GET to give an instantaneous jump to the required subroutine. The program waits for a key to be pressed and then jumps to the required subroutine.

```

10 PRINT "ENTER 1 OR 2 "
20 GET A$: IF A$="" THEN 20
30 IF A$<>"1" AND A$<>"2" THEN 20
40 ON VAL(A$) GOSUB 100,200
50 GOTO 10
100 PRINT "FIRST SUBROUTINE"
110 RETURN
200 PRINT "SECOND SUBROUTINE"
210 RETURN

```

N6: Subroutine use

As an example of the use of subroutines, here is a guess-the-number game. The program has three subroutines, one to get the number (lines 160 – 190), one to check the guess (lines 210 – 300), and one for the success message (lines 350 – 410), which sets the marker MARK to tell the main program, which is the loop between lines 50 and 140, whether the number N (computer's number) is the same as G (the player's guess). This defines whether the success subroutine has been called as a result of the conditional test in line 110.

```

5 REM**GUESSNUM**
10 MARK=0
20 TRIES=0
30 PRINT "GUESS MY NUMBER. ", "NUMBER IS
  BETWEEN 1 AND 99"
35 REM**GET NUMBER**
40 GOSUB 150
50 TRIES=TRIES+1
60 INPUT "ENTER YOUR GUESS"; G
80 REM**GOSUB CHECK**
90 GOSUB 200
100 REM**GOSUB SUCCESS**
110 IF DI=0 THEN GOSUB 350
120 REM**CHECK MARK**
130 IF MARK=1 THEN 500
140 GOTO 50
150 REM**GET NUMBER SUBROUTINE**
160 N=INT(RND(1)*99)+1
170 RETURN

```

```

200 REM**CHECK SUBROUTINE**
210 DI=ABS(G-N)
220 IF DI>50 THEN PRINT,,"FREEZING"
230 IF DI>25 AND DI<=50 THEN PRINT,,"COL
D"
240 IF DI>10 AND DI<=25 THEN PRINT,,"*WA
RM*"
250 IF DI>4 AND DI <=10 THEN PRINT,,"**H
OT**"
260 IF DI>0 AND DI<=4 THEN PRINT,,"** *B
OILING* **"
270 RETURN
350 REM**SUCCESS SUBROUTINE**
360 PRINT"#####";TAB(15
);"$ SUCCESS $";TAB(30);"#####";
370 PRINT"#####IT TOOK ";TRI
ES;" TRIES"
380 MARK=1
390 RETURN
500 REM**END/RE-RUN MOD **
510 INPUT"ANOTHER GO ? ";A$
520 IF A$="" THEN 510
530 IF A$="Y" THEN 10
540 PRINT"OK, BYE"
550 END

```

The structure of the program is thus:

Module 1:

- 1 Initialise success marker MARK and variable to store number of guesses made (TRIES)
- 2 Print instructions
- 3 Call GET NUMBER subroutine

Module 2 (main program loop):

- 1 Increment TRIES
- 2 Input guess
- 3 Call CHECK subroutine
- 4 Check if Guess equals Number. If it is, then call SUCCESS subroutine
- 5 Check marker. If Success subroutine has been called (MARK=1), then GOTO END/RERUN module
- 6 Loop back to Input guess again (1)

Module 3 (GET NUMBER subroutine):

- 1 Define random number 1 – 99 as number N
- 2 Return

Module 4 (CHECK subroutine):

- 1 Set variable DIFF equal to ABS difference of guess and number
- 2 Check value of DIFF, print appropriate message
- 3 Return

Module 5 (SUCCESS subroutine):

- 1 Print success message, number of guesses made
- 2 Set MARK equal to 1
- 3 Return

Module 6 (END/RERUN module):

- 1 Print prompt for input
- 2 Input response to Another go? (A\$)
- 3 If replay required (A\$ = "Y") then GOTO Module 1,3
- 4 If A\$ not "Y" then print end message
- 5 Stop

Notice that, although this program has been modularised rather artificially to demonstrate its principles, it consists of an introductory section and a main program loop with both conditional and unconditional calls to subroutines within a short main program loop. This makes the structure of the program clear, and minimises the use of GOTO statements, which would be required in profusion if the program were written in a linear, rather than a modular fashion. It is perfectly possible to write the program in this linear manner, but the structure will not be as visible.

You should also note that the END/RERUN module is not a subroutine, but uses GOTO to pass control to this section from the main program, with a conditional GOTO to pass control back to the main program if required. Conditional GOTOS are preferable program structures to unconditional GOTOS, and while the END module could be a subroutine, RETURNing to the main program loop, further conditions would need to be inserted to pass control to Module 1 for a new number to be defined. The subroutine would also need to be exited by a GOTO for the program to stop. There is another solution, however, involving a nested subroutine, which we will set as an exercise.

EXERCISE

Rewrite GUESSNUM with the END/RERUN module as a subroutine. The procedure should be as follows:

END/RERUN SUB

- Prompt for player input, and get response as before.
- If RERUN not required, bypass 3 and 4 below, by a GOTO the RETURN line.
- GOSUB to GETNUMBER subroutine. This is a nested subroutine. The new value of N will be set by this operation.
- Re-initialise TRIES as 0 and MARK as 0.
- Return.

The main program loop is then returned to. The main program must then test whether it is to exit (rerun not required) or continue (new game started). We could set another marker to test this, but in effect we have done this by re-setting MARK if a rerun is required.

Rewrite the main program loop, so that on return from END/RERUN subroutine, the program loops back only if MARK = 0. If MARK = 1 then the program will not loop back and you can either insert a GOTO to bypass all the subroutines to an end program procedure, or STOP the program before the subroutines.

Insert an additional subroutine which prints

```
1;"ST", 2;"ND", 3;"RD"
```

and then TH for other numbers into the FACTORS program (unit N2: Subroutine example) which you should have saved.

- Write a program which determines how many rolls of a die are required to produce a total score greater than 100. Use subroutines to produce the random numbers for the die rolls and print out the results.
- Let the computer choose a four digit number with no two digits alike. You try to guess the number chosen. The computer indicates H (too high), L (too low) or R (right) for each digit in turn and determines how many guesses are required to get the correct number. Use subroutines to create the number, input the operator's guess and give the response to each guess.

PART THREE

**THE COMPLETE
PROGRAMMING METHOD**

THE UNIVERSITY OF CHICAGO
LIBRARY

1957

Section O: Programming Methods II

O1: A recapitulation

Before you are introduced to advanced BASIC programming, it may be useful to recap what you have studied so far.

The method to design the solution or algorithm to a computational problem using *top down* analysis has been explained. You have seen how to break down a problem into sub-problems which form program modules (using tree diagrams). You know how to describe the algorithm in concise English sentences called pseudocode, and how to determine and illustrate the flow of control in the problem solution by drawing a flowchart. When designing the solution, you recognise the usefulness of the fundamental programming tools of:

- decision making
- branching as a result of decisions
- direct transfer from one point in the algorithm to another
- repetition

These control structures, as they are called, which are present in all computer languages, have been discussed in some depth, together with other important BASIC language fundamentals. The techniques of:

- decision making
- numeric processing
- character handling with strings
- looping through counting and condition testing
- handling of output by printing and plotting

and the realisation of modular techniques in programming by using subroutines have all been covered.

What's next?

The second phase of the programming method is producing the program itself. It is important to do this now, so that the programming tool kit is complete enough to investigate and use the more sophisticated information-handling facilities to be introduced a little later:

- logical operations on data
- character codes
- moving graphics
- graph plotting
- constructing and searching list and data arrays
- how to sort information into order

Once these skills have been mastered, the complete programming expertise can then be applied to real applications. First you must learn how to code the algorithms into BASIC language programs, and then debug, test and document

them. Further important design rules will be introduced, and finally a summary of our complete programming method will be provided with a flowchart and worked example.

So: given an algorithm – which is written in steps in a description called pseudocode – together with a flowchart – which shows how the steps of the solution are combined in sequence for the computer to solve the problem – we must now:

- a code the algorithm in Commodore BASIC
- b debug and test the program
- c document the program

O2: The rules of coding and design

Code on a one-to-one basis.

If the description of the algorithm is correct then coding on an almost one-to-one basis from statements in the pseudocode or the flowchart is possible. If you cannot code from the flowchart or pseudocode then further refinement of the algorithm is necessary.

Pseudocode descriptions in formal mode of the BASIC language control structures for decisions and loops are given later in this section. You will notice that the description itself is concise, with the terms almost the same as BASIC statements. This is not unusual, as BASIC was designed to do this very thing and is English-like in its syntax.

To be able to code at all you must of course

Know the BASIC language and its rules.

It is probably the right language for the job; on the Commodore you don't have much choice! Actually, it is a question of the ease of programming specific applications that generates different languages. Most things can be done in BASIC, although perhaps not efficiently or elegantly. It is often useful to identify the kind of processing that will be required. When designing the algorithm consider whether the problem is a scientific or a business application, whether extensive calculations will be performed or large amounts of list processing done, whether the data is extensively numeric or string, and whether the program will be interactive, with a lot of user dialogue.

When coding, avoid spelling and formatting mistakes.

Define and contain each module with REM statements.

For example:

```
100 REM * SORT MODULE *
200 REM * THIS MODULE SORTS STRINGS *
.....
.....
.....
500 REM * END SORT *
```

Terminate your program properly.

You may have noticed that 64 BASIC does have a special end-of-program statement which is the END statement. You can, however, put one in using a REM statement. For example:

```
REM ** End of program **
```

The Commodore does not process REM statements, but only notes them. When the above line runs, the program will finish elegantly with a READY message. You can also stop a program with the STOP statement. Main modules should finish like this, with subroutines programmed at higher line numbers terminated with a REM statement. When terminated with STOP a message BREAK AT LINE ... will be given.

```

5 REM * NAME OF PROG *
10 REM * MAIN MODULE *
20 GOSUB 500
30 STOP
40 REM * END MAIN *
500 REM * SUBROUTINE *
600 RETURN
700 REM * END SUBROUTINE *
800 REM * END OF PROGRAM *

```

You could also use a GOTO 800 at line 30 to terminate execution on the last program line, or better (because neater) an END statement.

Always code according to the logical order of processing.

This is usually ensured if you code from a flowchart, with your flowchart structured into modules, ie. if you have flowchart groups for the modules in the program. Take care with the control structures and avoid unnecessary branching, especially with GOTO instructions. Try to make your programs both readable and efficient – but above all readable!

User friendly programs

Design your programs with the user in mind – and that includes you! Directions to users should be concise and as few as necessary, both in the program and in the user guide, if your program is large enough to merit one. Where the user needs a number of instructions to operate the program, these can be built into an optional 'help' module or subroutine.

```

100 REM * USER INSTRUCTION *
110 REM * DIRECTS USER TO HELP SUBROUTINE *
120 PRINT "FOR INSTRUCTIONS PRESS H
        PRESS ANY OTHER KEY TO CONTINUE"
130 GET A$: IF A$ = "" THEN 130
140 IF A$ = "H" THEN GOSUB 1000
150 REM * END USER INST *

. . . . .
1000 REM * HELP SUBROUTINE *
. . . . .
. . . . .
1200 RETURN
1210 REM * END HELP *

```

Users need to know:

- how to run the program
- what form of input data is required
- what output is produced

Your program should check on the range and type of input data. If the input data is out of range or incorrect, the program should not stop with an error, but continue with a message to input correct data. After you have designed a program to do a specific task, it is often possible to make it more general – that is, to do several similar tasks. As you become more skilled and confident in programming you will be able to generalise, writing a subroutine that enables users to select options from a menu. This is similar to the exercise you have seen in multiple decision structures. More *user friendly* tips are given in the section on documentation, and useful routines in Unit V2.

Designing program layout

You must make your program readable. The program design will be modular, containing specific identifiable segments, subroutines and modules. These should be labelled in the design of the algorithm and transferred in the coding process.

Each module should be titled or labelled with its function.

For example, as in this program:

```
10  REM "AVERAGE"
20  REM * PROGRAM AVERAGES ANY NUMBERS INPUT *
30  REM
40  REM * USER ROUTINE *
50  REM * CHOICE OF NUMBERS INPUT *
60  INPUT "HOW MANY NUMBERS DO YOU WISH TO AVERAGE"; N
70  DIM A(N)
80  REM * INPUT ROUTINE *
90  REM * NUMBERS INPUT TO ARRAY *
100 PRINT "INPUT NUMBERS"
110 FOR I = 1 TO N
120 INPUT A(I)
130 NEXT I
140 REM
150 REM * PROCESSING ROUTINE *
160 REM * COMPUTES AVERAGE *
170 SUM = 0
180 FOR J = 1 TO N
190 SUM = SUM + A(J)
200 NEXT J
210 AVERAGE = SUM/N
220 REM
230 REM * OUTPUT ROUTINE *
240 PRINT "THE AVERAGE OF"
250 FOR K = 1 TO N
260 PRINT A(K); " ";
270 NEXT K
280 PRINT "IS"; AVERAGE
290 REM
300 REM * END AVERAGE *
```

Design your program so that related statements are together.

For example, input/processing/output statements:

- All input statements will be at the beginning of a simple sequential program, processing in the middle, and output normally at the end.
- For a modular program, input, processing and output routines will be separate modules or groups of statements within a single module.
- Subroutine modules will usually be placed separately at the end of a program.

Insert REM statements between program modules as separators.

Program modules are then easily identified. Use blank REM lines, lines of asterisks, or spaces.

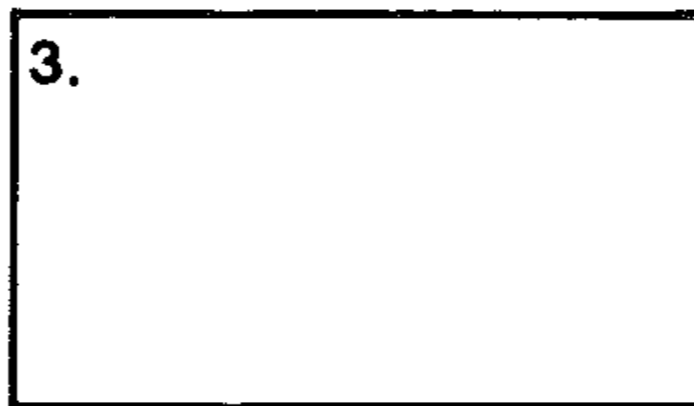
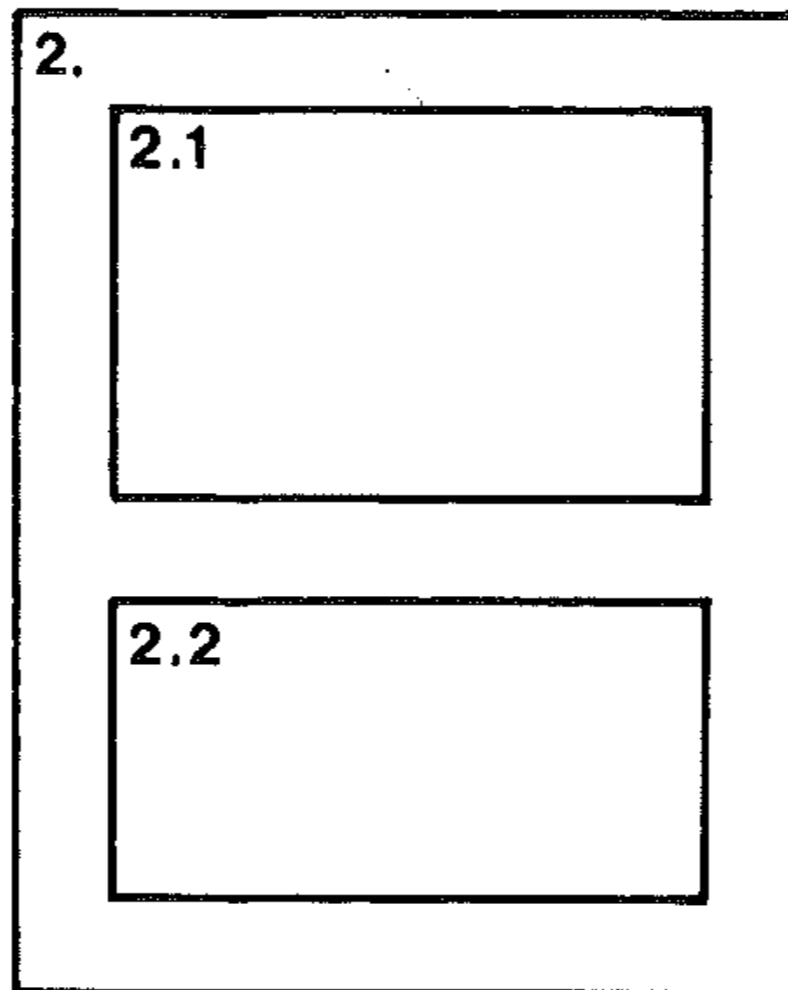
Plan your program layout before coding.

The printed listing of your program is important. Choose a maximum line width. Break longer lines into shorter ones in REM statements by using spaces. Compensate for overrun. For example:

```
REM * AAAAA
AAAAA +
```

Note: remember the maximum line length is 40 characters on the screen, and each BASIC line can only be two lines long.

The layout should reflect the modular structure of the program.



Designing program output

For the user, the output is the most important part of the program. Take time planning it. The output instruction in 64 BASIC is: PRINT, command.

Results should be output with related text.

Label all your numerical output. For example:

YEAR 1974 **NET INCOME** \$5678.65 instead of 1974 5678.65
AVERAGE AGE OF BOYS IS 15 YRS 3 MONTHS rather than 15 3

Display large amounts of output as a table, histogram or graph, and give titles.

For example:

TABLE 1: NET INCOME FOR B. JONES FOR YEARS 1978-1980

YEAR	NET INCOME
1978	\$2018.45

Box your tables if possible. The user should not have to look up the program listing to see what the numbers in the output mean.

Design your output to be easy to read.

Plan it to be attractive to any user of your program including, of course, yourself. Graphics is a powerful tool for this.

Align, space and justify the output.

Plan your output with reference to the screen size and divisions. For tables, align information central to the heading: justify characters left and numbers right, except align signs vertically (there are routines in the text for doing this). For example:

STUDENT CODE	NAME	NUMBERS
876-340	JIM SMITH	15.003
27-210	HUNG FO	815.231
453-003	SARAH JAY	-4.000
1-025	DRACULA	-100.100

Fill in with zeros to get decimal placing correct.

Use space carefully.

Print output horizontally wherever possible. For example:

TABLE OF POWERS OF 2

2
4
8
16
32 etc

should be:

TABLE OF POWERS OF 2

2	4	8	16	32
64	128	256	512	1024

Do not overdo explanations.

Be succinct.

Make your abbreviations clear.

NDTC=NUMBERS OF DAYS TO CHRISTMAS

Display input data as an option.

Allow checking of input data before processing. Make your program check for incorrect or bad input data.

Modular design

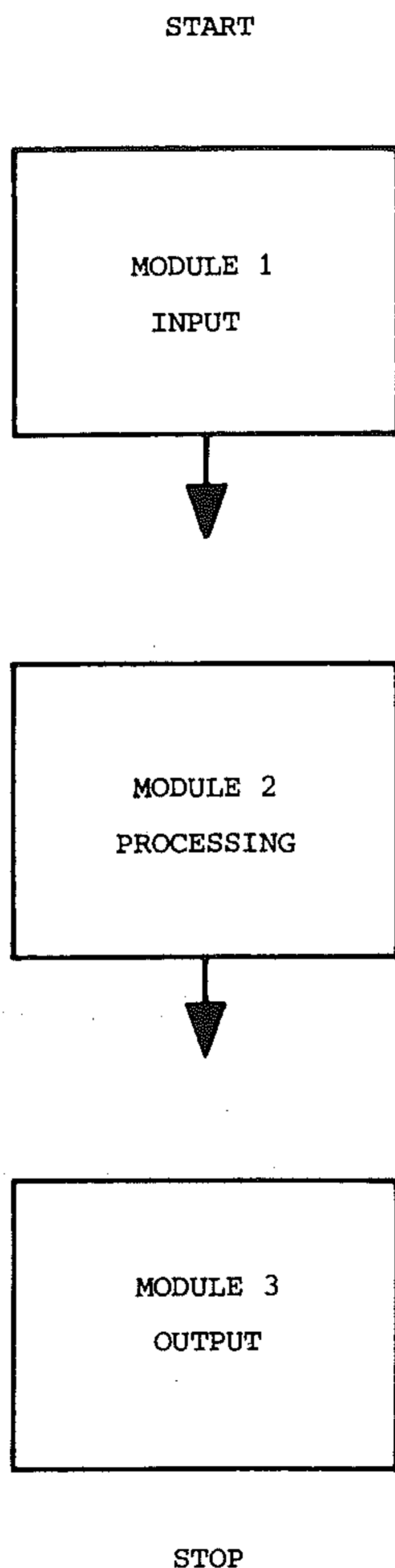
Break problems down into sequences of steps to produce programs in which different kinds of activities are separated out. These distinctive program modules are our *subroutines* or *subprograms*. Each module has its own name and address, but in BASIC they are usually referred to by address only: GOTO 100, GOSUB 3300, where the address is the line number of the first statement in the module.

There are good reasons for modular design and the use of subroutines and sub-program modules. They make the logic of the program – its flow – easier to follow. The clarity of the structure of the main program is improved, while program design is proceeding by referring to the number of the module initially, instead of starting to write out the code of the module at that point. The module can be coded as a separate entity.

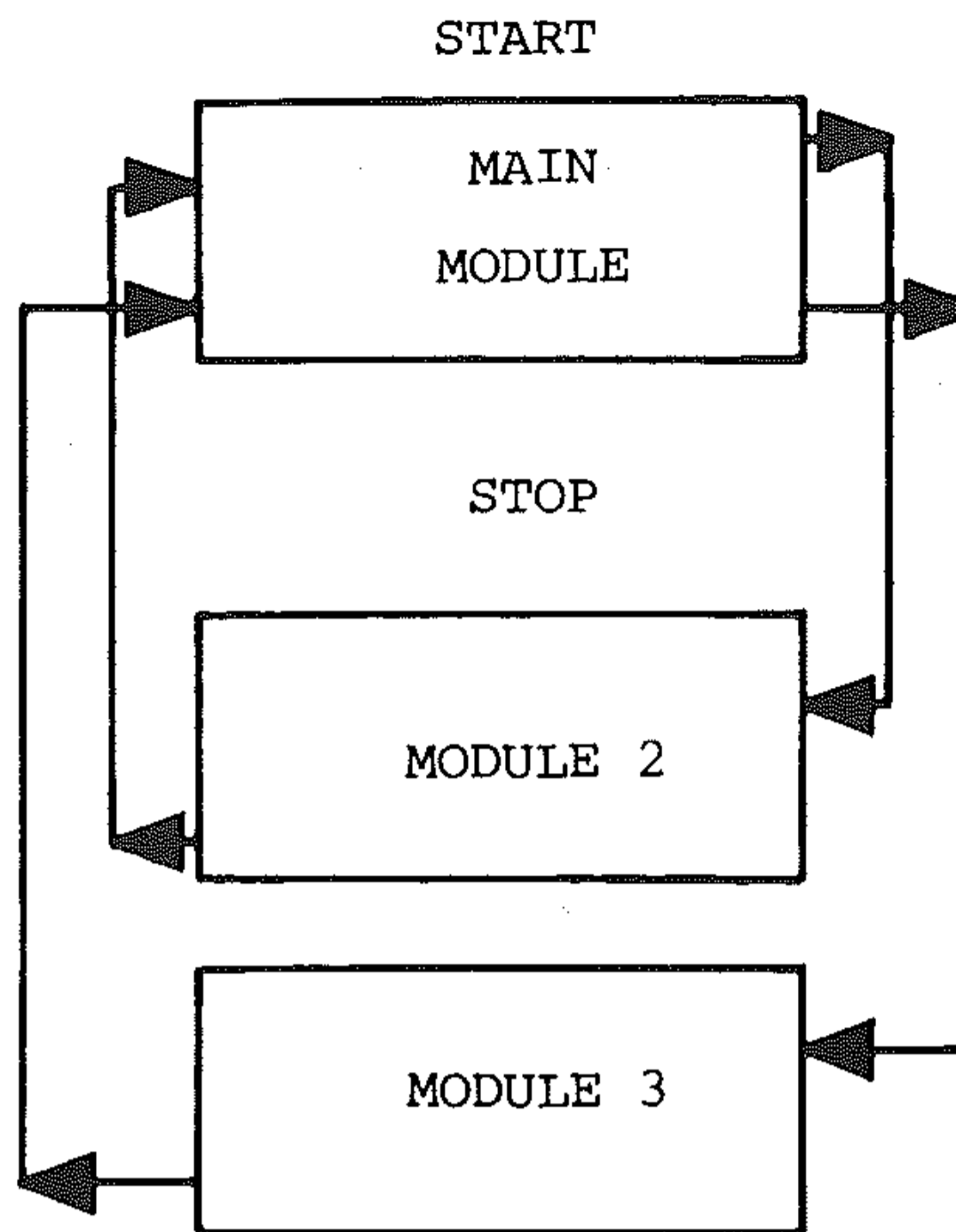
Independent testing of modules is possible, but care must be taken that all variables have been declared and have their correct values at the start of the module. Debugging is simpler with this approach, since the module is isolated. You can leave the coding of a module until a later stage, but you must know what it will do when coded.

If a module has to be used several times in a program from different places it need only be written once, and called into action from these points by reference to its line number.

Program modules can be designed to run sequentially:



This structure is convenient for simple programs. However, programs can be structured in terms of subroutines and subprograms being called from a short and simple main program module.



This structure is convenient for longer, more complicated programs with many modules and nestings. Subroutines automatically return to the next line in the main program through the RETURN statement. Other modules are *called* by GOTO (line number) and *return* by GOTO (line number) instructions. GOTO *must be used with thought and care and not excessively*. Use a GOSUB unless a return to a different point in the main module is needed or a multiple return is possible as a result of a decision to be made within the module.

Nested modules can be treated as other modules and called from within the subroutine or sub-program, by GOSUB and GOTO instructions. Nested loops *must* be contained within the same module, however.

O3: Control structures in 64 BASIC

- Each *control structure* is a *program module*.
- A formal pseudocode description of each structure is given of the general form of the control structure.
- A flowchart description is given of the general form.
- The BASIC version is given of the general form.
- A simple example illustrates the BASIC form of the control structure.
- Structures will be written in indented form in the pseudocode version for clarity. REM statements must be used to show the start and stop lines for program modules.
- P is a *processing operation*. It can be a single instruction, a statement or a group of statements.
- In the formal pseudocode each structure will commence with the title *module* (abbreviated to *mod*), and end with the statement *endmodule* (abbreviated to *endmod*),
- In BASIC each structure will be bounded by REM *STARTMOD* and REM *ENDMOD* statements.
- Flowcharts will be bounded by START and STOP symbols.

The structures summarised are:

Decision structures

- single decision IF...THEN structure
- double decision IF...THEN...ELSE structure
- multiple decision case structure

Loop structures

- repeat-forever structure
- repeat-until structure
- while-do structure
- FOR...NEXT structure

The names of the structures are implemented as actual programming language structures in other languages and some forms of BASIC. The FOR...NEXT structure is a special form of while-do loop, given a specific implementation in BASIC.

Decision structures

Single decision: The IF...THEN structure

IF (condition true) THEN (do something)

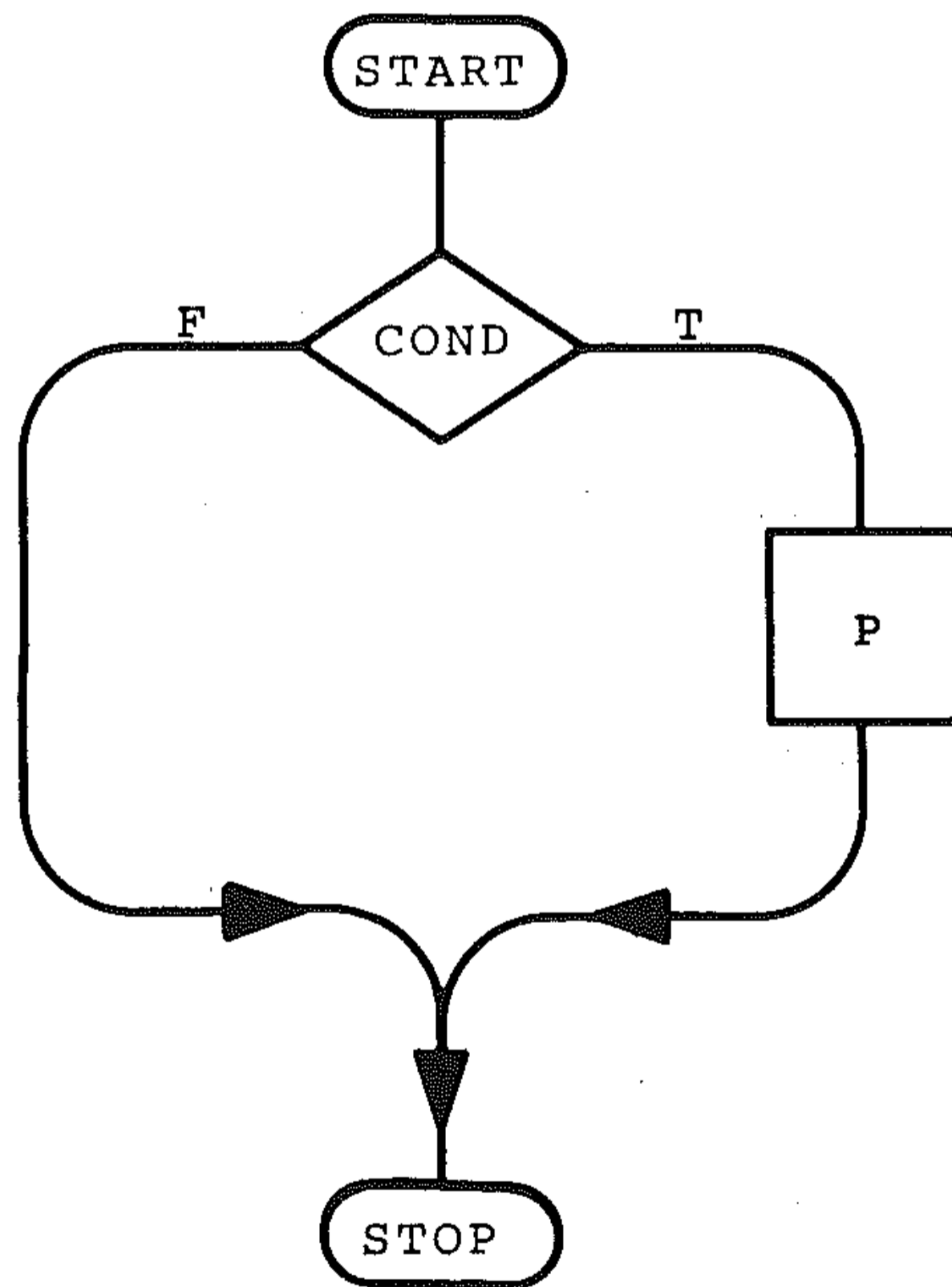
Pseudocode

```
mod
  if(condition)
    then P
  endif
endmod
```

BASIC

```
10 REM * START MOD *
20 IF (COND) THEN P
30 REM * ENDMOD *
```

Flowchart



For example: input a number and if it is positive, print it.

Pseudocode

```
mod
  input A
  if A > 0
    then print A
  endif
endmod
```

BASIC

```
10 REM * START MOD *
20 INPUT A
30 IF A > 0 THEN PRINT A
40 REM * END MOD *
```

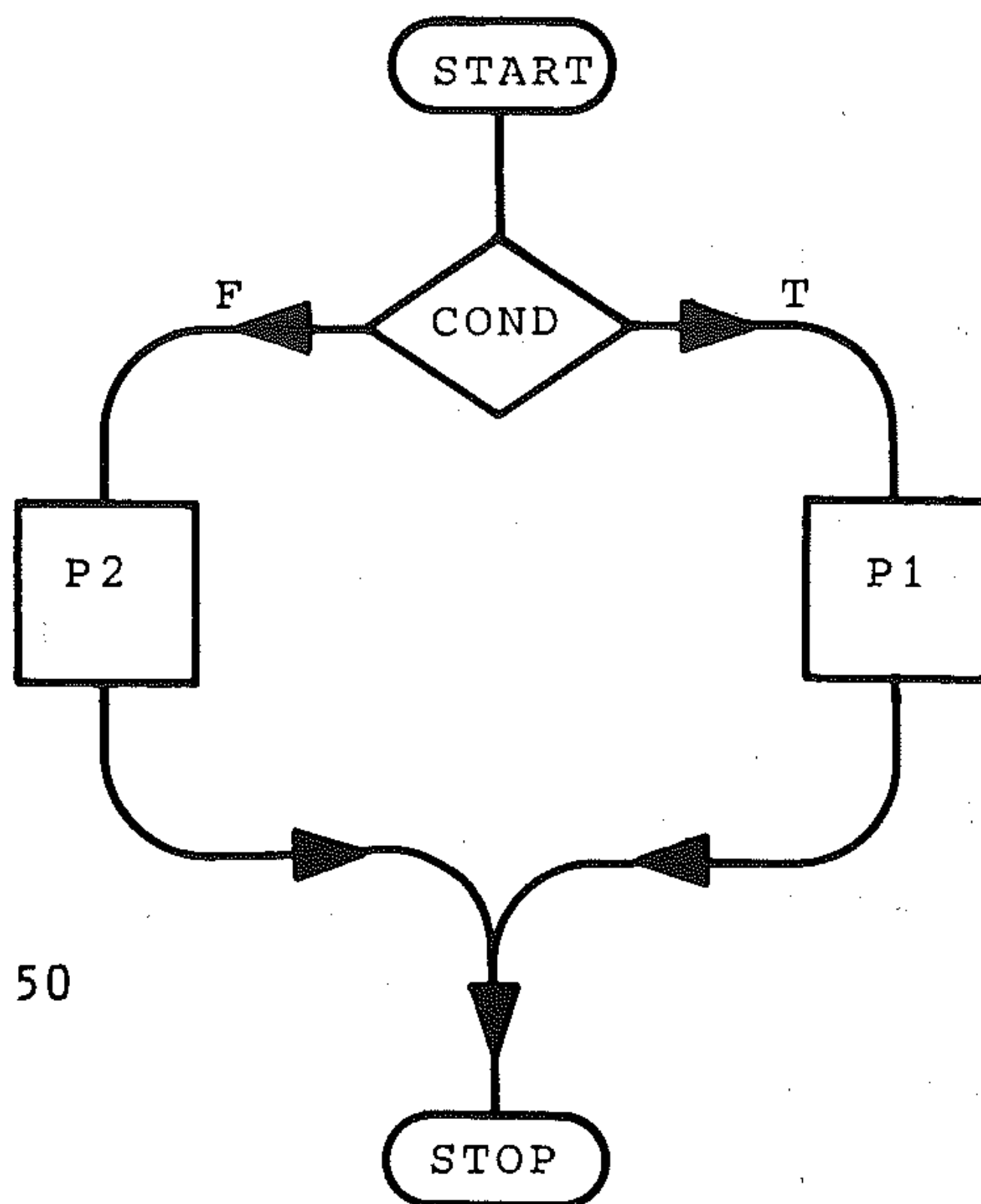
Double decision: The IF...THEN...ELSE structure

IF (condition true) THEN (do something) otherwise (if condition is false) do something else

Pseudocode

```
mod
  if(condition)
    then P1
    else P2
  endif
endmod
```

Flowchart



BASIC

```
10 REM * START MOD *
20 IF (COND) THEN GOTO 50
30 (FALSE TASK P2)
40 GO TO 60
50 (TRUE TASK P1)
60 REM * ENDMOD *
```

To perform the true task (P1 in the pseudocode) first, the BASIC implementation of the structure would test the complement of the condition, so that in the program below, for example, $A > B$ would be replaced by $A < B$, and lines 50 and 70 swapped. Note that the standard form of complement would be $B \leq A$, but we have defined the input numbers as unequal in this case.

For example: input two unequal numbers and print the largest.

Pseudocode

```
mod
input A, B
  if A > B
    then print A
    else print B
  endif
endmod
```

BASIC

```
10 REM * STARTMOD *
20 INPUT A, B
30 IF A > B THEN 70
40 PRINT B
50 GOTO 80
60 PRINT A
70 REM * ENDMOD *
```

Multiple decision structure: The case structure

With this structure the program must select and perform one of several alternative tasks. The conditions in this case structure are sequential, *not* nested and mutually exclusive.

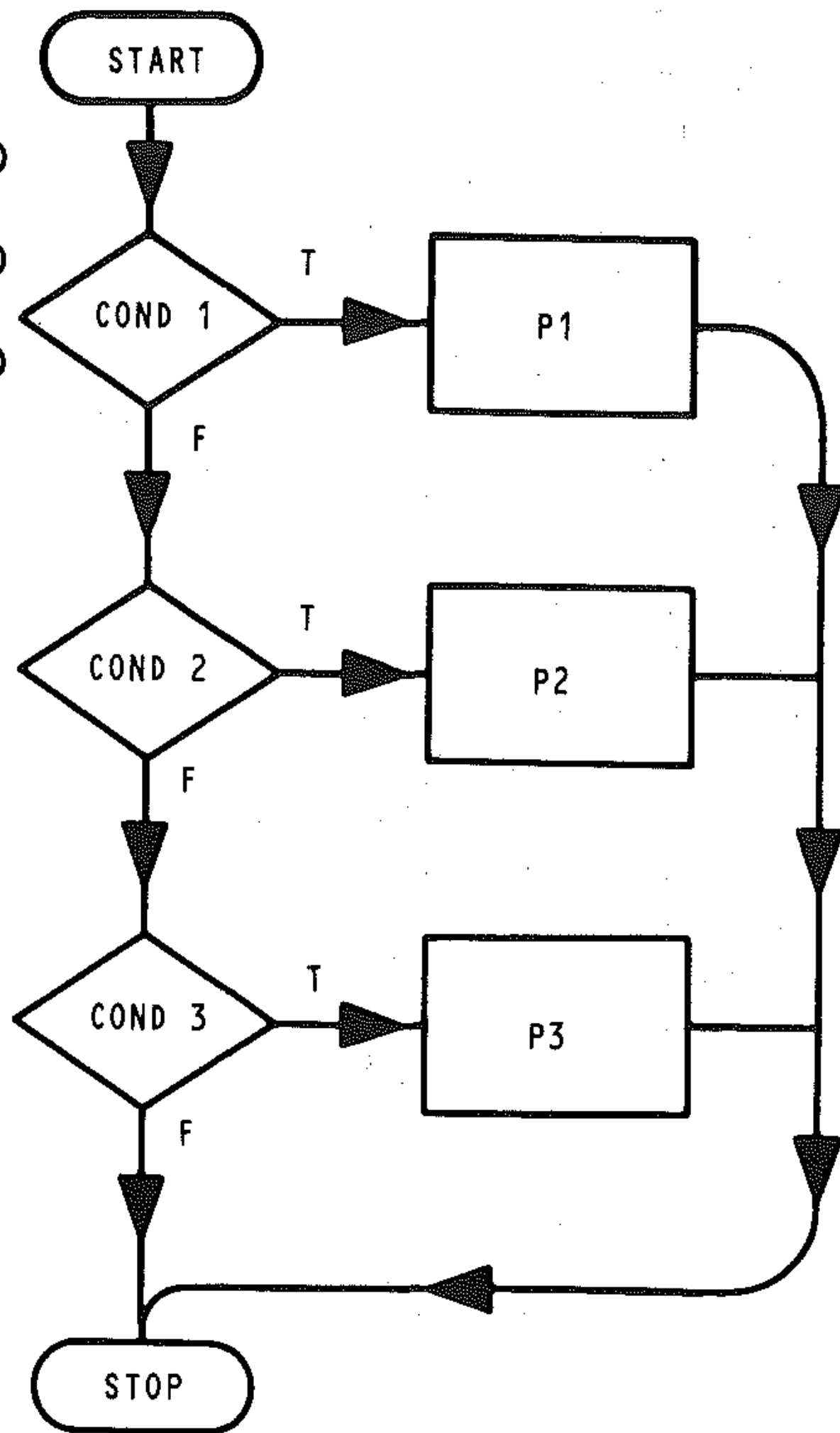
Pseudocode

```

mod
  case
    if(condition 1 is true)
      then P1
    if(condition 2 is true)
      then P2
    if(condition 3 is true)
      then P3
  endcase
endmod

```

Flowchart



BASIC

```

10 REM * STARTMOD *
20 IF C1 THEN P1
30 IF C2 THEN P2
40 IF C3 THEN P3
50 REM * ENDMOD *

```

For example: Test whether a number input is positive, zero, or negative, and print the result.

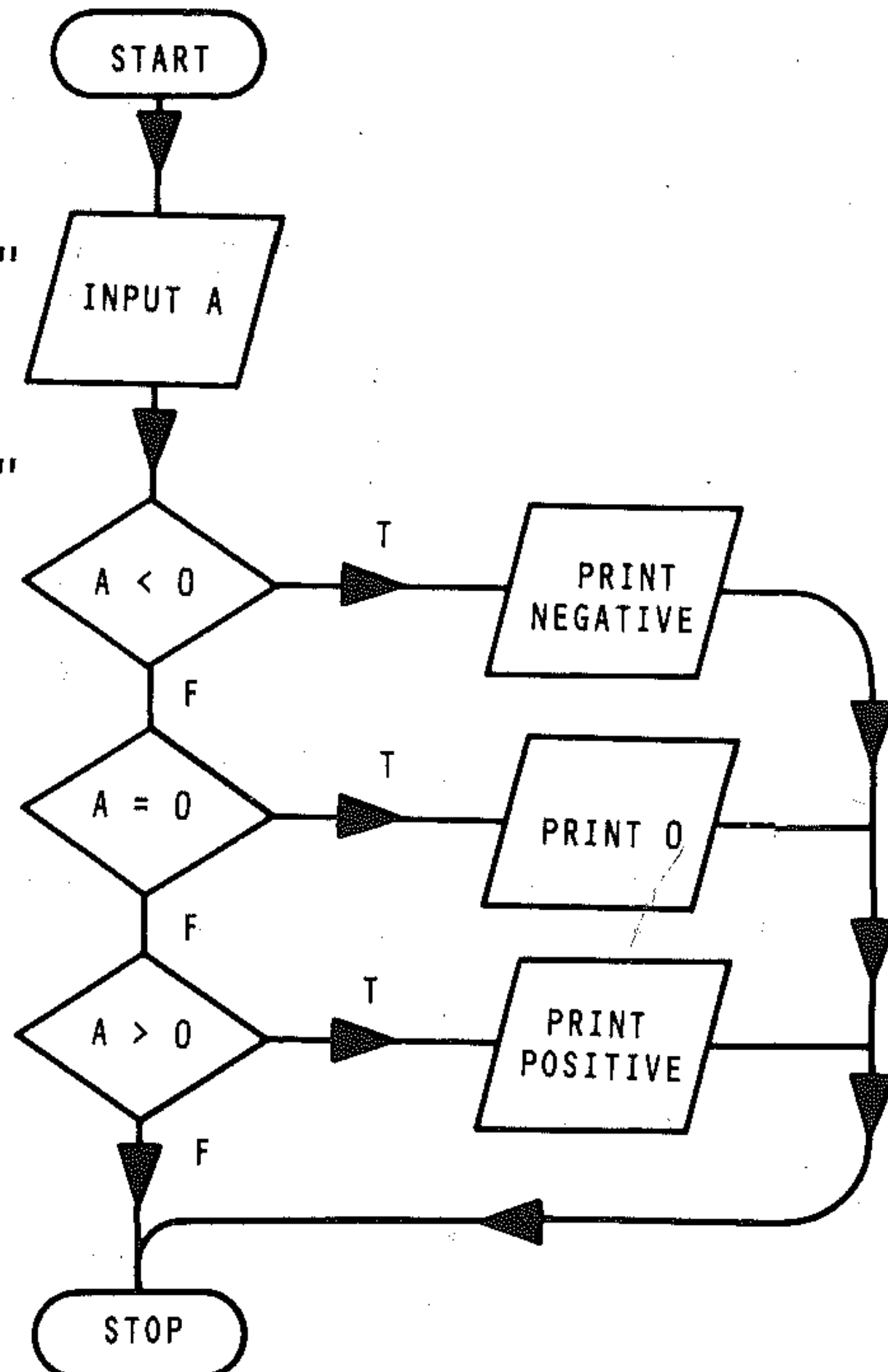
Pseudocode

```

mod
  input A
  case
    if A < 0
      then print "NEGATIVE"
    if A = 0
      then print "ZERO"
    if A > 0
      then print "POSITIVE"
  endcase
endmod

```

Flowchart



BASIC

```
10 REM * STARTMOD *
20 INPUT A
30 IF A < 0 THEN PRINT "NEGATIVE"
40 IF A = 0 THEN PRINT "ZERO"
50 IF A > 0 THEN PRINT "POSITIVE"
60 REM * ENDMOD *
```

Alternatively, conditional and unconditional GOTO statements could be used to implement this structure. This would be appropriate if the processing section of a program after the decision were several statements long.

BASIC

```
10 REM * STARTMOD *
20 INPUT A
30 IF A < 0 THEN 69          30 IF A<0 THEN PRINT "NEGATIVE"
40 IF A = 0 THEN 80 OR      40 IF A=0 THEN PRINT "ZERO"
50 IF A > 0 THEN 100        50 IF A>0 THEN PRINT "POSITIVE"
60 PRINT "NEGATIVE"        60 REM * ENDMOD *
70 GOTO 110
80 PRINT "ZERO"
90 GOTO 110
100 PRINT "POSITIVE"
110 REM * ENDMOD *
```

Loop structures

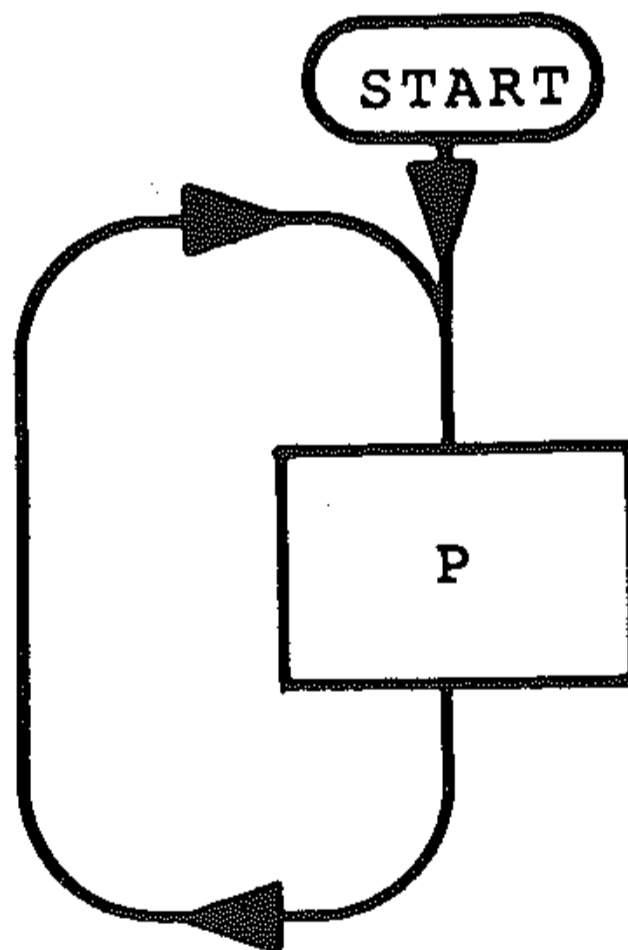
The repeat-forever loop

This loop is normally useless. The only conceivable result is to stop the program with an arithmetic overflow report or a user break.

Pseudocode

```
mod.
  repeat
  P
  forever
endmod
```

Flowchart



BASIC

```
10 REM * STARTMOD *
20 P
30 GOTO 20
40 REM * ENDMOD *
```

This structure is for demonstration only; it must be avoided in programs. It can sometimes occur in error. Use the **RUN/STOP** key if you suspect your program has entered such a loop (because nothing happens).

The repeat-until loop

Repeats processing until a condition is true. These structures loop until a specific termination condition is met, for example until a counter reaches a certain value or until a dummy or sentinel value is input. The important characteristic of this loop structure is that the repeat test (or exit test) is at the bottom of the loop, after

the processing *body*. The program lines making up the body of the loop (P) will be executed at least once. The repeat condition can use any conditional operator or its complement (reverse), for example equals = or not equal <>. Use of the complement often leads to a more elegant program.

Pseudocode

```

mod
  repeat
    P
  until(condition is true)
endmod

```

BASIC

```

10 REM * STARTMOD *
20 P
30 IF (COND) THEN 50
40 GOTO 20
50 REM * ENDMOD *

```

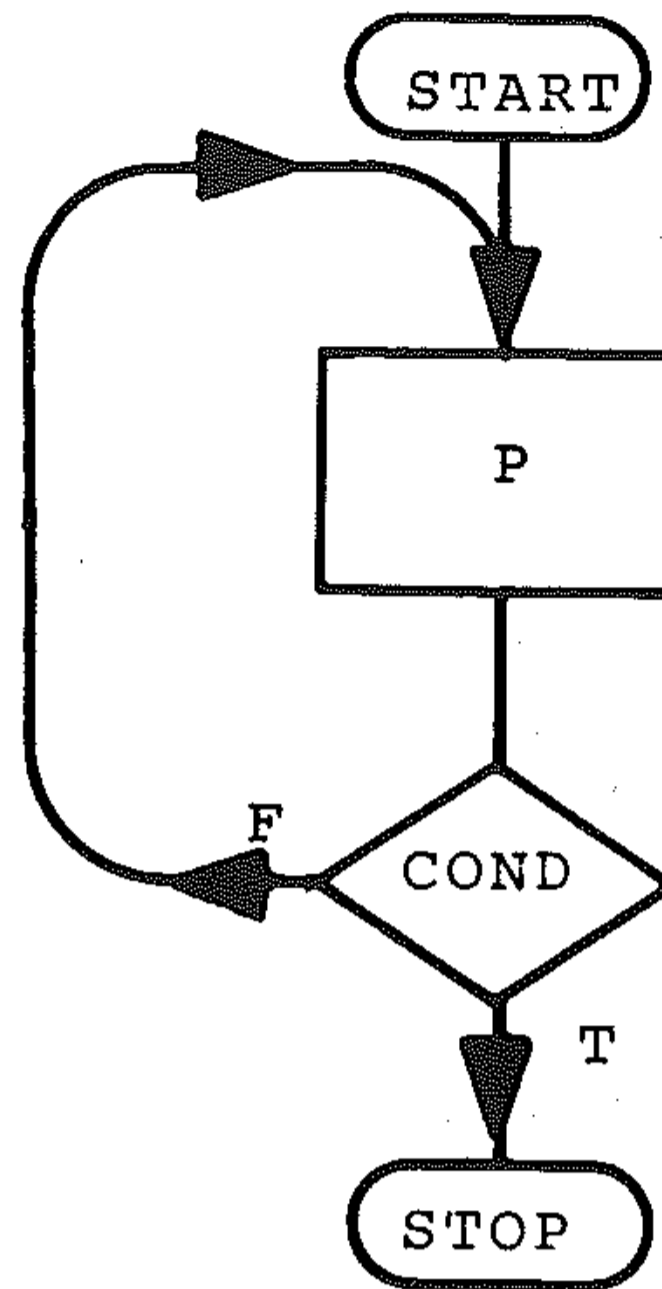
BASIC using complement

```

10 REM * STARTMOD *
20 P
30 IF (COMP COND) THEN 20
40 REM * ENDMOD *

```

Flowchart



Exit requires no specific instruction.

For example: input and print strings until the sentinel value LAST is input.

Pseudocode

```

mod
  repeat
    input A$
    print A$
  until A$="LAST"
endmod

```

BASIC

```

10 REM * STARTMOD *
20 INPUT A$
30 PRINT A$
40 IF A$ = "LAST" THEN 60
50 GOTO 20
60 REM * ENDMOD *

```

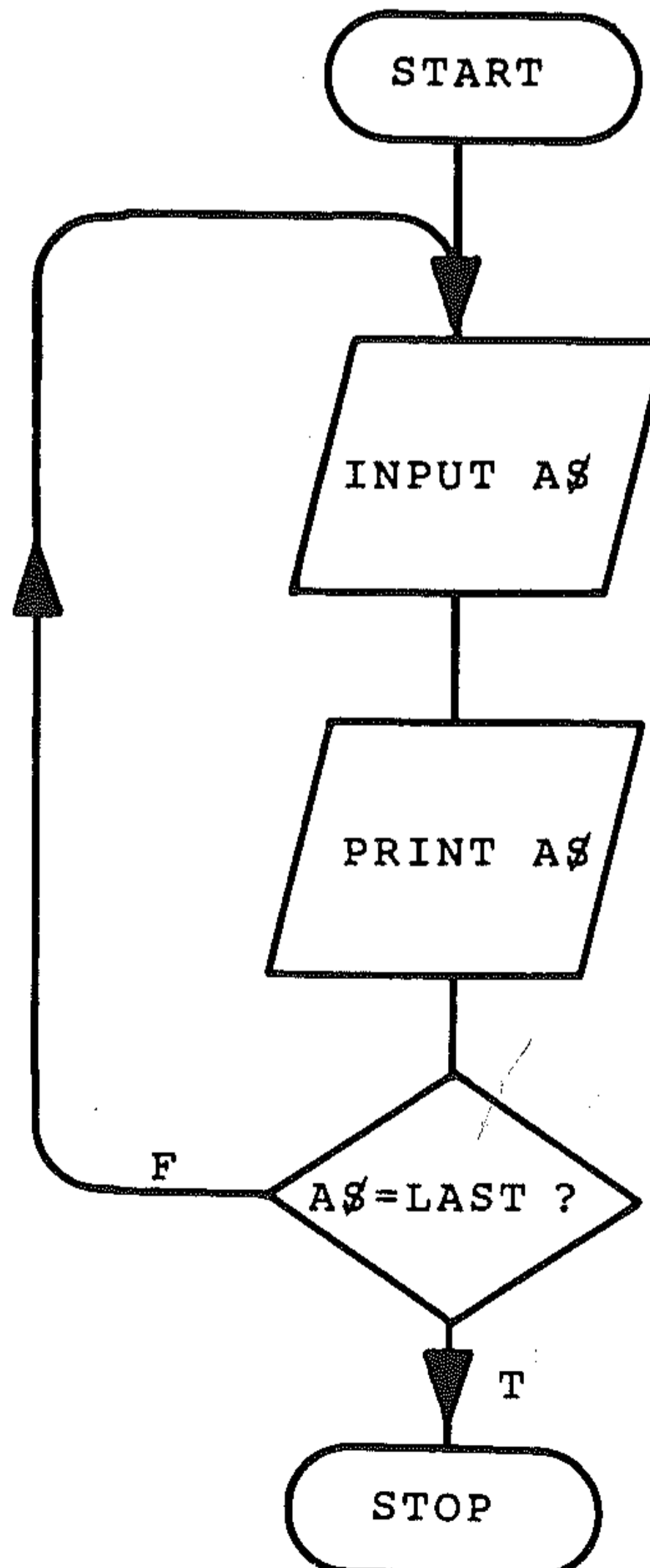
BASIC using complement

```

10 REM * STARTMOD *
20 INPUT A$
30 PRINT A$
40 IF A$ <> "LAST" THEN 20
50 REM * ENDMOD *

```

Flowchart



While-do structure

While a condition holds (TRUE), keeps repeating the processing until the condition is broken (FALSE). The condition can be, for example, that a loop-counter variable value is not equal to its final value (IF N<10 THEN...). The process will then repeat until it is. The condition may also be set so that a sentinel value has not occurred (IF N<>6 THEN...). These conditions are set so that the true pathway is the process task, and the false is the exit. The while-do loop is characterised by having the repeat test carried out prior to the body (i.e. at the top). No processing will happen if the repeat test is false at the first encounter, and the body of the loop will not be entered.

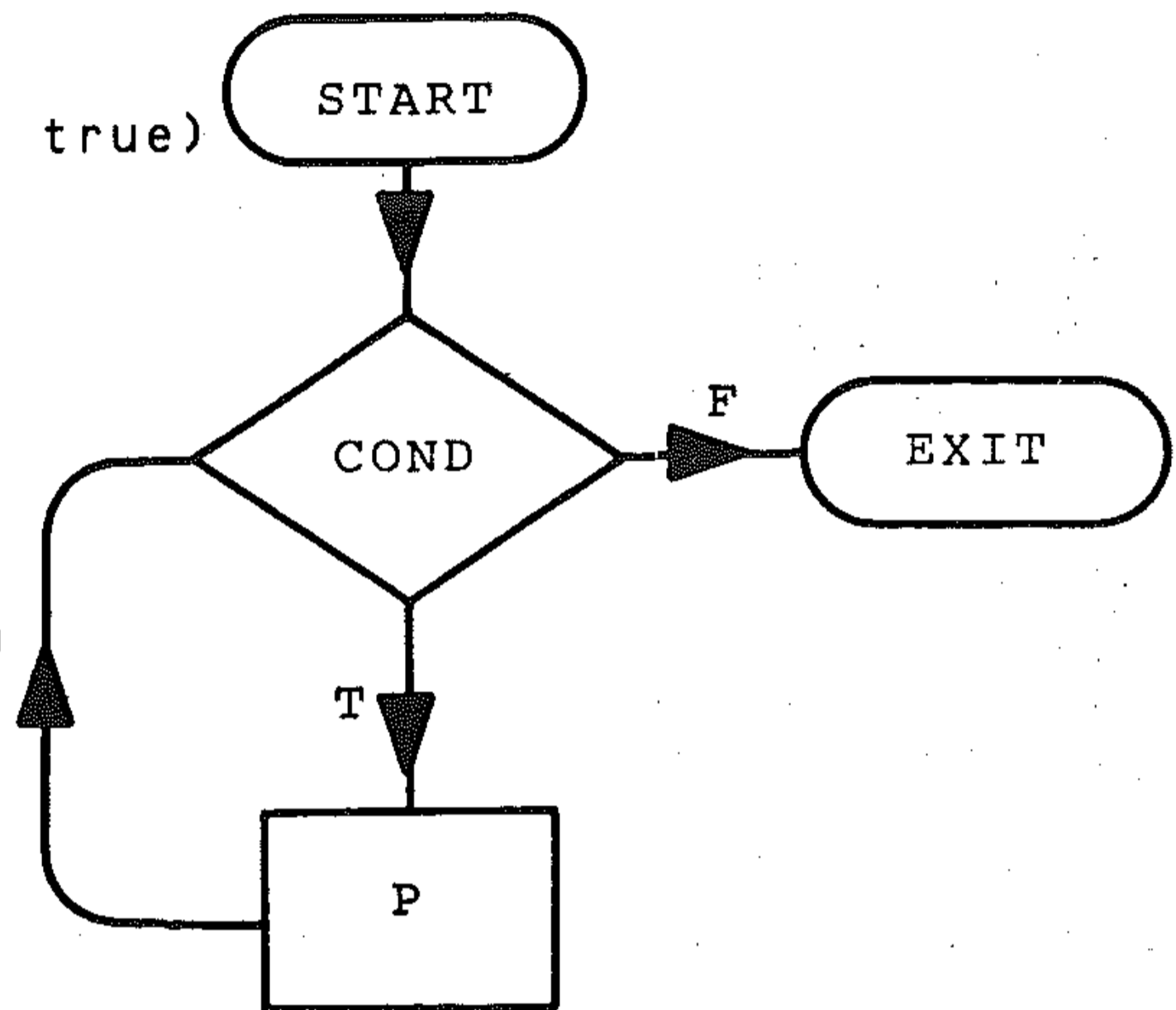
Pseudocode

```
mod
  while(condition is true)
    do P
  endwhile
endmod
```

BASIC

```
10 REM * STARTMOD *
20 IF (COND) THEN 40
30 GOTO 60
40 P
50 GOTO 20
60 REM * ENDMOD *
```

Flowchart



Using the complement of the repeat condition gives a neater program.

BASIC with complement

```
10 REM * STARTMOD *
20 IF (COND) THEN 50
30 P
40 GOTO 20
50 REM * ENDMOD *
```

For example: While the value of the square of consecutive integers is less than 100, print them on the screen.

Pseudocode

```
mod
  n = 1
  while n*n<=100
    do print n*n
    n=n+1
  end while
endmode
```

BASIC (complement)

```
10 REM * STARTMOD *
20 N = 1
30 IF N*N>100 THEN 60
40 PRINT N*N
45 N = N + 1
50 GOTO 30
60 REM * ENDMOD *
```

FOR...NEXT loop

Repeats a process a stated number of times. These are in fact while-do loops and have the repeat test at the top of the loop.

For example: Print the values of the first ten integers.

Pseudocode BASIC

```

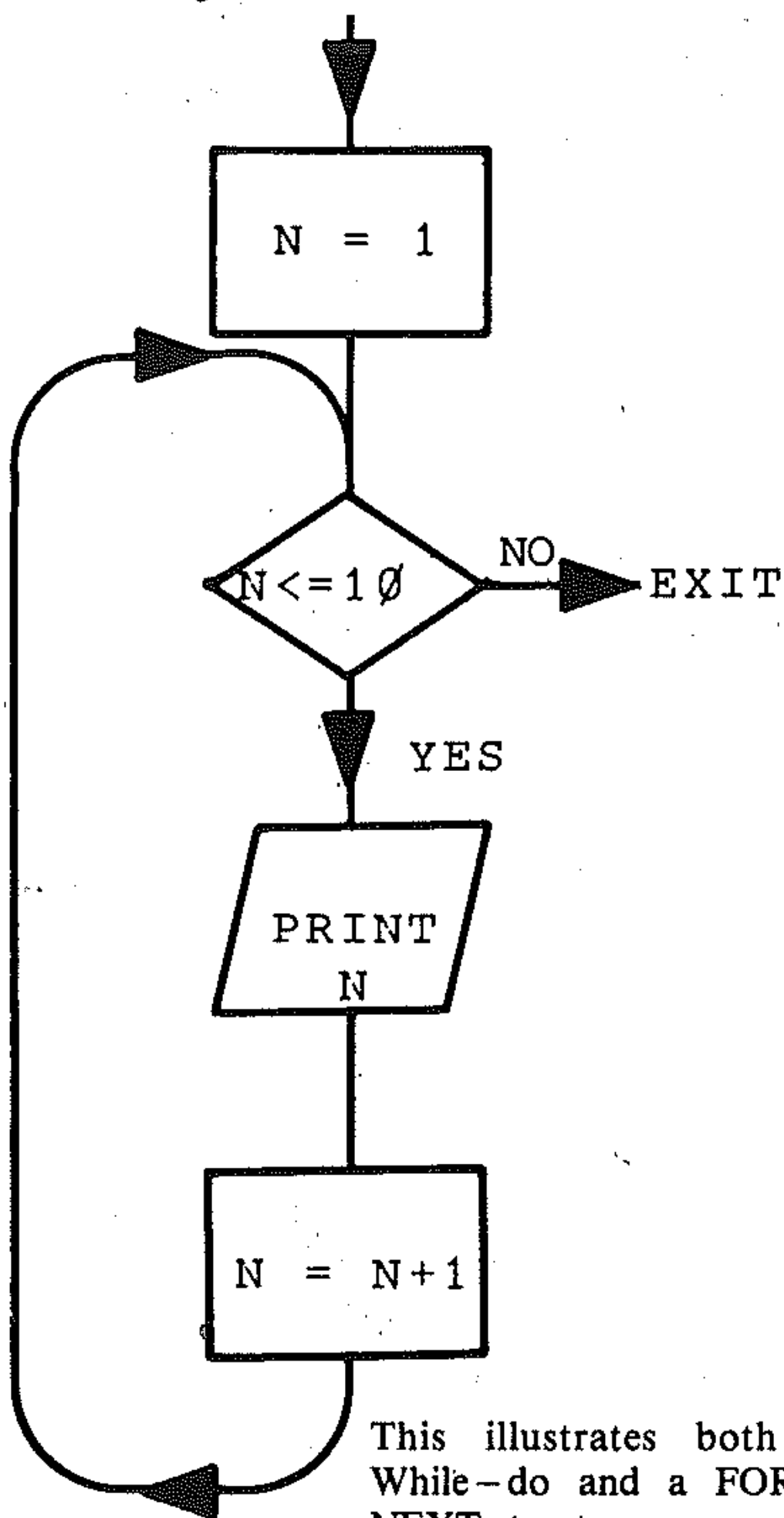
mod
  n = 1
  While n <= 10
    do print n
    n = n + 1
  end while
endmode
  
```

```

10 REM * STARTMOD *
20 FOR N = 1 TO 10
30 PRINT N
40 NEXT N
50 REM * ENDMOD *
  
```

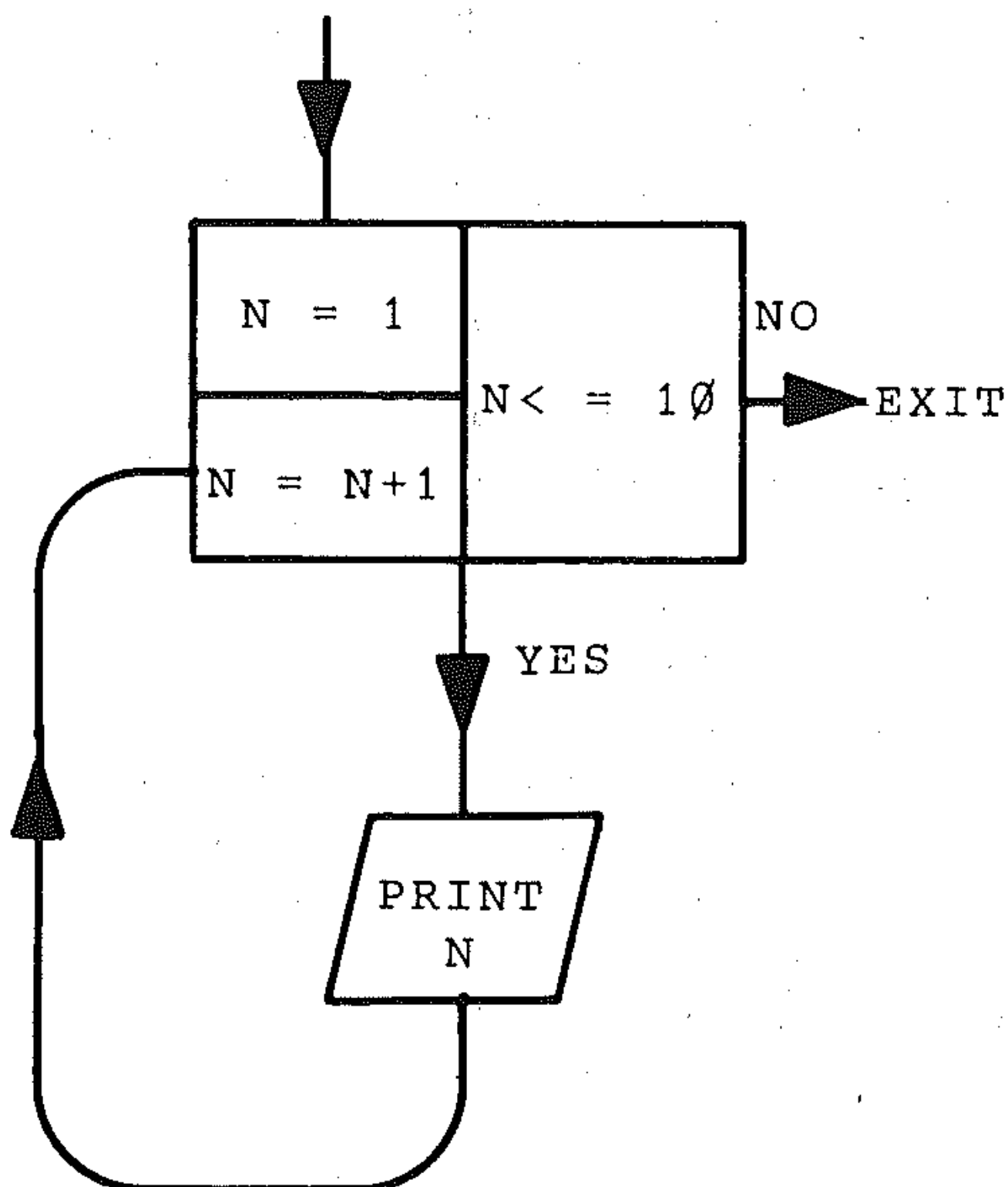
FOR...NEXT loops have their own flowchart symbol, because they are used so extensively in BASIC.

Ordinary



This illustrates both a While-do and a FOR-NEXT structure.

Special



This is a condensed version. It groups together the FOR-NEXT-STEP instruction elements, which the standard form separates.

O4: Program Development

Program development includes *debugging* the program of errors, *testing* to see if it behaves as specified and gives the desired results, and *documentation*, which tells users how to run the program.

Debugging

The 64 machine has good editing facilities and error messages. The error messages have a brief statement of the type of error. It seems inefficient to correct errors one at a time, because there is seldom only a single error; programming mistakes tend to compound one another. But error messages are produced singly, when an error stops the program. Thus you must deal with the errors as they occur. You may notice a number of errors on carefully looking through the listing; any you spot should be edited out at once.

Get to know your computer error codes.

This will happen automatically in time (as you make mistakes!) but it is worthwhile studying the codes. They define the ways in which *run-time* errors occur, and an understanding of them will help you avoid bugs.

Keep notes on mistakes you make and how you correct them.

This should become an automatic part of your personal documentation. Keep copies of old program listings. Record the errors you have made, the corrections you tried but which did not work, and what you learned in developing the program.

Trace the impact of any error through the problem.

Syntax errors

These are caused by BASIC statements you key in which do not obey the language rules of 64 BASIC. If there is an error the interpreter will say **SYNTAX ERROR IN LINE N** on the screen. To correct this type of error you must compare the syntax you have written with the rules of BASIC.

Program logic errors

These are the result of bad logical design of the program. They can be avoided if care is taken in design and coding; if a program produces incorrect results then there is an error in the flow of logic in the program. This may only occur with certain values of data.

If each program section or module has been tested independently, then the linking of the modules is incorrect. You can test program sections as follows:

- a Insert a temporary breakpoint into the program, at an appropriate point, using a STOP instruction.
- b Print out values of intermediate results to the screen.
- c It is important to print out the values of variables used in making a decision and those used in loops, either counter loops or FOR...NEXT loops.
- d Go back to the pseudocode or flowchart and modify the steps which are in error. Walk through the algorithm, using a flowchart, to check the step sequence, and hand trace the program with selected values of data and/or variables. Be careful! Often changes in one part of the algorithm cause changes in the others. Remember that solving one problem may cause a new one.
- e Change the documentation if necessary. Note down the changes you have made, or lines you have deleted. Keep program listings.
- f Re-test the complete program, using a variety of data.

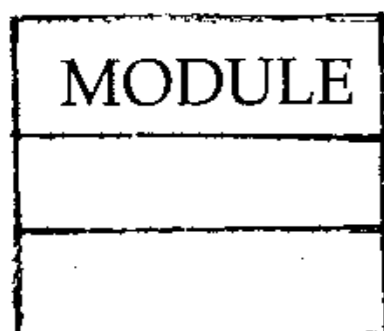
Each testing statement in a complex program should be headed by a remark statement.

```
1000 REM - DEBUG
-
-
-
-
- REM - END DEBUG
```

These temporary REM statements are later deleted by keying in their line numbers, as are the testing statements. It is very easy to leave in test instructions unless they are marked.

Inserting break points

You can stop a program at any point and obtain the values of variables, expressions, etc to test calculations or check for errors. This is done by inserting a group of statements which will output the values we want and then stop the program.



← Add test output of variables

← Insert a STOP statement

CONT will restart the program.

Individual modules or sections of programs can be tested this way. The program has to be RUN, of course, from the required module line number. Care should be taken when this is done that variables needed in the module have declared properly and that the values of parameters passed to the module are as required. Remember that you can INPUT the values of variables directly if necessary, using the command mode, and using LET statements, which is implied by the 64:

```
X(2) = 20 etc.
```

The value of any variable at the point the program crashed can also be obtained by keying a statement without the line number, and again using the computer in command mode:

```
PRINT A$  
PRINT X(3)
```

The commands RUN N (where N is the line number you wish to run the program from) and GOTO N enables you to run the program starting at any point. Using GOTO N does not negate the initialisation of variables that occurs if the program has already run. For example, if you input:

```
10 A = 1  
20 B = 2  
30 PRINT A, B
```

Enter GOTO 30. This prints 0, 0. Now enter RUN followed by GOTO 30; this prints 1, 2 as A and B have been assigned the values 1 and 2 respectively.

Run-time errors

These are a result of programmer carelessness and do not prevent the interpreter from translating the program. They make the program *crash* when you attempt to run it; that is, they prevent the program from running to completion. Common run-time errors include:

- arithmetic overflow
- lack of data for processing
- failure to complete loop increment and subroutine section statements
- subscript out of range
- memory full
- integer out of range

As you have seen, run-time errors cause diagnostic system messages to be printed. These appear on the screen and are called error messages. These errors can then be traced through the type of error given by the code and the line number at which the program stopped.

Error messages

Error messages or reports are presented on the screen when a program stops for any reason, either as a result of successful completion (no more program lines), an instruction or command (STOP, BREAK), or a run-time error. The 64 gives an error report code, with a brief statement in the form:

[XXXXXX ERROR] IN LINE N

where XXXXXX is type of error. Report codes are crucial aids to debugging programs; without them, you would know only that there was an error, but not where it occurred or what type of error it was. The error message at least gives clues. The cause of an error might be earlier in a program than the line where the program stopped; for example, where a numeric value wrongly defined or generated by the program causes another expression to cause an arithmetic overflow. But without the message you might have no idea where to start looking.

Lists of error messages and their meanings for the Commodore 64 is given in *Appendix IV*.

Testing and verification

Testing comes after debugging a program. Its purpose is to ensure that the program is logically correct, produces correct answers and meets the specification of its purpose.

First test each module separately.

Each procedure and subroutine should be treated as if it were a separate program. Test for:

- good data – the expected type and range of inputs
- bad data – out-of-range and incorrect type inputs

Try to ensure that each procedure *fails softly*. For bad data (particularly in a data entry module), following each input a check routine or procedure should be used to give an error message if the range is incorrect, or to check the type of input and correct syntax. This is best done with strings, which are more flexibly handled. See Section V, which deals with input checks at length.

Combine the modules and test the complete program.

If there is a logical error (that is, if the program does not produce the intended results) insert additional test statements which will:

- output intermediate results
- output values of variables at each stage
- output results of expressions at each stage
- output values of the loop counter at each pass
- output results of array manipulation after each operation
- output values of parameters before and after a subroutine's entry and return

Provide for exceptions.

- test all data in the program
- screen all data
- process only good data
- output bad data saying why it was bad

Let your program stop elegantly.

- When there is no data input or data available, the program should tell you so

- 64 BASIC programs are interactive. The user can control program continuation with:

```

910 PRINT "PROCESSING ENDED - MORE DATA ? YES OR NO"
920 INPUT A$
930 IF A$ = "YES" THEN 100
940 PRINT "GOODBYE"
950 END
960 REM PROGRAM END

```

Rewrite the program until you are satisfied with it.

Remember the program should be:

- structured
- easy to read
- easy to understand
- handle exceptions
- be as efficient as possible
- documented

and it must solve the problem as specified!

Put clarity before efficiency.

A good program algorithm does not have to be clever, difficult to understand or run super-fast. If you do not understand how the algorithm works, don't use it – rewrite it or use another method. Programs will work correctly if the rules of the language are obeyed, and the program will work to specifications if the algorithm is properly designed.

Documentation

Annotate and document your program to create a readable program.

Write an explanation for each program module or segment. At the beginning of each segment provide suitable comments which explain:

- the purpose of the algorithm
- the variables and their significance (the values they store)
- the results expected

Use comments only where necessary:

- don't comment each program line
- don't explain the obvious
- at the beginning of the program, provide a block of comments that explain the program at each module and provide a comment which explains what the module does in relation to the program.

Clear comments should appear separated from program code. The clearest comments are framed. For example:

```

10  REM  *      *      *      *      *      *      *      *
20  REM  *          SUBROUTINE TO
30  REM  *          CALCULATE N TO 2 D.P.
40  REM  *
50  REM  *      *      *      *      *      *      *      *

```

Lines of asterisks provide visible dividers between sections of program.

Use comment in the program and in the output to the screen or printer. Use blank REM lines or spaces as separators in the program. For large programs write a reference document:

- Describe the algorithm you used. If it is not original you should include a note of its source, author, version, and type of computer it was written on.
- Explain how you wrote the program, the reasons for writing it, the type of computer used and memory required.
- Make a note of areas that may need improving, or could be modified for different purposes.
- Which modules are general (menus, subroutines), and which require specific kinds of input.
- Explain the scope and limitations of the program.
- Include your name, and the date of production.

List the tests you made and data used. Reproduce some of the results of the tests.

List performance tests (eg. how long it takes the program to run).

Give user instructions and reproduce the output of a run and explain to the user how he uses the program.

Give the program characteristics. Explain any abnormal behaviour of the program (eg. response to bad input).

And finally, write a user guide, for an ordinary user, not for a computer expert. It should explain:

- the purpose of the program
- the algorithm
- how to run the program
- what input is needed
- what results are printed
- how to use the menu (if included)

O5: The complete programming method

Summary: the structured programming method

1 Produce the algorithm.

1.1 State the problem fully

1.1.1 State the problem

1.1.2 Understand what is to be done

1.2 Research the problem

1.2.1 Research and analyse the problem to see how the computer can handle it

1.2.2 Identify all formulae and relations to be used.

1.2.3 Identify all data involved

1.3 Design the algorithm, using top-down structured methods

1.3.1 Break the problem up into sub-problems or modules

1.3.2 Use a structure diagram or tree diagram to help in breaking down the problem

1.3.3 Start classifying modules or parts of modules as:

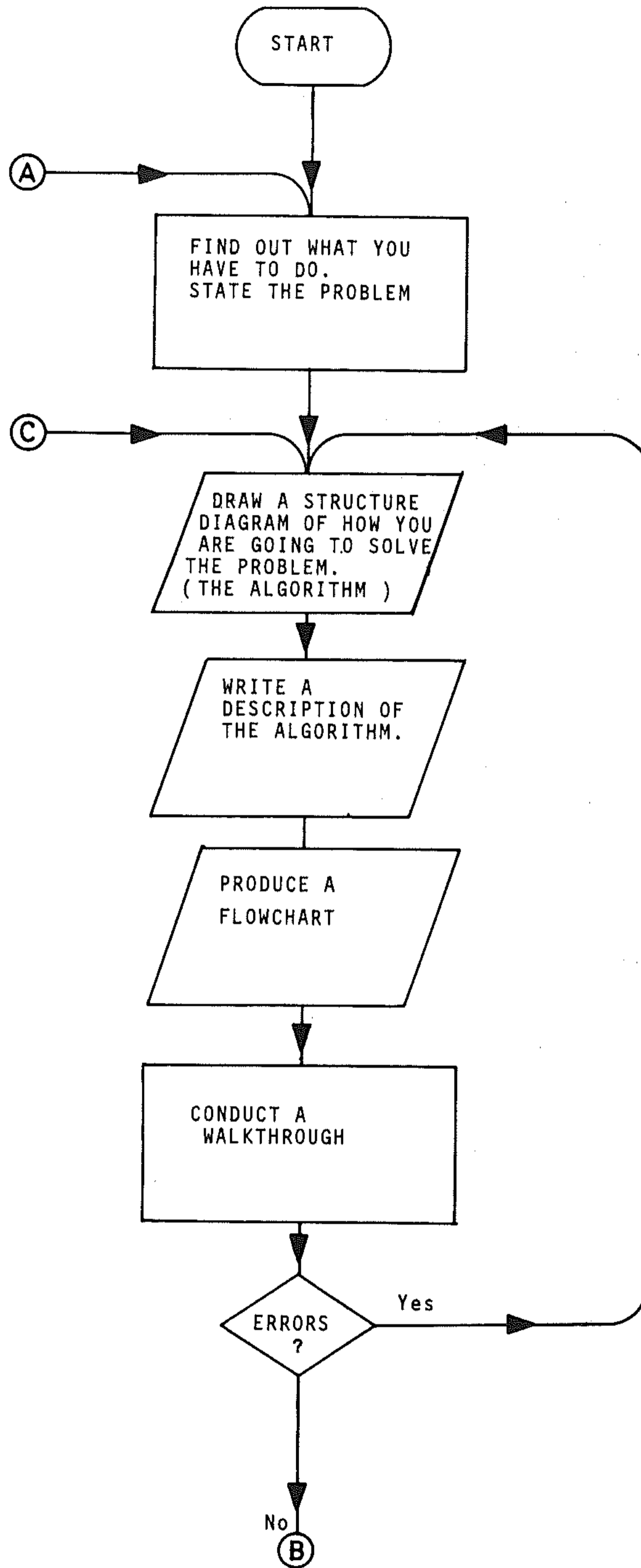
- output
- processing
- output

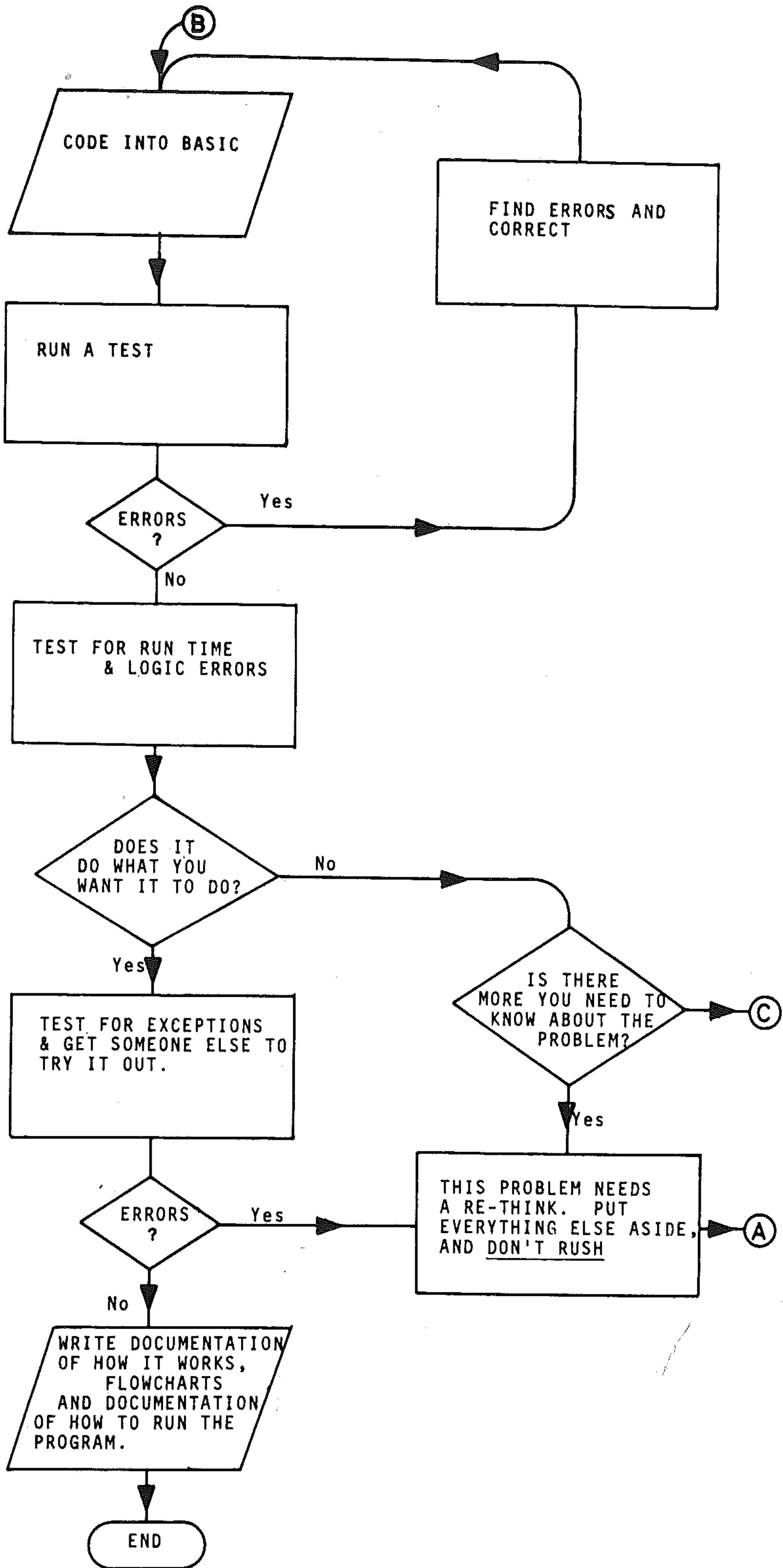
- 1.3.4 Utilise the fundamental control structures in the modules:
 - decision structures
 - transfer structures
 - loops
 - subroutines
 - nested structures
 - subprograms
- 1.3.5 Set up a DATA TABLE in which all data types are classified as:
 - variables
 - constants
 - counters
 - functions (if using a Commodore and the DEF FN instructions)
- 1.3.6 Define the algorithm further until coding it into a BASIC language program is an easy and obvious exercise.
- 1.4 Describe the algorithm in pseudocode and flowchart form
- 1.4.1 Write out the final algorithm (now in modular form) in small steps in an abbreviated English style called pseudocode. Each module should be treated separately and labelled.
- 1.4.2 Illustrate the logical flow of control in the algorithm by constructing a flowchart.
- 1.4.3 Test the algorithm, if necessary using a hand trace or walk through.

- 2 **Produce the program.**
- 2.1 Code the Algorithm in 64 BASIC
- 2.1.1 Code on a direct basis from the pseudocode or flowchart description in line-numbered BASIC statements, module by module.
- 2.1.2 Implement the fundamental control structures, used in their 64 BASIC versions.
- 2.2 Debug and test the program
- 2.2.1 Debug the program. Check the program variables against your algorithm test. Correct syntax, run time, and logical errors.
- 2.2.2 Test the program for further logical errors. Run the program with sample data.
- 2.3 Document the program. For a full documentation, you should:
- 2.3.1 Produce a programmers' guide, consisting of:
 - pseudocode
 - flowchart
 - variable table or data table
 - program listing
 - test results or sample printout
- 2.3.2 Detail the steps that producing the program involved.
- 2.3.4 Write a user guide.

Summary of the method in flowchart form

Here is a diagrammatic summary of structured programming:





O6: An example of structured design

1 Problem statement.

Write a program that computes and prints the *average* or *Mean* (M) and *Standard Deviation* (S) of a collection of N data items. To compute S use the formula:

$$\text{Standard Deviation} = \sqrt{\frac{\text{sum of squares of items} - (\text{Mean})^2}{N}}$$

2 Find out what you have to do (research the problem).

You are given most of the information in the question, but some is missing. The question does not tell you how to compute the Mean, or average. This is done using the formula:

$$\text{Mean} = \frac{\text{sum of all numbers}}{N}$$

You now have all the information you need to start designing the algorithm.

3 What is involved in this problem?.

You can now define the outline procedure:

- You have to INPUT the numbers, and
- perform two calculations on these numbers. First calculate the Mean and then use the Mean value to calculate the Standard Deviation, then
- output the results.

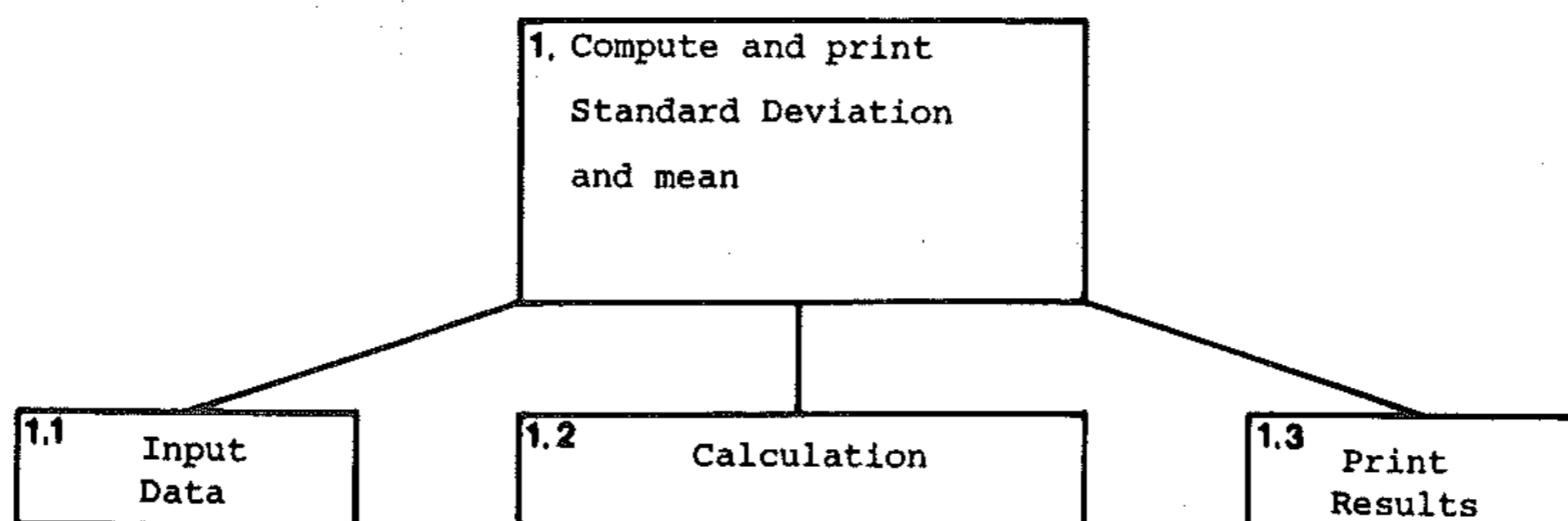
4 Design the algorithm.

This describes the detailed procedure for the steps needed to solve the problem:

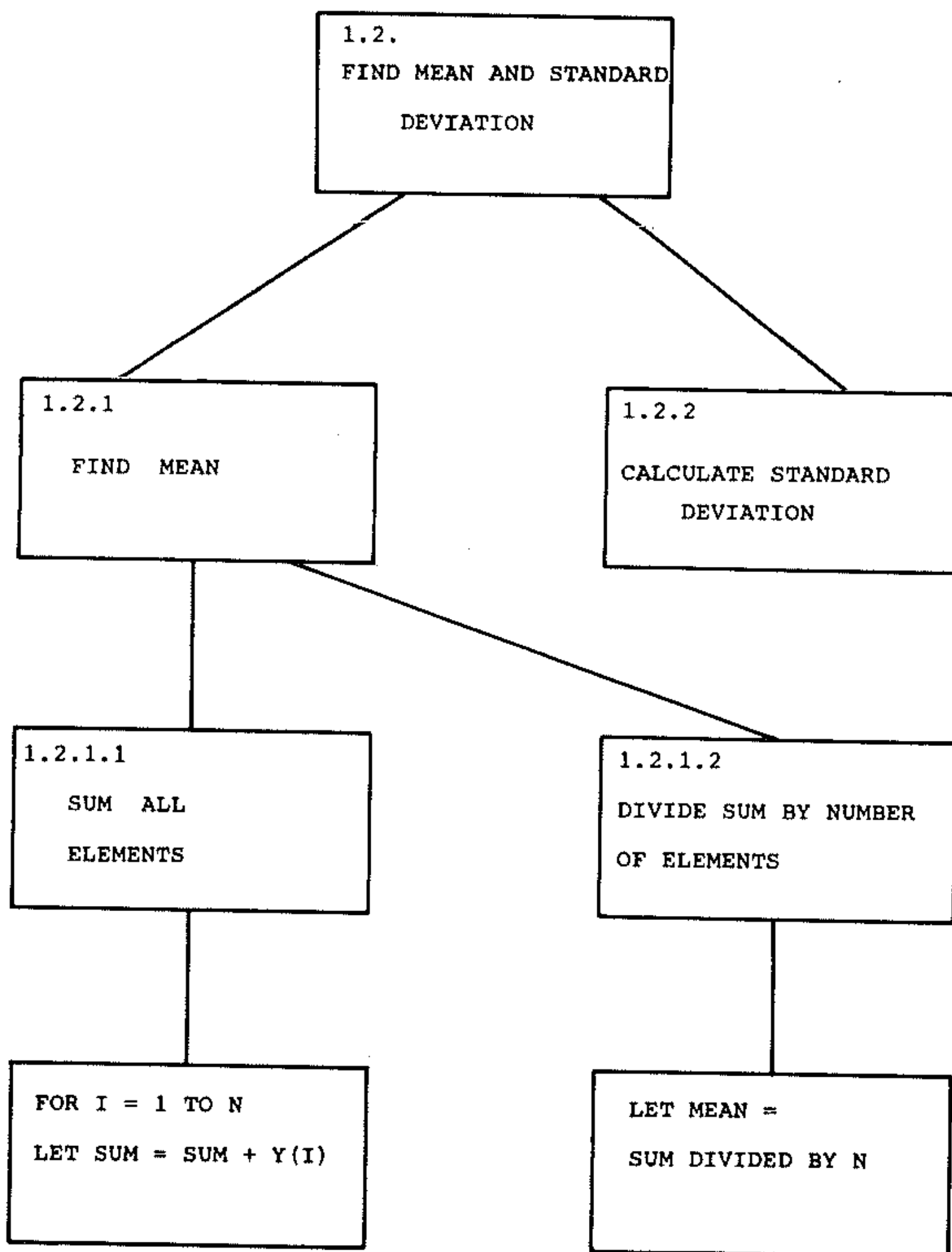
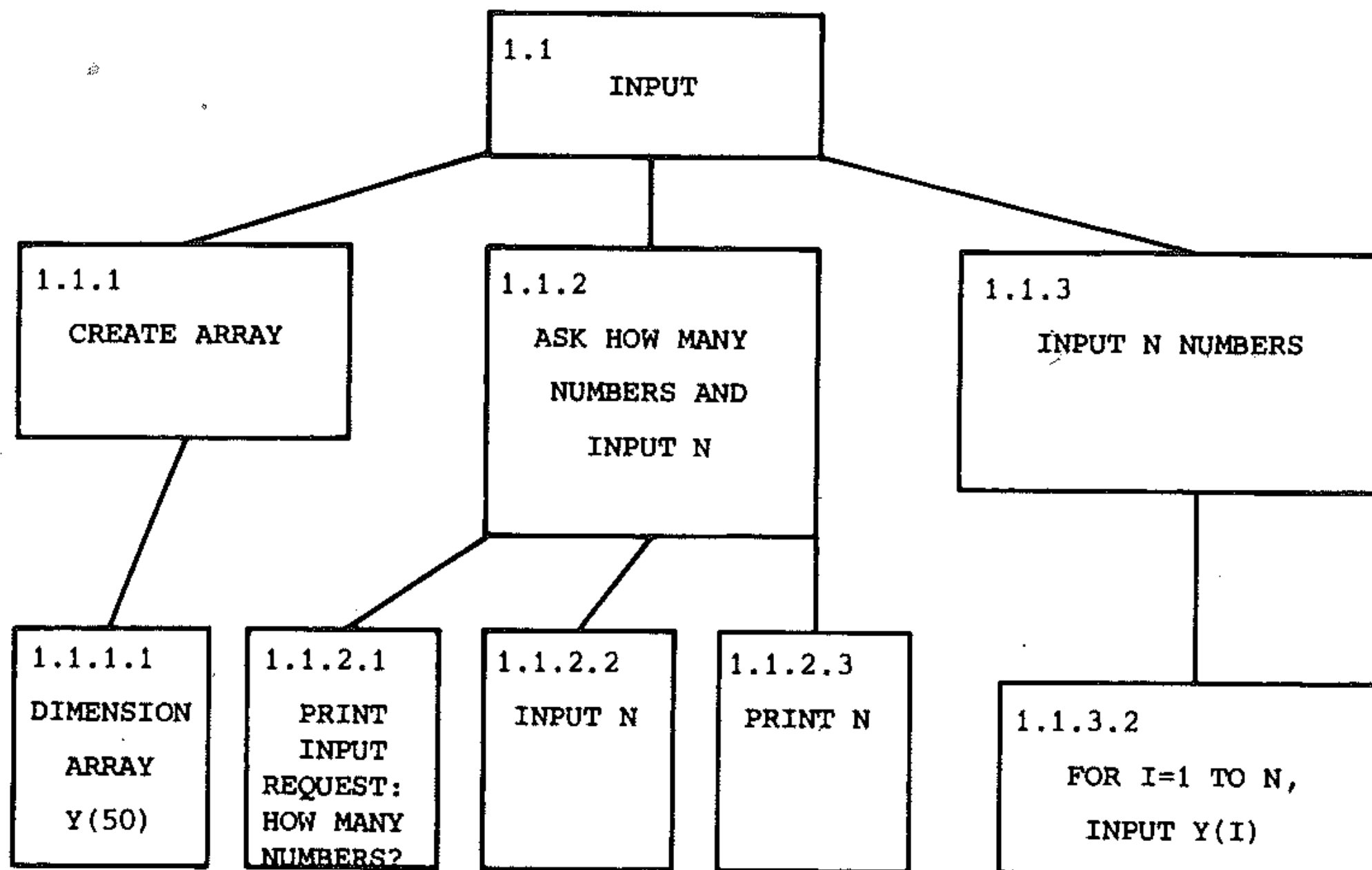
- INPUT: the numbers must be input into an array because they will be needed twice in the calculation module.
- Calculate the Mean: add all the numbers in the array and divide by N.
- Calculate the Standard Deviation: total the squares of all the numbers in the array and use the formula to calculate S.
- Output: the results should be printed on new lines with the words

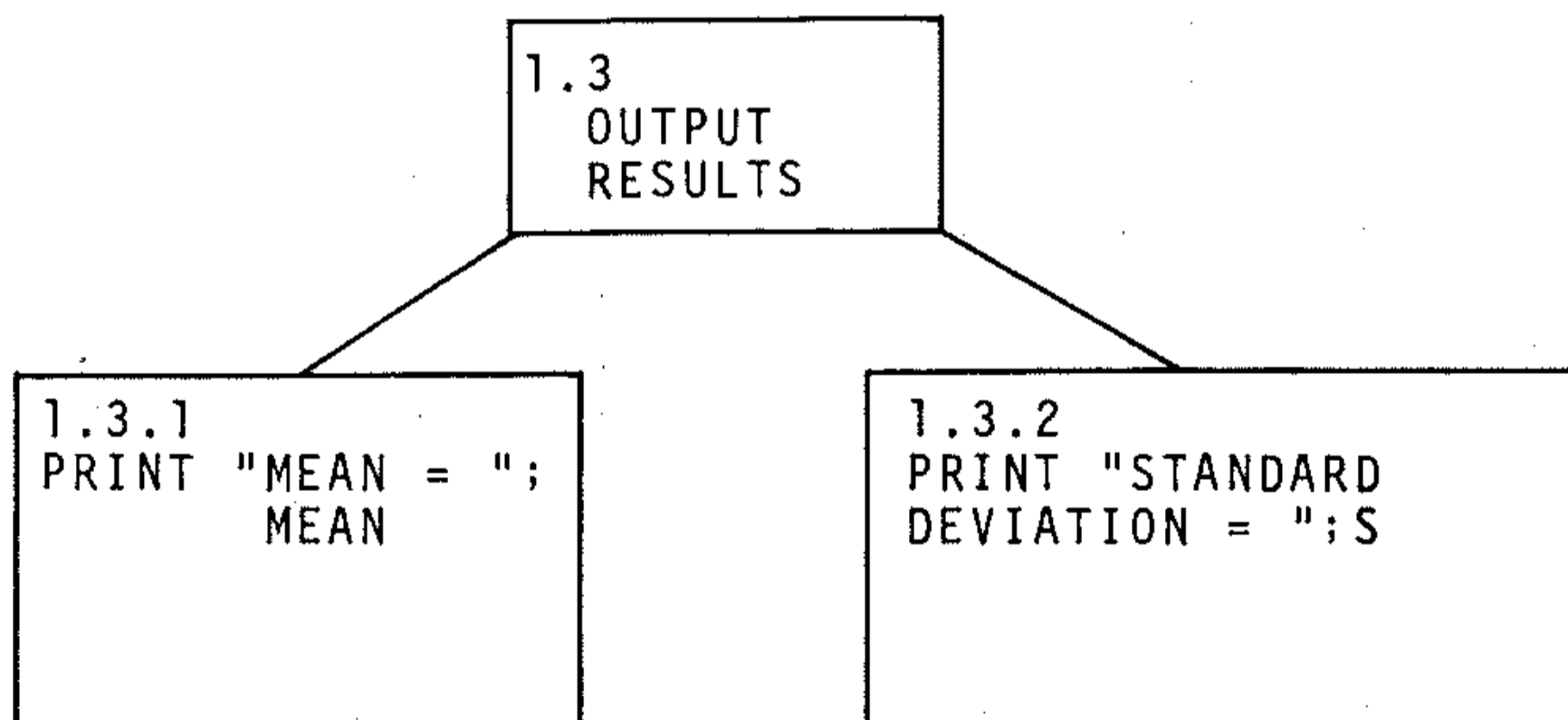
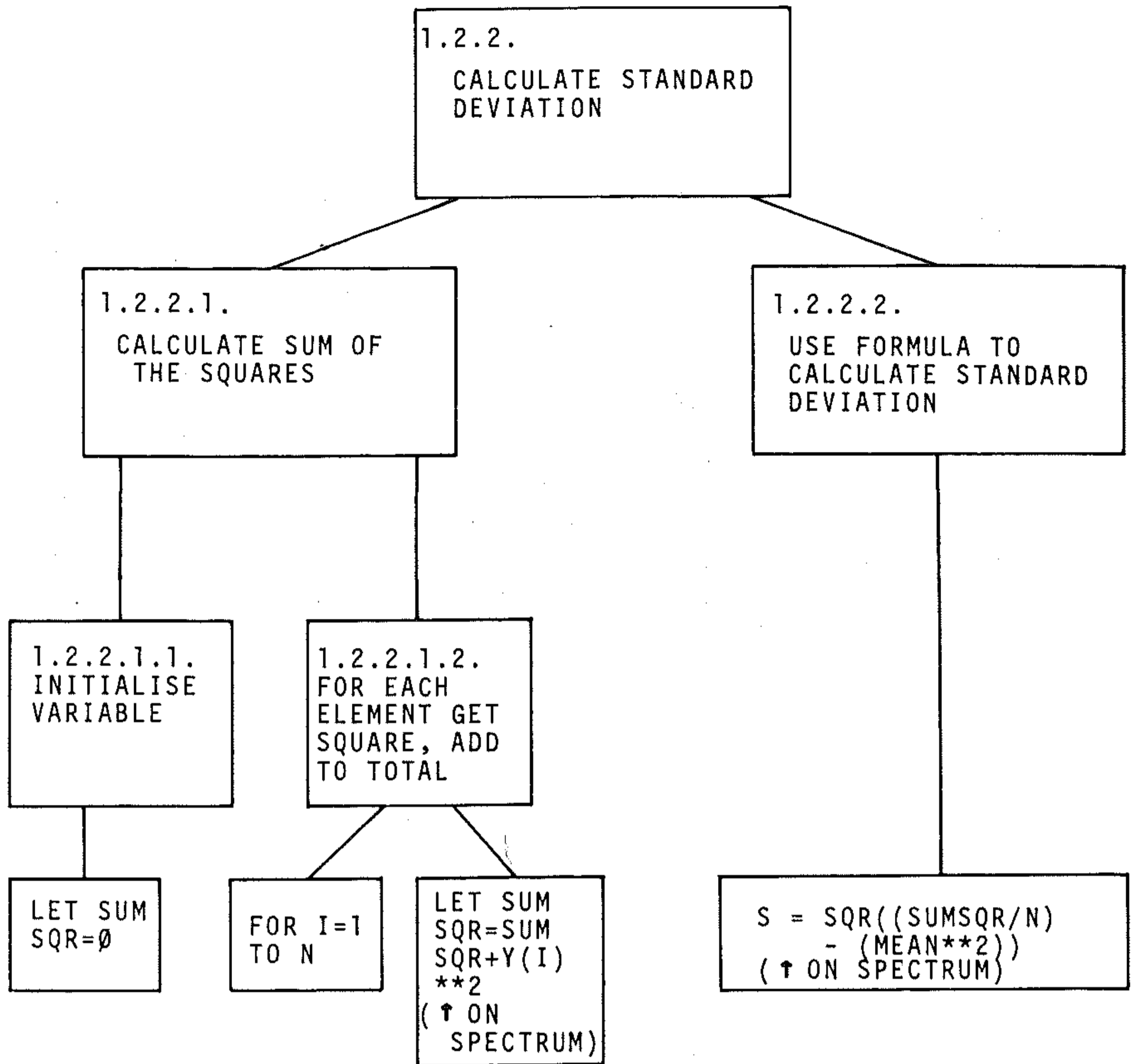
```
MEAN = (value)
STANDARD DEVIATION = (value)
```

5 The tree diagrams.



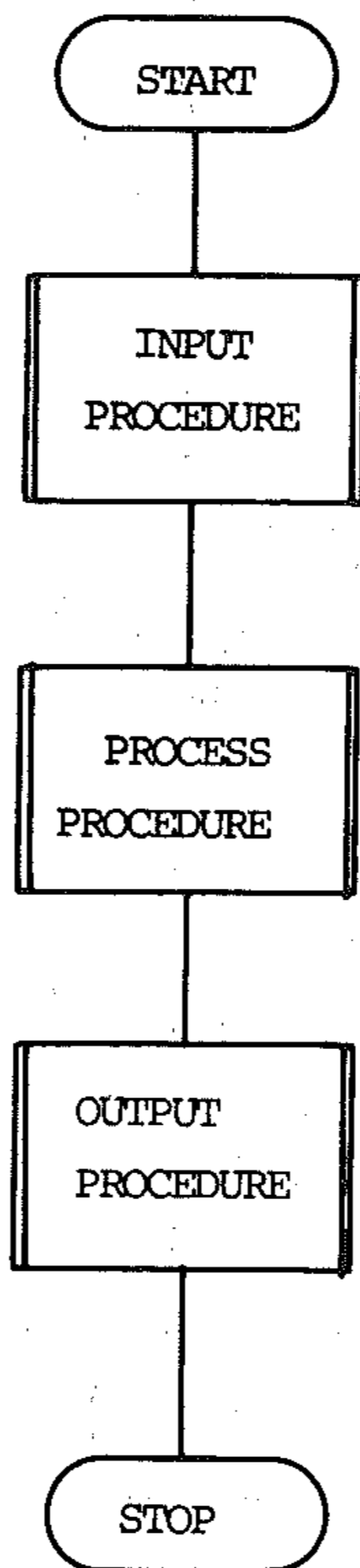
Each of modules 1.1, 1.2 and 1.3 will be subroutines. These will be called in the appropriate sequence by the main program module.



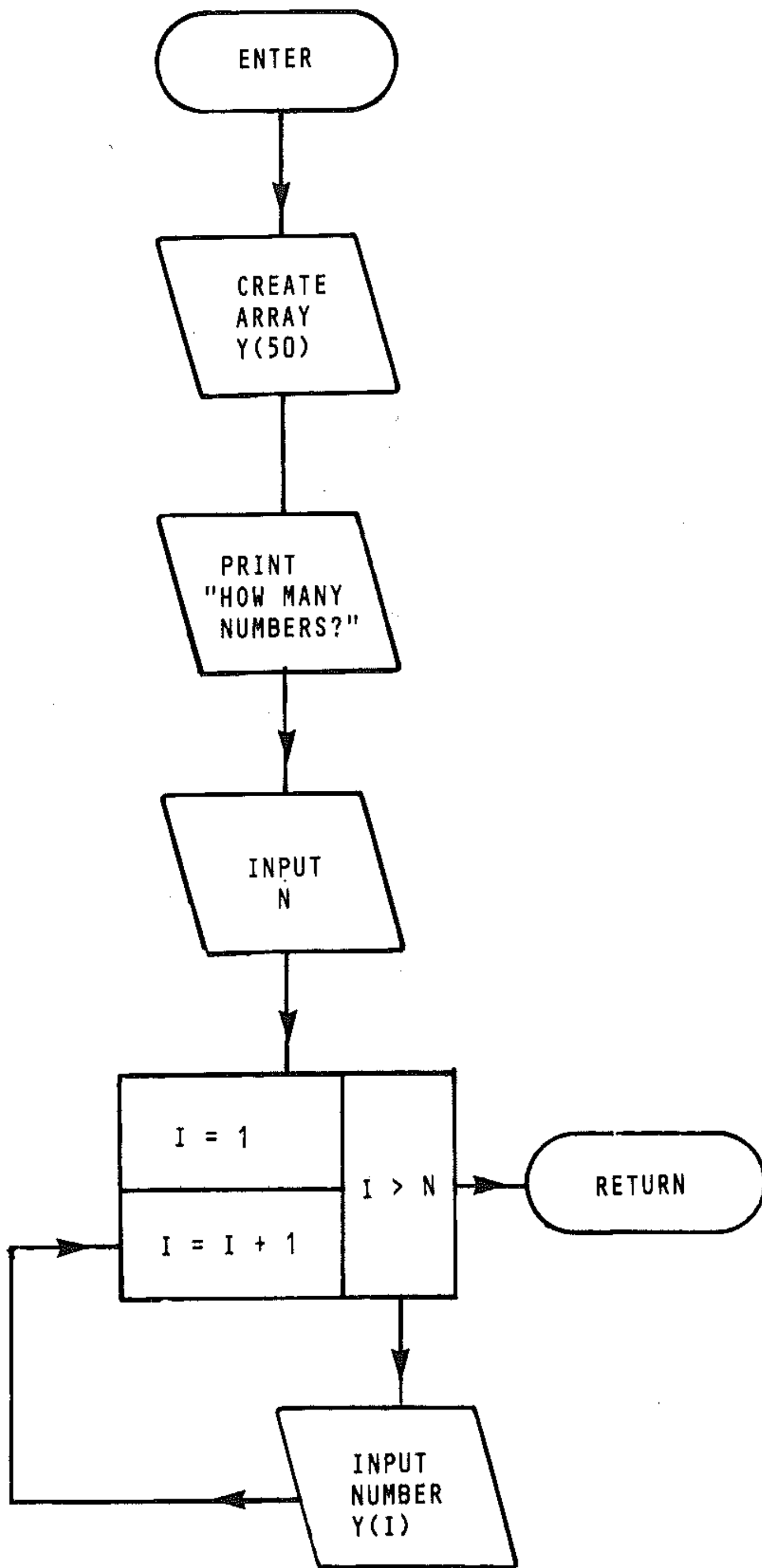


6 The flowcharts.

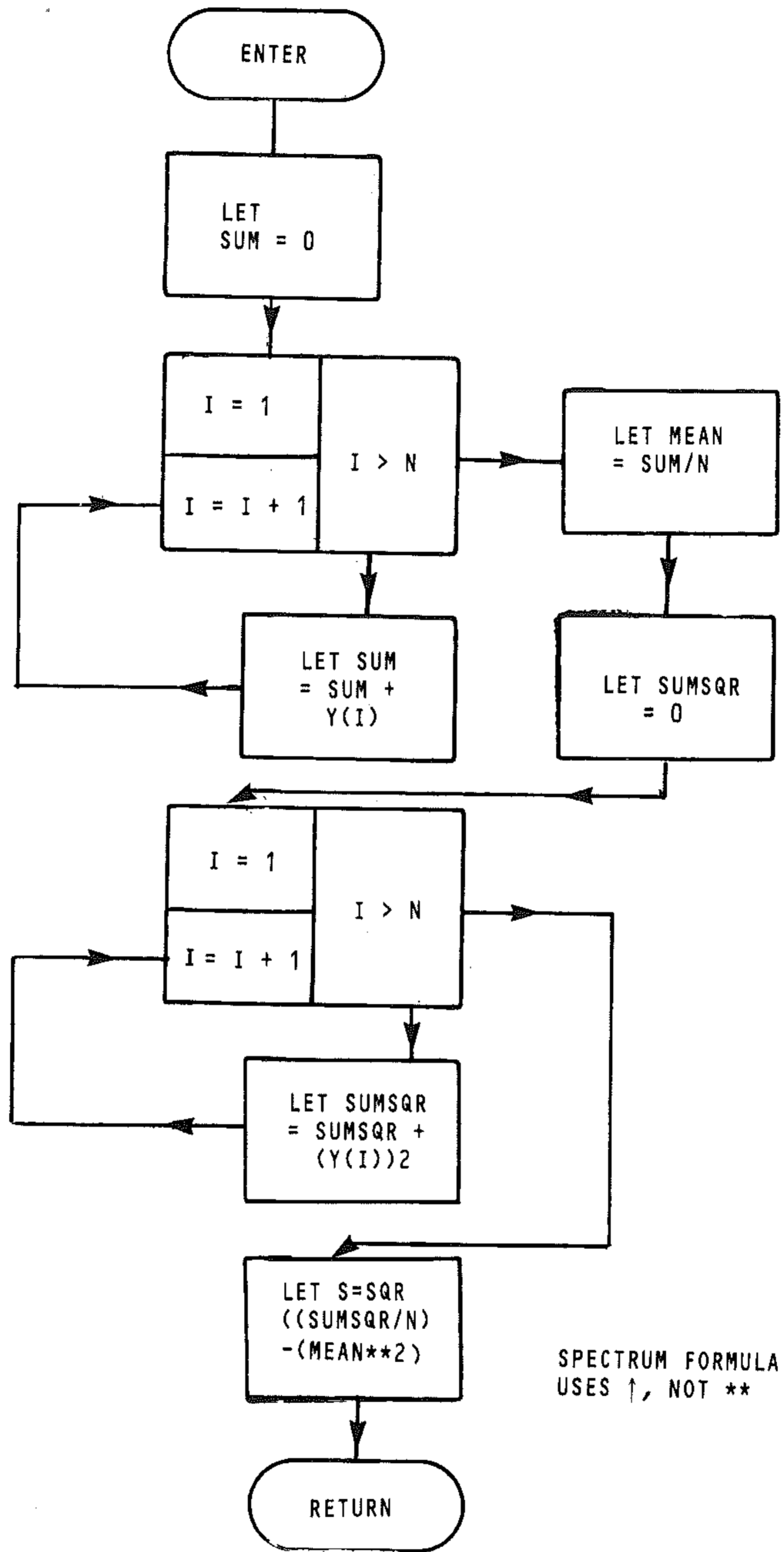
The main program module flowchart:



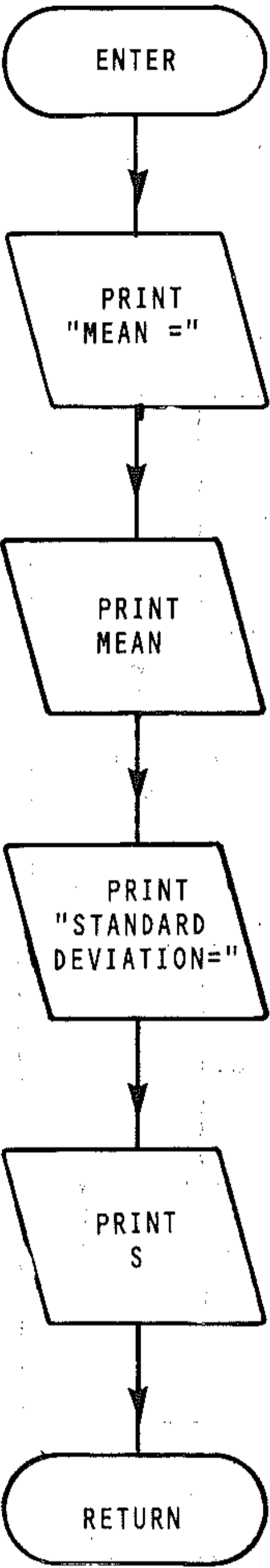
The INPUT subroutine flowchart:



The processing subroutine flowchart:



The output subroutine flowchart:



O7: The program

```
5 REM  **SDEVIATION          **
10 REM  **MAIN PROGRAM MOD**
20 REM  **INPUT DATA        **
30 GOSUB 100
40 REM  **CALCULATE          **
50 GOSUB 200
60 REM  **PRINT RESULTS      **
70 GOSUB 400
80 END
90 REM  **END MAIN           **
100 REM**INPUT SUBROUTINE**
105 DIM Y(50)
110 INPUT "HOW MANY NUMBERS"; N
120 FOR I=1 TO N
130 INPUT Y(I)
140 NEXT I
150 RETURN
160 REM**END INPUT SUB      **
200 REM**CALCULATION SUB   **
210 SUM=0
220 FOR I=1 TO N
230 SUM=SUM+Y(I)
240 NEXT I
250 MEAN=SUM/N
260 SUMSR=0
270 FOR I=1 TO N
280 SUMSR=SUMSR+(Y(I)2)
290 NEXT I
300 S=SQR((SUMSR/N)-(MEAN2))
310 RETURN
320 REM**END CALC SUB      **
400 REM**OUTPUT SUB       **
410 PRINT
420 PRINT "MEAN= "; MEAN
430 PRINT
440 PRINT "STANDARD DEVIATION= "; S
450 RETURN
460 REM**END OUTPUT SUB   **
```

O8: The documentation

- This program will compute and print the Mean and Standard Deviation of a collection of data items (numbers).
- It allows for a maximum of 50 items to be entered. You can increase the size of array Y if you wish to deal with more data.

- The numbers can be of any size, positive or negative, to the limit of the computer's handling capacity. This is large; you will not exceed it.
- To run the program key in RUN, and enter numbers one at a time, and press RETURN after each one has been keyed in.

Sample run to find Mean and Standard Deviation of 30, 31, 32, 5, 6, 7, 10, 13, 27, 3:

```
HOW MANY NUMBERS? 10
30 31 32 5 6 7 10 13 27 3
MEAN = 16.4
STANDARD DEVIATION = 11.45603
```

EXERCISE

The example program to compute and print the standard deviation of a set of data items does not include a pseudocode description of the algorithm, and the documentation process is incomplete in other ways, too. Complete the programming procedure by doing the following:

- Write out a pseudocode description of the algorithm.
- Perform a pre-coding walk through, checking the values of the variables, counters and expressions for each subroutine module.
- Key in the program and debug it.
- Insert breakpoints in each subroutine and perform a program trace. Insert PRINT statements to print out values of variables, counters and expressions.
- Obtain a program listing from the printer and run the program for a sample set of data. Keep a copy of the printer output.
- Document the program fully in your notebook.

Examples:

```
10 OPEN 1,1,1, "TAPE FILE":PRINT #1, "WRITE TO TAPE"  
10 OPEN 2,1,0, "TAPE FILE":INPUT #2 A$:REM**DATA**  
10 OPEN 3,4:PRINT #3, "OUTPUT TO PRINTER"  
10 OPEN 4,8,3,"0:DISK-FILE, S,W,":PRINT #4,"DATA TO DISK"
```

P3: Files with cassette recorder

Cassette tapes are useful for storage of files because they have an enormous capacity; the longer the tape, the greater the storage. The major drawback in using long tapes is the length of time it takes to find the required data. The most suitable method is to use C-12 or C-15 tapes for data storage and to keep data files on different tapes from your program tapes. For both types of tape you must keep an index (using the tape counter) to where the files are on the tapes.

Because data files on a cassette tape are sequential, if a file needs to be updated this can be done only by reading the whole file into memory, altering it and then saving it again. If your files grow too big to put in memory, there are two possible solutions: either split the file up into several files, or upgrade your system by getting a floppy disk drive. A disc system is expensive but allows greater flexibility than using cassettes.

Writing data to a file

The PRINT # statement is used.

Try this example:

```
10 OPEN 1,1,1,"ALPHABET"  
20 PRINT#1,"ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
30 PRINT#1,"END OF ALPHABET"  
40 CLOSE 1
```

Enter this program and run it.

Now enter the next program, rewind the tape to the beginning of the file you created above and run it.

```
10 OPEN 1,1,0,"ALPHABET"  
20 INPUT#1,A$  
30 INPUT#1,B$  
40 PRINT A$,B$  
50 CLOSE 1
```

The screen should show:

ABCD... (end of alphabet)

Now consider this program:

```

10 OPEN 1,1,1,"TEST"
20 A$="TEST"
30 B$="TEST1"
40 C$="TEST2"
50 PRINT#1,A$,B$,C$
60 CLOSE 1

```

This writes three strings to a file (or does it?). Enter it, and run it. Now rewind the tape and enter and run the next program.

```

10 OPEN 1,1,0,"TEST"
20 INPUT#1,A$,B$,C$
30 PRINT A$
40 PRINT B$
50 PRINT C$
60 CLOSE 1

```

Look at the output and the program carefully. Notice that A\$ contains Test Test1 Test2, and B\$ and C\$ contain nothing.

What has happened is as follows:

If you printed A\$, B\$, C\$ in the first program to the screen rather than to the cassette, the output would be separated by 1 to 10 spaces, depending on the length of each string. Thus, the screen would show:

```

Test (spaces) Test1 (spaces) Test2 (carriage return -
CHR$(13))

```

Now the cassette would contain a similar pattern.

In the above program, your input from the cassette, A\$ contains Test Test1 Test2 (carriage return) including the spaces. The input statement takes the carriage return on tape as an end of data marker. There are two solutions: either enter one line of data per print statement or use an end of data marker (a carriage return).

Consider, and run, each of these three programs. They do exactly the same thing.

```

10 OPEN 1,1,1,"TESTA"
20 A$="TEST"
30 B$="TEST1"
40 C$="TEST2"
50 PRINT#1,A$
60 PRINT#1,B$
70 PRINT#1,C$
80 CLOSE 1

```

```

10 OPEN 1,1,1,"TESTB"
20 A$="TEST"
30 B$="TEST1"
40 C$="TEST2"
50 D$=","
60 PRINT#1,A$,D$,B$,D$,C$
70 CLOSE 1

```

```

10 OPEN 1,1,1,"TESTC"
20 A$="TEST"
30 B$="TEST1"
40 C$="TEST2"
50 D$=CHR$(13)
60 PRINT#1,A$,D$,B$,D$,C$
70 CLOSE 1

```

- Line 10** opens a logical file number (channel) which will be referred to as 1.
- Line 20** then says that this file number (channel) will send output to the screen which is referred to as 3 (the device number of the screen). This is a *stream*. Thus the stream is associated with a channel. Line 20 prints information to logical file number 1, which outputs it to device 3 (the screen).
- Line 30** closes the channel and frees it for later use.

The following table shows the devices available:

Device	Device no.	Secondary number	String
Cassette	1	0=input 1=output 2=output with Eot	File Name
Printer	2	0	Control register
Screen	3	0, 1	
Printer	4 or 5	0=upper case/graphics 7=upper/lower case	Text
Disk	8 to 11	0=program file load 1=programme file save 2-14 = data channels 15 = command channel	Drive No. Program name as above. Drive No., file name, file type, read/write command.

Now enter this program, rewinding the tape to the beginning of file TESTA.

```

10 INPUT "ENTER A, B, OR C"; A$
20 IF A$ <> "A" AND A$ <> "B" AND A$ <> "C" TH
EN 10
40 OPEN 1, 1, 0, "TEST"+A$
50 INPUT #1, A$, B$, C$
60 PRINT A$
70 PRINT B$
80 PRINT C$
90 CLOSE 1

```

Run the above program 3 times. Enter first a, then b and finally c. The results should be the same.

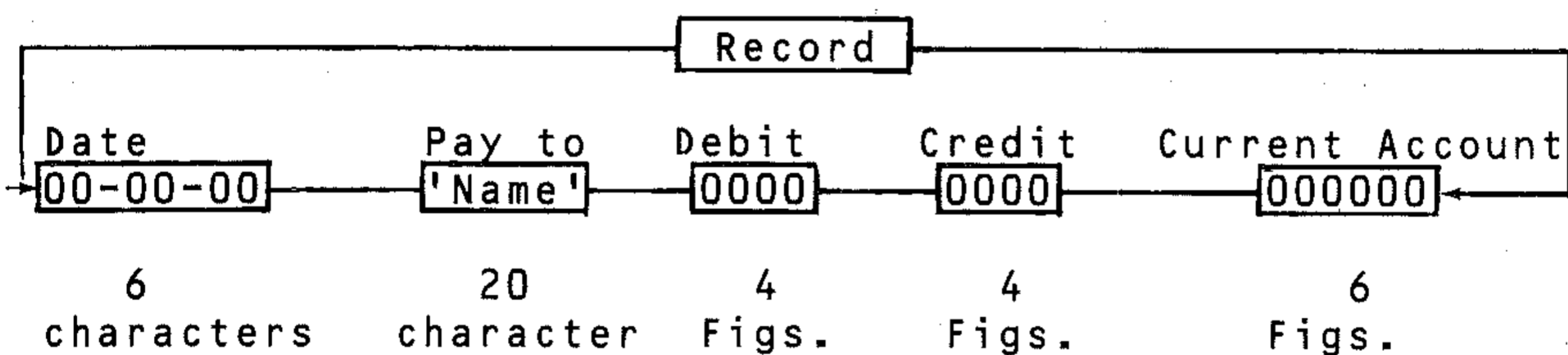
P4: Structure of files

Files should be an ordered collection of data. If you mix numeric, string or integer in your data items, make sure you know in what order they are written so that they can be read properly. For long files, it is a good idea to have two or more files containing the information.

For example, consider a payment account structure:

<u>Date</u>	<u>Pay to</u>	<u>Debit</u>	<u>Credit</u>	<u>Current account</u>
-------------	---------------	--------------	---------------	------------------------

If you construct one file this way:



...the file could become quite large. Each time an entry is to be added the whole file must be read into memory, updated and written back. As the file becomes bigger, memory will fill up and no more data can be accepted. One solution is to have a file for each item, read the file out, insert a new tape, write it out, insert old tape, write it out, etc. Another solution is to leave a large gap, read tape, rewind tape, rewrite, read next file, rewind to start of that file, rewrite, etc.

Yet another solution is to create one file, work out how much memory each record takes up, then limit the file to that number of records, create a new file and continue.

Thus,

Date	Pay to	Name	Debit	Credit	Current Account	
						Record 1
						Record 2
						Record N

File 1

Date	Pay to	Name	Debit	Credit	Current Account	
						Record N+1
						Last record

File 2

This can easily be achieved by keeping a record, in the file, of the number of entries, and updating it for each new record created.

Finally, it is important that a marker (record) is kept to indicate an end of file, otherwise you will get an error. The simplest and best way, when using string data, is to write EOF into the file and check for it when reading the file, using a statement like

```
IF A$ = "EOF" then.....
```

P5: Using a disc drive

The use of a disc drive is highly recommended, as it saves a lot of time in loading and saving programs. For serious programs and more specifically file handling applications, the use of a disc drive is essential. The cost of a single disc drive is higher than that of a cheap cassette recorder, but it is not a luxury.

Unlike most computer systems, the 64 does not possess a D.O.S (Disk Operation System). Commodore calls their disc unit *intelligent*. This means that you can send it commands. For example: to create a sequential file, the sequence is as follows:

```
OPEN 1,8,2,"0:File Name,S,W"  
PRINT#1, "Data"
```

where

```
File Name = name of the file  
S         = sequential  
W         = write
```

will create a file which is sequential and can be written to.

When you close the file (CLOSE 1) your file can now only be read.

```
OPEN 1,8,2,"0:File Name, S,R"  
INPUT#1,Data
```

(where R = read) will read data from the file.

The Commodore disc drive uses 5 1/4 inch *single sided single density* magnetic discs. Before these can be used they must be *formatted* for the Commodore (the same applies to any other computer.) The surface of the disc must be divided into tracks; for the Commodore 64 there are 35 tracks each subdivided into sectors. Formatting tells the disc where each track is and how many sectors. Also, each disc is given a name and an identification code (ID).

To format a disc:

```
OPEN 1,8,15
PRINT #1, "NO: File Name, ID"
CLOSE #1
```

where:

```
1      = file number
8      = disc unit
15     = command channel
N      = new
O      = drive 0
```

This takes about one minute and the disc is ready to be used. It is a good idea to format discs as you buy them, so that you know they are all formatted and to save time later on.

P6: Loading and saving programs

When a formatted disc is first inserted into the Commodore 64 disc drive, and on every subsequent use of it, you *must* execute the following statements:

```
OPEN 1,8,15
PRINT #1, "I"
CLOSE 1
```

This must be done every time a disc is inserted into the drive. Basically, I (initialise) reads the name and ID of the disc and allows its use.

To save a program, use:

```
SAVE "File Name", 8
```

and to load:

```
LOAD "File Name", 8.
```

P7: Errors

When an error occurs, the indicator light on the front of the disc drive will flash. This means that you must clear the error before any other operation is done with the drive. To accommodate Commodore 'intelligent drive' you must run the following program:

```
10 OPEN 1,8,15
20 INPUT#1, A$, B$, T, S
30 PRINT A$, B$
40 PRINT T, S
50 CLOSE 1
```

A\$ and B\$ contain the error number and message respectively. T and S contain the track (T) and sector (S) where the error occurred; 0 indicates a syntax error. If you execute the above program when no error has occurred, you should get this printed on the screen:

```
0 OK
00 00
```

showing no error has occurred. Whenever any files are used, it is a good idea to check the command channel (as above) to see if any errors have occurred. This should be done after each read or write operation.

P8: Working with files

The Commodore drive supports the following files:

Program (load/save)
Seq (sequential)
Rel (random access)

You have read in the cassette section above about sequential files, but there are a few differences when using the disc drive. Use OPEN LFN, 8, channel, 0:name, type, direction. For example, to write a file:

```
OPEN 1,8,10,"0:File,S,W"
```

and to read this file, use this:

```
OPEN 1,8,10"0:File,S,R"
```

Try this example:

```
10 OPEN 1,8,15
20 OPEN 2,8,10,"FILE,S,W"
25 INPUT#1,A$,B$,T,S
26 IF A$<>"0" THEN 90
30 FOR I=1 TO 10
40 PRINT#2,I
50 NEXT
60 INPUT#1,A$,B$,T,S
70 IF A$<>"0" THEN 90
80 PRINT"DATA WRITTEN":GOTO 100
90 PRINT"ERROR      "A$;B$;T;S
100 CLOSE 2:CLOSE1
110 END
```

It is unlikely that you will get an error. If you do, the most likely cause is that that file already exists. Enter this:

```
OPEN 1,8,15
PRINT #1,"S0:File"
CLOSE 1
```

Now enter the above program, changing the following lines:

```
20 OPEN 2,8,10,"File,S,R"
40 INPUT #2,I
45 PRINT I
80 PRINT "Data read"
```

P9: Random and relative files

The advantage of a random file is that it removes the necessity of reading all the data from the beginning of the file in order to find the data that you want. Random files are organised in records, and each record can be accessed individually. The best way to work with random access files is to use *relative files*. Each relative file can have 720 records, each record having up to 254 characters.

To create a relative file, use this format:

```
OPEN file #, device #, channel #, name,L, + CHR$(record length)
```

Thus:

```
OPEN 2,8,2"File,L," + CHR$(100)
```

this file will be associated with device 8 (disk), channel 2, and will contain 100 records.

Once such a file has been created, it can be referenced using a simpler form of OPEN:

```
OPEN file #, device #, channel #, "Name"
```

This format knows the file is relative, and allows both reading and writing.

Using relative files

Before you can read or write to a relative file, you must position the record pointer to the required record. Use a statement like this:

```
PRINT #15, "P" CHR$(2) CHR$(1) CHR$(0) CHR$(20)
```

in which

```
P CHR$(2)      is the channel number
CHR$(1) CHR$(0) is the record number in low-high order
CHR$(20)      is the position in record
```

You must give record numbers in low-high order because although you can access 720 records, CHR\$ can only access from 0 to 255. Thus, for example, to access record 256, the low (rec#lo) is 0 and the high (rec#hi) is 1. To convert a number to low-high, divide it by 256 and take the integer. To get the low, multiply the integer by 256 and subtract from original number. For example (rec is the record number):

```
rechi = INT(Rec/256)
reclo = rec - 256 * Rechi
```

Also, a relative file is like a sequential file in that you must write into it a separator, comma or carriage return, so that input picks up the right data. This must be taken into account when working out the record length. For example:

Record structure	name,	address,	phone	no.				
No. of characters	20,	30,	10					
Total characters	20	+ 30	+ 10	+ 3	=	63		

including the two commas and a carriage return.

P10: The disc unit

This section is only for reference for those who wish to experiment with the disc unit. It gives the error codes and a brief description of them, and the format of disc storage. Further details can be found in the handbook delivered with the drive.

Disc formats: Structure of individual directory entries

Byte	Definition
0	File flag OR'ed with \$80 0 = DELETED 1 = SEQ 2 = PRG 3 = REL
1 - 2	First data block track and sector.
3 - 18	Reserved for file name.
19 - 20	Track and sector of first side sector block (REL files only).
21	Record size for REL files.
22 - 25	Not used.

- 26 - 27 Track and sector of replacement file using '@' operations.
28 - 29 Number of blocks in file.

Standard file formats

<i>Byte</i>	<i>Definition</i>
0 - 1	Track and sector of next sequential block.
2 - 255	Reserved for file storage.

P11: Output to a printer

There are two ways of connecting a printer, depending on whether you are using the Commodore printer or some other make, for which you would have to buy a standard RS232 plug-in module.

To operate a printer with the Commodore 64 you have to learn another command: the CMD command. As the Commodore has no directed List command (List# output device), a channel to the printer is opened using:

```
OPEN 4,4,0 or 4,5,0 (upper case/graphics)
OPEN 4,4,7 or 4,5,7 (upper/lower case)
```

Then enter the CMD command:

```
CMD 4
followed by
LIST
```

CMD directs all output that would go to screen to the device selected (in this case the printer). You can also use the PRINT# command to print information to the printer. For example:

```
10 OPEN 4,4,0
20 INPUT "Enter your name" JAS
30 PRINT "Your name is" jAS
40 PRINT#4, "Your name is" JAS
50 CMD 4
60 LIST
70 PRINT#4
80 CLOSE 4
```

Finally, this section does not cover two specialised features of the disc unit, as they are really beyond the scope of this text. These are the commands to program the disk controller, and the block commands. Use of the block commands to access the track and sectors is messy, and best avoided unless you wish to experiment. These things are covered in detail in the drive manual.

Section Q: Colour and Sound

The Commodore 64 provides 16 colours that can be used with characters, border, screen, and various graphics modes (such as programmable characters, sprites, and high resolution graphics). This is a powerful and useful facility which will add to your enjoyment in programming. It will greatly increase the visual impact of games you may write and can be used to make program output clearer by colouring output messages and results. This section deals with the standard character mode and round-up of colours; graphics are covered in the next section.

Q1: Standard character mode

The Commodore 64 assumes this mode when you switch on, with the border (outer part) and screen (inner part) colours being light blue and blue respectively. The colour of these areas of screen can however be changed to any of the available 16 colours. This is done by POKEing the code (see table Q1) of the required colours into the registers at locations 53280 and 53281. The following colours are available:

COLOUR CODE	COLOUR	OBTAINED BY
		CTRL +
0	black	[1] or CHR\$(144)
1	white	[2] or CHR\$(5)
2	red	[3] or CHR\$(28)
3	cyan	[4] or CHR\$(159)
4	purple	[5] or CHR\$(156)
5	green	[6] or CHR\$(30)
6	blue	[7] or CHR\$(31)
7	yellow	[8] or CHR\$(158)
		C = +
8	orange	[1] or CHR\$(129)
9	brown	[2] or CHR\$(149)
10	light red	[3] or CHR\$(150)
11	grey 1	[4] or CHR\$(151)
12	grey 2	[5] or CHR\$(152)
13	light green	[6] or CHR\$(153)
14	light blue	[7] or CHR\$(154)
15	grey 3	[8] or CHR\$(155)

TABLE Q1: COLOUR TABLE AND CODES

So to change the border colour to black, all we have to do is:

```
POKE 53280,0
```

and to change the screen colour to white:

```
POKE 53281,1
```

Now to see all the possible combinations of border and screen colours, run the following program:

```

10 FOR I=0 TO 15
20 FOR J=0 TO 15
30 POKE 53281,I
40 POKE 53280,J
50 FOR K=1 TO 500
60 NEXT K,J,I

```

The outer loop (I) changes the screen colour once for every 16 border colour changes by loop (J). It is possible to change the text colour too. This can be done either by using the **CTRL** or the **C=** along with the number keys, or by POKEing the colour code into location 646 which determines the current colour for characters.

Now add the following lines to the program above:

```

25 FOR L = 0 TO 15
45 POKE 646, L
46 PRINT "[CLR][CD][CD][CD][CR][CR]COMMODORE 64"

```

and change line 60 to read:

```

60 NEXT K,L,J,I

```

and run the program again. The prompt COMMODORE 64 will be printed in all the available colours.

Q2: Screen and colour memory

The screen on the Commodore 64 has 40 columns and 25 rows, making the screen size 1000 bytes long (see Fig. Q1). These 1000 bytes are located at addresses 1024 to 2023 on the screen memory. Associated with these 1000 bytes of screen memory are 1000 bytes of colour memory, which are located at 54272 locations further on; that is, the colour memory starts at location 55296 through to 56295 (see Fig. Q2). Characters are displayed on the screen by either PRINTing them there, or by POKEing the respective memory location with the screen character code (see list of codes in the Appendix). Note that the screen character code is different from the ASCII character code. But you will not see anything on the screen just by POKEing the screen memory with the character code, since you have not POKEd the corresponding colour memory with the colour code. To work out positions on the screen memory and the corresponding colour memory, use the following formulas:

Screen position	= 1024+C+40 * R
Colour position	= screen position + 54272

where C and R are column and row respectively. So to put a purple A at column 20, row 12, use:

```
P = 1024+20+12 * 40
```

```

POKE P,1
POKE P+54272,4

```

or

```

POKE 1024+20+12 * 40,1
POKE 55296+20+12 * 40,4

```

The following program will display all the screen characters and their codes:

```

10 PRINT" "
20 P=1024+20+1*40
30 POKE 54272+P,4
40 FOR I=1 TO 255
50 POKE P,I
60 PRINT"CODE= ";I" "
70 FOR J=1 TO 200:NEXT J,I

```

This method of outputting characters on the screen can be used to produce animation. This effect is demonstrated by the next program. The cursor keys are used to make an asterisk move around the screen.

```

10 PRINT" "
20 R=0:C=0
30 IF C<0 THEN C=39:R=R-1
40 IF C>39 THEN C=0:R=R+1
50 IF R<0 THEN R=24:C=C-1
60 IF R>24 THEN R=0:C=C+1
70 P=1024+C+R*40
80 POKE 54272+P,3
90 POKE P,42
100 GET A$:IF A$="" THEN 30
110 IF A$=CHR$(17) THEN R=R+1
120 IF A$=CHR$(29) THEN C=C+1
130 IF A$=CHR$(145) THEN R=R-1
140 IF A$=CHR$(157) THEN C=C-1
150 POKE P,32:GOTO 30

```

Demonstrate the importance of the statement POKE P,32 at line 150 by changing the line to read:

```
150 GOTO 30
```

Now run it again. As you move the character around, you keep duplicating the character in the new position, as previously you were erasing the old position by putting a space at that position.

In the next example, a microsnake moves at random around the screen. The snake is a 10-element string of characters. Each element is printed individually in sequence as the direction of the snake changes.

```

10 PRINT" " : POKE 53280,5:POKE53281,3
20 DIM R(12),C(12)
30 REM SNAKE CHARACTERS
40 FOR I=1 TO 10:READ A$:S$=S$+A$:NEXT
50 REM START COORDINATES
60 FOR I=1 TO 12:READ R(I),C(I):NEXT
70 GOSUB 230
80 REM NEW DIRECTION
90 FOR I=1 TO 100
100 SR=INT(RND(1)*3+1):SC=INT(RND(1)*3+1
)

```

COMMODORE SCREEN AND COLOUR MEMORY MAPS

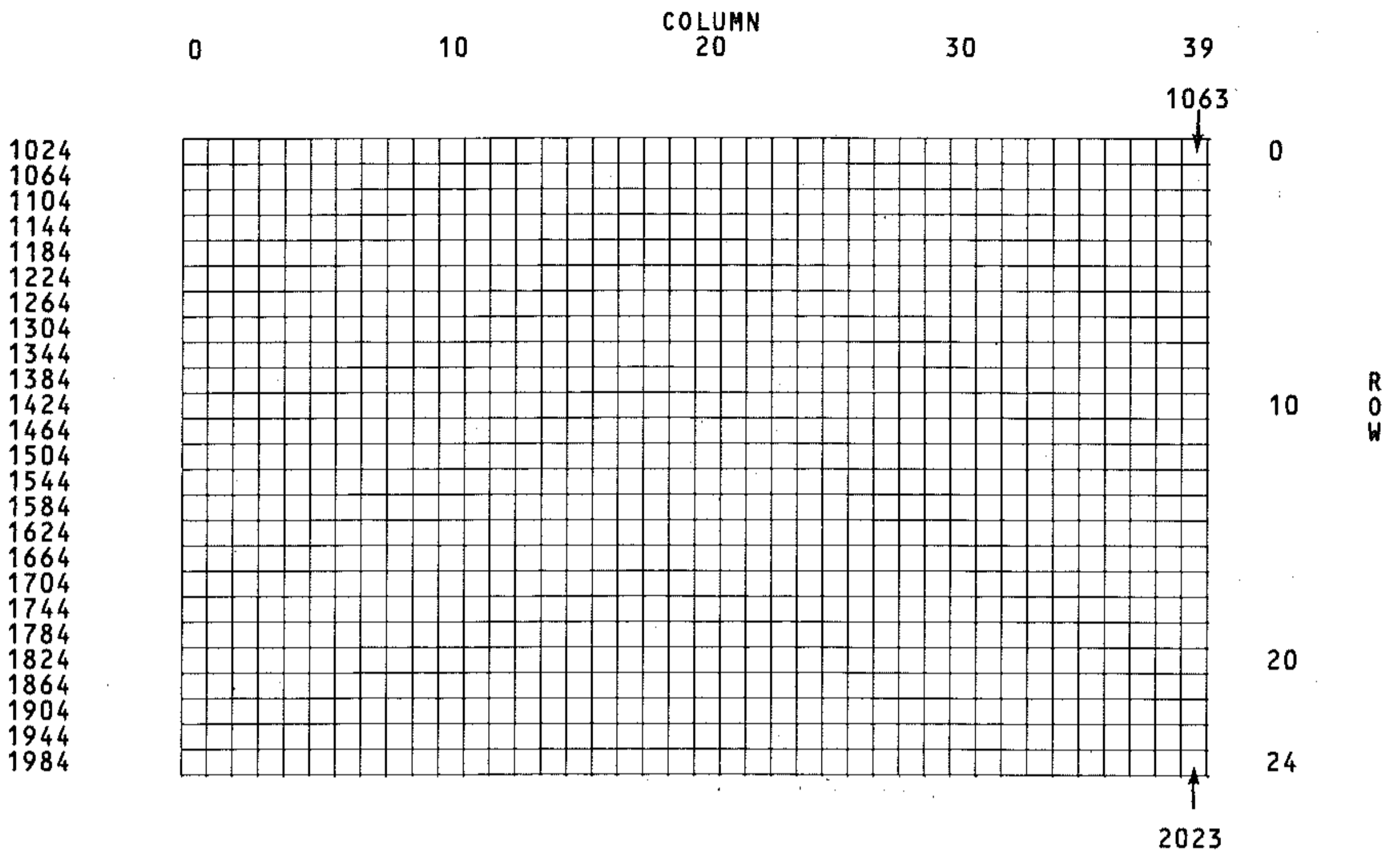


Figure Q1: Screen memory map

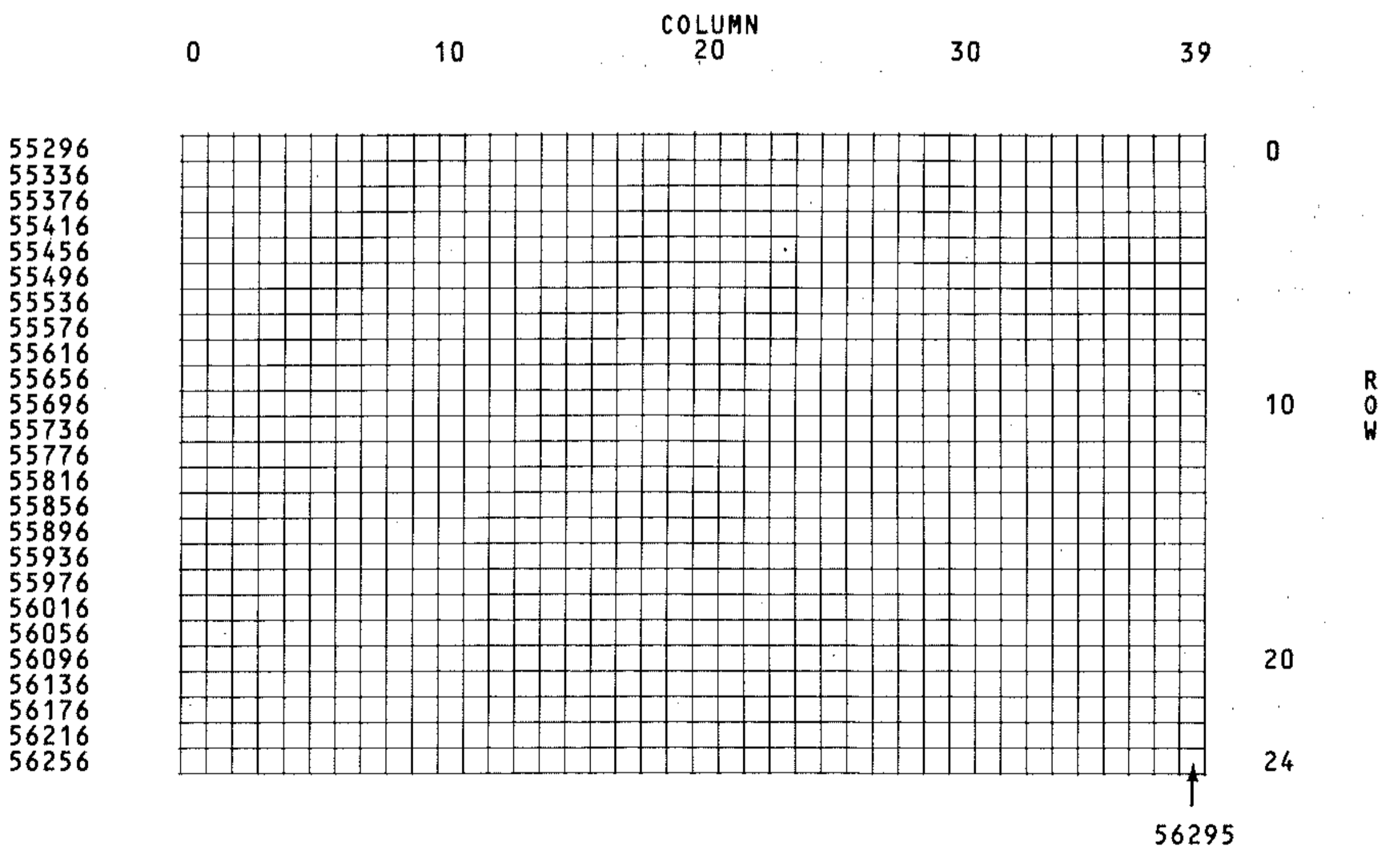


Figure Q2: Colour memory map


```

110 FOR J=1 TO 3
120 R(1)=R(1)+(1 AND SR=1)+(0 AND SR=2)+
(-1 AND SR=3)
130 C(1)=C(1)+(1 AND SC=1)+(0 AND SC=2)+
(-1 AND SC=3)
140 REM CHECK EDGE
150 IF R(1)<0 OR R(1)>23 OR C(1)<0 OR C(
1)>39 THEN 90
160 REM SHIFT COORDINATES
170 FOR K=11 TO 1 STEP -1
180 R(K+1)=R(K):C(K+1)=C(K):NEXT
190 GOSUB 230
200 NEXT J,I
210 PRINT" "
220 END
230 REM SUNROUTINE PRINT
240 POKE 781,R(12):POKE 782,C(12):SYS655
20:PRINT" "
250 FOR I=10 TO 1 STEP -1
260 POKE 781,R(I+1):POKE 782,C(I+1):SYS6
5520:PRINT" "MID$(S$,I,1)
270 NEXT
280 RETURN
290 DATA M,I,C,R,O,S,N,A,K,E
300 DATA 11,22,11,21,11,20,11,19,11,18,1
1,17,11,16,11,15,11,14,11,13,11,12,11,11

```

Here is how the program works:

Line 20	dimensions the row and column arrays
Line 40	reads the 10 character string S\$
Line 60	reads the start coordinates for printing the snake
Lines 90 – 130	select a new direction from randomly generated coordinates of the snakes head
Line 150	edge boundary value is checked
Lines 160 – 180	give new print coordinates for the array

Q3: Sound and music

Sound on the 64 is created by the powerful 6581 SID chip, which is capable of producing three different voices, each having eight octaves. Of these voices one, two or all three can be played simultaneously. Production of sound is done by POKEing values (0 – 255) into the 28 registers concerned with sound generation. These registers are located at memory locations 54272 to 54300 in the memory map. The registers are:

<i>Register</i>	<i>Action</i>
Voice 1	
0	low frequency value of note
1	high frequency value of note
2	low pulse
3	high pulse
4	waveform
5	attack/decay
6	sustain/release
Voice 2	
7	low frequency value of note
8	high frequency value of note
9	low pulse
10	high pulse
11	waveform
12	attack/decay
13	sustain/release
Voice 3	
14	low frequency value of note
15	high frequency value of note
16	low pulse
17	high pulse
18	waveform
19	attack/decay
20	sustain/release
21	low frequency cut-off (0 – 7)
22	high frequency cut-off (0 – 255) resonance (bits 4 – 7) bit 0, filter voice 1 (turn off voice 1) bit 1, filter voice 2 (turn off voice 2) bit 2, filter voice 3 (turn off voice 3)
24	volume control voice 1, voice 2 and voice 3 (bits 0 – 3)
25	access to output of envelope generator voice 3
26	digitalized output from voice 3
27	digitalized output from envelope generator 3

Q4: Playing a note

In order to be able to play a note on the 64, we must set parameters such as attack/decay, sustain/release, volume, etc of the voice concerned. These parameters are as follows (enter the program lines as they are given):

Volume

The 4 least significant bits of the register of location $S + 24$ control the volume for all three voices, where $S = 54272$. Since only 4 bits control the volume, therefore the maximum volume setting is 15 (ie 0 lowest and 15 highest). If a value higher than 15 is used, depending on what bit is set, then filtering operation will be carried out on the respective voice output. So the volume is set using the following command:

10 S = 54272

20 POKE S + 24, 15; REM SET VOLUME TO MAX.

Attack/decay

Controls the rate at which the note rises to its peak volume and then falls again. The 4 L.S.B. of the registers at locations S + 5, S + 12 and S + 19 control the decay for voices 1, 2 and 3 respectively, and the 4 M.S.B. of these registers control the attack.

location	ATK4	ATK3	ATK2	ATK1	DEC4	DEC3	DEC2	DEC1
54277	128	64	32	16	8	4	2	1
54284	128	64	32	16	8	4	2	1
54291	128	64	32	16	8	4	2	1

30 POKE S + 5, 9 : REM A = 0, D = 9

This produces no attack, but a fairly long decay for voice 1.

Sustain/release

As in attack/decay there are three registers for the three voices, and the sustain and release for each voice is controlled by one register. These registers are located at addresses S + 6, S + 13 and S + 20 for voices 1, 2 and 3 respectively. The sustain/release register prolongs a note at a certain volume and releases it.

location	SUS4	SUS3	SUS2	SUS1	REL4	REL3	REL2	REL1
54278	128	64	32	16	8	4	2	1
54285	128	64	32	16	8	4	2	1
54292	128	64	32	16	8	4	2	1

40 POKE S + 6, 40 : REM 8 = 32, R = 8

This produces low sustain and high release for voice 1.

Waveform

The waveform for each voice is set separately. There are four types available: triangle, sawtooth, pulse and noise.

location	triangle	sawtooth	pulse	noise
54276	17	33	65	129
54283	17	33	65	129
54290	17	33	65	129

50 POKE S + 4, 33 ; REM SAWTOOTH WAVEFORM

This provides voice 1 with a sawtooth waveform.

High frequency/low frequency

There are three pairs of registers for the three voices. Each pair provides the low and high frequency value of the note to be played. The value to be POKEd into these registers can be obtained from the frequency value table in the Appendix.

60 POKE 54272, 37 : POKE 54273, 17

This will play middle C note on voice 1 provided that the other parameters have already been set. Try running the program given so far. Note that all you hear is a continuous sound. Now add the following lines and try again:

```
70 FOR I = 1 TO 100: NEXT
80 FOR I = S TO S + 6 : POKE I, 0: NEXT
```

Line 70 determines how long the note is played for and line 80 resets all the registers associated with voice 1. Going through all this to play one note seems funny, but once you have set the parameters they need not be altered unless it is necessary for some special effect.

Now try the following program, which plays a tune on voice 1.

```
5 S=54272
10 POKE S+24,15
20 POKE S+5,29:POKE S+6,9
30 FOR J=1 TO 8
40 POKE S+4,32
50 READ LF,HF,D
60 POKE S+1,LF:POKE S,HF:POKE S+4,33
70 FOR I=1 TO D*20
80 NEXT I,J
90 FOR K=S TO S+6:POKE K,0:NEXT K
100 DATA 17,37,10,21,154,10
110 DATA 25,177,10,28,214,10
120 DATA 30,141,10,28,214,10
130 DATA 25,177,10,21,154,10
```

Here is how the program works:

Line 10	sets volume to maximum
Line 20	sets attack/decay and sustain/release
Line 40	selects sawtooth waveform for voice 1
Line 50	sets the low frequency, high frequency and duration
Line 60	plays notes on voice 1
Line 70	provides appropriate delay between notes

Now add the following DATA statements to the program of example 1 to play a nice tune. Note that you should change the loop value at line 30, since there is more data: change line 30 to read:

```
30 FOR J = 1 TO 87
```

The data for the above program is as follows:

```
100 DATA 17,37,10,21,154,10
110 DATA 25,177,10,28,214,10
115 DATA 30,141,10,28,214,10
117 DATA 25,177,10,21,154,10
120 DATA 17,37,10,21,154,10
125 DATA 25,177,10,28,214,10
126 DATA 30,141,10,28,214,10
130 DATA 25,177,10,21,154,10
140 DATA 22,227,10,28,214,10
145 DATA 34,75,10,38,126,10
146 DATA 40,200,10,38,126,10
```

```

150 DATA 34,75,10,28,214,10
160 DATA 17,37,10,21,154,10
165 DATA 25,177,10,28,214,10
166 DATA 30,141,10,28,214,10
170 DATA 25,177,10,21,154,10
180 DATA 25,177,10,32,94,10
185 DATA 38,126,10,43,52,10
186 DATA 45,198,10,43,52,5
190 DATA 38,126,10,32,94,10
191 DATA 34,75,18
200 DATA 17,37,10,21,154,10
205 DATA 25,177,10,28,214,10
206 DATA 30,141,10,28,214,10
210 DATA 25,177,10,21,154,10
220 DATA 30,141,18,28,214,5
225 DATA 30,141,5,28,214,5
226 DATA 30,141,5,28,214,5
227 DATA 25,177,18
230 DATA 22,227,10,28,214,10
235 DATA 34,75,10,38,126,10
236 DATA 40,200,10,38,126,10
240 DATA 34,75,10,28,214,10
250 DATA 40,200,18,38,126,5
255 DATA 40,200,5,38,126,5
256 DATA 40,200,5,38,126,5
257 DATA 34,75,18
260 DATA 25,177,10,32,94,10
265 DATA 38,126,10,43,52,10
266 DATA 45,198,10,43,52,10
267 DATA 38,126,10,32,94,10
270 DATA 45,198,18,43,52,5
275 DATA 45,198,5,43,52,5
276 DATA 45,198,5,43,52,5
280 DATA 38,126,18,34,75,18

```

Q5: Multiple voices

So far you have heard only one of the three individually programmable voices. As mentioned earlier, two or all three voices could be used simultaneously. To use all three voices together, set the parameters of voices 2 and 3 as you did with voice 1.

Now try the following program, which is an extended version of one of those above:

```

5 S=54272
10 POKE S+24,15
20 POKE S+5,46:POKE S+6,0
21 POKE S+12,29:POKE S+13,127

```

```

22 POKE S+19,36:POKE S+20,31
30 FOR J=1 TO 24
40 POKE S+4,32:POKE S+11,64:POKE S+18,12
  8
50 READ LF,HF,D,L1,H1,L2,H2
60 POKE S+1,LF:POKE S,HF:POKE S+4,33
63 POKE S+8,L1:POKE S+7,H1:POKE S+11,65
66 POKE S+15,L2:POKE S+14,H2:POKE S+18,1
  29
70 FOR I=1 TO D*10
80 NEXT I,J
90 FOR K=S TO S+24:POKE K,0:NEXT K
100 DATA 17,37,10,4,73,68,149,21,154,10,
  0,0,0,0
110 DATA 25,177,10,6,108,0,0,28,214,10,0
  ,0,0,0
115 DATA 30,141,10,7,163,115,88,28,214,1
  0,0,0,0,0
117 DATA 25,177,10,6,108,0,0,21,154,10,0
  ,0,0,0
120 DATA 17,37,10,4,73,68,149,21,154,10,
  0,0,0,0
125 DATA 25,177,10,6,108,0,0,28,214,10,0
  ,0,0,0
126 DATA 30,141,10,7,163,115,88,28,214,1
  0,0,0,0,0
130 DATA 25,177,10,6,108,0,0,21,154,10,0
  ,0,0,0
140 DATA 22,227,10,7,163,115,68,28,214,1
  0,0,0,0,0
145 DATA 34,75,10,6,108,0,0,38,126,10,0,
  0,0,0
146 DATA 40,200,10,5,185,91,140,38,126,1
  0,0,0,0,0
150 DATA 34,75,10,8,147,0,0,28,214,10,0,
  0,0,0

```

Q6: Filtering

Another feature of the SID chip is the provision of filtering facilities. Filtering simply means changing the harmonic contents of a waveform. There is a low pass filter, a band pass filter and a high pass filter. These are controlled by bits 4,5 and 6 of the register at location 54296. So if we wish to turn on the high pass filter, we must POKE the register with 64. There are other filters that can be used to 'turn off' voices; these are controlled by bits 0, 1 and 2 of the register at location 54295.

Section R: Graphics

The Commodore 64 is a powerful graphics computer, capable of displaying up to 16 colours. The graphics capabilities of the 64 are provided by the powerful 6567 video interface chip, also known as the VIC-II chip. This is a multi-purpose colour video controller, with 47 controllable registers that can be accessed by the commodore processor chip, the 6510; and it is capable of addressing 16K of memory at a time. It provides a variety of graphics modes, including the 40 column by 25 lines character display mode, 320 by 200 fixed high-resolution display mode, and the moveable object blocks called sprites.

The graphics display modes are:

Character display modes

- Standard character mode
- Multicolour character mode
- Extended background colour mode

High resolution (bit map) mode

- Standard bit map mode
- Multicolour bit map mode

Sprites

- Standard sprites
- Multicolour sprites

R1: User defined (programmable) characters

The VIC-II chip gets its character from the character generator ROM, which starts at location 53248. This chip contains the patterns required to form all the characters and symbols that can be seen on the keyboard.

The Commodore 64 allows manipulation of the patterns located in RAM for games, letters in different alphabets, mathematical symbols and other special symbols and shapes, all called *user-defined* or *programmable* characters. A normal character set contains 256 characters, each character being made up of an 8x8 pixel array. This means that a full character set requires 2K (256x8) bytes of memory. This 2K can be accommodated in any one of the 4 possible blocks of 16K memory, so a method is needed to tell the VIC-II chip which block to look for. The action of switching from one block to the other is known as *banking*, and each block is called a *bank*. Switching banks requires control of bits 0 and 1 of port A of CIA#2 chip at location 56576. This is done as follows:

```
POKE 56576, (PEEK(56576) AND 252) OR A
```

where A can be one of the following values:

Value of A	Bits	Bank	Starting location	VIC II chip range
0	00	3	49192	49192 – 65535
1	01	2	32768	32768 – 49191
2	10	1	16384	16384 – 32767
3	11	0	0	0 – 16383

Bits 0 and 1 of port A are set to output by setting bits 0 and 1 of location 56578 to a 1. So we must use the following instruction before we change banks:

```
POKE 56578,PEEK(56578) OR 3
```

Telling the VIC-II chip which bank to look at for data is not enough. It must also be told where in the bank data the character memory is. This is done by controlling the register at location 53272. The lower 4 bits of this register is used to control where the character set is located (only bits 3,2 and 1 are used), while the upper 4 bits control the location of the screen memory. To change the character memory location, use the following statement:

```
POKE 53272,(PEEK(53272) AND 240) OR A
```

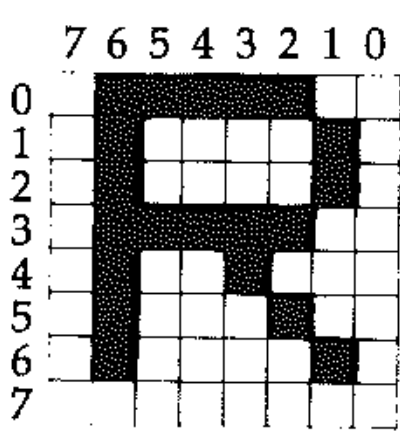
where A has one of the following values:

Value of A	Bits	Location of character memory
0	XXXX000X	0
2	XXXX001X	2048
4	XXXX010X	4096
6	XXXX011X	6144
8	XXXX100X	8192
10	XXXX101X	10240
12	XXXX110X	12288
14	XXXX111X	14336

X = DON'T CARE

R2: Defining characters

A character on the Commodore 64 is made up of an 8x8 pixel array. Here is the character R as it appears on the screen (but greatly magnified). The character is produced by linking in individual pixel elements:

	<i>Binary data</i>	<i>Decimal Data</i>
	01111100 01000010 01000010 01111100 01001000 01000100 01000010 00000000	124 66 66 124 72 68 66 0

Each character row has 8 bits or 1 byte. Each bit is either 1 if it is set, or 0 if it is unset. Therefore each character row can be described as a combination of 1's and 0's. These combinations are seen in the array of 1's and 0's next to the character.

Notice that the 1's in the diagram above give the character R. The binary number of row 0 is:

```
01111100
```

This 8-bit binary number has to be represented as a decimal number. To find it, multiply each binary bit or digit by a power of 2 according to its position.

7	6	5	4	3	2	1	0
2	2	2	2	2	2	2	2

Only the binary 1's attribute; add these to get the decimal value. So:

$$0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \quad \text{binary}$$

$$= 2^6 + 2^5 + 2^4 + 2^3 + 2^2$$

$$= 64 + 32 + 16 + 8 + 4 = 124 \quad \text{decimal.}$$

Check that the decimal values for each character row byte is correct for R. Once these decimal values for producing character R have been calculated, they have to be put into memory. This is done as in this program:


```

10 PRINT"!"
20 REM CHARACTER MEMORY AT 8192
30 POKE53272,(PEEK(53272) AND 240)+8
40 FOR I=0 TO 7
50 READ C:POKE8192+I,C:NEXT
60 DATA 124,66,66,124,72,68,66,0

```

Run the program. Note that the writing on the screen goes funny. Now press some keys: all you get is useless shapes. However, if you press the @ key you will see R is displayed on the screen. This is because @ is the first character of the screen characters (see list of characters in the Appendix) and what you have done is to reprogram it to represent character R. Here's how the program works:

Line 30 tells the VIC-II chip that the character memory starts at location 8192.
Line 40 sets up the loop for reading the data for character R.
Line 50 reads the data and reprograms the @ key to represent character R.

In the above program location 8192 of BANK 0 was used as the character's memory location. If, for example, you wished to use location 8192 of BANK 1 as the starting character memory location, you would add the following statements to the program:

```

15 POKE 56578,PEEK(56578) OR 3
16 POKE 56576,PEEK(56576) OR 2

```

and change line 50 to read:

```
50 READ C:POKE 24576 + I,C:NEXT
```

Location 24576 is obtained by the following formula:

memory location = BANK number * 16384 + character memory.

Since we are using bank 1:

location = 1 * 16384 + 8192 = 24576.

To program the R key to represent our defined character we use the following formula:

character address = character code * 8 + memory location = 18 * 8 + 24576 = 24720

so line 50 is changed again to read:

```
50 READ C:POKE 24720 + I,C : NEXT
```

Now by pressing the R key, the character R will be printed on the screen.

To program more than one character, add 8 more data values for each character, and increase the loop value by 8. Code in the following program and run it. Press the @, A and B keys, and notice that each key is programmed to represent different shapes.

```

10 PRINT"!"
20 REM CHARACTER MEMORY AT 8192
30 POKE53272,(PEEK(53272) AND 240) OR 8
40 FOR I=0 TO 23
50 READ C:POKE8192+I,C:NEXT
60 DATA 124,66,66,124,72,68,66,0
70 DATA 255,129,129,129,129,129,129,255
80 DATA 60,60,24,255,24,36,68,129

```

It seems pointless to program characters that are already available to us from the Commodore's character set; the whole idea of programmable characters is to define characters that don't already exist. To make use of some of the characters in the character set, they must be copied across from ROM into RAM memory.

Try the following program, which copies the first 128 characters of the ROM characters into RAM. LIST the program on the screen and run it. Note the changes occurring on the patterns displayed on the screen as each character is copied across.

```
10 POKE53272,(PEEK(53272)AND 240)OR 8
20 POKE56334,PEEK(56334)AND254
30 POKE1,PEEK(1)AND 251
40 FOR I=0 TO 1023
50 POKE 8192+I,PEEK(53248+I)
60 NEXT
70 POKE 1,PEEK(1)OR 4
80 POKE56334,PEEK(56334) OR 1
```

Here is how the program works:

Line 10	tells the VIC-II chip to look at location 8129 onwards for its character data.
Line 20	turns the keyscan interrupt off.
Line 30	turns the I/O ROM off and switches in the character ROM.
Lines 40 – 60	copy 128 characters (1024 bytes) from ROM into RAM.
Line 70	turns the I/O ROM on.
Line 80	turns the keyscan interrupt on.

When the program is first run, the screen display goes strange. This is because line 10 instructs the VIC-II chip to look at locations 8192 onwards for its character data, and since you have not copied any character across or programmed any character at this stage, contents of RAMs chosen at random are displayed.

Now any key that we press will be displayed on the screen, except when using the **CTRL 9** keys to obtain reverse characters. Pressing **CTRL 9** keys followed by any key will not display that character in inverse video, but a strange looking pattern, because only 128 out of 256 characters are copied to RAM.

R3: Multicolour characters

Normally Commodore 64 displays its characters in only one of the 16 available colours. It is, however, possible to have a character displayed in 4 out of the 16 available colours, hence the name multicolour characters. The Commodore is put into multicolour character mode by setting or unsetting bit 4 of the multicolour character register at location 53270. This is done as follows:

```
POKE 53270,PEEK(53270) OR 16
```

To turn multicolour mode off, use:

```
POKE 53270,PEEK(53270) AND 239
```

Once you are in in multicolour mode, the character space that was 8×8 dots becomes 4 pairs of dots across by 8 dots down. These 4 bit pairs will determine the colours in which characters are displayed on the screen.

Bit pair	Description	location
00	dot colour same as screen colour	53281
01	dot colour specified by the contents of register #1	53282
10	dot colour specified by the contents of register #2	53283
11	dot colour specified by bits 0,1 and 2 of the colour memory (foreground colour)	colour RAM

So to display the programmed character in multicolour, first put the Commodore into multicolour mode. This is done as follows:

```
POKE 53270,PEEK(53270) OR 16
```

Then use the **CTRL** key with a numbers key. This changes the foreground colour. Now if you press any of the programmed keys (ie. @, A and B), you'll see the shapes represented by these keys displayed in more than one colour. The colours at which characters are displayed depends on the bit pair pattern. Alternatively, you can provide the desired colours for colour registers #1 and #2. Add the following lines to the program of example 2 and RUN it:

```
35 POKE 53280,4:POKE 53281,3
36 POKE 53282,2:POKE 53283,0
37 POKE 53270,PEEK(53270) OR 16
```

Line 35 sets border and screen colours to purple and cyan respectively.

Line 36 sets register #1 colour (bit pair 01) and register #2 colour (bit pair 10) to red and black respectively.

Line 37 puts commodore into multicolour mode.

Here is another example program. Key it in and try it.

```
10 PRINT" "
20 POKE53270,PEEK(53270) OR 16
30 POKE53281,0:POKE53282,7:POKE53283,4
40 PRINT:PRINT"A B C D E F G H I J K L M
N O"
45 REM SELECT RANDOM FOREGROUND COLOUR
50 FOR I=1 TO 15
60 POKE55296+40+I,INT(RND(1)*7+1)
70 NEXT
80 GOTO 80
```

Line 20 sets bit pairs 01 to yellow and bit pairs 10 to purple.

Lines 50 - 70 select random foreground colour (bit pair 11) for the 15 characters printed on the screen.

R4: Extended background colour

This mode enables you to control the background, as well as the foreground colour of each character on the screen. As usual there is a price to be paid for this extra facility; you are restricted to use of only the first 64 characters of character ROM or the first 64 characters of programmable characters, instead of the usual 256. The reason for this is that two bits (6 and 7) of the character codes are used to select the background colour. This means that a screen character code of greater than 63 will put on the screen the character corresponding to itself in the first 64 characters, but in a different background colour.

For example, if you POKE 1 into the screen memory, letter A will be displayed, and if you POKE 64, then again letter A will be displayed, but in a different background colour. The following table gives the registers associated with each set of 64 characters.

Character code	Bit 7	Bit 6	Background colour register
0 - 63	0	0	53281
64 - 127	0	1	53282
127 - 191	1	0	53283
192 - 255	1	1	53284

Only characters with code greater than 63 will have different background colour, because the register at location 53281 sets the screen colour. Thus characters with codes 0 - 63 will have screen colour as their background colour.

The extended colour mode is switched on by setting bit 6 of the register at location 53265. This is done as follows:

```
POKE 53265,PEEK(53265) OR 64
```

to turn off this mode we have to reset the bit 6 as follows:

```
POKE 53265, PEEK (53265) AND 191
```

Note: this mode does not work with multicolour characters. Key in the following program, which displays the letter A on the screen in different background colour.

```
5 PRINT "A";
10 POKE 53265,PEEK(53265) OR 64
20 POKE53281,0:POKE53282,3
30 POKE53283,15:POKE53284,7
40 PRINT"BIT 7      BIT 6"
50 PRINT"      0          0"
60 POKE 1095,1
70 PRINT:PRINT"      0          1"
80 POKE 1095+80,65
90 PRINT:PRINT"      1          0"
100 POKE 1095+160,129
110 PRINT:PRINT"      1          1"
120 POKE 1095+240,193
```

R5: High resolution (bit map) mode

This mode is useful when writing games, plotting graphs (generally short recursive routines, such as drawing lines, circles, cubes, etc), because it can manipulate every individual dot (pixel) on the screen. The high resolution screen consists of 320 (40 × 8) horizontal pixels by 200 (25 × 8) down, giving a total of 64000 pixels on the screen. These figures come from the fact that the normal screen display on the 64 is 40 character spaces across by 25 down, with each character space being made up of 8 bytes or 64 bits.

In order to manipulate every pixel on the screen, a single bit of memory must be allocated to each pixel. This, however, requires 8000 (64000/8) bytes of memory. When the Commodore is put into high resolution mode, it is asking the VIC-II chip to display the contents of 8000 bytes of memory on the screen, so it must tell the VIC-II chip which 8K section of memory to display. This is done as follows:

```
POKE 53272,(PEEK(53272) AND 240) OR 8
```

Now the Commodore is put into high resolution mode by setting bit 5 of the register at location 53265. This is done as follows:

```
POKE 53265,PEEK(53265) OR 32
```

To come out of high resolution mode bit 5 of the register is reset as follows:

```
POKE 53265,PEEK(53265) AND 223
```

To see what happens when you turn on high resolution mode, run the following program:

```
5 REM SWITCH TO BANK 1
10 POKE 56576,(PEEK(56576) AND 252) OR 2
15 REM HIGH RES. SCREEN STARTS AT 8192 OF BANK 1
(I.E. 24576)
20 POKE 53272,8
25 REM SWITCH TO HIGH RES. MODE
30 POKE 53265,PEEK(53265) OR 32
```

The pattern displayed on the screen is unusual shapes of random colours, which can be cleared by POKEing the desired colour code into character screen memory. Colour in high resolution mode is not provided by colour memory, as is the case with character modes, but it is taken from the character screen memory (the first 1000 bytes of each bank, except bank 0 which starts at 1024).

The upper 4 bits of character screen memory becomes the colour of bit(s) that is/are set (i.e. foreground) in the 8×8 character space, while the lower 4 bits become the colour of bits that is/are unset (ie background). So the following formula is used to calculate the value for colour to be POKEd:

Value = foreground colour code * 16 + background colour code

To have a blue background, with light blue foreground, POKE the character screen memory with 230 (14 * 16 + 6). This is done as follows:

```
40 FOR I = 0 TO 999:POKE 16384 + I, 230:NEXT
```

Now press **STOP/RUN** and **RESTORE** keys together to get the Commodore to standard character mode. Add line 40 to the program and run it again. Notice that those unusual square shapes disappear, but there are still screenfuls of garbage. What you see is the contents of 8K RAM, and it must be cleared so that you can draw your desired shapes on the screen. To clear this 8K screen you must unset these locations – POKE 0 into them. This takes rather a long time, so be patient. Again, get out of high resolution mode and add the following line to the program:

```
50 FOR I = 0 TO 8000:POKE 24576,0:NEXT
```

Watch the screen being cleared gradually. Once the screen is completely free of garbage, you can start plotting shapes on the screen. The process of clearing the screen and setting the desired colour using BASIC is rather slow and time consuming, especially if you have to clear the screen every time you wish to draw a new graph; to speed it up a machine code routine is used that does it in a matter of seconds. Full description of this routine is given in Section U; for now it saves a considerable amount of time.

Having cleared the high resolution screen we can now plot graphs or draw lines or circles by turning on pixels of the high resolution screen. This is what high resolution screen looks like:

byte 0	8	16	-	-	-	-	-	-	312
byte 1	9	-	-	-	-	-	-	-	313
byte 2	10	-	-	-	-	-	-	-	314
byte 3	11	-	-	-	-	-	-	-	315
byte 4	12	-	-	-	-	-	-	-	316
byte 5	13	-	-	-	-	-	-	-	317
byte 6	14	-	-	-	-	-	-	-	318
byte 7	15	-	-	-	-	-	-	-	319
byte 320									
byte 321									
byte 322									
byte 323									
byte 324									
byte 325									
byte 326									
byte 327									
-									
-									
-									
-									
-									
-									
-									
-									
byte 7680		-	-	-	-	-	-	-	7992
byte 7681		-	-	-	-	-	-	-	7993
byte 7682		-	-	-	-	-	-	-	7994
byte 7683		-	-	-	-	-	-	-	7995
byte 7684		-	-	-	-	-	-	-	7996
byte 7685		-	-	-	-	-	-	-	7997
byte 7686		-	-	-	-	-	-	-	7998
byte 7687		-	-	-	-	-	-	-	7999

To set a pixel on the screen, you must know how to find the correct bit in the character memory. This becomes a fairly simple task by using the following formula:

```

ROW = INT(X/8)      finds row
COL = INT(Y/8)      finds the character position
LIN = (Y AND 7)     finds the line of that character position (0 - 7)
BIT = 7 - (X AND 7) finds bit of that byte

```

Putting these together provides the following formula, which calculates the byte in which character memory point (X,Y) is located:

```

BYTE=start of high resolution screen (24576)+ROW*20+COL*8+LIN

```

Now to set a pixel at point (X,Y), use:

```

POKE BYTE,PEEK(BYTE) OR 2-BIT

```

Now that you can control any pixel on the screen, try plotting something on the screen. You can now use routines to draw lines, circles, ellipses, spirals, cubes, etc. The following is a program that draws a line between two specified points:

```

5 REM CLEAR SCREEN AND PUT COLOUR USING
MACHINE CODE (LINES 10-50)
10 PRINT "ENTER COLOUR AS 16*FOREGROUND C
ODE + BACKGROUND CODE  "
20 INPUT CC
30 FOR I=0 TO 36:READ J:POKE49152+I,J:NE
XT
40 POKE49175,CC
50 SYS(49152)
55 REM START & END POINTS OF LINE
60 PRINT "ENTER XS,YS,XE,YE";
70 INPUT XS,YS,XE,YE
80 POKE56576,(PEEK(56576)AND 252)OR 2
90 POKE53272,B
100 POKE53265,PEEK(53265) OR 32
110 XA=XE-XS:YA=YE-YS
120 A=(XA AND (ABS(XA)>=ABS(YA)))+(YA AND
(ABS(XA)<ABS(YA)))
130 X=0:Y=0:X=X+XS:Y=Y+YS
140 FOR I=1 TO ABS(A)
150 X1=INT(X/8):B=7-(X AND 7)
160 Y1=INT(Y/8):L=Y AND 7
170 C=Y1*320+X1*8+L
180 POKE24576+C,PEEK(24576+C) OR 2+B
190 X=X+XA/ABS(A):Y=Y+YA/ABS(A)
200 NEXT
210 DATA 160,32,162,0,169,0,157,0,96,232
,208,250,238,8,192,136,208,244
220 DATA 160,4,162,0,169,0,157,0,64,232,
208,250,238,26,192,136,208,244,96

```

The logic in line 120 checks which is the greater of the distances to be covered between the points, and makes A equal to that, since the smaller value will be in a false statement, and will be evaluated as 0. The program library (see Appendix) has an expanded version of this program (called CUBE).

Now key in the following program, which uses the cursor control keys to draw shapes on the screen.

```

5 REM CLEAR SCREEN AND PUT COLOUR USING
MACHINE CODE (LINES 10-50)
10 PRINT "ENTER COLOUR AS 16*FOREGROUND C
ODE + BACKGROUND CODE  "
20 INPUT CC
30 FOR I=0 TO 36:READ J:POKE49152+I,J:NE
XT
40 POKE49175,CC
50 SYS(49152)
60 PRINT "ENTER START POINT X & Y ";
70 INPUT X,Y

```

```

80 POKE56576,(PEEK(56576)AND 252)OR 2
90 POKE53272,8
100 POKE53265,PEEK(53265) OR 32
110 X1=INT(X/8):B=7-(X AND 7)
120 Y1=INT(Y/8):L=Y AND 7
130 C=Y1*320+X1*8+L
140 POKE24576+C,PEEK(24576+C) OR 2+B
150 GET A$:IF A$="" THEN 150
160 IF A$=CHR$(17) THEN Y=Y+1
170 IF A$=CHR$(29) THEN X=X+1
180 IF A$=CHR$(145) THEN Y=Y-1
190 IF A$=CHR$(157) THEN X=X-1
200 GOTO 110
210 DATA 160,32,162,0,169,0,157,0,96,232
,208,250,238,8,192,136,208,244
220 DATA 160,4,162,0,169,0,157,0,64,232,
208,250,238,26,192,136,208,244,96

```

Note that most program lines are common to both programs, as is true for the programs to follow. The differences are in the actual routines that are required to draw lines, circles, ellipses etc. It can be seen from these programs that most drawing is done utilising loops. To draw a circle you need to choose a suitable STEP value for a loop that runs either from 0 to 360 (degrees), or 0 to $2 * \pi$ (radians). You also have to set a centre to put the circle where you want it, and a radius such that it will fit the screen.

```

5 REM CLEAR SCREEN AND PUT COLOUR USING
MACHINE CODE(LINES 10-50)
10 PRINT"ENTER COLOUR AS 16*FOREGROUND C
ODE + BACKGROUND CODE "
20 INPUT CC
30 FOR I=0 TO 36:READ J:POKE49152+I,J:NE
XT
40 POKE49175,CC
50 SYS(49152)
60 PRINT"ENTER CENTRE CORD.(XC,YC)";
70 INPUT XC,YC
80 INPUT"ENTER RADIUS(RD) ";RD
90 POKE56576,(PEEK(56576)AND 252)OR 2
100 POKE53272,8
110 POKE53265,PEEK(53265) OR 32
120 FOR D=0 TO 360 STEP 0.5
130 RA=D*PI/180
140 X=XC+RD*COS(RA)
150 Y=YC+RD*SIN(RA)
160 X1=INT(X/8):B=7-(X AND 7)
170 Y1=INT(Y/8):L=Y AND 7
180 C=Y1*320+X1*8+L
190 POKE24576+C,PEEK(24576+C) OR 2+B
200 NEXT

```



```

210 DATA 160,32,162,0,169,0,157,0,96,232
,208,250,238,8,192,136,208,244
220 DATA 160,4,162,0,169,0,157,0,64,232,
208,250,238,26,192,136,208,244,96

```

Lines 130 - 140 converts to radians, in which all the trigonometric functions of the computer work, and calculates the horizontal component, and line 150 the vertical. Having set the basic values, you can introduce variations within the loop into the above program. You can calculate the radius value and get a spiral plot. Alter line 120 as follows and run it:

```
120 RA = D * π / 180: RD = RA * 10
```

Calculating additional values, all within the loop, and using the loop values as a basis, provides complex shapes fairly easily. Alter line 120 to read:

```
120 RE = D * π / 180: RD = 90 * SIN(RA * 6)
```

Run the program. From a simple circle, you now have the basis for a polar graph plot and can identify a scale of values for plotting that will fit the screen. To draw an ellipse, for example, you have to provide a vertical radius and a horizontal radius. So make the following changes in the program of example 8 for drawing an ellipse:

```

75 INPUT "ENTER HORIZ. & VERT. RADIUS";HR,VR
140 X = XC + HR * COS(RA)
150 Y = YC + VR * SIN(RA)

```

where HR and VR are the horizontal and vertical radiuses. The programs of examples 9, 10, and 11 provide further patterns utilising the SIN and COS functions:

```

5. REM CLEAR SCREEN AND PUT COLOUR USING
MACHINE CODE (LINES 10-50)
10 PRINT "ENTER COLOUR AS 16*FOREGROUND C
ODE + BACKGROUND CODE "
20 INPUT CC
30 FOR I=0 TO 36:READ J:POKE49152+I,J:NE
XT
40 POKE49175,CC
50 SYS(49152)
80 POKE56576,(PEEK(56576)AND 252)OR 2
90 POKE53272,B
100 POKE53265,PEEK(53265) OR 32
110 FOR X=20 TO 300 STEP 0.2
120 IF X=160 THEN 150
130 I=(X-160)/3
150 Y=130-ABS(90*SIN(I)/I)
160 X1=INT(X/8):B=7-(X AND 7)
170 Y1=INT(Y/8):L=Y AND 7
180 C=Y1*320+X1*B+L
190 POKE24576+C,PEEK(24576+C) OR 2+B
200 NEXT

```

```
210 DATA 160,32,162,0,169,0,157,0,96,232
,208,250,238,8,192,136,208,244
220 DATA 160,4,162,0,169,0,157,0,64,232,
208,250,238,26,192,136,208,244,96
```

```
5 REM CLEAR SCREEN AND PUT COLOUR USING
MACHINE CODE(LINES 10-50)
10 PRINT"ENTER COLOUR AS 16*FOREGROUND C
ODE + BACKGROUND CODE "
20 INPUT CC
30 FOR I=0 TO 36:READ J:POKE49152+I,J:NE
XT
40 POKE49175,CC
50 SYS(49152)
80 POKE56576,(PEEK(56576)AND 252)OR 2
90 POKE53272,8
100 POKE53265,PEEK(53265) OR 32
110 FOR X=0 TO 320 STEP 0.5
120 Y=100+40*SIN(X/8):GOSUB 160
130 Y=100+15*COS(X/8):GOSUB 160
140 NEXT:END
160 X1=INT(X/8):B=7-(X AND 7)
170 Y1=INT(Y/8):L=Y AND 7
180 C=Y1*320+X1*8+L
190 POKE24576+C,PEEK(24576+C) OR 2+B
200 RETURN
210 DATA 160,32,162,0,169,0,157,0,96,232
,208,250,238,8,192,136,208,244
220 DATA 160,4,162,0,169,0,157,0,64,232,
208,250,238,26,192,136,208,244,96
```

```
5 REM CLEAR SCREEN AND PUT COLOUR USING
MACHINE CODE(LINES 10-50)
10 PRINT"ENTER COLOUR AS 16*FOREGROUND C
ODE + BACKGROUND CODE "
20 INPUT CC
```

```

30 FOR I=0 TO 36:READ J:POKE49152+I,J:NE
XT
40 POKE49175,CC
50 SYS(49152)
80 POKE56576,(PEEK(56576)AND 252)OR 2
90 POKE53272,8
100 POKE53265,PEEK(53265) OR 32
110 FOR X=0 TO 320 STEP 0.2
120 PI=4*ATN(1):M=X*PI/180
130 Y=90*SIN(15*M)*COS(1.2*M)
140 Y=Y+100
160 X1=INT(X/8):B=7-(X AND 7)
170 Y1=INT(Y/8):L=Y AND 7
180 C=Y1*320+X1*8+L
190 POKE24576+C,PEEK(24576+C) OR 2+B
200 NEXT
210 DATA 160,32,162,0,169,0,157,0,96,232
,208,250,238,8,192,136,208,244
220 DATA 160,4,162,0,169,0,157,0,64,232,
208,250,238,26,192,136,208,244,96

```

R6: Sprites

A sprite is a small graphics object which can be displayed anywhere on the screen. Under normal circumstances up to 8 sprites can be displayed at any one time and they can be mixed with either graphics or text. The colour, movement, magnification etc. of each sprite are controlled by 46 special registers called sprite registers. These registers are located at memory locations 53248 to 53294 in the memory map. Each sprite has its own definition location, colour register, position register, and has its own bit for collision detection.

Sprite registers

Memory location	Description
53248	sprite 0 X position
53249	sprite 0 Y position
53250 – 53263	sprites 1 – 7 X and Y positions
53264	most significant bit of X position (1 bit per sprite)
53265	bit map mode and vertical pixel scrolling
53266	raster register
53267	light pen X position
53268	light pen Y position
53269	turn sprite ON/OFF
53270	multicolour character mode and horizontal pixel scrolling
53271	sprite expand in Y direction
53272	character memory pointer
53273	interrupt request

Memory location	Description
53274	disable interrupt
53275	sprite priority
53276	multicolour sprite mode
53277	sprite expand in X direction
53278	sprite sprite collision
53279	sprite background collision
53280	border colour
53281	background colour 0
53282	background colour 1
53283	background colour 2
53284	background colour 3
53285	sprite multicolour 0
53286	sprite multicolour 1
53287 – 53294	colour for sprites 0 – 7

R7: Defining a sprite

Sprites are defined in a similar fashion to that of programmable characters, except that they are 24 by 21 dots requiring 63 (24 * 21/8) bytes of memory. This 63 bytes of memory form a block in which sprite data is stored.

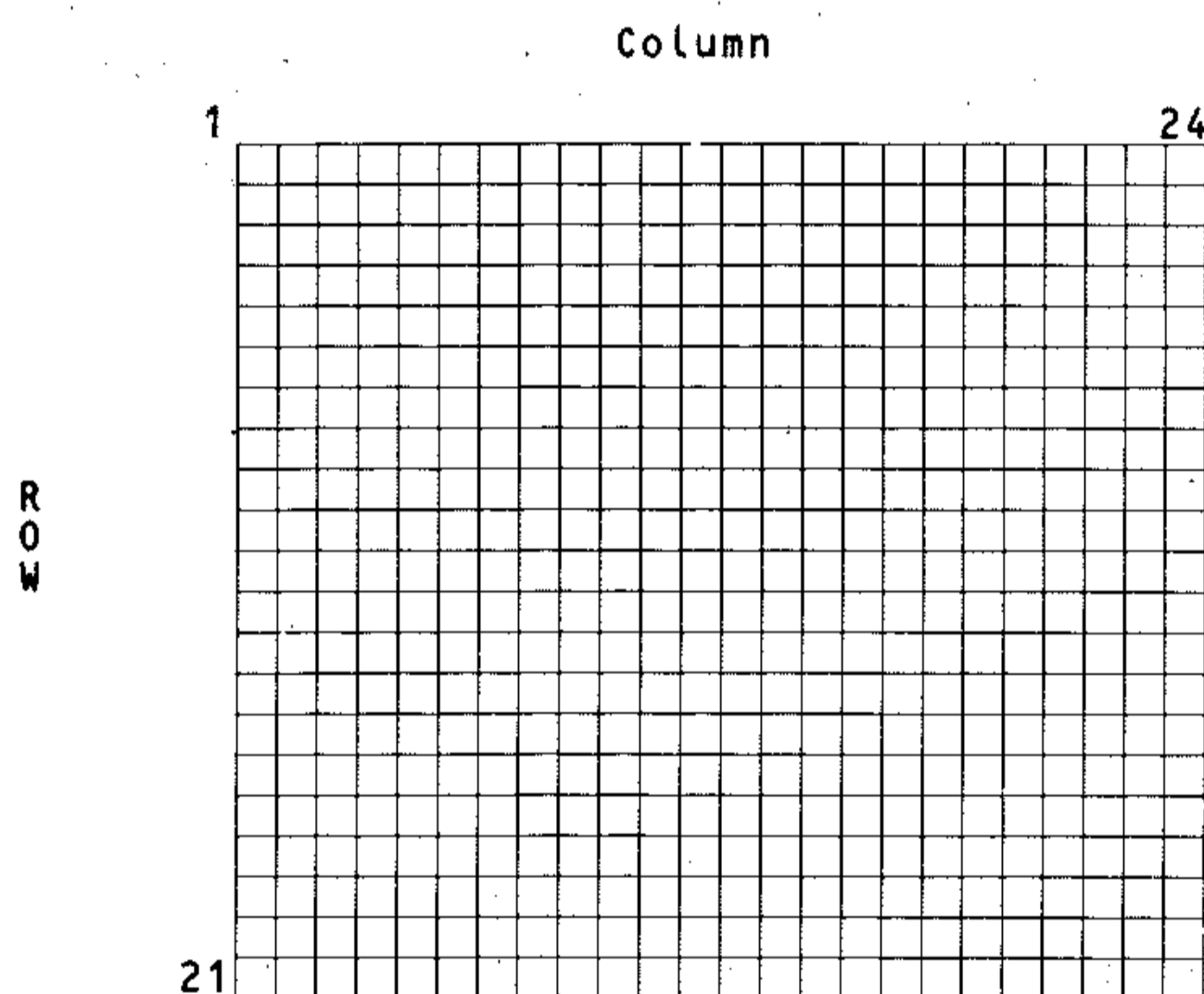


Figure R1: Sprite definition block

The block consists of 21 rows and each row contains 3 bytes. A sprite definition looks as follows:

byte 1	byte 2	byte 3
byte 4	byte 5	byte 6
.	.	.
.	.	.
.	.	.
byte 61	byte 62	byte 63

Each bit set to 0 will display whatever data is behind the block, and each bit set to 1 will display the sprite foreground colour.

R8: Sprite formation

To form a sprite you must take a series of steps. The text outlines these steps below, and creates a program in the process.

First, work out the data for the required sprite and place it in a 64 byte block in memory. To do this you must first draw the sprite on a 24 by 21 point grid and add on the values of all the bits set to 1, just as you did for a programmable character.

Consider the problem of creating a car as our sprite object. The data for such an object is as listed in lines 120 to 150 of the following program.

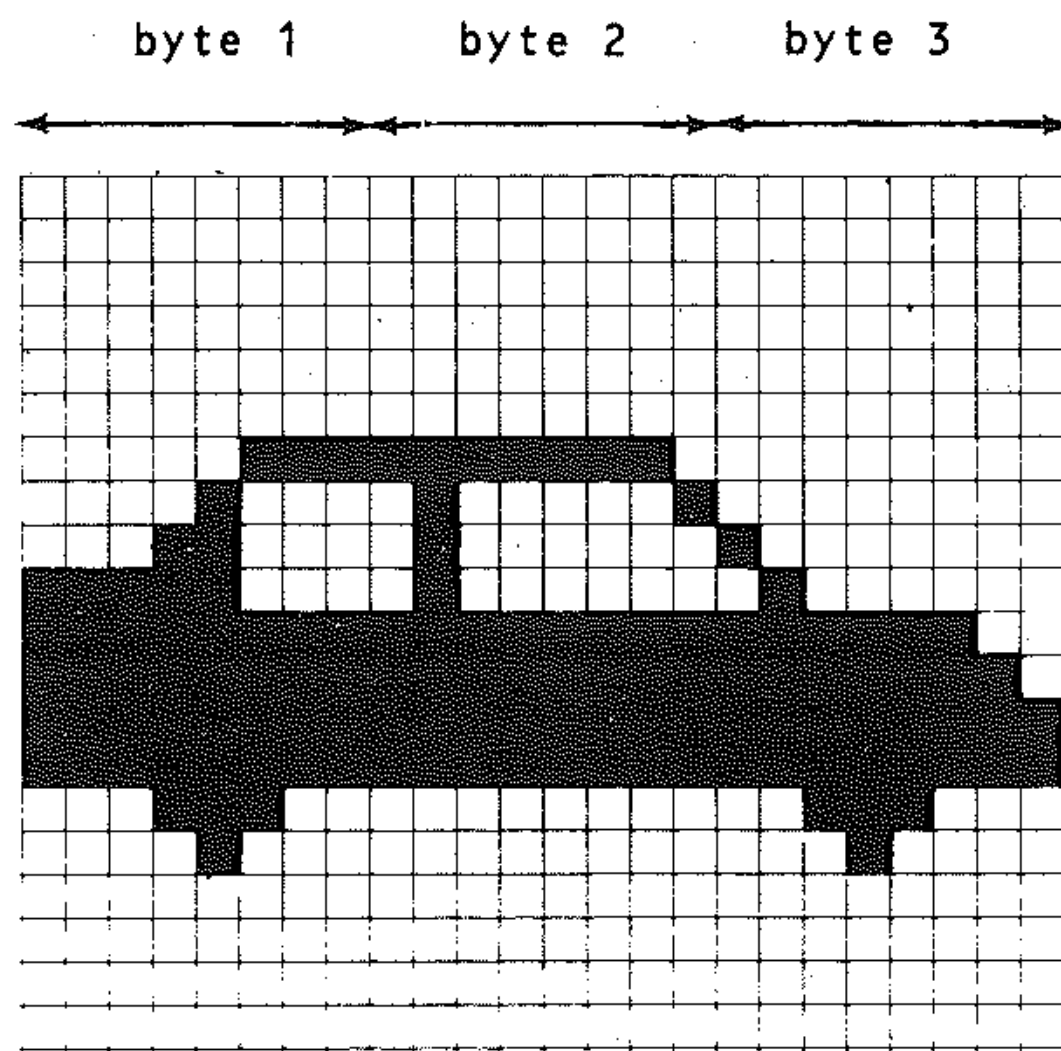


Figure R2: The car sprite

Now you must store these data in memory. Allocate a block in memory where data will be stored. These blocks start at location 0 and go up to 16320. This means that there are 255 blocks in each 16K memory. However, the first block that can be used safely is block 11 and then blocks 13 onwards. Locations 1024 to 2047 are also unuseable, because the screen memory resides there. The following equation gives the block starting address:

$$\text{address} = \text{block} * 64$$

So if you were to use block 11 for the sprite, the starting address would be:

$$\text{address} = 11 * 64 = 704$$

Now that you have your data and the starting address of the block, POKE them into the block as follows:

```
10 FOR M = 704 TO 766
20 READ SDAT
30 POKE M,SDAT: NEXT M
```

Having stored the data in memory, next you must tell the *sprite pointer* which block contains our data. Since there can be 8 sprites displayed at any one time, there are therefore 8 memory locations associated with the 8 sprites. These bytes are always located as the last 8 bytes of the 1K screen memory. Normally, on the Commodore 64, this means that they begin at location 2040. Remember that if we move the screen memory, the sprite pointers also move.

The following table gives the addresses associated with the 8 sprites.

address	2040	2041	2042	2043	2044	2045	2046	2047
sprite number	0	1	2	3	4	5	6	7

So to set the sprite pointer of sprite 7 to the 11th block in memory, use:

```
40 POKE 2047, 11
```

Note that it is possible to allocate the data in block 11 to more than one sprite. So if you wish to create two identical sprites, you must POKE the block number into the respective memory location.

Next you must turn on the required sprite. To turn on any sprite you must POKE into the *sprite enable* register at location 53269. This register has a bit allocated to each of the 8 sprites, and the value to be POKEd depends on the sprite number.

sprite number	7	6	5	4	3	2	1	0
value	128	64	32	16	8	4	2	1

So to turn on sprite 7, set that bit to a 1.

```
50 POKE 53269,128
```

To turn on more than one sprite, you must add up the values of the bits set to 1 and then POKE it into the sprite enable register. For example, to turn on sprite 0 and 7, use:

```
POKE 53269,129
```

A sprite is turned off by setting the bit corresponding that sprite to a 0. The following statement will do this:

```
POKE 53269,PEEK(53269) AND (255 - value)
```

So to turn off sprite 7, use:

```
POKE 53269, PEEK(53269) AND 127
```

Next, add the colours. A sprite can be in any of the 16 colours available on the 64. Each sprite has its own colour register and these are located at addresses 53287 to 53294 respectively. POKEing the colour code into the respective colour register will display all the bits set to 1 in the specified colour, and all the bits set to 0 will be transparent; that is, the data behind the sprite will be displayed. So to set the foreground colour of the sprite to purple we use:

```
60 POKE 53294,4
```

This allows our sprite to be displayed in only one colour. It is, however, possible to use up to four colours in our sprite. The multicolour mode will be discussed later.

Finally, move the sprite onto the screen. If the program is run as it is so far, you will not see your sprite. This is because it is positioned off the visible part of the screen. To move it onto the visible section of the screen, POKE the X and Y coordinates into the X and Y position registers. There are a pair of these registers associated with each sprite. So to move the sprite to point (150,200), use the following statement:

```
70 POKE 53262,150:POKE 53263,200
```

Enter the above line and re-run the program. The sprite is now displayed at position 150,200. To make the sprite move across the screen, you must POKE into the X register a value between 0 and 320. But the maximum value that an 8 bit register can handle is 255. To get around this problem, use the X MOST SIGNIFICANT BIT register location 53265. This register has its 8 bits associated with each sprite respectively. By setting the bit corresponding to your sprite to a 1, you now can handle a number as large as 512; thus it gives you 512 possible positions in the X direction. However, only values between 0 and 343 are needed for the present display area. To demonstrate the point, add the following lines to the program and run it.

```
80 FOR I = 0 TO 343
100 POKE 53262,I:POKE 53263,200:NEXT I
```

The program stops with the message ILLEGAL QUANTITY ERROR IN 100, for the reason specified above. Now add the following lines and try it again.

```
90 MSB = INT(I/256):X = I - 256 * MSB
100 POKE 53262,X:POKE 53263, 200
110 POKE 53264,MSB + 127:NEXT I
```

Line 90 assigns value 0 or 1 to variable MSB and also works out the X position of the sprite on the screen. Line 110 sets the bit 8 of the X MOST SIGNIFICANT BIT register to a 1 or 0 depending on the value of I.

R9: Sprite expansion

Having defined a sprite and displayed it on the screen, you can now double it up in height, width or both, by *expanding* it. Each dot in the sprite doubles in dimension.

To expand horizontally, set the corresponding bit of the SPRITE EXPAND IN X DIRECTION register at location 53277 to a 1. This is demonstrated in line 170 of our next program

sprite no	7	6	5	4	3	2	1	0
X expand value	128	64	32	16	8	4	2	1

So if you wish to expand sprites 5 and 7 in X direction, use the following command:

```
POKE 53277,160
```

To unexpand a sprite in X direction, set the corresponding bit of the register to a 0.

```
POKE 53277,PEEK(53277) AND (255 - value)
```

Expanding and unexpanding of a sprite in Y direction is done in a similar manner, except that you must set or unset the SPRITE EXPAND IN Y DIRECTION register at location 53271 (see line 170). To expand sprites 3 and 7 in Y direction, use the following command:

```
POKE 53271,136
```

and to unexpand a sprite in Y direction use the following command:

```
POKE 53271,PEEK(53271) AND (255 - value)
```

```
5 PRINT " "
6 REM STORE SPRITE DATA IN BLOCK 11
10 FOR M=704 TO 766
20 READ SDAT
30 POKE M,SDAT:NEXT M
35 REM TURN ON SPRITES 1,3,5 AND 7
40 POKE53269,170
45 REM SET SPRITE POINTERS TO BLOCK 11
50 POKE2041,11
60 POKE2043,11
70 POKE2045,11
80 POKE2047,11
85 REM SET SPRITE COLOUR
90 POKE 53288,7
100 POKE 53290,13
110 POKE 53292,3
120 POKE 53294,10
125 REM POSITION SPRITES ON SCREEN
130 POKE53250,160:POKE53251,52
140 POKE53254,160:POKE53255,86
150 POKE53258,150:POKE53259,150
```


7 into multicolour mode, we must POKE $128 + 8 = 136$ into the multicolour register. To do this, add the following line to the program of example 14:

```
121 POKE 53276,136
```

Also, since the bit pairs 01 and 11 will take their colour from multicolour registers 0 and 1 respectively, you must therefore POKE these locations with the desired colour code. Add the following line to the program and run it.

```
122 POKE 53285,1:POKE 53286,14
```

To turn off the multicolour mode, use the following command:

```
POKE 53276,PEEK(53276) AND (255 - value)
```

R12: Collision detection

This is another important feature provided by the VIC chip. Two registers at locations 53278 and 53279 detect sprite-to-sprite collision, and sprite-to-background object collision respectively. Each sprite has a bit associated with it in each of these registers and the bit will be set if a collision occurs. For example if sprites 3 and 5 are involved in a collision, then the bits associated with the two sprites in the sprite-to-sprite collision register will be set. The bits in this register will remain set until the register is PEEKed. PEEKing this register will automatically unset the register. The sprite-to-background object detection is done in the same way.

Section S: Lists and arrays

S1: Introduction

You have been studying simple variables to handle strings and numbers, but there are many occasions when the programmer needs to use lists of strings or numbers, such as, for example, a class list of students and examination marks. A simple variable may be allowed to represent several values, but only one at a time, because each variable has only one part of memory allocated to it.

Consider the following short lists of names and marks.

SMITH	50
JONES	45
BROWN	82
BLACK	36

To represent this list completely using simple variables, you would need 4 string variables and 4 numeric variables. With lists containing several hundred items, this becomes unmanageable. To provide a more convenient method of accessing this type of information, the computer needs to do two things:

- reserve enough storage in memory for all the items in the list (called the *dimensioning* arrays), and
- provide some way of identifying each item in the list (subscript or index).

S2: Dimensioning

The dimension statement DIM is used to reserve storage space for a *list* or *array* to contain numbers. DIM A(N) sets up an array A with space for N numbers. A may be any legal variable. The Commodore will accept both upper and lower case letters; a(N) will signify the same array as A(N). N may be a number, a numeric variable or an expression.

A dimension statement must be declared before the array can be used. This is usually done at the beginning of a program, unless the value N is to be set equal to an expression or a variable which will be calculated later in the program.

These statements in a program –

```
10 DIM A(10)
20 DIM B(15)
30 DIM C(30)
```

or

```
10 DIM A(10),B(15),C(30)
```

– will reserve storage for a list A containing 10 numbers, a list B containing 15 numbers and a list C containing 30 numbers. The values of each *element* (number) in an array are automatically set (initialised) as zero. The message **OUT OF MEMORY ERROR** will be displayed if there is no room for the array (that is, if N is too large).

S3: The index variable

The index variable N is used to locate a member of a list. The form $A(N)$ is used to locate the N 'th number of a list $A(L)$, where $1 \leq N \leq L$. If $N=5$, then $A(N)$ refers to $A(5)$, the fifth number in the list. The program below establishes a four element list, so that:

```
A (1) = 1
A (2) = 4
A (3) = 9
A (4) = 16
```

and prints out the second and fourth elements.

```
10 DIM A(4)
20 FOR N=1 TO 4
30 A(N)=N*N
40 NEXT N
50 PRINT"SECOND ELEMENT IS ";A(2)
60 PRINT"FOURTH ELEMENT IS ";A(4)
```

S4: Lists

Many types of problem involve a set of values, and it is convenient to store such items in a list. The next program illustrates the idea. Assuming that a list of the squares of the first 20 integers is required, it is necessary to reserve storage for the twenty numbers (1,4,9 etc. up to 400) and this is done in line 20. The loop (lines 30 to 50) puts $A(1)=1$, $A(2)=4$. . . $A(20)=400$ and it is thus possible to use any item of this list at a later date, using the index variable. Line 60 prints out 4 and 16 and the loop (lines 80 – 100) will print out the complete list of numbers.

```
10 REM**LIST**
20 DIM A(20)
30 FOR N=1 TO 20
40 A(N)=N*N
50 NEXT N
60 PRINT A(2);A(4)
70 PRINT
80 FOR N=1 TO 20
90 PRINT A(N)
100 NEXT N
```

Here are some example programs illustrating the use of lists.

Simple allocation of elements in a list

Look at the program. What will be printed out when the program is run? Check by entering and running the program.

```

10 REM**LIST1**
20 DIM A(4)
30 A(1)=10
40 A(2)=58
50 A(3)=72
60 A(4)=20
70 PRINT A(1)*A(2)
80 PRINT
90 PRINT A(3)-A(2)

```

Allocating elements in an array

This technique uses the the READ and DATA statements. What will be printed out when the program is run?

```

10 DIM B(6)
20 FOR N=1 TO 6
30 READ B(N)
40 NEXT N
50 PRINT 3*B(2)
60 PRINT B(6)-B(4)
70 PRINT 2*B(3)-B(1)
80 DATA 14,25,36,47,58,69

```

Allocation of values to the elements in a list

Here a loop and the INPUT statement are used. The value of the control variable (N) of the loop is used to specify each element of the list in turn. Again, study the program and try to work out the results, then check by keying it in and running it.

```

10 REM**LIST2**
20 DIM A(4)
30 FOR N=1 TO 4
40 PRINT"ENTER A(";N;") ";
50 INPUT A(N)
60 NEXT N
70 PRINT A(1)*A(3)
80 PRINT
90 PRINT A(3)-A(2)

```

Dimensioning lists in a program

All lists used in a program must be dimensioned. Hand trace this program and decide what are the 10 elements in list B(N). Check by entering and running the program.

```

5 REM**LIST3**
10 DIM A(20),B(10)
20 FOR N=1 TO 20
30 A(N)=N*N
40 NEXT N
50 FOR N=1 TO 10
60 B(N)=A(2*N)-A(2*N-1)
70 NEXT N
80 FOR N=1 TO 10
90 PRINT B(N)
110 NEXT N

```

Using a variable in a DIM statement

A variable may be used in a DIM statement, provided its value is assigned before the DIM statement is reached. Enter the next program and run it for X=20.

```

5 REM**LIST4**
10 INPUT X
20 DIM A(X)
30 FOR N=1 TO X
40 A(N)=SQR(N)
50 PRINT A(N)
60 NEXT N

```

Use of lists to store data

The program OHMS LAW illustrates the use of lists to store data from a set of electrical circuit experiments. The voltmeter and ammeter readings from each experiment are stored in the lists A(N) and V(N) as they are input. Notice that it is essential to dimension storage space for the derived lists of results R(N), (line 60). The loop (lines 60 to 90) enables the readings to be stored for use in the later loop (lines 130 to 170), which performs the calculation and prints out the results of each experiment, giving the current in amps, the voltage in volts, and the resistance in ohms derived by the formula $R=V/I$ in line 140. Line 120 initialises a variable T which has each resistance in turn added to it. This enables line 200 to print the average resistance value.

Notice that if line 50 came before the DIM statements, the program could use DIM A(X), etc. to set the size of the arrays, as in program LIST4 above.

```

10 REM**OHMS LAW**
20 PRINT"OHMS LAW RESULT"
30 PRINT"UPTO 20 PAIRS OR READINGS"
40 DIM A(20),V(20),R(20)
50 INPUT"ENTER NUMBER OF READINGS";X
60 FOR N=1 TO X
70 INPUT"ENTER CURRENT IN AMPS";A(N)
80 INPUT"ENTER VOLTAGE IN VOLTS";V(N)
90 NEXT N

```



```

100 PRINT "AMPS"; TAB(8); "VOLTS"; TAB(16); "
OHMS"
110 PRINT "*****"
120 T=0
130 FOR N=1 TO X
140 R(N)=V(N)/A(N)
150 T=T+R(N)
160 PRINT A(N); TAB(8); V(N); TAB(16); R(N)
170 NEXT N
180 PRINT "*****"
190 PRINT
200 PRINT "AVERAGE RESISTANCE "; T/X " OHMS
"

```

Another example of storing data

This program shows image positions (V) and magnifications (M) for a convex lens, given the focal length of the lens (F) and the object distance (U).

```

5 REM**CONVEXLENS**
10 PRINT "THIS PROGRAM SHOWS THE POSITIO
N AND MAGNIFICATION OF THE IMAGE ";
15 PRINT "PRODUCED BY A CONVEX LENS"
20 PRINT "*****
*****"
30 DIM U(12),V(12),M(12)
40 INPUT "ENTER FOCAL LENGTH IN CM."; F
50 PRINT "ENTER OBJECT DISTANCE IN CM"
60 FOR N=1 TO 12
70 INPUT U(N)
80 V(N)=U(N)*F/(U(N)-F)
90 M(N)=V(N)/U(N)
100 NEXT N
110 PRINT "U"; TAB(8); "V"; TAB(22); "M"
120 FOR N=1 TO 12
130 PRINT U(N); TAB(8); V(N); TAB(22); M(N)
140 NEXT N

```

The screen display will look like this :

THIS PROGRAM SHOWS THE POSITION AND
MAGNIFICATION OF THE IMAGE PRODUCED
BY A CONVEX LENS

```
*****  
U          V          M  
10         -13,333333  -1.3333333  
20         -40          -2  
30         -120         -4  
39.555     -3555.5056  -89.887639  
40.555     2922.8829   72.0772073  
50         200          4  
60         120          2  
70         93.333333   1.3333333  
80         80           1  
90         72           0.8  
100        66.666667   0.66666667  
120        60           0.5
```

S5: String arrays

The DIM statement for string arrays has the form

```
DIM A$(N)
```

where N = number of strings.

DIM A\$(3) will reserve storage space for 3 strings A\$(1), A\$(2), A\$(3), each of any length up to 255 characters. Each letter of each string can be accessed separately, as with a string variable. Substrings may also be allocated using the LEFT\$, RIGHT\$ and MID\$ functions, which will return the specified characters. For example, if

```
A$(2) = "EFGH"
```

then

```
MID$(A$(2),2,3) = "FGH"
```

The two programs below show these operations. Key them in and run them. Note that space may be included as letters, as can any other character usable in a string.

```
5 REM**STRING ARR1**  
10 DIM A$(4)  
15 PRINT"ENTER ANY CHARACTERS"  
20 FOR N=1 TO 4  
30 INPUT A$(N)  
40 NEXT N  
50 PRINT MID$(A$(4),3,1); " ";MID$(A$(3),  
2,1)  
60 PRINT  
70 PRINT A$(1); " ";A$(2); " ";A$(3)
```

```

10 REM**STTR ARR2**
20 DIM A$(3)
30 A$(1)="ABCD"
40 A$(2)="EFGH"
50 A$(3)="IJKL"
60 PRINT MID$(A$(2),4,1); " ";MID$(A$(3),
2,1)
70 PRINT
80 PRINT MID$(A$(2),2,3)
90 PRINT MID$(A$(1),3,2)

```

S6: Two-dimensional arrays

A 2-D (two-dimensional) numeric array is dimensioned by the statement:

```
DIM A(R,C)
```

where A is any legal variable, R is the number of rows and C is the number of columns. All elements are set as zero.

The simple array is one-dimensional, and contains just a linear sequence of items. But arrays can have more than one dimension:

4	6	8
10	12	14
16	18	20
22	24	26

This is a numeric array consisting of 4 rows of numbers in 3 columns. Storage would be reserved by the statement:

```
10 DIM A(4,3)
```

In an array A(R,C) any element can be accessed, so that in the array above $A(2,1) = 10$, $A(3,2) = 18$, etc. An array of two (or more) dimensions is also known as a *matrix* (plural matrices).

The following program establishes an array and prints out two selected elements and then the complete array:

```

10 REM**ARRAY**
20 DIM A(10,9)
30 FOR R=1 TO 10
40 FOR C=1 TO 9
50 A(R,C)=R*C
60 NEXT C,R
70 PRINT A(10,6),A(5,3)
80 PRINT
90 FOR R=1 TO 10
100 FOR C=1 TO 9
110 PRINT TAB(3*C);A(R,C);
115 IF A(R,C)<10 THEN PRINT " ";
120 NEXT C
130 PRINT:PRINT:PRINT TAB(3);
140 NEXT R

```

Line 20 allocates the appropriate storage. Nested loops (line 30 to 60) are used to allocate values to the elements in the array. Line 70 prints out two elements in the array. Nested loops (lines 90 to 140) print out the complete array.

Why is line 130 required? A system is needed to keep track of the elements in the array. In general, it is easiest to use *R* to represent the rows and *C* the columns and always to access the rows before the columns. Note the use of the **TAB** function to give a clear printout.

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45

etc.

String arrays can also have more than one dimension. 2-D String arrays are dimensioned by a statement of the form

```
DIM A$(R,C)
```

where *R* is the number of rows, *C* is the number of columns. Try this program:

```
10 REM**2DSTRING**
20 DIM A$(3,3)
30 PRINT"ENTER ANY CHARACTERS MAX 5 LET
TERS "
40 FOR R=1 TO 3
50 FOR C=1 TO 3
60 PRINT"ROW ";R;"COL ";C
61 PRINT
65 INPUTA$(R,C)
70 NEXT C,R
80 FOR R=1 TO 3
90 FOR C=1 TO 3
100 PRINT TAB(C*7);A$(R,C);
110 NEXT C
120 PRINT
130 NEXT R
```

S7: Multidimensional arrays

Multidimensional arrays are available for both numbers and strings, although a 3-D string array is rarely needed in a program. The easiest way of thinking of these arrays is as follows:

$A(P,R,C)$ is a 3 dimensional array: page, row, column.

$A(B,P,R,C)$ is a 4 dimensional array: book, page, row, column.

A 5-dimensional array would be a library, on the basis of this analogy. These arrays require DIM statements to reserve the necessary storage. A simple example of a 3 dimensional array (which needs 3 nested loops) is given below.

```
10 REM**3DLIST**
100 DIM A(3,2,4)
110 FOR P=1 TO 3
120 FOR R=1 TO 2
130 FOR C=1 TO 4
140 A(P,R,C)=P*R*C
150 NEXT C,R,P
160 PRINT A(2,1,3)
170 PRINT
180 FOR P=1 TO 3
190 FOR R=1 TO 2
200 FOR C=1 TO 4
210 PRINT A(P,R,C); " ";
215 IF A(P,R,C) < 10 THEN PRINT " ";
220 NEXT C
230 PRINT
240 NEXT R
250 PRINT
260 NEXT P
```

The similar program below, extended by a dimension, shows a 4-dimensional array printed out in a suitable form.

```
10 REM**4DARRAY**
100 DIM A(2,3,4,5)
110 FOR B=1 TO 2
120 FOR P=1 TO 3
130 FOR R=1 TO 4
140 FOR C=1 TO 5
150 A(B,P,R,C)=B*P*R*C
160 NEXT C,R,P,B
170 PRINT A(1,2,3,4); " "A(2,2,3,3)
180 PRINT
190 FOR B=1 TO 2
200 FOR P=1 TO 3
210 PRINT "BOOK"; B; "/PAGE"; P
220 FOR R=1 TO 4
230 FOR C=1 TO 5
240 PRINT A(B,P,R,C) " ";
245 IF A(B,P,R,C) < 10 THEN PRINT " ";
250 NEXT C
260 PRINT
270 NEXT R
280 PRINT
290 NEXT P
300 NEXT B
```

Here is a 3-D version to try.

```
5 REM**3D STRING ARRAY**
10 DIM A$(3,3,3)
20 FOR P=1 TO 3
30 FOR R=1 TO 3
40 FOR C=1 TO 3
50 INPUT A$(P,R,C)
60 NEXT C,R,P
70 FOR P=1 TO 3
80 FOR R=1 TO 3
90 FOR C=1 TO 3
100 PRINT A$(P,R,C); " ";
110 NEXT C
120 PRINT
130 NEXT R
140 PRINT
150 NEXT P
```

S8: Use of arrays

A simple example of the use of 2-D arrays is shown in the seat booking program below. A small theatre consists of 10 rows of seats with 6 seats in each row. Some seats may already be reserved. These are input when the program is run. When a new booking is made the requested seat, if available, is sold. If the seat is not available the customer is asked to choose another row. The sections of the program are as follows:

- 1 Initialise an array to represent the 10 rows of 6 seats (line 30). All elements in this array are 0, and represent unbooked seats.
- 2 Input seats already booked (lines 40 – 130).
 - 2.1 Input row and seat number of booked seats. If input is 0 for the row number, program goes to 2.3.
 - 2.2 If seat already booked (array element = 1), print message to user. Seat is booked by placing a 1 in the appropriate array element.
 - 2.3 Program prints prompt, then halts until C is input.
- 3 Customer request for seat is input (lines 165 – 240).
 - 3.1 Row and seat required are input. If seat already booked (array element = 1), program goes to 4.
 - 3.2 Seat is booked.
 - 3.3 Menu is printed to enable user to choose to book another seat or to view seating plan.
 - 3.4 If seat booking is requested, program returns to 3.1. If seating plan option is chosen, program goes to 5.
- 4 Seat available module (lines 300 – 450).
 - 4.1 Seat unavailable message is printed, then variable SEATS is set at zero, and the loop checks if at least one seat is free in this row, setting SEATS = 1 if seat is free (Current Row R, checked for S(R,1) to S(R,6)).
 - 4.2 If no seats in this row (Seats = 0), program passes to line 440 and prints message, then returns to menu (3.3).
 - 4.3 If at least one seat is free, the loop at lines 390 to 410 prints out the numbers of the seats free, and the program returns to the menu.

- 5 View seat plan module (lines 500–600).
- 5.1 Nested loops are used to display seat plan, row 10 being at the top, as 0's and 1's.
- 5.2 Copy option is given, to print out the seating plan.
- 5.3 Menu presented for end of program or return to book seats. Program goes to 3.1, or proceeds to 6.
- 6 Program ends. Instructions given to restart if required without using RUN and clearing the data stored in the array.

```

10 REM**THEATRE**
20 REM**INITIALISE ARRAY**
30 DIM S(10,6)
35 REM**INSERT SEATS ALREADY**
36 REM**BOOKED**
40 PRINT"INPUT SEATS THAT ARE BOOKED.",
INPUT 0 TO FINISH"
50 INPUT"ROW ?";R
60 IF R=0 THEN 130
80 INPUT"SEAT ?";C
100 IF S(R,C)=1 THEN PRINT"ROW ";R;" SEA
T ";C;"ALREADY BOOKED"
110 S(R,C)=1
120 GOTO 50
130 PRINT""
140 INPUT"ENTER C TO PROCEED TO BOOKING"
;A$
150 PRINT""
160 REM**CUSTOMER REQUEST FOR**
165 REM**SEAT**
170 INPUT"ENTER ROW REQUESTED ";R
180 INPUT"SEAT NUMBER ";C
190 IF S(R,C)=1 THEN 300
200 S(R,C)=1
210 PRINT"THIS SEAT FREE. NOW BOOKED"
220 PRINT"ROW ";R;" SEAT "C
250 PRINT
260 PRINT"BOOK ANOTHER SEAT (S) OR VIEW
","SAETING PLAN (P) ? ENTER S OR P "
270 INPUT A$
280 IF A$="S" THEN 170
290 GOTO 500
300 REM**SEAT UNAVAILABLE**
310 PRINT"REQUESTED SEAT NOT AVAILABLE"
320 SEATS=0
330 FOR N=1 TO 6
340 IF S(R,N)=1 THEN 360
350 SEATS=1
360 NEXT N
370 IF SEATS=0 THEN 440
380 PRINT"SEATS FREE: ";
390 FOR N=1 TO 6

```

```

400 IF S(R,N)=0 THEN PRINT N; " ";
410 NEXT N
420 PRINT ". "
430 GOTO 450
440 PRINT "NO SEATS ARE FREE IN ROW ";R
450 GOTO 260
490 REM**SEATING PLAN**
500 PRINT " "
510 FOR R=1 TO 10
520 X=11-R
530 PRINT TAB(5); "ROW "; X; TAB(12);
540 FOR C=1 TO 6
550 PRINT S(X,C);
560 NEXT C
570 PRINT
580 NEXT R
590 INPUT "ENTER E TO END, S TO BOOK SEA
TS"; A$
600 IF A$="E" THEN 630
610 PRINT " "
620 GOTO 170
630 PRINT "PROGRAM STOPPED USE GOTO 170 T
O RESTART"
640 REM**END OF PROGRAM**
650 END

```

This is the screen display at the end of the seat plan print routine (the first prompt has been responded to with a user input):

```

ROW 10  000000
ROW 9   000000
ROW 8   000000
ROW 7   000000
ROW 6   000000
ROW 5   000000
ROW 4   000100
ROW 3   100000
ROW 2   111111
ROW 1   000000

```

```

INPUT C TO COPY, ANY TO PROCEED
INPUT E TO END, S TO BOOK SEATS

```

The variables used in the theatre booking program are as follows:

S(10,6) array to store 10 rows of 6 seats (value 1 when set booked booked, 0 when free).

R current row of array in processing.

C current seat number in processing.

A\$ user input string for menu choices.

SEATS marker used to indicate whether seats are free in current row. Set to 0 when no seats free, 1 when seats available.

N loop variable. Value used in processing inside loops to check availability, and print seat numbers.

- R** loop variable used to print seat plan. Note this is the same as the variable for Rows above. This name may be used in two different ways in this program because the value of the simple variable R is re-initialised by the input line 170 on return to the seat selection routine.
- X** variable used for reverse printing of the seating rows.
- C** Loop variable for seats in seat plan printing. As with R above, re-initialised as simple variable on return to to seat selection routine.

Note that the use of variables in two ways, as with R and C in this program, is possible only if the simple variables will be re-initialised every time they are used; otherwise problems can arise. A loop variable erases a simple variable of the same name. It would be better practice to use different names for the two types of variable.

Note: For simplicity this chapter assumes there are N elements in an array A(N). In fact there are N+1 elements as the Commodore creates the element A(0) too. If your application is short of memory, don't waste this extra element.



Section T: Sorting, Searching and Storing Arrays

T1: Searching and sorting

Searching a list of numbers (or strings) for specified values can obviously be done much more efficiently if the numbers (or strings) are sorted according to some specified order, commonly alphabetical order or ascending numerical order. In electronic data processing, groups of records (files) can be handled more efficiently if the records are pre-sorted into a specified order (e.g. merging transaction files into a master file). Various techniques have been developed to sort data, and several of the simpler methods are illustrated in this section together with two simple methods of searching lists.

There is a considerable difference in the efficiency of the various sorting techniques, depending on the type and volume of data to be sorted. A technique which is good for a random list of numbers may not be appropriate for a list in which only one number is out of sequence. For random lists the *quick sort* and *shell sort* techniques are very much faster than a *bubble sort*. Deciding on which is the most suitable method is largely a matter of experience, and you should experiment, using the different techniques for equivalent sets of numbers, and timing the sort procedures.

Many sorting algorithms exist, of which the simplest is the bubble sort.

The bubble sort is used for sorting numbers (or strings with appropriate alterations) into ascending or descending order. The principle of the bubble sort is to compare adjacent numbers and change positions if they are in the incorrect order. This is done for elements 1 and 2, then 2 and 3, 3 and 4...X-1 to X at the end of which the highest number is in the Xth position. This is repeated (and the next highest number bubbles up to the X-1th position) and repeated again, until the ordering is complete.

The following program is a bubble sort to put numbers into ascending order. The sorting routine itself is in lines 130 to 225.

```
10 REM**BUBBLE**
20 PRINT"NUMBER OF ITEMS TO BE SORTED"
30 INPUT"MAXIMUM NUMBER IS 50 ";X
40 IF X>50 THEN 30
45 DIM A(X)
50 PRINT"ENTER NUMBERS ONE AT A TIME "
60 FOR N=1 TO X
70 INPUT A(N)
80 NEXT
90 PRINT "#####UNSORTED LIST"
100 FOR N=1 TO X
110 PRINT A(N); " ";
120 NEXT
130 PRINT
140 REM**SORTING ROUTINE**
150 FOR N=1 TO X-1
160 FOR M=1 TO X-N
```

```

170 C=A(M)
180 D=A(M+1)
190 IF C<=D THEN 220
200 A(M)=D
210 A(M+1)=C
220 NEXT M,N
230 REM**END OF SORT**
240 PRINT "UNSORTED LIST"
250 FOR N=1 TO X
260 PRINT A(N); " ";
270 NEXT
280 PRINT
290 END

```

Here is a diagram in table form of the operations performed in the course of the sort:

Table Operations

	Pass 1				Pass 2			Pass 3	
	N = 3				N = 2			N = 3	
START	M=1	M=2	M=3	START PASS 2	M=1	M=2	START PASS 3	M=1	FINISH
A(1) 129	129			129	56		56	41	41
A(2) 267	267	56		56	129	41	41	56	56
A(3) 56		267	41	41		129	129		129
A(4) 41			267	267			267		267

Sample printout:

```

UNSORTED LIST
129 267 56 41 69 43 99 90 4 8

```

```

SORTED LIST
4 8 41 43 56 69 90 99 129 267

```

To illustrate the operation of the program, take the first four of these numbers and see how the program sorts them:

Bubble sort for 4 numbers A(1), A(2), A(3), A(4) input as 129, 267, 56, 41

The procedure is as follows:

The program goes through the list comparing successive number pairs. For example: A(1) and A(2), then A(2) and A(3). If A(1) > A(2) then they are swapped, so that A(2) becomes A(1) and A(1) becomes A(2). If A(1) < A(2) then they are left as is. The largest number in the list will finally be in the highest position, ie. A(4).

On the first pass three comparisons are made, and the largest number will end as A(4). On the second pass two comparisons are made, and the largest number will be in position A(3). On the third pass we make one comparison. The larger number will be A(2). There is no need for any more passes; the smallest number will be A(1).

There are four numbers, so $X = 4$.

$X - 1$ passes are needed, so $N = 1$ TO $(X - 1) = 1$ TO 3 passes

For *each pass* from 1 to $X - N$ comparisons are needed, so $M = 1$ to $(X - N)$ comparisons.

T2: Bubble sort with flag

```
10 REM**BUBBLE**
20 PRINT"NUMBER OF ITEMS TO BE SORTED"
30 INPUT"MAXIMUM NUMBER IS 50 ";X
40 IF X>50 THEN 30
45 DIM A(X)
50 PRINT"ENTER NUMBERS ONE AT A TIME "
60 FOR N=1 TO X
70 INPUT A(N)
80 NEXT
90 PRINT "UNSORTED LIST"
100 FOR N=1 TO X
110 PRINT A(N); " ";
120 NEXT
130 PRINT
140 REM**SORTING ROUTINE**
150 FOR N=1 TO X-1
160 S=0
170 FOR M=1 TO X-N
180 C=A(M)
190 D=A(M+1)
200 IF C<=D THEN 240
210 A(M)=D
220 A(M+1)=C
230 S=1
240 NEXT M
250 IF S=0 THEN 280
260 NEXT N
270 REM**END OF SORT**
280 PRINT "SORTED LIST"
290 FOR N=1 TO X
300 PRINT A(N); " ";
310 NEXT
320 PRINT
330 END
```

In order to ensure that the sort is completed as quickly as possible, a *flag* (in this case the variable S) is introduced, to indicate if it has been necessary to swap

elements in the list; S=1 when a swap has occurred and sorting will continue until S=0 at line 215. This prevents unnecessary sorting taking place. The procedure and program are otherwise the same as the bubble sort. The lines 145, 205 and 215 have been inserted into the above BUBBLE program.

EXERCISE

Draw up a table of operations for this program, as was done for the BUBBLE.

T3: Alphabetic sort

The bubble sort (and all other sorts) may be used to sort strings by using appropriate string variables and string arrays.

```
5 REM**ALPHASORT**
10 PRINT "HOW MANY STRINGS "
20 PRINT "MAXIMUM 10 CHARACTERS"
30 INPUT X
35 PRINT ""
40 DIM A$(X)
50 FOR N=1 TO X
60 INPUT A$(N)
70 NEXT
80 PRINT "": PRINT "UNSORTED LIST"
90 FOR N=1 TO X
100 PRINT A$(N); " ";
110 NEXT
120 PRINT
130 REM**SORTING ROUTINE**
140 FOR M=1 TO X-1
150 FOR N=1 TO X-M
160 IF A$(N+1) >= A$(N) THEN 200
170 T$=A$(N+1)
180 A$(N+1)=A$(N)
190 A$(N)=T$
200 NEXT N,M
210 PRINT "SORTED LIST"
220 FOR N=1 TO X
230 PRINT A$(N); " ";
240 NEXT
250 PRINT
260 END
```

Some care must be exercised if the above sort is to be used on numbers entered as strings. The example given below shows that it will work, *provided* one ensures that all numbers entered have the *same number* of figures. To illustrate:
Incorrect use:

```
HOW MANY STRINGS
MAXIMUM 10 CHARACTERS
```

UNSORTED LIST

123 99 543 6 456 897 567 21 345 45

SORTED LIST

123 21 345 45 456 543 567 6 897 99

Correct use:

HOW MANY STRINGS
MAXIMUM 10 CHARACTERS

UNSORTED LIST

123 099 543 006 456 897 567 021 345 045

SORTED LIST

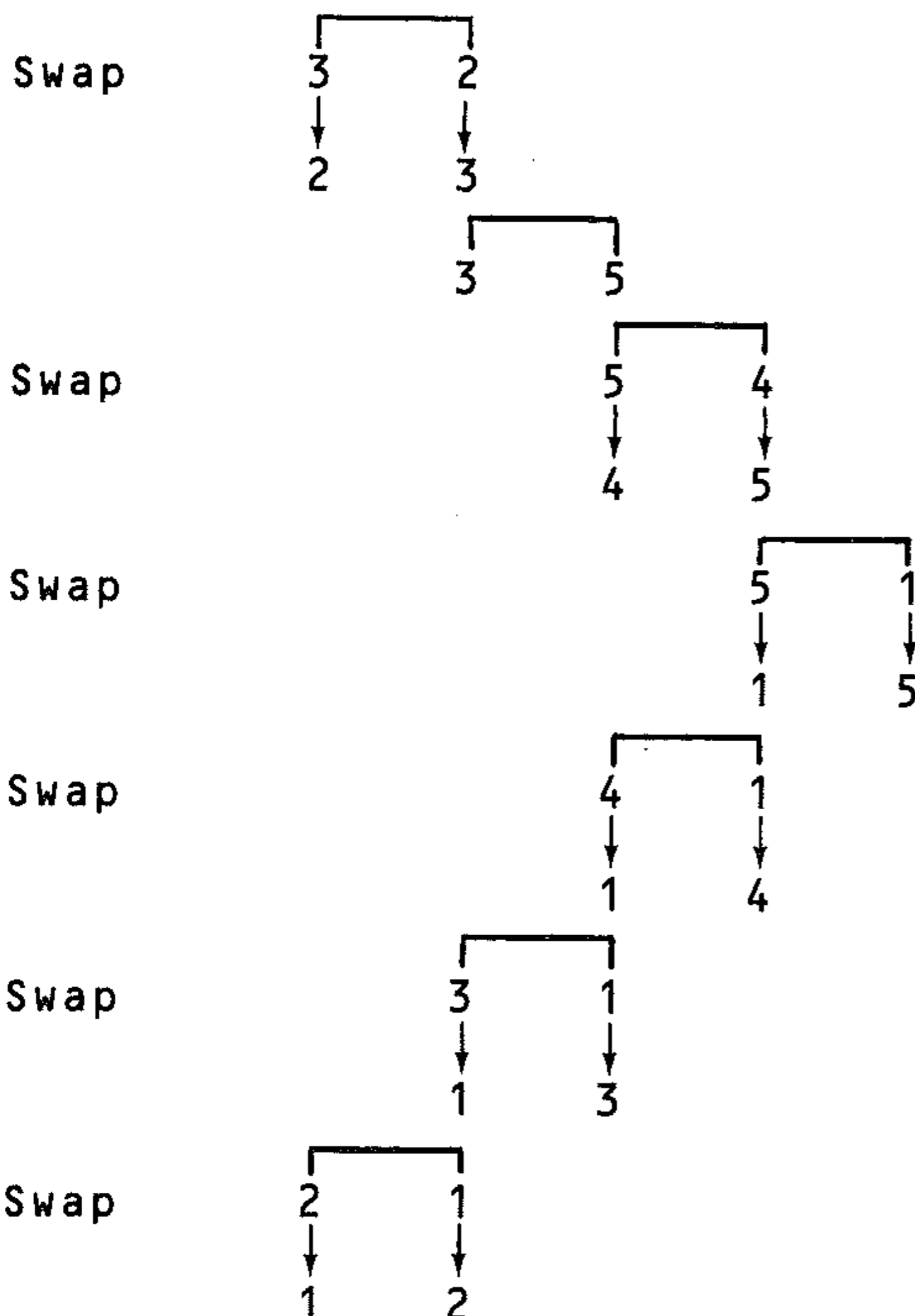
006 021 045 099 123 345 456 543 567 897

T4: Insertion sort

This is a sort which is more efficient than the bubble sort and is also the basis of an even faster sort called a shell sort. Speed is a prime consideration when sorting large amounts of data. Consider the list of numbers:

3 2 5 4 1

Starting with the first entry in the list, a comparison is made with the second. Then they are swapped if necessary. Then the second is compared with the third, a swap performed if required, and if a swap was made, the first and second are compared again, and swapped if necessary. Then the third item is compared with the fourth, and so on. The list above will be sorted like this:



Consider the list $A(1), A(2) \dots A(X)$. To insert item $A(I+1)$ in the correct position:

Let $T = A(I+1)$, then if $T \geq A(I)$ no swap is necessary and no further comparisons are required. If $T < A(I)$ let $A(I+1) = A(I)$ and move on to $A(I-1)$; then if $T \geq A(I-1)$, let $A(I) = T$ and insertion is complete. If $T < A(I-1)$, let $A(I) = A(I-1)$, and so on down the list.

The various steps the program will make are therefore as follows:

- 1 Set $J = I$ and $T = A(I+1)$
- 2 If $T \geq A(J)$ let $A(J+1) = T$ and stop
- 3 Let $A(J+1) = A(J)$
- 4 Let $J = J - 1$
- 5 If $J < 1$ let $A(J+1) = T$ and stop. If not, go to (2).
- 6 Repeat for each value of I (from 1 to $N-1$) where $N =$ number in list.

A trace of the program, using the example list again, can be shown like this:

	I=1	I=2	I=3		I=4				
Start	J=1	J=2	J=3	J=2	J=4	J=3	J=2	J=1	J=0
	T=3	T=5	T=4	T=4	T=1	T=1	T=1	T=1	T=1
A(1)=3	2								1
A(2)=2	3	3						2	
A(3)=5		5		4			3		
A(4)=4			5			4			
A(5)=1					5				

```

5 REM**INSERT**
10 INPUT "ENTER NUMBER OF NUMBERS "; X
20 DIM A(X)
30 PRINT "ENTER NUMBERS"
40 FOR N=1 TO X
50 INPUT A(N)
60 NEXT
70 PRINT "UNSORTED LIST"
80 FOR N=1 TO X
90 PRINT A(N); " ";
100 NEXT
110 PRINT
120 REM**SORT ROUTINE**
130 FOR I=1 TO X-1
140 J=I
150 T=A(I+1)
160 IF T>=A(J) THEN 200
170 A(J+1)=A(J)
180 J=J-1
190 IF J>=1 THEN 160
200 A(J+1)=T
210 NEXT
220 PRINT "SORTED LIST"

```



```

230 FOR N=1 TO X
240 PRINT A(N); " ";
250 NEXT
260 PRINT
270 END

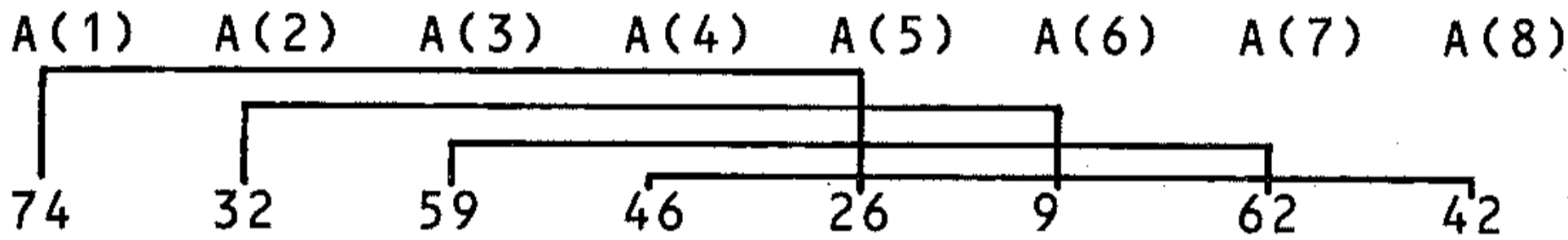
```

T5: Shell sort

The procedure in this sort is to precede an insertion sort by a process which, considering a list of numbers to be placed in ascending order left to right, will move low values to the left and high values to the right more quickly.

Consider an 8 element list $A(8)$, holding the values: 74, 32, 59, 46, 26, 9, 62, 42. The sort proceeds in the following stages:

- a Divide the 8 by 2 and compare elements 4 positions apart in the list, swapping if necessary:



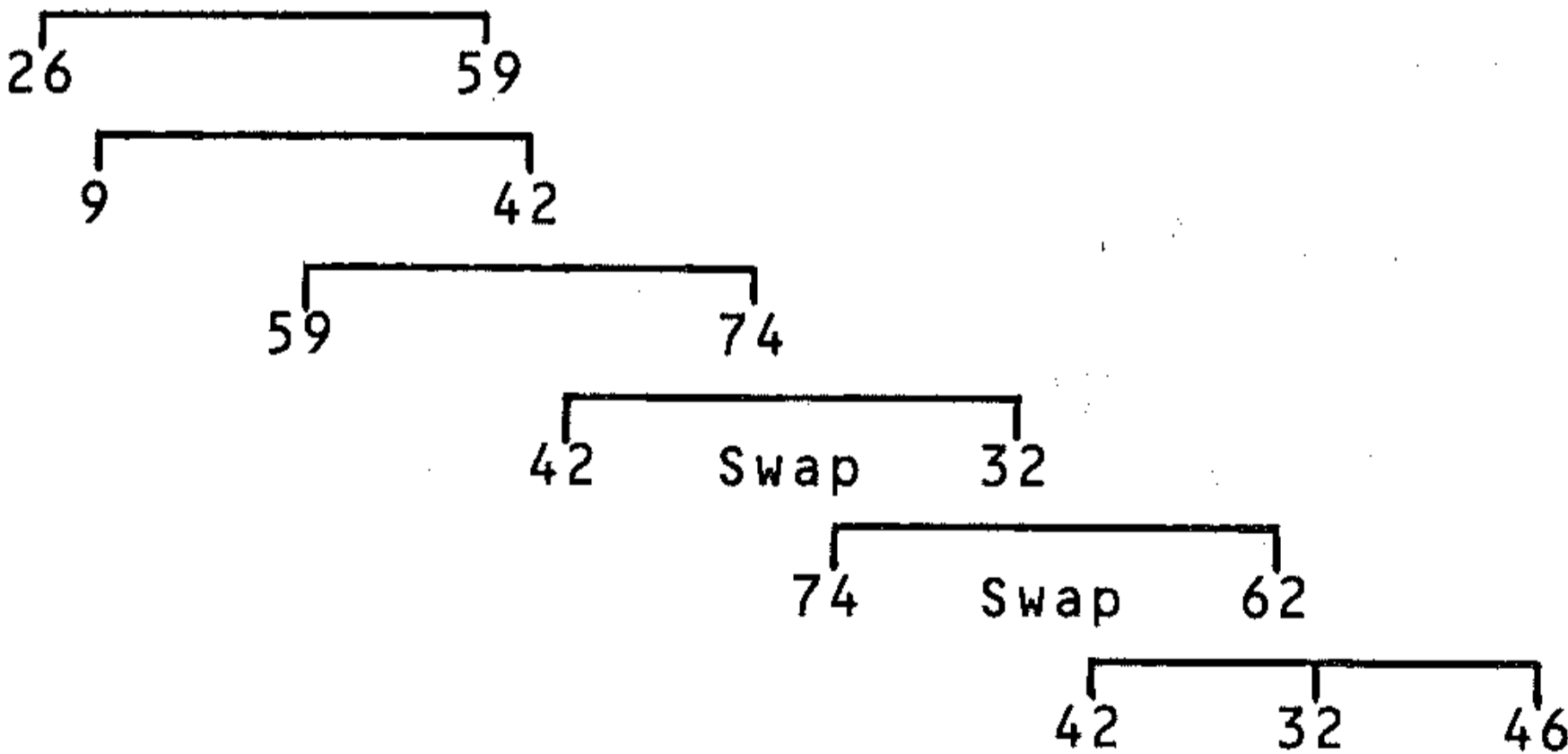
Compare:

- A(1) and A(5) Swap
- A(2) and A(6) Swap
- A(3) and A(7) In order – leave
- A(4) and A(8) Swap

New list:

26 9 59 42 74 32 62 46

- b Divide the 4 by 2 and compare the elements 2 positions apart in the list and swap if necessary:



New list:

26 9 59 32 62 42 74 46

- c Divide 2 by 2 and compare elements 1 apart in the list, which is using the equivalent of an insertion sort to give the final order:

9 26 32 42 46 59 62 74

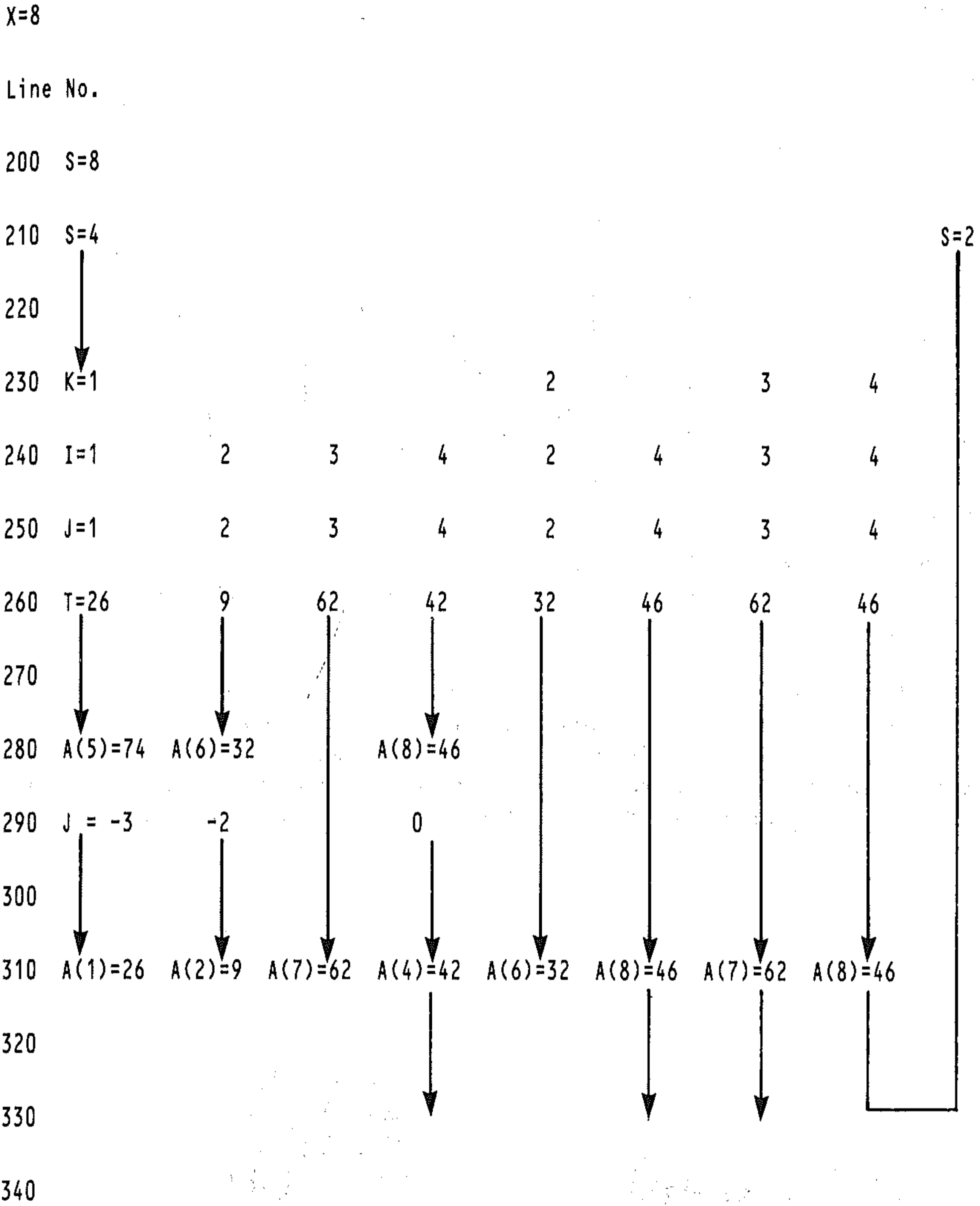
The steps we make in the program are as follows:

- a Select an integer S (number of positions apart from comparison). This is usually taken as $\text{INT}(N/2)$ where N = number of items in the list.
- b Sort the lists of items S positions apart, by comparing and swapping if necessary.
- c If $S < 1$ then stop, since the list is sorted.
- d If $S \geq 1$ then pick a new value of S (usually $\text{INT}(S/2)$), and repeat steps b) to d) as often as necessary.

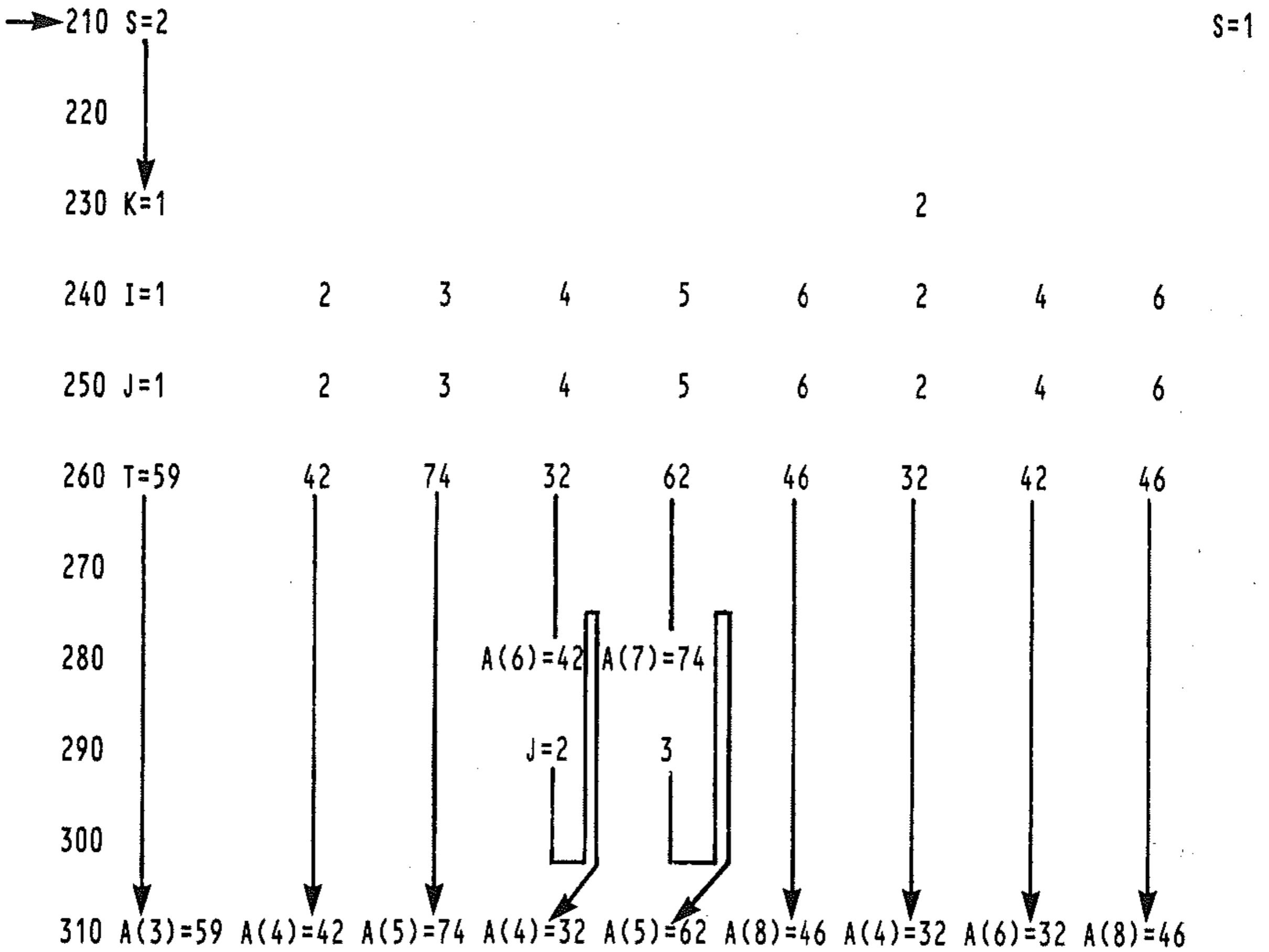
```
10 REM**SHELL**
20 INPUT "ENTER HOW MANY NUMBERS"; X
30 DIM A(X)
40 PRINT "ENTER NUMBERS ONE AT A TIME"
50 FOR N=1 TO X
60 INPUT A(N)
70 NEXT
80 PRINT
90 PRINT "UNSORTED LIST"
100 FOR N=1 TO X
110 PRINT A(N); " ";
120 NEXT N
130 PRINT
190 REM**SORTING ROUTINE**
200 S=X
210 S=INT(S/2)
220 IF S<1 THEN 400
230 FOR K=1 TO S
240 FOR I=K TO X-S STEP K
250 J=I
260 T=A(I+S)
270 IF T>=A(J) THEN 310
280 A(J+S)=A(J)
290 J=J-S
300 IF J>=1 THEN 270
310 A(J+S)=T
320 NEXT I,K
330 GOTO 210
340 REM**END OF SORT**
400 PRINT
410 PRINT "SORTED LIST"
420 FOR N=1 TO X
430 PRINT A(N); " ";
440 NEXT
450 REM**END**
```

Hand trace of shell sort

Consider the 8 element list 74, 32, 59, 46, 26, 9, 62, 42.

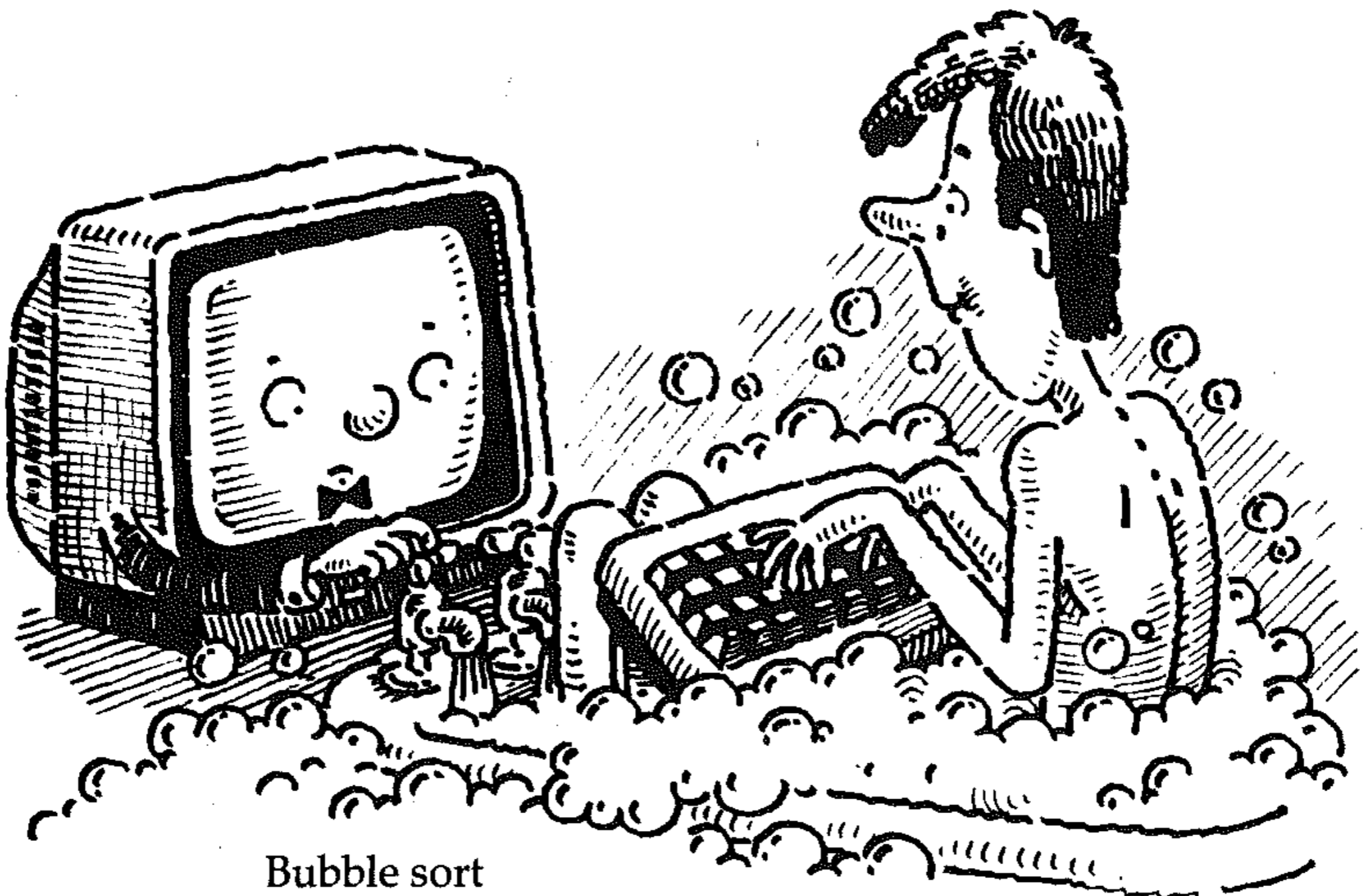


At this stage (1) the list is 26, 9, 59, 42, 74, 32, 62, 46.



At this stage (2) the list is 26, 9, 59, 32, 62, 42, 74, 46.

The final stage is the comparison of neighbouring elements, using the insertion sort technique, to which the Shell sort routine is equivalent when S=1. This is perhaps more easily seen considering the trace below of the operation of the Shell sort program on a simple 5-item list, in which only two passes need to be made.



Pass 1					Pass 2					
S = INT5/2 = 2					S = INT 2/2 = 1					
K = 1					K = 1					
I = 1	I = 2	I = 3	I = 2	I = 2	Start Pass 2	I = 1	I = 2	I = 3	I = 4	Finish
J = 1	J = 2	J = 3	J = 2	J = 2		J = 1	J = 2	J = 3	J = 4	
T=A(3)	T=A(4)	T=A(5)	T=A(4)	T=A(5)		T=A(2)	T=A(3)	T=A(4)	T=A(5)	
Start										
A(1)=2	1				1	1				1
A(2)=4		3		3	3	3	2			2
A(3)=1	2		2		2		3	3		3
A(4)=3		4		4	4			4	4	4
A(5)=5			5		5				5	5

The two methods of tracing a program illustrated here should show you the method by which a systematic analysis can be made of the changing values of variables in a program as processing proceeds. This is a procedure you should put to use when designing a program (i.e. in checking that the algorithm will work as intended) and when checking the operation of other people's programs that you wish to analyse. The procedure is also a great help in debugging a program. Break points inserted into the program (STOP commands), after which you can print the values of variables by direct commands, or PRINT statements, inserted as appropriate to print the values of variables at each step in the program, will enable you to check that the values occurring in the program are the same as the ones your trace diagram shows.

T6: Quick sort

This is a fast sorting technique which works by subdividing the list into two sub-lists and then subdividing the sub-lists. The principle of the quick sort is as follows: consider a list A(X) containing X numbers. In this example, X=8 and the numbers are as shown below. The following steps are carried out:

- 1 Initialise two pointers, I and J, at opposite ends of the list. Let X(I) be the reference number. In the example, this is 63.

I
J
63 27 43 96 72 31 82 43

- 2 Compare the two numbers indicated by the pointers and swap if necessary.

I
J
43 27 43 96 72 31 82 63

- 3 Move the pointer opposite the reference number one place towards it.

I
J
43 27 43 96 72 31 82 63

- 4 Repeat steps 2 and 3 until I = J.

I
J
43 27 43 96 72 31 82 63

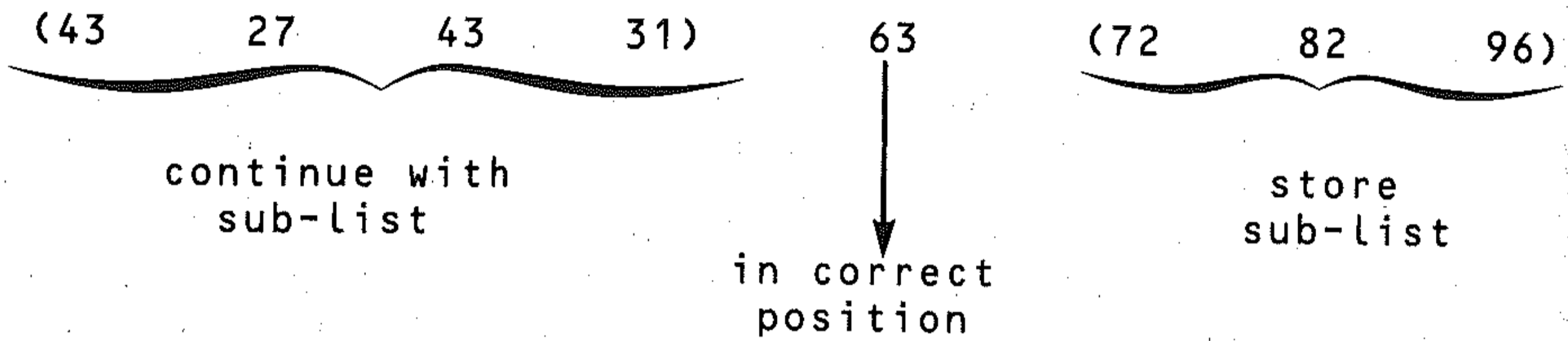
I
J
43 27 43 96 72 31 82 63

I
J
43 27 43 63 72 31 82 96

			I			J	
43	27	43	<u>63</u>	72	31	82	96
				I	J		
43	27	43	31	72	<u>63</u>	82	96
				I			
43	27	43	31	<u>63</u>	72	82	96

When this stage has been reached the list has been split into two sub-lists. The reference number is now in its correct position in the list, and the sub-lists are the numbers to the left and right of this position.

- 5 One of the lists is stored for future sorting (see below) and the other is taken through steps 1 to 4 above.



I			J
<u>43</u>	27	43	31
31	I		J
	27	43	<u>43</u>
31	27	I	J
		43	<u>43</u>
			↓
			in correct position

New Sub-list

[31	27	43]
<u>31</u>	27	43
31	27	43
27	<u>31</u>	43
	↓	
	in correct position	

- 6 This process is repeated, in each case storing a sub-list where necessary, and finally going back and sorting all stored sub-lists so that eventually each number is in the correct position.

The left-hand and right-hand numbers of a list are denoted by subscripts L and R respectively. The pointer positions are denoted by I and J and a flag S is set to indicate the pointer at the reference number, so that:

$$S = 1 \quad \text{if reference number at pointer I}$$

$$S = -1 \quad \text{if reference number at pointer J}$$

If at the end of step 4 the reference number is at I (as in the example) then the list has been split into:

sub-list	reference	sub-list
(L, ... , I - 1)	I	(I + 1, ... , R)

The right-hand list is remembered by setting up a *stack*, using an array $S(P, 2)$ with P initially set to zero. As each sub-list is stored, let:

$$P = P + 1, \quad S(P, 1) = I + 1 \text{ and } S(P, 2) = R$$

The array S is initially dimensioned as $S(X, 2)$ for a list with X elements. P indicates the number of the sub-lists; $S(P, 1)$ the left-hand element and $S(P, 2)$ the right-hand element of the sub-list. In the example given, at step 5 you will have the first sub-list generated stored by setting:

$$P = 1, \quad S(P, 1) = 6 \text{ and } S(P, 2) = 8$$

Thus each list to be stored is placed in sequence into the array, this process being known as **PUSHing** on to the **STACK**.

When the sub-list that the program continues with finally has only one number, you must return to sort the stored lists. Retrieve a stored sub-list (**POP** a list out of the **STACK**) by letting:

$$L = S(P, 1), R = S(P, 2) \text{ and } P = P - 1$$

and continuing until all the lists are sorted.

```
10 REM**QUICK**
20 INPUT "ENTER HOW MANY NUMBERS"; X
30 DIM A(X), S(X, 2)
40 PRINT "ENTER NUMBERS ONE AT A TIME"
50 FOR N=1 TO X
60 INPUT A(N)
70 NEXT
80 PRINT
90 PRINT "UNSORTED LIST"
100 FOR N=1 TO X
110 PRINT A(N); " ";
120 NEXT
130 PRINT
180 REM**SORTING ROUTINE**
190 P=0
200 L=1
210 R=X
220 I=L
230 J=R
240 S=-1
250 IF A(I)<= A(J) THEN 300
260 T=A(I)
270 A(I)=A(J)
280 A(J)=T
290 S=-S
300 IF S=1 THEN I=I+1
310 IF S=-1 THEN J=J-1
320 IF I<J THEN 250
```

```

330 IF I+1 >=R THEN 370
340 P=P+1
350 S(P,1)=I+1
360 S(P,2)=R
370 R=I-1
380 IF L<R THEN 220
390 IF P=0 THEN 450
400 L=S(P,1)
410 R=S(P,2)
420 P=P-1
430 GOTO 220
440 REM**END OF SORT**
450 PRINT
460 PRINT "#####SORTED LIST"
470 FOR N=1 TO X
480 PRINT A(N); " ";
490 NEXT
500 REM**END**

```

- Lines 190 – 230 initialise P, and set values for L, R and the pointers I and J.
Line 240 sets the flag which indicates the position of the reference pointer (S = -1).
Lines 250 – 290 make the interchange of A(I) and A(J) if necessary and reset the flag.
Lines 300 – 320 move whichever of I and J is to be moved, according to how the flag is set.
Line 330 checks if I is at the end of the list, bypassing the sub-list storage routine.
Lines 340 – 360 'push' sub-lists on to the stack.
Lines 370 – 380 check if the sub-list has more than one element, sending control back to line 220 if it has.
Line 390 sends control to the print routine if no sub-lists are stored.
Lines 400 – 420 'pop' sub-lists out of the stack.
Line 430 starts the sort routine for the 'popped' sub-list.

T7: Index sort

When data records or files contain several *fields* of information, it is often necessary to sort them according to one particular item. An index sort routine makes this possible. For example, consider a series of records, each containing a reference number, name, sex, age, home town and occupation:

```

10 SMITH MALE
32 ENFIELD GROCER
21 OXFORD BUTCHER
20 JONES FEMALE

```

Each record contains six fields. It may be required to sort these records alphabetically by name (field 2) or numerically by age (field 4). A sort of the type presented here enables sorting to be done for any of the fields. Since it uses string arrays to hold records and performs an *alphabetical* sort, it is necessary for all numerical items in any field to contain the same number of digits: for example 010, 020, 100, 200 instead of 10, 20, 100, 200, using leading zeros to maintain the value.

The procedure is as follows:

- 1 Set up an array N(N,L)$ containing N records, each of L fields. For example, an array N(10,5)$, to represent 10 records, each with 5 fields.
- 2 Decide on the *key field* – the fields you wish to sort – say the J th field. You must then set up an array K(N)$ to store it, and let K(R) = N(R,J) for the number of records (FOR R=1 TO N) so that the list K(N)$ will then contain the items you wish to arrange in order.
- 3 Sort the key field into ascending order. This is done in the subroutine starting at line 900 by counting how many times each element in the array K(N)$ is \geq the other elements (including itself). This sort uses a numerical array $X(N)$ to store the result of this count (P) for each element, by setting $X(P)$ to equal N , when K(N)$ is the item being checked.

First set $P = 1$ (since each item is equal to itself) and then check through the other items of $K$$ (lines 920 to 970), making the count by letting $P = P+1$ when the element is \geq another element. If the elements are equal, the original order in $N$$ is kept (line 960): (ie. test first element of $K$$ and set $X(P) = 1$, reset P , test second element of $K$$ and set $X(P) = 2$, etc.) For example, with 10, 30, 20, 40 as the $K$$ list, the array $X(4)$ would hold the values:

```
10 X(1) = K$(1)
30 X(3) = K$(2)
20 X(2) = K$(3)
40 X(4) = K$(4)
```

Printing out K(X(1))$ to K(X(4))$ in order will give the sorted order of $K$$ elements.

- 4 The array N(X(N), L)$ will now consist of the records sorted in the appropriate order, according to the field chosen, and is printed out using the loop variables R and I to access N(X(R), I)$ in lines 290 to 340.

```
10 REM**INDEX**
20 PRINT"   SORTING RECORDS"
30 PRINT"ENTER MAXIMUM NUMBER OF CHARACT
ERS IN ANY ITEM"
40 INPUT C
50 INPUT"ENTER NUMBER OF RECORDS";N
60 INPUT"ENTER NUMBER OF ITEMS IN RECORD
";L
70 REM**INITIALISE ARRAYS**
80 DIM N$(N,L),K$(N),X(N)
90 REM**INPUT RECORDS**
100 PRINT"   "
110 FOR R=1 TO N
120 PRINT"ENTER ";L;" ITEMS FOR RECORD "
;R
130 FOR I=1 TO L
140 INPUT N$(R,I)
150 N$(R,I)= LEFT$(N$(R,I),C)
160 NEXT I,R
170 PRINT
180 INPUT"   WHICH ITEM IS SORTING KEY";J
```

```

190 FOR R=1 TO N
200 K$(R)=N$(R,J)
210 NEXT
220 GOSUB 900
230 PRINT
240 PRINT "SORTED RECORDS ARE"
250 PRINT
260 FOR R=1 TO N
270 FOR I=1 TO L
280 PRINT N$(X(R),I); " ";
290 NEXT I
300 PRINT
310 NEXT R
320 INPUT "DO YOU WISH TO CONTINUE Y/N"; Y$
330 IF Y$="Y" THEN 180
340 STOP
400 REM**PROGRAM END**
890 REM**SORTING SUBROUTINE**
900 FOR A=1 TO N
910 P=1
920 FOR B=1 TO N
930 IF K$(A)>K$(B) THEN P=P+1
940 IF K$(A)=K$(B) THEN 960
950 GOTO 970
960 IF A>B THEN P=P+1
970 NEXT B
980 X(P)=A
990 NEXT A
1000 RETURN
1010 REM**END OF SUBROUTINE**

```

For example, if you input a storage array of 4 records with 3 fields, maximum 6 characters in any item, and use as input data:

SMITH	460	OXFORD
JONES	080	LEEDS
BROWN	730	YORK
WHITE	095	BATH

results are as follows:

a using field 1 as key sorted records are:

BROWN	730	YORK
JONES	080	LEEDS
SMITH	460	OXFORD
WHITE	095	BATH

b using field 2 as key sorted records are:

JONES	080	LEEDS
WHITE	095	BATH
SMITH	460	OXFORD
BROWN	730	YORK

c using field 3 as key sorted records are:

WHITE	095	BATH
JONES	080	LEEDS
SMITH	460	OXFORD
BROWN	730	YORK

T8: Linear search

The most straightforward way of looking for a particular number in a list of unsorted numbers is to examine the list one by one, in each case comparing with the 'wanted' number.

In this program a set of random numbers is created between 100 and 199 and it is arranged so that there is only a single occurrence of each number. This is in lines 40 – 60. The list is printed out in lines 70 – 110. The search routine is then carried out in lines 200 – 300. Clearly a number near the beginning of the list is found quickly but one at the end rather slowly. For a 50 element list the average number of searches will be 25.

```
10 REM**SEARCH1**
20 DIM A(100)
30 INPUT "ENTER NUMBER LESS THAN 100";N
40 IF N>100 THEN 30
50 FOR M=1 TO N
60 A(M)=INT(100*RND(1))+100
65 IF M<>1 THEN 70
66 NEXT M
70 FOR R=1 TO M-1
80 IF A(M)=A(R) THEN 60
90 NEXT R,M
100 PRINT "UNSORTED LIST"
110 FOR M=1 TO N
120 PRINT A(M); " ";
130 NEXT
140 PRINT
200 REM**LINEAR SEARCH**
210 PRINT "ENTER NUMBER BETWEEN"
220 INPUT "100 AND 199";X
230 FOR I=1 TO N
240 IF X=A(I) THEN 290
250 NEXT
260 PRINT "NUMBER NOT IN LIST"
270 PRINT "AFTER ";N;" SEARCHES"
280 GOTO 400
290 PRINT "FOUND ";X;" AFTER ";I;" SEACH
ES"
400 REM**END**
```

If a number occurs more than once in the list, it will be necessary to go through the complete list and test each item. Make the following modifications to the program:

Delete lines 70, 80 – this allows numbers to occur several times. Edit line 90 to read:

```
90 NEXT M
```

Insert line 225, a flag to test if number is present (1) or not present (0):

```
225 S = 0
```

Replace 240:

```
240 IF X = A(I) THEN PRINT 'NUMBER';X;" IS ELEMENT";I;" OF LIST"
```

Insert 245:

```
245 IF X = A(I) THEN S = I:REM FLAG SET
```

Replace 270:

```
270 IF S = 0 THEN PRINT "NUMBER NOT IN LIST"
```

Delete lines 280, 290, 400.

T9: Binary search

This is a much faster search technique than the linear search but can only be used for a list that has already been put in order. In many applications you will be dealing with an ordered list and under such circumstances this is the appropriate method to use.

In the program the binary search technique is in lines 500 to 600. The idea is to compare the wanted number with the middle item of the ordered list. The wanted item is then either smaller (in which case it is in the first half of the list) or larger (in which case it is in the second half of the list) than the middle item (unless it is equal to it, in which case the search is already completed). The process is repeated, in each case halving the list. Consider a search for 30 in the following list:

```
1  2  4  6  8  10  12  14  16  18  20  24  28  30  36
```

The first choice is 14 (middle). The list is now:

```
16  18  20  24  28  30  36
```

and the next choice is 24 (middle). The list is now:

```
28  30  36
```

Now 30 (middle) is selected, and the number is found in three searches (compared with fourteen using the linear search).

In the program the following sections occur:

- Setting up and printing initial unordered list (lines 10 – 150).
- Sorting this list into order and printing it (lines 200 – 330).
- Binary search with printout (lines 500 – 680).

```
10 REM**SEARCH2**
20 DIM A(50)
30 INPUT "ENTER NUMBER LESS THAN 50";N
40 IF N>50 THEN 30
50 FOR M=1 TO N
60 A(M)=INT(100*RND(1))+100
70 IF M<>1 THEN 90
80 NEXT M
90 FOR R=1 TO M-1
```

```

100 IF A(M)=A(R) THEN 60
110 NEXT R,M
120 PRINT"UNSORTED LIST"
130 FOR M=1 TO N
140 PRINT A(M); " ";
150 NEXT
160 PRINT
200 REM**INSERTION SORT**
210 FOR I=1 TO N-1
220 J=I
230 T=A(I+1)
240 IF T>=A(J) THEN 280
250 A(J+1)=A(J)
260 J=J-1
270 IF J>=1 THEN 240
280 A(J+1)=T
290 NEXT
300 REM**END OF INSERTION SORT**
310 PRINT"SORTED LIST"
320 FOR M=1 TO N
330 PRINT A(M); " ";
340 NEXT
350 PRINT
360 PRINT"ENTER NUMBER REQUIRED BETWEEN
100 AND 199"
380 INPUT"TO FINISH ENTER 999"; X
390 IF X=999 THEN 700
400 PRINT
410 PRINT"SEARCH "; N; " ITEM LIST"
500 REM**BINARY SEARCH**
510 L=1
520 H=N
530 C=0
540 M=INT((H+L)/2)
550 C=C+1
560 IF X=A(M) THEN 630
570 IF L>=H THEN 660
580 IF X>A(M) THEN 610
590 H=M-1
600 GOTO 540
610 L=M+1
620 GOTO 540
630 PRINT"NUMBER FOUND "; X
640 PRINT"AFTER "; C; " SEARCHES"
650 GOTO 350
660 PRINT"NUMBER NOT FOUND"
670 PRINT"AFTER "; C; " SEARCHES"
680 GOTO 350
690 REM**END SEARCH**

```

Results:

TYPE NUMBER <50

UNSORTED LIST

144	128	117	118	101	189	111	198
150	107	188	197	172	106	157	148
168	160	130	181	100	165	175	133
186	190	155	167	199	138	122	163
131	142	113	143	154	194	119	153

SORTED LIST

100	101	106	107	111	113	117	118
119	122	128	130	131	133	138	142
143	144	48	150	153	154	155	157
160	163	165	167	168	172	175	181
186	188	189	190	194	197	198	199

TYPE NUMBER REQUIRED BETWEEN 100 AND 199
TO FINISH TYPE 999

SEARCH 40 ITEM LIST
FOUND 198 AFTER 5 SEARCHES

Section U: Memory and Machine Code

U1: Binary systems

Digital computers operate with sequences of numbers in the binary number system. Binary numbers are numbers to base 2, while the 'normal' number system is decimal (base 10). The binary system uses only two digits, 0 and 1. These are binary digits (bits). The computer holds a bit as a voltage level (+5v or 0v) in a switched pathway.

In the decimal system, a number, for example 418, is *coded* as a number using the digits 0 to 9. The coding is based on powers of ten. 418 means: (4 times 10 to the power 2) + (1 times ten to the power 1) + (8 times ten to the power 0).

In the decimal system, a number, for example 418, is *coded* as a number using the digits 0 to 9. The coding is based on powers of ten. 418 means: (4 times 10 to the power 2) + (1 times ten to the power 1) + (8 times ten to the power 0).

$$\begin{array}{r r r r r r} (4 \times 10^2) & + & (1 \times 10^1) & + & (8 \times 10^0) & \\ 400 & + & 10 & + & 8 & = & 418 \end{array}$$

The binary system of coding uses powers of *two* in exactly the same way. The number 13 is represented as:

$$\begin{array}{r r r r r r r r r r} (1 \times 2^3) & + & (1 \times 2^2) & + & (0 \times 2^1) & + & (1 \times 2^0) & & & & \\ 1 & & 1 & & 0 & & 1 & & & & \text{Binary number 1101} \\ 8 & + & 4 & + & 0 & + & 1 & = & 13 & \text{Decimal equivalent} \end{array}$$

The binary number 101110 is evaluated as:

$$\begin{array}{r r r r r r r r r r r r} (1 \times 2^5) & + & (0 \times 2^4) & + & (1 \times 2^3) & + & (1 \times 2^2) & + & (1 \times 2^1) & + & (0 \times 2^0) & \\ 32 & + & 0 & + & 8 & + & 4 & + & 2 & + & 0 & = & 46_{10} \end{array}$$

Key in the following program, which converts decimal numbers to their binary representation:

```

10 REM CONVERTS DECIMAL TO BINARY
20 INPUT "ENTER DECIMAL NUMBER"; N
30 M=N: B$="": Z$="0000000"
40 L=INT(N/2)
50 B=N-2*L
60 IF B=1 THEN A$="1"
70 IF B=0 THEN A$="0"
80 B$=A$+B$
90 N=L
100 IF L>0 THEN 40
110 B$=LEFT$(Z$,8-LEN(B$))+B$
120 PRINT "M; " IS " ; B$ ; " IN BINARY"
130 INPUT "ANOTHER GO (Y/N) "; A$
140 IF A$="Y" THEN 20
    
```

Input sequences of numbers to familiarise yourself with the binary system. The program only deals with positive whole numbers. Trace the program to see how it works, using the examples 13 and 46 given above as inputs. Non-integer numbers are dealt with by using an exponent, as with the E notation system for decimals. Binary numbers have a *binary point*, and bits to the right of the point are binary fractions, representing the reciprocal power of two. The binary number 1.101, for example, represents:

$$\begin{array}{rcccccc} (1 \times 2^0) & + & (1/2^1) & + & (0/2^2) & + & (1/2^3) \\ 1 & + & .5 & + & 0 & + & .125 & = & 1.625 \end{array}$$

You may have noticed that binary numbers as seen above are all positive. Negative numbers are dealt with using a particular form of binary representation. The method used by the Commodore to store numbers is described later in this Section.

A *byte* is a sequence of 8 bits. The sequence of 8 bits represents a number between 0 and 255 decimal, 00000000 and 11111111 binary. The Commodore uses 8 bit *words*, ie. 1 byte.

U2: The memory map

Memory in computers is organised as a linear sequence of *addresses*. Each address is a memory location or memory cell holding a single byte. The binary numbers in these locations are interpreted as numbers, characters or instructions, depending on their context in the computer memory. The organisation of the memory is constant, but the space occupied by each area of memory varies according to the program and its requirements. As an obvious example, a long program takes up more space for storage than a short one. Memory is of two types:

Read Only Memory (ROM) is fixed and cannot be altered. It contains the BASIC interpreter program, and is built into the computer in manufacture.

Random Access Memory (RAM) is variable, multi-purpose memory that holds the current program and all the other elements of data required to run the program.

Data can only be extracted (*read*) from ROM and is permanent. RAM memory can be both read from and written to. Inserting a value into a memory location in RAM (writing or storing) will wipe out or overwrite the existing data at that address.

Memory capacity is referred to as the number of *Kilobytes* (k) involved. 'Kilo' means one thousand, but a kilobyte is actually 1024 (2^{10}) bytes (the closest binary number to 1000).

The Commodore 64 has a complex memory map. It is called the 64 because there is 64k of memory available to the processor and also because it has 64k of RAM. This may seem strange because when the machine is powered up it says `38911 Basic Bytes free`.

Consider the fundamental memory map (as shown in Fig. U1). The numbers shown on the left are the hexadecimal starting addresses of the blocks of memory. Notice that there is both an 8k basic section (A000 – BFFF) and the kernal section (E000 – FFFF). The kernal will be dealt with later; without going into detail here, it is possible to reorganise memory using a special port on the 6510 microprocessor to select different arrangements.

The basic memory map (Default) is shown in Fig. U2. The figures shown in the box will be explained later, but they refer to various flags in the machine. For the moment it is sufficient to assume the memory is as in the map.

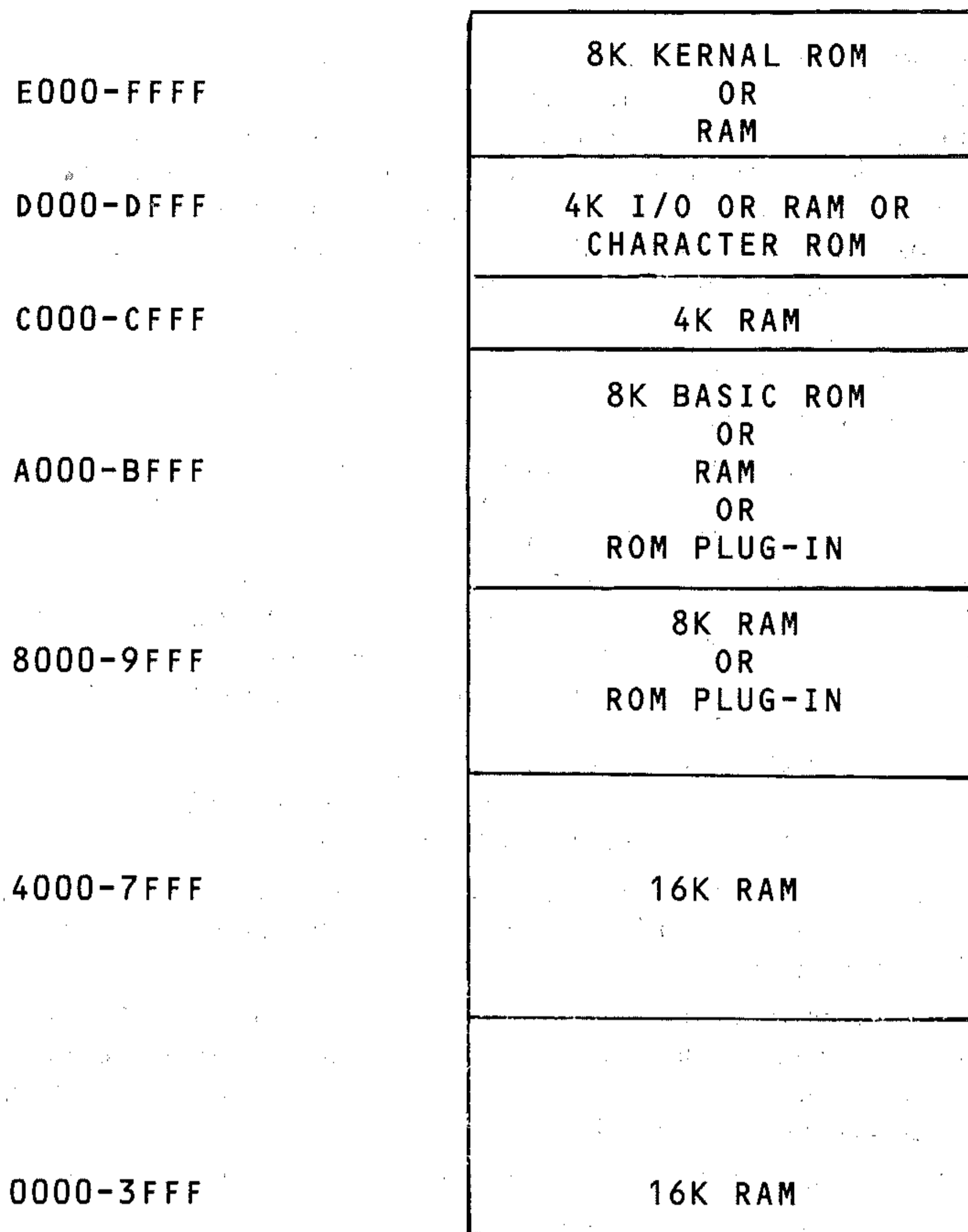
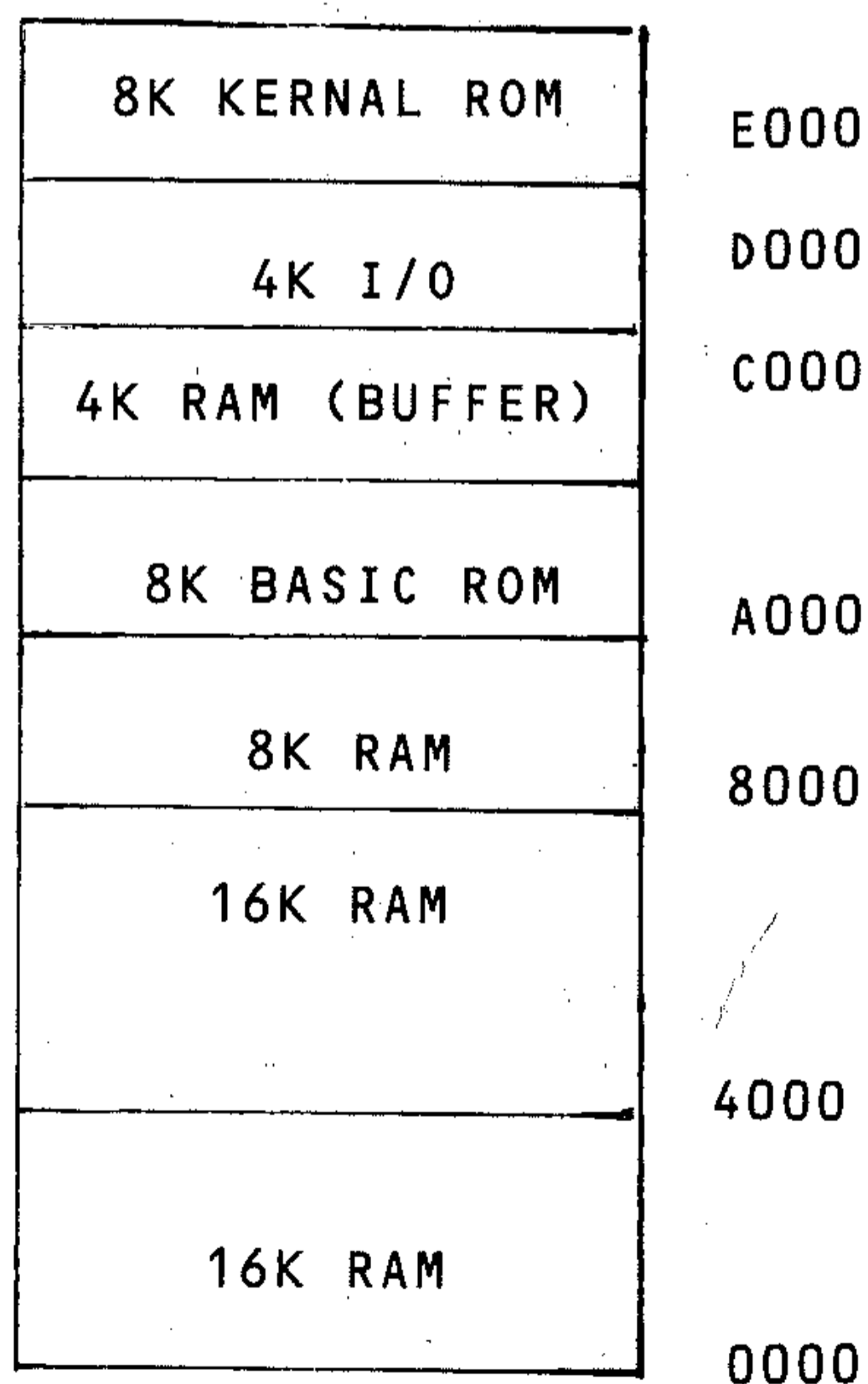


Figure U1: Fundamental memory map



This is the default BASIC memory map which provides BASIC 2.0 and 38K contiguous bytes of user RAM.

Figure U2: Basic memory map

The complete memory organisation of the Commodore 64 is listed in the Commodore manual under the heading *Commodore 64 Memory Map*. Column 1 shows the label, column 2 the hex address, 3 the decimal equivalent and 4 a short description. The label is called a *mnemonic* – this is a short easily-remembered name, for example, Lastpt (last temporary string address). Most of the locations (labels) will have little meaning to you, but some can be used to gain a better understanding of the Commodore.

Warning: Locations 0000 and 0001 should not be altered. If you accidentally alter these, you will probably need to switch the machine off.

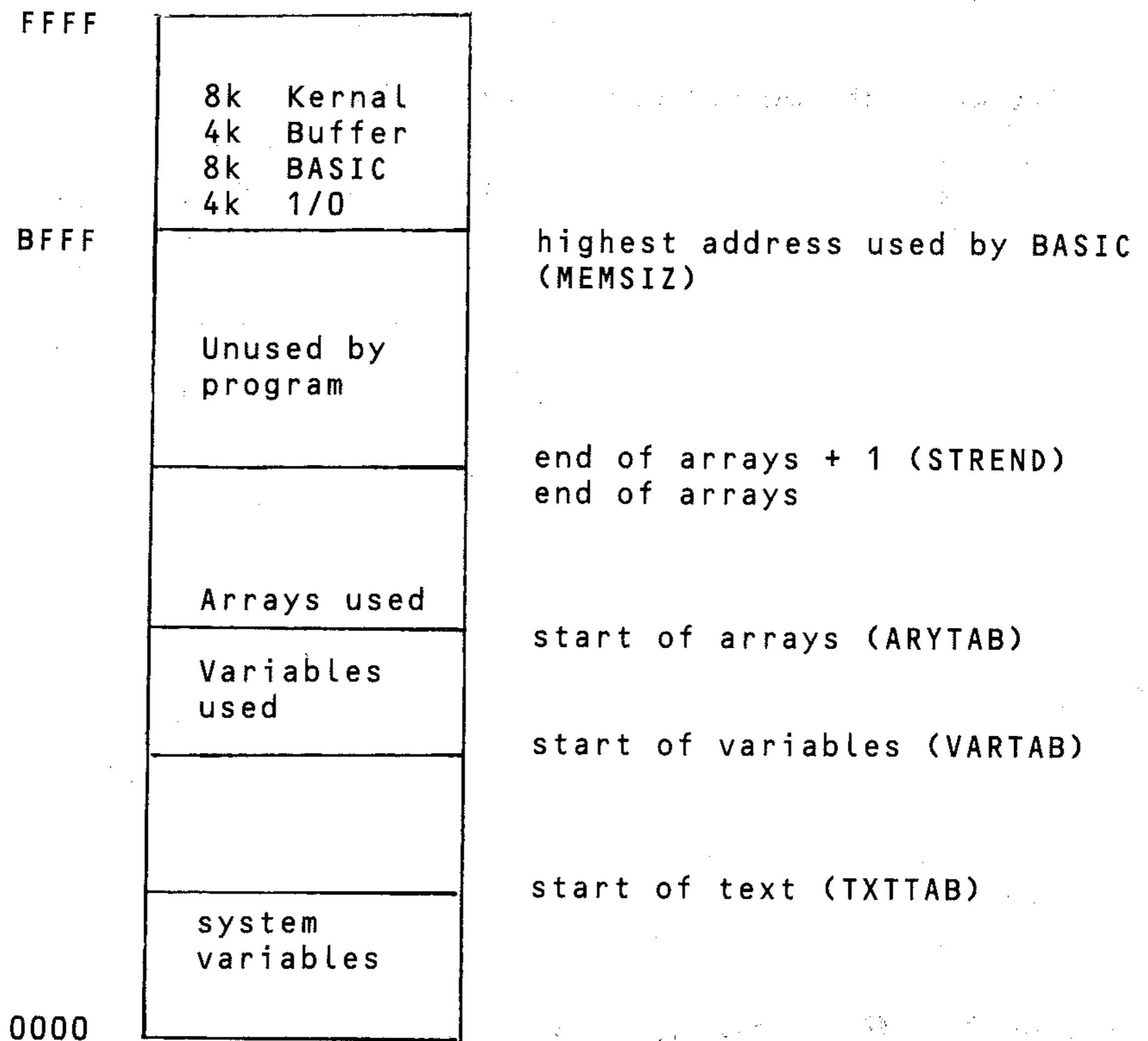
Here is a list of the most useful locations.

TXTTAB	43 – 44	Pointer to start of BASIC text.
VARTAB	45 – 46	Pointer to start of BASIC variable.
ARYTAB	47 – 48	Pointer to start of BASIC arrays.
STREND	49 – 50	Pointer to end of arrays plus one.
MEMSIZ	55 – 56	Highest address used by BASIC (Ramtop).
CURLIN	57 – 58	Current BASIC Line number.
OLDLIN	59 – 60	Previous BASIC Line number.
STACK	256 – 511	Stack.
KEYD	631 – 640	Keyboard buffer queue (FIFO).
COLOR	646	Current character colour code.
GDCOL	647	Background colour under cursor.

Note: The keyboard buffer is called a FIFO (First In First Out); this means that the first character typed in will be removed first. The stack is an area of memory reserved for special needs.

When BASIC calls a subroutine, it puts its return address of the stack. When it hits an RTS (return from subroutine) statement, BASIC 'pops' the return address of the stack. The stack is called a LIFO (Last In First Out), meaning that the last entry is removed before any other.

Your program is stored in memory (RAM) in the following way.



VARTAB, TXTTAB, ARYTAB and STREND are not fixed but are controlled by your program. In computer terms they are *dynamic variables*. The start of text is usually fixed but the start of variables (and consequently start of arrays) are moved up in memory. Thus when you run a program it creates its variables and arrays at the end of your program in memory. When your program stops execution (the ready message appears), try entering a new line and use the GOTO statement; it works, but all variables are now undefined. Run this program:

```
10 INPUT A
20 PRINT A
30 PRINT A↑2
40 PRINT A↑3
```

Now enter GOTO 20 at the end and run it again. The same results will be printed. Now add this line:

```
50 PRINT A↑4
```

and run it with the GOTO 20. Notice the difference. The variables have been reset to zero because a new line has been entered, showing that the memory used to store variables is placed at the end of your program and is overwritten by new program lines.

U3: PEEK and POKE

The PEEK and POKE instructions have been introduced for specific purposes in connection with the display file and character storage. This should have given you some understanding of their uses. Now that you have been introduced more generally to the way memory is stored in a computer, you will notice that these instructions provide direct access to the memory of the computer.

PEEK(N) returns the value (in decimal notation) of the number stored in binary form in the memory byte of address N.

For the Commodore, ROM extends from E000 – FFFF and A000 – BFFF, and RAM from 0000 to A000 (address in hexadecimal).

POKE M,N places into the memory address M the binary form of the decimal number N. N must be in the range 0 to 255 (to fit in a single byte), whereas M must be in the range 0 to 65535. ROM may not have values POKEd into it.

In general, PEEK is used to extract from memory any values useful to a program, and POKE to insert values into memory. Remember that the values (entered and returned in decimal notation) can be numbers or characters. Machine-code programming is performed by POKeIng into a specified sequence of addresses the values which correspond to instructions which the 6510 central processor chip understands. This sequence of instructions is then called from within a BASIC program, in a similar fashion to calling a subroutine, and is executed. At the end of the machine-code program, a return instruction passes back control to the BASIC program.

PEEK will be used to investigate how numbers and program lines are stored in the Commodore. Different types of number have different formats in which they are held. The normal number in memory is held by 5-byte *floating point binary form*. Thus 5 bytes of memory are used.

U4: The program area

The line numbers in a program are stored in 2 bytes of memory. Numbers in a program listing are stored as their printed characters, then in 5-byte form.

Enter the program below. It PEEKs the memory locations of the program storage area (from the start of the program listing), after printing out the numbers as instructed in the first three lines. For each memory location it prints out the address, the contents of the address as a decimal number, and the character string corresponding to it.

```
10 PRINT 23
20 PRINT 123 E 8
30 PRINT 123 E -8
40 PRINT ""
50 A=PEEK(43)+256*PEEK(44)
60 PRINT A;TAB(10);PEEK(A);TAB(20);
70 IF PEEK(A) >= 32 OR PEEK(A) < 127 THEN
PRINT CHR$(PEEK(A))
80 PRINT"CONTROL CHARACTERS OR KEYWORDS"
90 A=A+1
100 PRINT:GOTO 60
```

At first sight the listing displayed does not make much sense. Press the **RUN/STOP** key and run the program again, but press the **CTRL** key and hold it down so that it slows down the listing on the screen, so that you can see what is happening. The PEEKs in line 50 set A to the value of the first memory address of the program storage area.

U5: Systems variables

The system variables area of memory is a fixed area. These generally occupy either one or two bytes.

Single byte variables are stored as a number between 0 and 255 decimal, and two-byte variables between 0 and 65535. In a two-byte variable there are 16 bits which can store $2^{16} - 1$, which is 65535. When PEEKing a two byte number in memory, you must remember that the first value returned multiplied by 256 plus the next value give the required number.

In the case of line numbers the procedure is slightly different; the value of the first number returned plus 256 multiplied by the second gives the line number.

Summary

Two byte number $A*256+b$
Line numbers $A+256*b$

U6: The hexadecimal system

Hexadecimal (often abbreviated to hex) is another numbering system much used in computing, although it has no practical application for beginning programmers since it is primarily used in machine-code programming. Whereas binary is base 2, and decimal base 10, hexadecimal is base 16. This is a convenient system for computers using 8-bit words, since $16 \times 16 = 256$. Any value which is held in a single byte can thus be represented by a two-digit hex code. The system uses the digits 0 to 9, and goes on with A, B, C, D, E and F, to represent the numbers 1 to 16.

Here is how the system counts:

Decimal	Hexadecimal	
0	0	(0×16^0)
1	1	(1×16^0)
.	.	
.	.	
.	.	
9	9	(9×16^0)
10	A	(10×16^0)
.	.	
.	.	
15	F	(15×16^0)
16	10	$(1 \times 16^1) + (0 \times 16^0)$
17	11	$(1 \times 16^1) + (1 \times 16^0)$
.	.	
25	19	$(1 \times 16^1) + (9 \times 16^0)$
26	1A	$(1 \times 16^1) + (10 \times 16^0)$
.	.	
.	.	
.	.	
160	A0	$(10 \times 16^1) + (0 \times 16^0)$
.	.	
.	.	
250	FA	$(15 \times 16^1) + (10 \times 16^0)$
.	.	
254	FE	$(15 \times 16^1) + (14 \times 16^0)$
255	FF	$(15 \times 16^1) + (15 \times 16^0)$

As with any number system, you could go on (256 is 100 hex, etc), but the use of hexadecimal is in representing binary numbers in a more convenient form than strings of 1's and 0's, which are difficult to read and easy to make mistakes with (unless you are a computer).

The advantage of hexadecimal notation is that any 8-bit binary number is convertible to hex far more easily than into decimal, due to the relationship of base 2 and base 16 numbers. Base 16 is base 2 to the 4th power, and this means that the 8 bits of a byte can be divided into two sets of four bits (remember 1111 binary = 15 decimal = F hex) and converted to the corresponding two hex digits. For example, the number 116 decimal is in binary 01110100. Split into two groups of four bits, 0111 and 0100, each group is converted to a hex digit.

0111 binary (7 decimal) is 7 hex

0100 binary (4 decimal) is 4 hex

The number in hex is 74. Check this:

$$(7 \times 16^1) + (4 \times 16^0) = 112 + 4 = 116.$$

To avoid possible confusion when both hexadecimal and binary numbers are being used, a small *h* should be used after a hexadecimal number. The examples above would be 74h and 5Dh. Hex numbers are grouped in twos, and the leading zero should be used for numbers less than 16 decimal, so that 12 decimal should be written 0Ch, not C or Ch.

Similarly, numbers up to 65535 decimal (held in two bytes of binary), are representable with four hex digits. Thus 1111111111111111 binary is FFFF hex, and 1010110100111110 is 1010 (A)/1101 (D)/0011 (3)/1110 (E): AD3Eh. Work out the value of this in decimal.

The program library (in the Appendix) has programs which convert decimal to hex and vice versa (HEXDEC and DECHEX). Analyse these programs to see how they work. The ASC and CHR\$ functions are used to check the hex notation, or produce it.

The last number system commonly used in computing is the OCTAL numbering system. Octal uses the digits 0 – 7 inclusive. Thus 10 octal is 8 in decimal, ie. $1*8 + 0*8$ which gives 8 (any number raised to the power 0 gives 1), also 15 octal is $1*8 + 5*8$ which gives $8 + 5$ which is 13 decimal.

Summary

Number systems covered:

hexadecimal	(16)
decimal	(10)
octal	(8)
binary	(2)

Consider:

$$..R^3 + R^2 + R^1 + R^0 .+ R^{-1} + R^{-2} + R^{-3} ..$$

where R is the radix or base of the number system. Thus if $R = 2$ (binary) and you had a number 1011, this could be represented as

$$1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$$

which gives $8+0+2+1 = 11$ decimal.

The above notation is useful as a way to convert between real numbers of different bases.

U7: Machine code programs

Although it is beyond the scope of this book to explain the concepts of machine code programming, a brief introduction to the use of machine code programs is useful. When you have entered a machine code you execute it using the following command:

```
SYS(X)
```

where X is the address of the start of your routine.

You can enter machine code programs in a number of ways, then enter the decimal equivalent and POKE into memory. If the machine code routine is given in hexadecimal code, then it must be converted into a decimal equivalent manually or by including a hex-dec conversion routine. The use of an *assembler* is highly recommended, as it makes the writing of machine language routines considerably easier. They are not expensive. Also, if you are interested in machine code, it is worth getting an assembler and an *editor*. (An assembler allows you to look at machine code programs in mnemonic form rather than decimal or binary, and an editor allows you to edit and examine the running of your program.)

It is important that you save a program containing machine code before you execute it, or you will lose it if it doesn't work first time; the Commodore will not respond to you. This is because once you execute a SYS statement, the monitor program is halted while your routine executes. If your routine errors, it won't be able to return to monitor or BASIC and so the Commodore will crash. This won't cause any damage, but the only solution is to switch off and then on again (as the Commodore has no reset button). The RUN/STOP key is called a *soft reset*, but it only works when either BASIC or monitor is in operation.

The writing of machine code routines without an assembler and an editor can be tedious, but in the long run is worth while.

Next follows an introduction to the 6510 processor and to running machine code on the Commodore, followed by two programs for you to try. After this is a

list of monitor routines and a short description of their uses. For further information a *disassembler* should be obtained and further investigation will be needed to fully understand machine code programming.

U8: The 6510 microprocessor chip

The Commodore 64 has a CPU (Central Processing Unit) called a 6510, an upgraded version of the 6502. It is not the purpose of this book to reach the basics of machine codes; there are a number of excellent books that cover the 6502. But here is a short introduction to the 6510.

The 6510 is an eight bit processor. This means it can handle numbers from 0-255 directly, in binary 11111111 to 00000000 (ie. 8 bits). It also has a 16 bit address bus. Thus it can address $2^{16} - 1$ memory locations of 64k. All the major operations are carried out in a register called the accumulator, including adding, subtracting and logic operators. It also has two 8 bit index registers referred to as the x and y registers. These are used mainly to act as counters or index and are often used in association with the accumulator.

Any machine code program must start and end somewhere in memory. To keep track of your program, there is a register called the program counter, which keeps track of where the next instruction will be found in memory.

The final register is the stack pointer. The 6510 has a special area of memory called the *stack*; it is very useful when writing machine code programs. It operates on the principle of Last In First Out (LIFO). This means that if two numbers are pushed onto the stack, the last number on the stack is the first out. Thus the stack can be used as a convenient store for numbers (i.e. entered information can be put on the stack for processing later).

As in BASIC, machine code programs use subroutines. When a subroutine is called it pushes its return address onto the stack. (The address in the program counter.) When returning from a subroutine the return address is pulled off the stack and put in the program counter and execution of the code is resumed from there.

The 6510 can address 64k (64 x 1024 bytes) of memory, each 1024 bytes is split into 4 units of 256 bytes, which are called pages. Thus there are 64 times 4 pages which is 256 pages. The two most important pages are page 0 and page 1.

Page 1 is the stack (explained above) and page 0 is referred to as zero page. Zero page can be used as extra registers to the 6510 – using its addressing modes for example.

The 6510 CPU has a limited instruction set (58 instructions) but has 13 possible addressing modes. While it is not the purpose of this book to go into great detail of the CPU, to give a general introduction, a few example programs and all the information needed to use the 64 in machine code are included.

For all but the shortest programs, an assembler should be obtained, as writing programs, converting to decimal and using BASIC to POKE them into memory is slow and very tedious.

There follows a couple of useful programs that clear the high resolution screen and change the screen colour. (You may remember that in the graphics section using BASIC to do this was very slow and time consuming.) These programs are first given in assembly listing, followed by a short BASIC program to demonstrate its use. Both routines are essentially the same, the differences being the amount of memory cleared and characters POKEd (ie. to clear screen poke 0, to change colour poke colour code).

Routine for clearing the high resolution screen

HEX ADDRESS	HEX CODE	DECIMAL	ASSEMBLER MNEMONICS
C000	A0 20	160 32	LDY #\$20
C002	A2 00	162 0	LDX #\$00
C004	A9 00	169 0	LDA #\$00
C006	9D 00 60	157 0 96	L1: STA \$6000,X
C009	E8	232	INX
C00A	D0 FA	208 250	BNE L1
C00C	EE 08 C0	238 8 192	INC L1+2
C00F	88	136	DEY
C010	D0 F4	208 244	BNE L1
C012	60	96	RTS

The X register is used as an index to store the accumulator in 6000h upwards (STA \$6000,X); when X=1, value in A is stored at 6001h. The X register can only hold up to 255; then it cycles to zero again. It is incremented and tested for zero. This is done at lines 5 and 6 (using INX and BNE LI, where LI is a label). When X does not equal zero it will jump to LI, store A at address+ X and continue. When it equals zero it will increment the byte (memory location) 2 bytes on from LI, the high order byte. Thus 60 becomes 61 after 255 increments for X, and now the accumulator will store from 6355 + X. Next the Y register is decreased by one. As Y contains 32 it will next contain 31. Consider that X goes from 0 to 255, 32 times; this is 256 x 32, which is 8k. Thus this routine will clear 8k of memory for the high resolution screen. The last two instructions are BNE LI and RTS. The BNE LI will branch when Y register does not equal zero; when it does the RTS (return from subroutine) will be executed. This is necessary because BASIC calls this routine as a subroutine, and when the RTS is executed it returns to BASIC.

The next routine changes the colour of the high resolution screen. The only differences between the two routines is that this one clears 1K of memory; and that thus LDY #\$4 and the colour code (xx=16* foreground + background) is loaded into the accumulator. Note that the area of memory used for the two routines is different (see the high resolution graphics for details).

Routine for changing colour of the high-res screen

HEX ADDRESS	HEX CODE	DECIMAL	ASSEMBLER MNEMONICS
C000	A0 04	160 4	LDY #\$04
C002	A2 00	162 0	LDX #\$00
C004	A9 xx	169 xx	LDA #\$xx
C006	9D 00 40	157 0 64	L1: STA \$4000
C009	E8	232	INX
C00A	D0 FA	08 250	BNE L1
C00C	EE 08 C0	238 8 192	INC L1+2
C00F	88	136	DEY
C010	D0 F4	208 244	BNE L1
C012	60	96	RTS

Here is a BASIC program that loads these routines into memory and executes them.

```

10 For A=0 to 18
20 READ B
30 POKE A + LOC,B
40 NEXT B
50 SYS(LOC)
60 POKE LOC + 1,4
70 POKE LOC + 8,64

```



```

75 INPUT"ENTER COLOUR AS 16*FOREGROUND
   COLOUR + BACKGROUND COLOUR";COLOUR
80 POKE LOC + 5,COLOUR
90 SYS(LOC)
100 END
110 DATA 160,32,162,0,169,0,157,0,96,232
120 DATA 208,250,8,192,136,208,244,96

```

Note that LOC is the location when the machine code is stored (normally C000h i.e. 49152 decimal).

U9: The kernal

As far as the user is concerned, the Commodore 64 machine seems only to work in BASIC. But BASIC is only one aspect of the machine. BASIC is a language, defined in terms of the operations and structures it performs and allows; BASIC does not handle input or output to any device (screen, keyboard, printer etc). These and many other tasks are performed by the *kernal*.

The kernal can be considered the heart of the machine; all input/output, memory management and the running of the BASIC language is done through the kernal. Thus it is possible to replace BASIC with another language, which is then linked into the kernal. From the user's point of view, the kernal's routines are used by calling them to perform various tasks.

In the Commodore 64 manual you will find a table of user callable kernal routines and a short description of their use; they are given for your reference and experimentation. To fully understand them, it would be necessary to use a disassembler to follow what these routines do.

U10: Input/output memory locations

The Commodore 64 manual includes a complete list of all addresses in the 64 that affect colour, sound, high resolution, and other special features. These include using light pens, joysticks and a real time clock (in hours, minutes, seconds and tenths of a second). Again, it is beyond the scope of this book to explain their use, but by all means experiment with them till you gain some understanding of them.

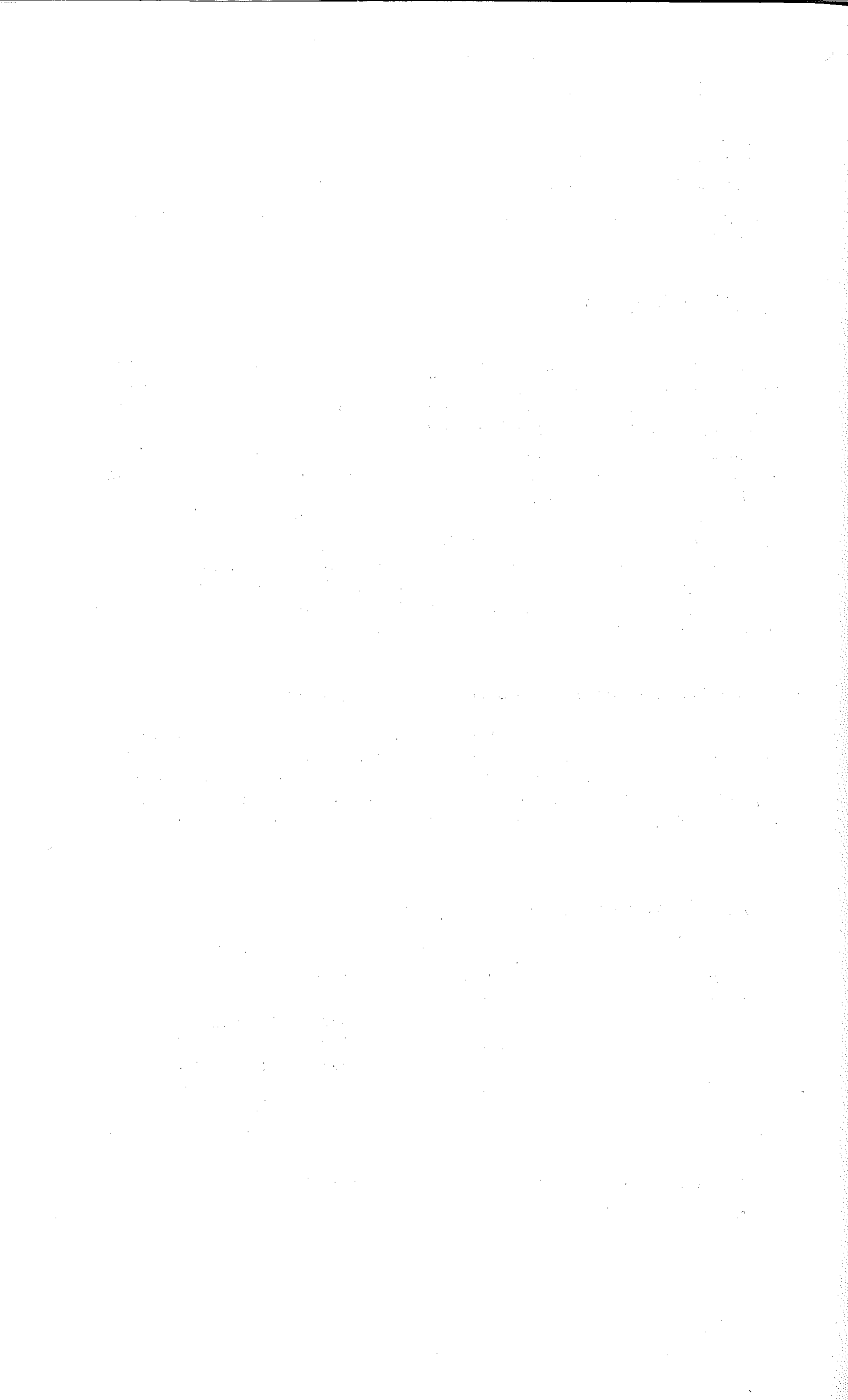
U11: Complete memory maps

The following locations are used by the 64 for setting up its memory map.

<i>Name</i>	<i>Bit</i>	<i>Direction</i>	<i>Description</i>
LORAM	0	Output	Control for RAM/ROM at A000 to BFFF (BASIC)
HIRAN	1	Output	Control for RAM/ROM at E000 to FFFF (kernal)
CHAREN	2	Output	Control for I/O ROM at D000 to DFFF

These are the first 3 bits of the 6510 Input/Output control port situated at location 0001.

The Commodore 64 manual includes tables showing how memory can be arranged in the Commodore 64.



PART FOUR

APPLICATION PROGRAMS

THE UNIVERSITY OF CHICAGO

Section V: Applications Programs

V1: Programming for applications

You have been introduced to the full set of Commodore BASIC instructions. A wide range of operations has been covered, including loops, lists, array manipulation, sorting and subroutines, as well as the implementation of control structures. These are the raw material of programming. The combination of what you have learned about program design and the task you wish the computer to perform produces an applications program.

There are no rules to derive algorithms. If there were, they could be coded into a master program that would write your programs for you. This book illustrated ways of thinking about problems, but each program you wish to write presents a unique set of circumstances. Familiarity with the language and control structures, and with existing solutions to a variety of problems, make it easier to program; but the art and craft of programming becomes easier only with practice. The importance of keeping notes about programs you have written, and on solutions to problems you have found in analysing other programs, is that it will prevent you having to re-invent the wheel. As you write more programs yourself, you will find that you come to recognise the method (or methods – there is seldom only one way to perform a given task!) by which you can implement and code each module of your program.

Modular, structured program design methods help to break down a programming problem to these recognisable chunks, and as you gain experience you will begin to recognise aspects of programs as problems you already know how to solve. You will also become familiar with the types of data structure required in a program to make it possible to manipulate the data efficiently, and grasp more quickly and clearly that, for example, a given set of data is more efficiently handled in a multi-dimensional array than in separate lists, or that a similar routine in different program modules could be handled by a single subroutine if suitable variables were initialised before calling the subroutine.

Experience cannot be transferred, and there is no substitute for practice, but examples can be given. In this final part of the book, you will find examples of the important topic of user-friendly programs, and then examples of programs written to perform specific tasks, to illustrate the process of designing applications programs. Games programming is briefly dealt with, too, since games are a good testing ground for problem-solving and programming techniques.

V2: Instructions and input checks

If you have written your own program, you know which inputs the program requires, in what form and when. You know, for example, that when ANOTHER GO? appears on the screen, you must press the Y key to run the program again.

Now consider what happens if someone else wishes to use the program (or if you return to it after some weeks). There is not enough information available to the user. The term *user-friendly* is applied to programs which have sufficiently clear and precise instructions to tell someone who has never seen the program running exactly what to do. You should always attempt to make our programs reasonably accessible. To continue the example of running a program again, the line:

```
60 PRINT "AGAIN?"
```

could be followed by

```
70 INPUT A$
80 RUN
```

or

```
70 IF A$="" THEN 70
80 RUN
```

or

```
70 IF A$="" THEN 70
80 IF A$="Y" THEN RUN
```

These need different responses. You could use for the first **PRESS RETURN TO RUN AGAIN**, for the second **HIT ANY KEY TO RUN AGAIN**, for the third **AGAIN? (PRESS Y OR N)**, or similar instructions appropriate to the program.

The user recognises that an input prompt requires string or numeric input, according to its form, but (as you shall see later) there may be reasons for requesting a number in string form, and in any case it must be made clear what is required. You should be careful to use, for example:

"INPUT FIRST WORD" not "INPUT A\$"

"ENTER A NUMBER 1 TO 10" not "INPUT X"

"MONTH (1 TO 12)" not "MONTH?"

since if the user does not know what A or A\$ are in the context of the program he or she is unlikely to respond correctly, and might try entering JAN or MARCH for the month.

You should also avoid the use of instructions grouped together:

```
10 PRINT"INPUT CURRENT,P.D, KNOWN AND UNKNOWN RESISTOR"
20 INPUT A,B,C,D
```

or

```
10 DIM A(10,3)
20 PRINT "INPUT MATRIX"
30 FOR F = 1 TO 10
40 FOR N = 1 TO 3
50 INPUT A(F,N)
60 NEXT N
70 NEXT F
```

It is very easy for the user to forget which of the inputs is currently required. The information also fails to include the units of the values required, and does not print the input values on the screen. This would be a better format, providing both clear instructions and visible input values:

```
10 PRINT "INPUT CURRENT IN AMPS"
20 INPUT A
30 PRINT "CURRENT ="; A; "AMPS"
40 PRINT "INPUT P.D. IN VOLTS"
50 INPUT B
60 PRINT "P.D. ="; B; "VOLTS"
```

A look at any reasonably complex user-friendly program will show you that a significant portion of any application program is instructions. Instructions should be concise, but only to a degree that still provides adequate information. Expanded instructions can form part of the documentation of a program, but the program itself must contain the basic instructions required to ensure correct input and manipulation. The program **MATTMULT** in the program library (see *Appendix*) has a good approach to the array entry problem. Try to write one yourself, then compare the two routines.

The combination of good instructions and input checks is the best method of reducing user error. The human being is less reliable than the computer, and far more inaccurate results or program crashes occur due to input error than to bugs in the program (assuming it has successfully completed a sequence of dry runs).

Checks to reduce the possibility of human error, or to prevent bad effects from it, are the means by which a program is 'idiot-proofed' or 'mug-trapped'. Commercial programs designed for inexperienced users often have as much space devoted to input checks as to the program proper. You can assume some awareness in the users of your programs, and trust that they will enter 2 and not TWO, for example, but check routines can ensure that simple keyboard errors are not passed over. Subtle errors or straightforward mistakes are less easy to deal with.

It is a simple matter to check that an entered value is within an acceptable range:

```
10 INPUT "INPUT MONTH (1 TO 12)";M
20 IF M>12 OR M<1 THEN 10
```

Since the month is to be input as an integer, you could add a line to check this:

```
30 IF INT(M)<>M THEN 10
```

In fact one line will do it all:

```
20 IF INT(M)<>M OR M>12 OR M<1 THEN 10
```

It is better to have a statement specifically stating that there was an error in input. To continue with the same example, you could have these lines:

```
20 IF INT(M)=M AND M<=12 AND M>=1 THEN 50
30 PRINT "INPUT ERROR:RE-ENTER MONTH"
40 GOTO 10
50 PRINT "INPUT YEAR (AS 82 FOR 1982, ETC.)"
60 ..... (rest of program)
```

The user is informed what is wrong, and told what to do. Make sure you see why the new line 30 had to have both the relational and logical operators switched round to make the program work.

To enable re-use of check or error routines, it is convenient to place them in subroutines. The following date entry routine uses a subroutine to print an error message for a few seconds (line 500) which is used if any of the checks shows an error. The routine checks the following:

- day of month between 1 and 31 (line 40)
- month between 1 and 12 (line 90)
- year between 1911 and 1998 (line 140)
- whether the year is a leap year, and if it is not, that 29 February has not been entered (line 190)
- that days which do not exist in some months have not been entered (line 210).

Check the logic used in these lines to see how it works. The lines are good examples of how multiple conditions can be combined, but for that reason they are a little difficult to follow.

```
10 PRINT "ENTER DATE"
20 PRINT "DAY?"
30 INPUT D$:D=VAL(D$)
40 IF D>=1 AND D<=31 THEN 70
50 GOSUB 500
60 GOTO 20
70 PRINT "MONTH? (1 TO 12)"
```

```

80 INPUT M$:M=VAL(M$)
90 IF M>=1 AND M<=12 THEN 120
100 GOSUB 500
110 GOTO 70
120 PRINT "YEAR? (AS LAST 2 DIGITS)"
130 INPUT Y$:Y=VAL(Y$)
140 IF Y>10 AND Y<99 THEN 170
150 GOSUB 500
160 GOTO 120
170 REM *CHECK DAY US MONTH*
180 REM *LEAP YEAR*
190 IF INT((Y+1900)/4)<>(Y+1900)/4 AND M
=2 AND D=29=-1 THEN 220
200 REM *SHORT MONTHS*
210 IF((M=2 AND D>29) OR ((M=4 OR M=6 OR
M=9 OR M=11)AND D=31))=0 THEN 240
220 GOSUB 500
230 GOTO 10
240 REM ....PROGRAM
245 T$="/"
248 YY$="/19"
250 PRINT D$;T$;M$;YY$;Y$
260 REM.....
270 REM.....
400 GOTO 999
490 REM **ERROR NOTICE**
500 PRINT "****INPUT ERROR****"
502 PRINT "PLEASE FOLLOW INSTRUCTIONS"
504 PRINT "PRESS ANY KEY TO START "
510 GET A$:IF A$="" THEN 510
520 PRINT " "
530 RETURN
999 END

```

It is also a simple matter to put in input checks that print the input, and invite the operator to check if it is correct, and to re-input if an error has been made. This is important where multiple data entries are being made, since an error would otherwise require entering everything again. As an example, here is a check routine for string input:

```

10 INPUT"HOW MANY NUMBERS";X
40 FOR N=1 TO X
50 PRINT "INPUT STRING";N
60 INPUT W$(N)
70 PRINT W$(N)
80 PRINT "IF INCORRECT PRESS E TO RE-ENT
ER"

```



```

85 PRINT "PRESS ANY OTHER KEY TO CONTINU
E IF OK."
87 GET A$
90 IF A$="" THEN GOTO 87
100 IF A$="E" THEN GOTO 50
110 NEXT N

```

Rather than check each value, it is sometimes better to wait until all entries have been made, and then print them out for checking. This routine does this for a list of numbers:

```

5 Z$=""
10 PRINT "-----"
20 PRINT "ENTER DATA ITEM : "
30 PRINT "-----"
40 PRINT "-----"
50 FOR A=1 TO 4
60 PRINT " | | | | "
62 PRINT " | | | | "
65 NEXT
70 PRINT "-----"
80 DIM A$(20):FOR A=1 TO 20
85 PRINT "-----";
86 FOR B=1 TO 5
90 GET A$:IF A$="" THEN 90
100 IF ASC(A$)=46 THEN 140
120 IF A$=CHR$(13) THEN 155
130 IF ASC(A$)<48 OR ASC(A$)>57 THEN 90
135 A$(A)=A$(A)+A$
140 PRINT A$;
150 NEXT B
155 PRINT "-----DATA COR
RECT:ENTER N IF NOT"
156 GET A$:IF A$="" THEN 156
157 PRINT "-----"
158 IF A$="N" THEN PRINT "-----
":A$(A)="" :GOTO 85
160 PRINT "-----"

```

```

170 NEXT A
300 C=1
310 PRINT "*****"; : FOR A=1 TO 4
331 FOR B=1 TO 5
332 F=LEN(A$(C))
333 IF F=5 THEN 335
334 A$(C)=A$(C)+LEFT$(Z$,5-F)
335 PRINT "  "; A$(C); " ";
336 C=C+1
337 NEXT B:PRINTCHR$(13); " ";
340 NEXT A
350 PRINT "-----"

```

The virtue of using string input is that instead of stopping the program with an error message if an invalid entry is made (a letter, character that is non-numeric, or more than one decimal point), as occurs with a numeric input, the string input can be accepted whatever the characters input. The inputted string must be checked, however, or else the user gets an error message when using VAL to convert to a numeric value. This requires a routine like the following:

```

5 REM "STRINGNUM"
7 X$="*****"
8 Y$="*****"
9 PRINT ""
10 PRINT LEFT$(Y$,0);LEFT$(X$,5);"INPUT
NUMBER"
20 INPUT N$
30 DP=0
40 IF N$="" THEN 100
50 FOR F=1 TO LEN(N$)
60 IF ASC(MID$(N$,F))=47 OR ASC(MID$(N$,
F))<46 OR ASC(MID$(N$,F))>57 THEN 100
70 IF ASC(MID$(N$,F))=46 THEN DP=DP+1
80 NEXT F
90 IF DP>1 THEN 100
95 GOTO 130
100 PRINT LEFT$(Y$,0);LEFT$(X$,5);"ERROR
IN INPUT"
105 PRINT "PRESS ANY KEY TO CONTINUE"
110 GET A$:IF A$="" THEN 110
115 GOTO 9
130 REM .. REST OF PROGRAM..
150 PRINT "NUMBER ENTERED IS ";VAL(N$)
180 PRINT "PRESS ANY KEY TO CONTINUE"
190 GET A$:IF A$="" THEN 190
200 GOTO 9

```

Line 20 inputs the string. Line 30 sets a variable to store the number of decimal points. Line 40 checks that **RETURN** key alone was not pressed, and passes control to line 100 to print an error message if it was. Lines 50 and 80 set a loop for the number of characters in N\$, and line 60 uses **ASC** to check whether characters other than numbers and the decimal point are present, and goes to 100 for an error message if they are. Line 70 adds 1 to the variable **DP** for each decimal point found in N\$. After the loop, line 90 sends control to 100 for an error message if there is more than one decimal point. Line 95 bypasses the error routine, and line 150 uses **VAL** to return the number for printing. At this point, if the number were needed for calculation, a variable could be set ($N = \text{VAL}(N\$)$) to store the value.

V3: Example programs

Here are some examples of applications programs of various types, as follows:

- **REACT**: reaction time testing.
- **BINGO**: creation, calling and checking of cards for playing Bingo on the computer.
- **REF.INDEX**: the calculation of refractive indices from the angle of deviation and prism angle data produced by spectrometer experiments.
- **SERIES**: the summing of a convergent series to a given degree of accuracy.
- **ELEMENT**: the calculation of empirical chemical formulae from the percentage composition of compounds, or the percentage composition from the numbers of atoms of each element in the molecule.

None of these programs are particularly complex, and they deal with fairly straightforward applications. However, the principles involved are valid for any size of program, and the programs themselves demonstrate many of the techniques and procedures introduced earlier in the text. More examples of applications programs and useful subroutines are to be found in the program library provided in the Appendix, but they are not as fully annotated. The programs here are presented as problems and solutions, with some discussion of the approach to the problem. The procedure is then presented, and the derived program.

Please remember that any program can be written in different ways, even given that the algorithm is exactly the same. This variety of solutions means that there is never only one correct program.

"REACT"

Problem: to use the computer to assess response times in reaction to a signal. An average should be taken of a number of timings.

Research the problem: the timing function can be used to calculate the time preceded by a random delay to prevent anticipation. Computing time must be allowed for in the result. The number of timings desired should be input and used to set up a loop.

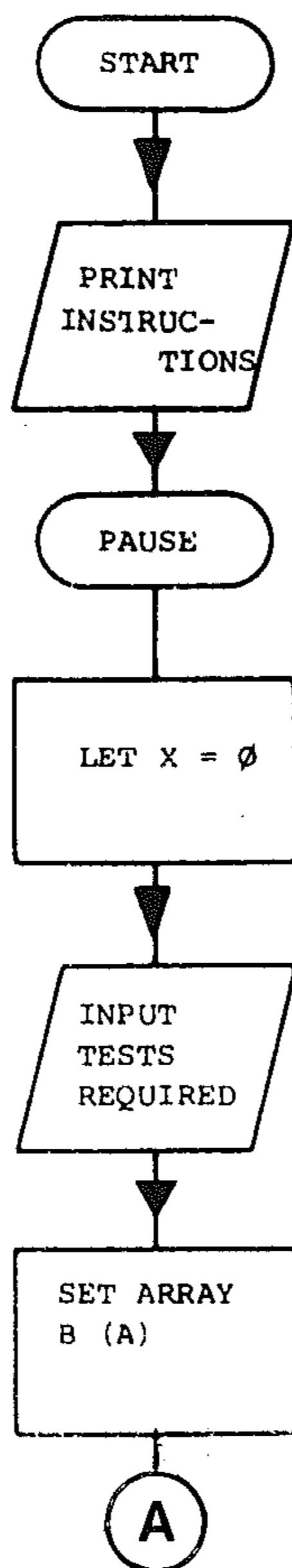
Procedure: the program will have a loop structure, determined by the input of the number of tests required. Outside this loop will be the instructions at the beginning of the program, and the output of average reaction time.

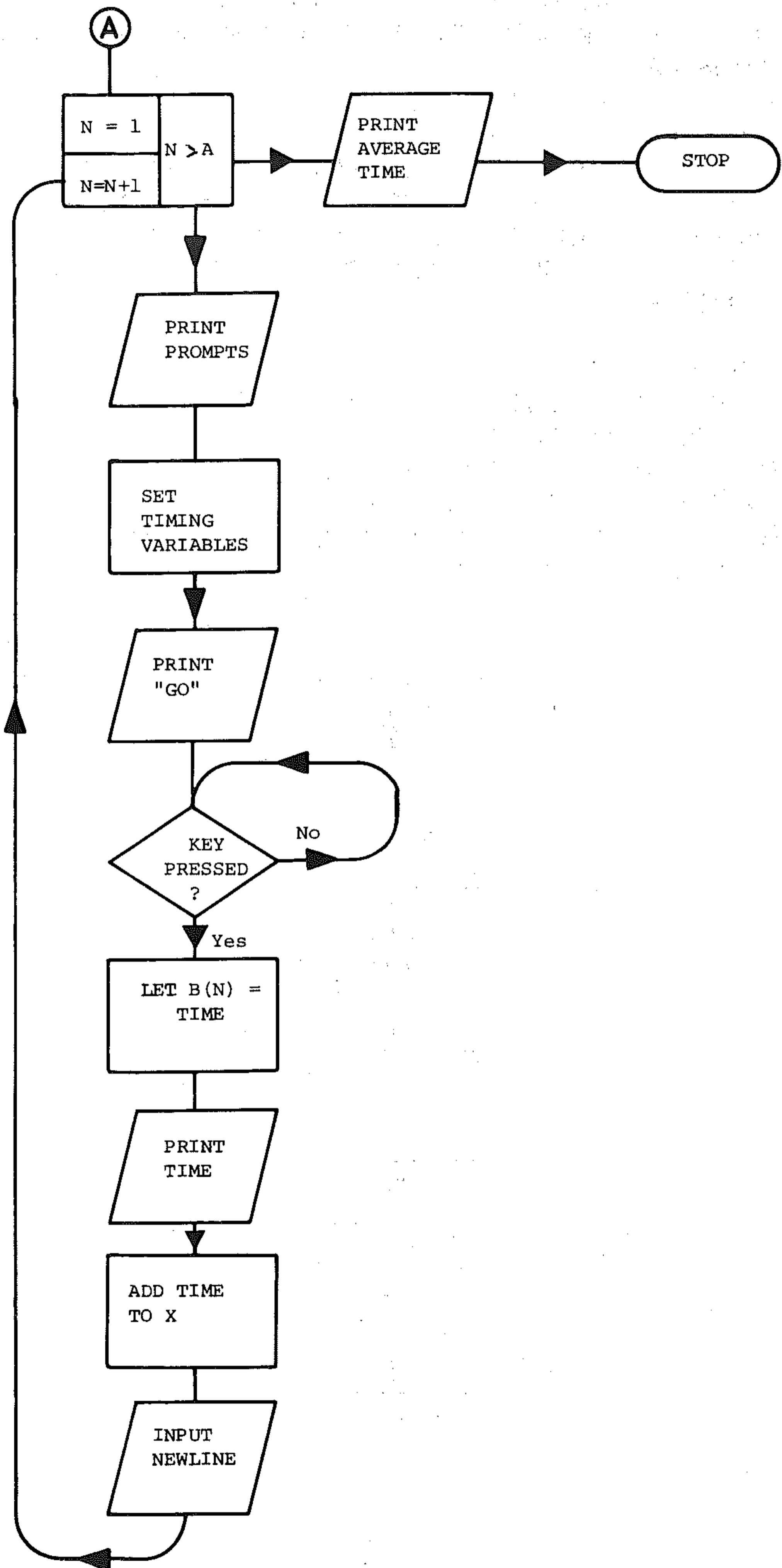
This timing module is the core of the program. The input module (instructions, 'get ready' messages and an input for the number of timings required) and the output module (average time) are easily built around this. You can proceed to key in a version of this module, having decided your structure for the program, and test/debug the timing module, before using the editing facilities to


```

5 PRINT " "
7 Y$=" "
8 X$=" "
10 PRINT "REACTION TEST"
20 PRINT "-----"
30 PRINT "SCREEN WILL SHOW <GET SET>"
40 PRINT "PRESS ENTER EACH REACTION TIME
  AND THE AVERAGE WILL BE DISPLAY"
50 INPUT "HOWMANY TESTS REQUIRED ";A
60 DIM B(A)
70 PRINT " ":X=0
80 REM...
100 REM ** TEST LOOP **
110 FOR N=1 TO A
120 PRINT "ON YOUR MARKS"
130 FOR I=1 TO 1500:NEXT I
140 PRINT "PRESS ENTER WHEN READY"
150 PRINT "<GET SET>"

```





```

160 FOR L=1 TO RND(1)*500:NEXT L
170 REM...
180 REM CALCULATE REACT. TIME
190 REM ....
200 TI$="000000":PRINT" "
210 PRINT LEFT$(Y$,12);LEFT$(X$,20);"**
GO**"
220 B(N)=TI/60
230 GET A$:IF A$<>CHR$(13) THEN 220
240 PRINT LEFT$(Y$,20);LEFT$(X$,15);B(N)
;" SECONDS"
250 X=X+B(N)
260 PRINT "PRESS ANY KEY TO CONTINUE"
270 GET A$:IF A$="" THEN 270
280 PRINT " ":NEXT N
290 REM...
300 REM * END STOP *
310 PRINT " ":PRINT
320 FOR N= 1 TO A
330 PRINT "B(";N;")=";B(N);" SECONDS"
340 NEXT N:PRINT
350 PRINT "AVERAGE TIME WAS ";X/A;"SECON
DS"

```

Comments: the use of an array (B(A)) allows flexibility in adding further manipulations (full printout or standard deviation, for example) if desired, by adding a further module to the program.

The same principle could be used to time other processes. The accuracy of the allowance for the delay due to computing time becomes less important as the time being measured increases. To derive a stop-watch program you would need a start and stop routine. Try to write such a routine, and note how the program is set so that the computer waits until a key is pressed.

"BINGO"

Problem: in a Bingo game, the caller shouts out the numbers between 1 and 99 in a random order and each player has a card with a set of numbers (say 15) in this range. The cards for each player contain different sets of numbers. The winner of the game is the player whose list of numbers is called first. The program should play the game for up to four players and check the validity of the winning player's card.

Outline procedure: modules are required as follows:

- 1 Set up game
- 2 Play game
- 3 Result of game

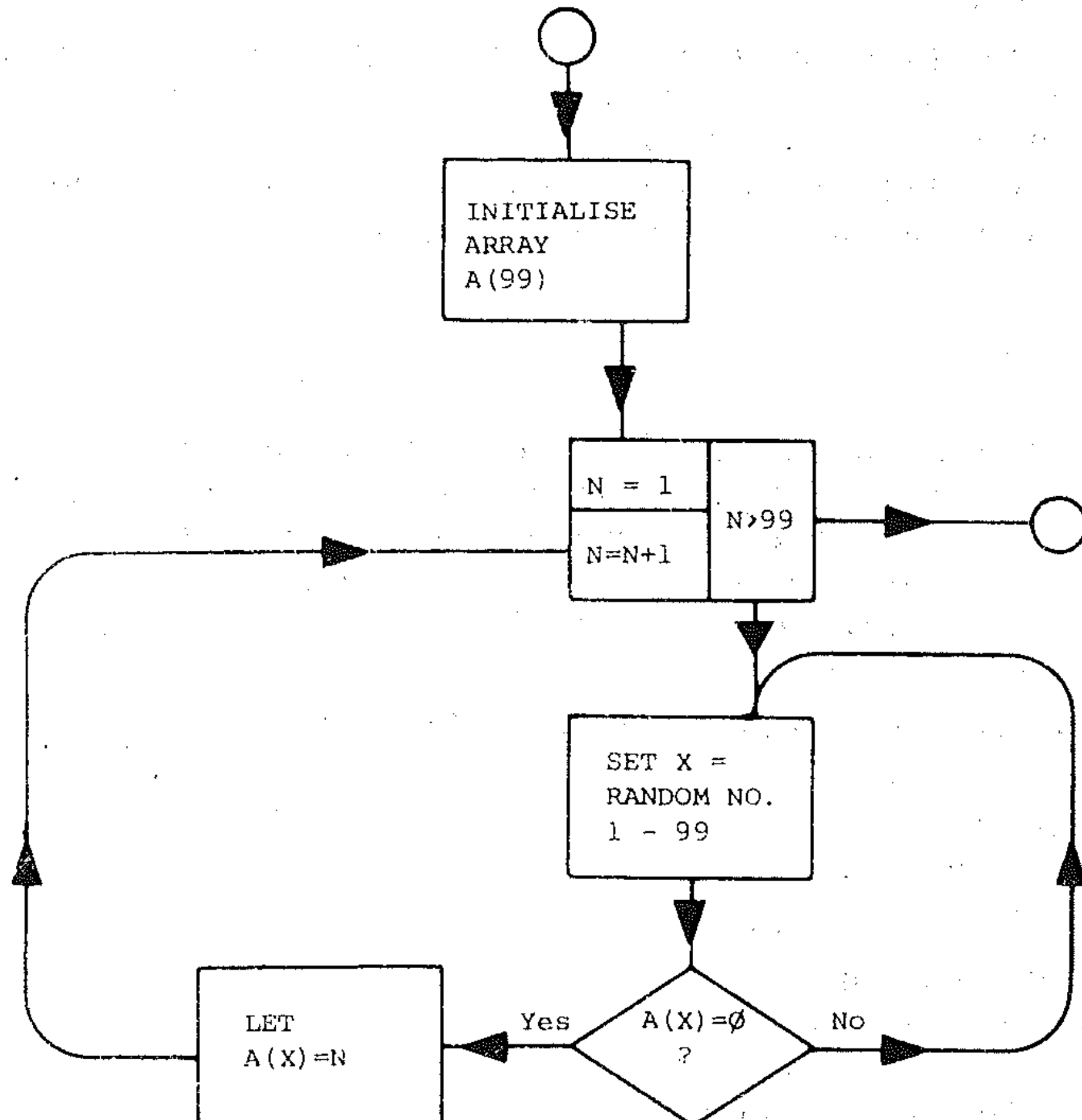
Procedure will be as follows:

- 1.1 Write preliminary instructions
- 1.2 Set up caller's numbers
- 1.3 Set up players' numbers
- 1.4 Display players' numbers

- 2.1 Display caller's numbers (one at a time)
- 2.2 Allow interruption by player
- 3.1 Display players' numbers
- 3.2 Display numbers called
- 3.3 Check winning card
- 1.1 These are the *instructions* needed to start the game. At each stage in the program it is essential to give clear directions to the user on how to proceed.
- 1.2 The *caller's numbers* require a random list of the integers 1 to 99 (each number occurring only once) which will be put in array A(99).
- 1.3 The *players' cards* require 4 sets (*cards*) of 15 random integers in the range 1 to 99 (ie. different numbers). The cards should have an ordered list of numbers and each list is a different set of numbers. These will be put into arrays Q(15), R(15), S(15) and T(15).
- 1.4 Display the *players' cards* on the screen and allow time for the players to take down the numbers.
- 2.1 Display the *caller's numbers* one at a time on the screen in large form.
- 2.2 Allow the display to be *interrupted* when a player calls 'house' (ie. thinks that all the numbers on his card have been displayed).
- 3.1 Repeats 1.4 for players to check their numbers.
- 3.2 The *numbers called* (i.e. those actually displayed in 2.1) are sorted into numerical order (and put into array P(99) which contains between 15 and 99 numbers) and displayed on screen.
- 3.3 The *winning card* is selected and the numbers on this card are put into the array V(15). These numbers are then compared with the numbers called in P(99) to check that they are correct.

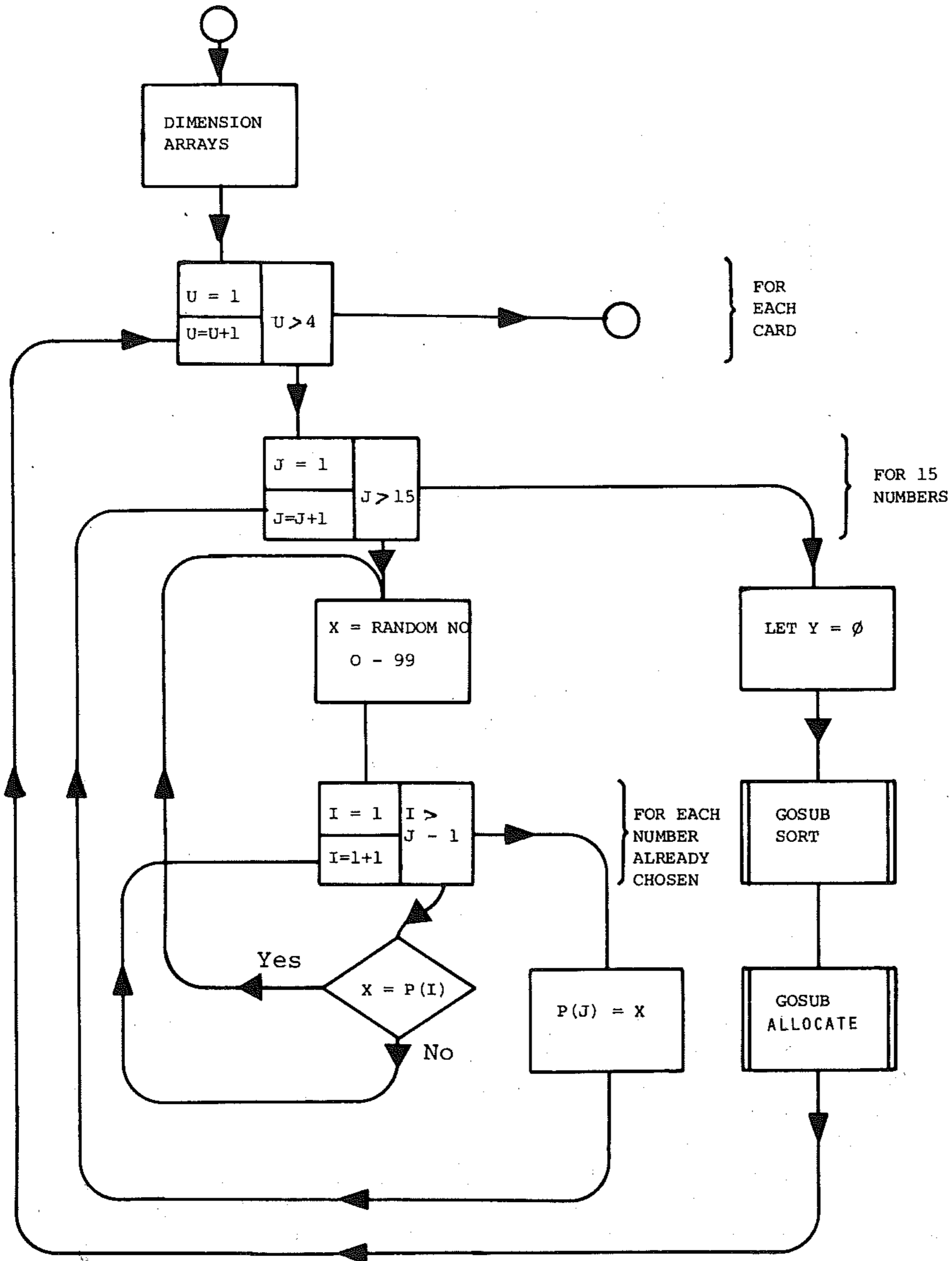
Algorithm description

- 1.1 This section gives the minimum instructions required to play. You may want to key in fuller details of the game of BINGO (Lines 350 to 490 in program).
- 1.2 Set up the caller's numbers. Flowchart:



This routine starts at line 1350, clearing the screen. The above routine is contained within the caller's numbers subroutine (lines 1350 to 1420), but note that this could be a subprogram sequence as part of the main program, as it is only executed once.

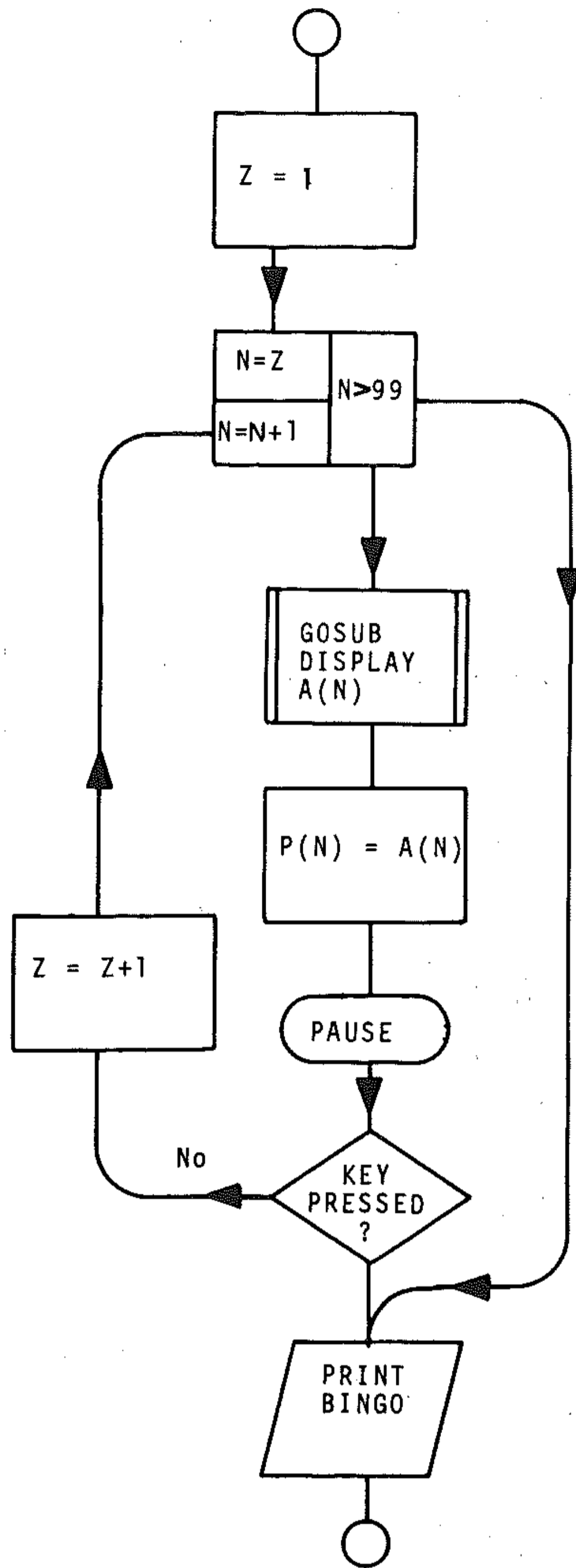
1.3 *Setting up (the BINGO cards).* Flowchart:



In the program this routine begins at line 700. Note array P(99), which is used in the sorting routine (although containing only 15 numbers) and is then used again later for the caller's numbers. Lines 720 - 800 carry out the selection of the 15 random numbers. Subroutine 1470 - 1600 is a shell sort as described in Section T (a fast sort is needed as at a later stage we are sorting nearly 100 numbers.) Subroutine 890 - 970 puts the numbers into the appropriate array.

1.4 This subroutine (lines 1020 - 1130) displays the numbers on the cards in a tabular form.

2.1 *Display caller's numbers and interrupt if 'House' is called.* Flowchart:



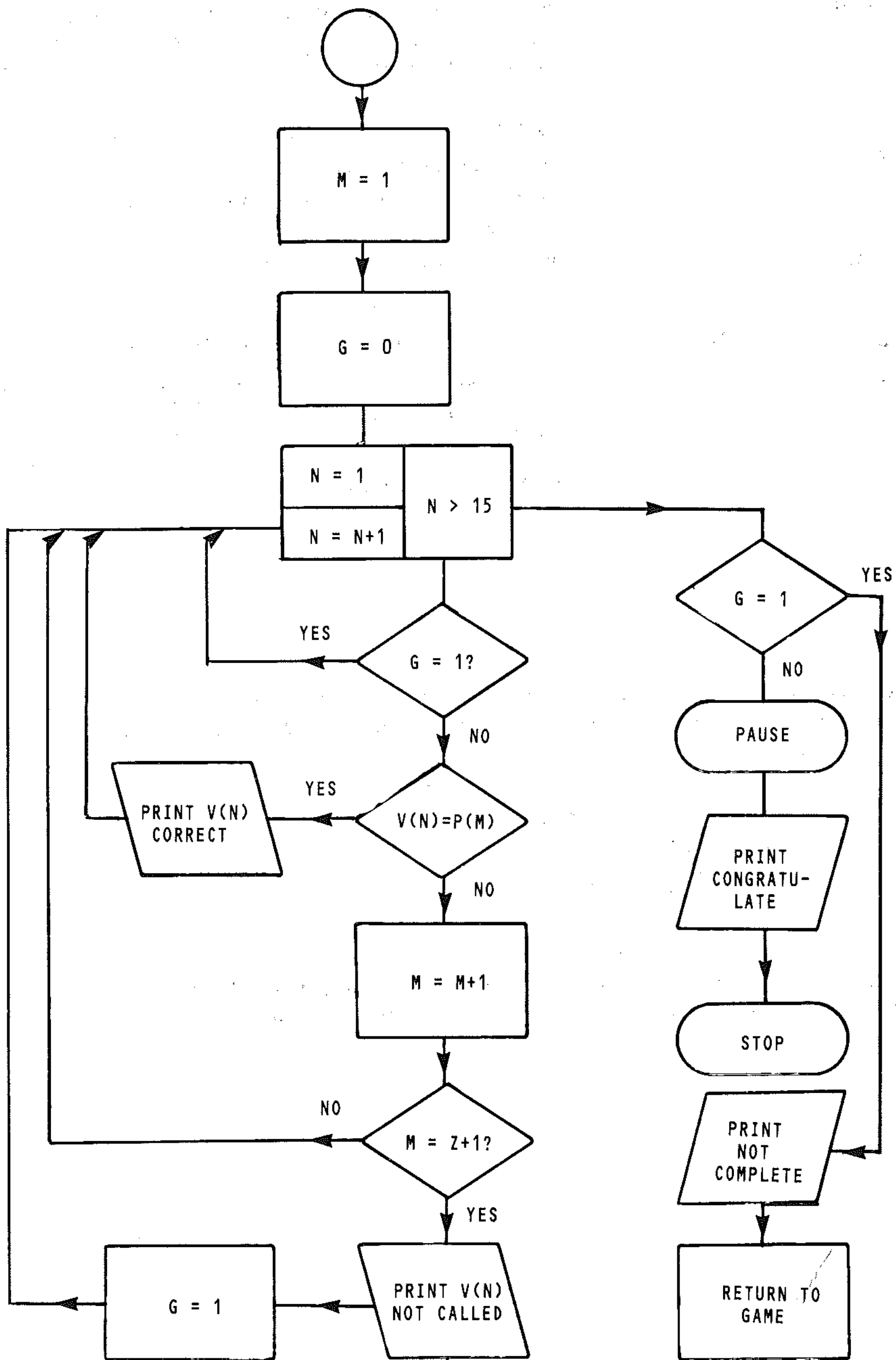
This subroutine takes numbers generated and displays them 16 times their normal size and places them in an array P. Variable Z counts the number of numbers called. At the end of this subroutine the numbers called have been put into the array P (which will contain Z numbers).

- 3.1 This repeats step 1.4.
- 3.2 This routine (lines 540 – 650) calls a sort subroutine at line 1470 which sorts the list P(Z) into numerical order and prints it out on the screen.
- 3.3 This subroutine (lines 1650 – 1960) has two parts.
 - a Select the winning card and arrange for the winning list of numbers to be put in the array V(15) (lines 1700 – 1720).
 - b Check the numbers on the winning card. It is necessary to search the ordered list P containing Z numbers for the numbers in the ordered list V containing 15 numbers. If any number is missing the card is not complete. If all numbers are present then CONGRATULATIONS is printed.

The quickest search method is as follows:

- Search for V(1) in list P until it is found, say P(12).
- Search for V(2) in list P beginning at P(13) etc.

The search will end as soon as a number in list V is not in list P but will continue for all 15 numbers in list if they appear in list P. The flowchart for this search is given below:



Data table

The following variables are used:

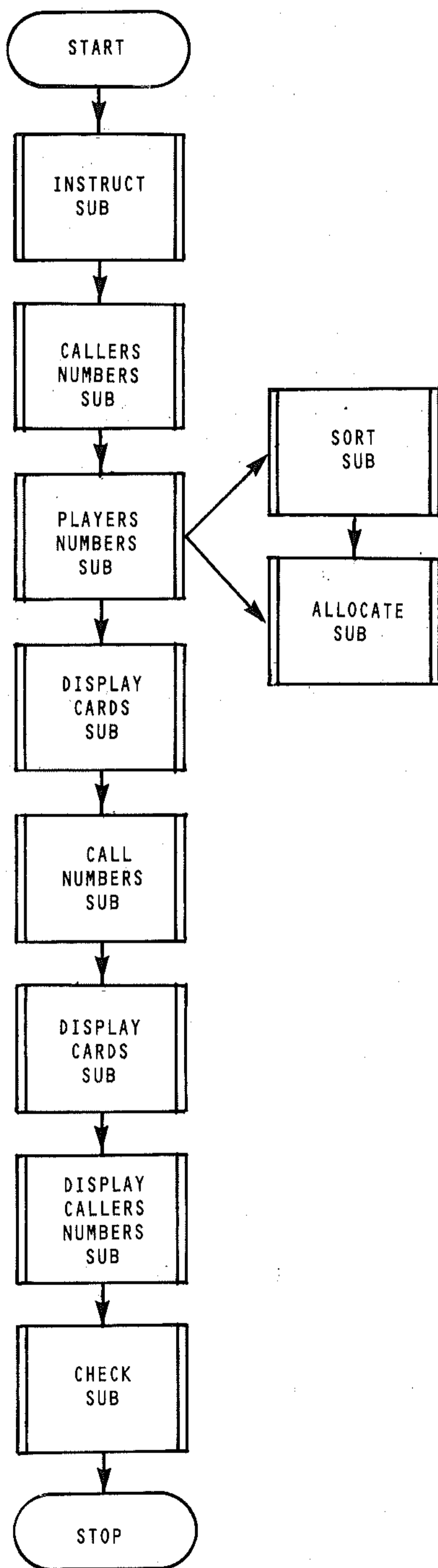
Q(15), R(15), S(15), T(15)	are the arrays containing the players' numbers.
A(99)	is the array containing the numbers for the caller.
P(99)	is the array containing the numbers actually called, and is also used as a temporary storage in setting up the players' numbers and in the sorting routine.
X	is a random number between 1 and 99.
Z	is a counter for numbers called.
Y	is the number of elements present in an array to be sorted
V(15)	is an array containing 'winning' list numbers
G	is a flag used in subroutine CHECK.
U, J, I, K, A, N	are used as loop variables in the program.
S	is the search position pointer in subroutine SORT.
B	is used as an array index pointer in subroutine SORT.
M	is the code for array P in the 'check numbers' section.

Comments: If you wished to play with your own Bingo cards and let the computer be the caller only, then you could delete these modules of the program:

- 1.3 Set up players' numbers.
- 1.4 Display players' numbers.
- 3.1 Display players' numbers.

These latter two modules are the same subroutine. You could also revise the program and include, as an option in the user instructions routine at the beginning, the choice of the two different modes of play.

Here are the complete flowchart and the program for BINGO:



```

5 REM *** B I N G O ***
7 X$="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
8 Y$="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
10 DIM P(99),Q(15),R(15),S(15),T(15),V(15)
20 REM *** 1.1 INSTRUCTION ***
30 GOSUB 330
40 REM *** 1.2 CALLERS NUMBERS ***
50 GOSUB 1350
60 REM *** 1.3 PLAYERS NUMBERS ***
70 GOSUB 700
80 GOSUB 890
90 REM *** 1.4 DISPLAY CARDS ***
100 GOSUB 1020
110 REM *** 2.1 CALL NUMBERS ***
120 REM *** 2.2 INTERRUPT ***
125 REM CALL FIRST CARD
130 Z=1
140 GOSUB 1180
150 PRINT ""
160 PRINT LEFT$(Y$,14);LEFT$(X$,14);"***
*****"
170 PRINT TAB(14);"***BINGO***"
180 PRINT TAB(14);"*****"
190 PRINT:PRINT TAB(7);"CHECK YOUR NUMBE
RS"
200 FOR I=1 TO 500 :NEXT I
210 REM *** 3.1 DISPLAY CARDS AGAIN ***"

220 GOSUB 1020
230 REM *** 3.2 DISPLAY NUMS. CALLED ***
"
240 GOSUB 540
250 REM ***3.3 CHECK WINNING CARD***"
260 GOSUB 1650
270 GET K$ :IF K$="" THEN 270
280 IF G=1 THEN 140
290 END
300 REM *** E N D M A I N ***
310 REM ..
320 REM....
330 REM *** INSTRUCTION SUB ***
340 REM ...
350 PRINT ""
370 PRINT TAB(2);"*****INSTRUCTIONS**
*****":PRINT
380 PRINT TAB(2);"FOUR BINGO CARDS ARE G
IVEN"
400 PRINT TAB(2);"LISTS Q,R,S AND T"

```

```

410 PRINT TAB(2); "CHOOSE YOUR CARD AND C
OPY"
420 PRINT TAB(2); "NUMBERS DOWN WHEN SHOW
N"
430 PRINT TAB(2); "*****"
****"
440 PRINT TAB(2); "A$ NUMS. CALLED CROSS
THEM"
450 PRINT TAB(2); "DEF YOUR LIST. THE WIN
NING"
460 PRINT TAB(2); "CARD IS THE FIRST COMP
LETED"
470 PRINT TAB(2) "WHEN READY PRESS A KEY"

480 GET E$: IF E$="" THEN 480
490 RETURN
500 REM..
510 REM...
520 REM..
530 REM..
540 REM *** DISPLAY CALLERS ***
545 REM *** NUMBERS SUB ***
550 REM ..
560 PRINT " "
565 PRINT LEFT$(Y$,6); LEFT$(X$,4); "CALLE
RS NUMBERS"
570 PRINT TAB(6); "*****"
580 PRINT , , , : PRINT TAB(10); "****WAIT**
**": Y=Z
590 GOSUB 1470
600 FOR N=1 TO Z: PRINT P(N); TAB(5);
610 NEXT N: PRINT
620 PRINT TAB(2); "PRESS A KEY TO CHECK R
ESULT"
630 GET G$: IF G$="" THEN 630
640 PRINT " "
650 RETURN
660 REM..
670 REM..
680 REM...
690 REM ...
700 REM **PLAYERS NUMBERS SUB **
710 REM...
720 FOR U=1 TO 4
730 FOR J=1 TO 15
740 X=INT(RND(1)*99)
750 S=0: REM FLAG OFF
760 FOR I=1 TO J-1
770 IF X=P(I) THEN S=1
780 NEXT I

```

```

790 IF S=1 THEN 740
800 P(J)=X: NEXT J
810 Y=15: GOSUB 1470
820 GOSUB 890
830 NEXT U
840 RETURN
850 REM..
860 REM..
870 REM...
880 REM ..
890 REM ***   ALLOCATE SUB   ***
900 REM..
910 FOR J=1 TO 15
920 IF U=1 THEN Q(J)=P(J)
930 IF U=2 THEN R(J)=P(J)
940 IF U=3 THEN S(J)=P(J)
950 IF U=4 THEN T(J)=P(J)
960 NEXT J
970 RETURN
980 REM..
990 REM..
1000 REM..
1010 REM..
1020 REM ***   DISPLAY CARDS SUB   ***
1030 REM..
1040 PRINT "☐": PRINT: PRINT
1050 PRINT TAB(2); "HAND Q"; TAB(10); "HAND
R"; TAB(18); "HAND S"; TAB(26); "HAND T"
1060 FOR J=1 TO 15
1070 PRINT TAB(5); Q(J); TAB(13); R(J); TAB(
21); S(J); TAB(29); T(J)
1080 FOR I=1 TO 500: NEXT I
1090 NEXT J
1100 PRINT "PRESS A KEY TO CONTINUE"
1110 GET D$: IF D$="" THEN 1110
1120 PRINT "☐"
1130 RETURN
1140 REM..
1150 REM..
1160 REM..
1170 REM..
1180 REM ***   CALL NUMBERS SUB   ***
1190 REM..
1210 PRINT "PRESS ANY KEY TO STOP WHEN Y
OUR CARD IS COMPLETE"
1220 PRINT "☐" LEFT$(Y$, 15); LEFT$(X$, 11);
A(Z)
1240 P(Z)=A(Z)
1250 FOR I=1 TO 2000: NEXT I
1260 PRINT "☐" LEFT$(Y$, 14); LEFT$(X$, 11);

```



```

"      "
1270 GET C$: IF C$<>" " THEN 1300
1275 Z=Z+1
1280 IF Z=100 THEN PRINT "ALL NUMBERS CA
LLED":GOTO 1300
1290 GOTO 1220
1300 RETURN
1310 REM..
1320 REM..
1330 REM..
1340 REM..
1350 REM *** CALLERS NUMBERS SUB ***
1360 DIM A(99)
1370 PRINT " ":PRINT LEFT$(Y$,14);LEFT$(
X$,10);" ** WAIT **"
1380 FOR N=1 TO 99
1390 X=INT(RND(1)*99)
1400 IF A(X)<>0 THEN 1390
1410 A(X)=N:NEXT N
1420 RETURN
1430 REM..
1440 REM..
1450 REM..
1460 REM ...
1470 REM**** SORT SUB ****
1480 REM..
1490 S=Y
1500 S=INT(S/2): IF S>=1 THEN 1520
1510 GOTO 1600
1520 FOR K=1 TO S
1530 FOR A=K TO Y-S STEP K
1540 B=A:T=P(A+S)
1550 IF T>=P(B) THEN 1580
1560 P(B+S)=P(B):B=B-S
1570 IF B>=1 THEN 1550
1580 P(B+S)=T:NEXT A,K
1590 GOTO 1500
1600 RETURN
1610 REM..
1620 REM..
1630 REM..
1640 REM..
1650 REM ***CHECK WINNING CARD***
1660 REM..
1670 PRINT " ":PRINT
1680 PRINT TAB(2);" TYPE NAME OF WINNING
CARDS";
1690 INPUT A$: IF A$<>"Q" AND A$<>"R" AND
A$<>"S" AND A$<>"T" THEN 1680
1695 PRINT " "

```

```

1700 FOR N=1 TO 15
1710 IF A$="Q" THEN V(N)=Q(N)
1714 IF A$="R" THEN V(N)=R(N)
1718 IF A$="S" THEN V(N)=S(N)
1719 IF A$="T" THEN V(N)=T(N)
1720 NEXT N
1730 M=1 :REM CALLERS ARRAY INDEX
1740 G=0:PRINT
1750 FOR N=1 TO 15:REM N CARD ARRAY INDEX
1760 IF G=1 THEN 1810
1762 IF V(N)=P(M) THEN PRINT V(N); "CORRECT"
1770 IF V(N)=F(M) THEN 1810
1780 M=M+1: IF M=Z+1 THEN PRINT V(N); "NOT CALLED"
1790 IF M=Z+1 THEN G=1:REM INCORRECT FLAG
1800 GOTO 1760
1810 NEXT N
1820 IF G=1 THEN PRINT "CARD "; A$; " NOT COMPLETED TRY AGAIN":RETURN
1830 GET A1$: IFA1$="" THEN 1830
1840 PRINT ""
1850 PRINT LEFT$(Y$, 8); LEFT$(X$, 5); "*****
*****"
1860 PRINT TAB(8); "**CONGRATULATIONS**"
1870 PRINT TAB(8); "*****"
1880 PRINT TAB(8); "*****BINGO*****"
1890 PRINT TAB(8); "*****"
1900 PRINT TAB(8); "****CARD "; A$; " WINS****"
1910 PRINT TAB(8); "*****"
1950 PRINT :PRINT "GAME ENDED USE RUN TO RESTART"
1960 RETURN

```

"REF. INDEX"

Problem: in a laboratory experiment with a spectrometer, a series of measurements are made of the prism angle A and the angle of minimum deviation D for various prisms. The refractive index of each is then determined using the formula:

$$N = \frac{\sin\left(\frac{A+D}{2}\right)}{\sin\left(\frac{A}{2}\right)}$$

A table of results is required, and an average of the refractive index results for each prism. Six prisms and four measurements for each prism are to be allowed for.

Research the problem: since we are using up to six prisms and each one can have four readings it is convenient to use nested loops and two-dimensional arrays to store the input. In this way $A(3,4)$ can, for example, represent the third prism, fourth reading of prism angle. Lists will be used to refer to the prisms and the quantities associated with them so that, for example, $Z(3)$ can represent the average refractive index of the third prism.

Angles will be input in degrees. Since the formula for the calculation requires the use of the Sine function SIN , the program will need to convert to radians. This will have to be done before the refractive index is calculated. A simple input check can be provided in a subroutine. Zero entries in the input loops will signify end of prisms or end of readings.

Outline procedure:

- 1 Input: data
- 2 Calculate: refractive indices and averages
- 3 Output: results in suitable tabular form

Detailed procedure:

Input

- 1.1 Arrays and Lists
- 1.2 FOR each Prism (1 to 6)
- 1.3 Set Z as zero
- 1.4 Input Prism number
- 1.5 IF Prism = 0, GOTO 3.1
- 1.6 FOR each Reading (1 to 4)
- 1.7 Input Prism Angle
- 1.8 If Angle = 0, GOTO NEXT Prism
- 1.9 Input Minimum Deviation

Processing

- 2.1 Convert angle to radians
- 2.2 Convert minimum deviation to radians
- 2.3 Calculate Refractive Index
- 2.4 Round to 3 decimal places
- 2.5 Add Refractive Index to Z
- 2.6 NEXT Reading
- 2.7 Average $R.I = Z$ divided by number of readings
- 2.8 Let Average $R.I = Z(N)$
- 2.9 NEXT Prism

Output

- 3.1 Print Headings
- 3.2 FOR Each Prism
- 3.3 FOR Each Reading
- 3.4 If Reading no 0, print Prism, Number, Angle, Minimum Deviation, Refractive Index
- 3.5 NEXT Reading
- 3.6 NEXT Prism
- 3.7 Print Headings
- 3.8 FOR Each Prism
- 3.9 Print Prism Number, Refractive Index
- 3.10 NEXT Prism

Correct

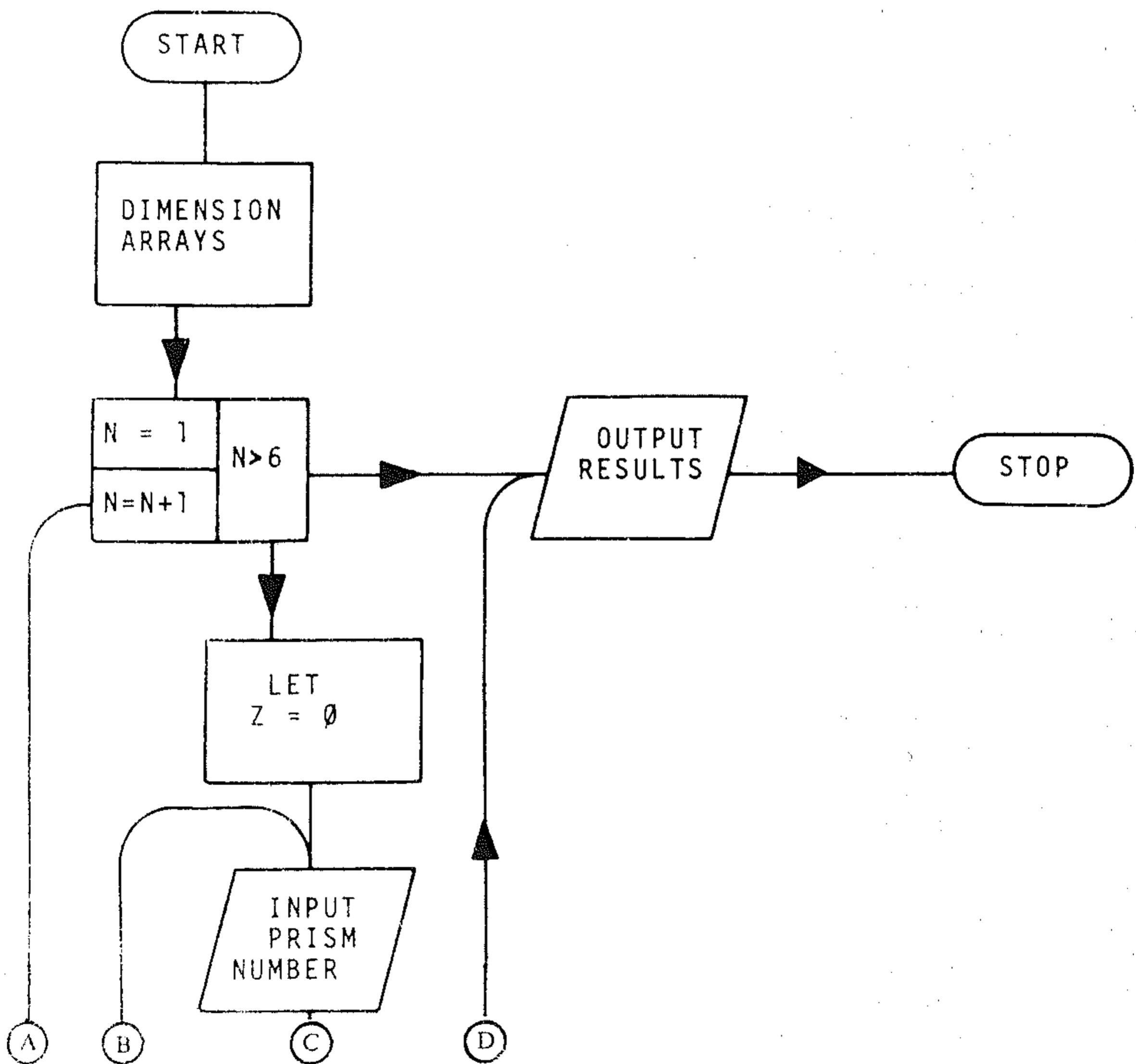
- 4.1 Input null string if error, C if correct Error subroutine
- 4.2 Return

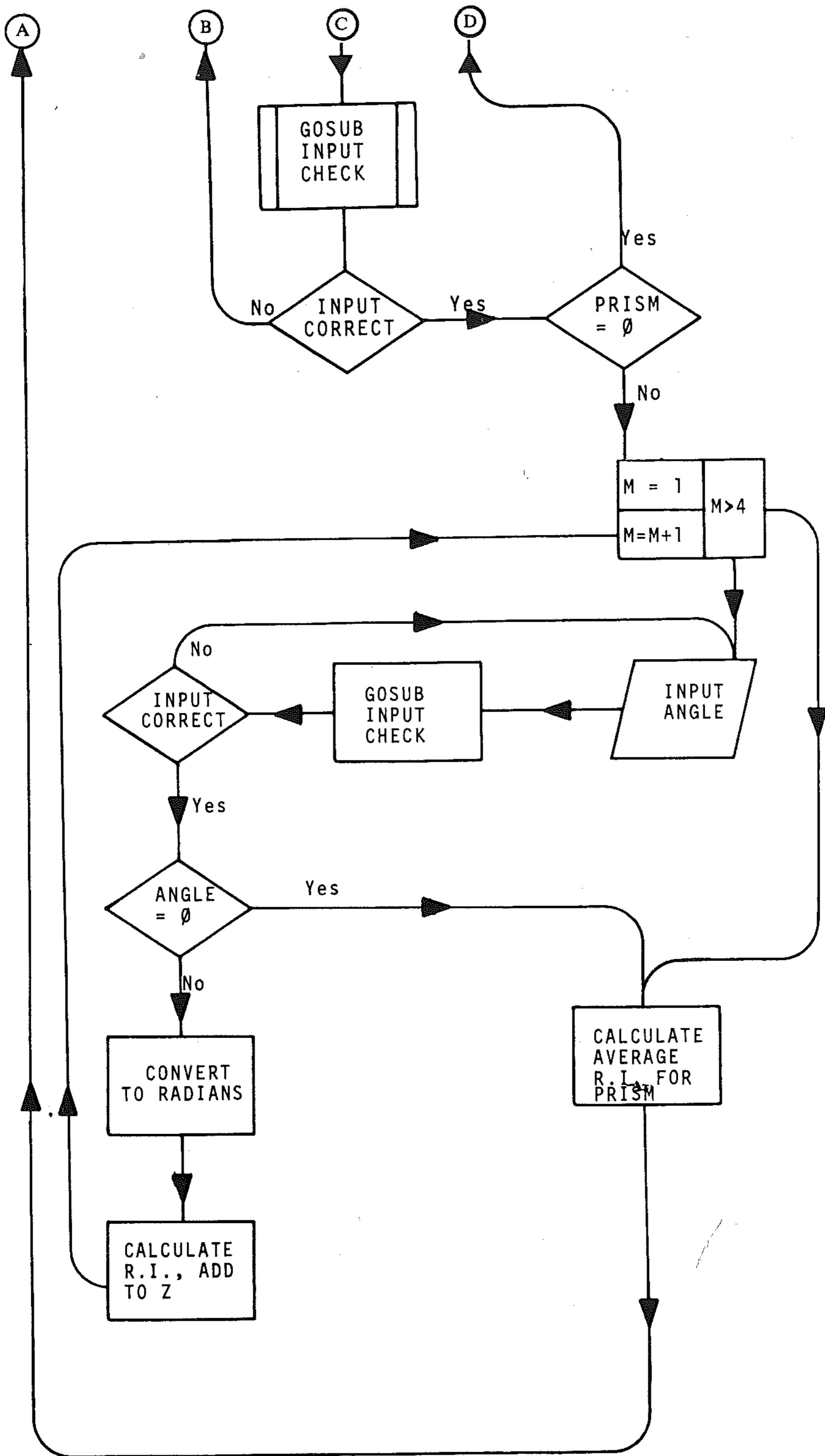
Note: subroutine called after each input. All entries can be re-input if incorrect. See program listing.

Data table

- P(6) Prism number
- A(6,4) Four Angles of measurement for each of 6 prisms
- D(6,4) Minimum Deviations, for each value of A(6,4)
- N(6,4) Results of refractive index calculation for each experiment, derived from the values stored in arrays A and D
- Z Variable to store totals of minimum deviations
- Z(6) Average refractive index for each prism, from the average of the four values stored in array N.
- N,M Used as loop variables for input
- X,Y Used as loop variables for printout
- A\$ Used as input for error check subroutine. Null string if incorrect, C if correct.

Flowchart:





```

20 REM ** REF. INDEX **
30 PRINT "REFRACTIVE INDEX", "USIN
G SPECTROMETER"
40 PRINT "ALLOWS UP TO SIX PRISMS AND F
OUR SETS OF RADING FOR EACH"
50 REM ** DIMENSION ARRAYS **
60 DIM Z(4), P(6), A(4,6), D(6,4), N(6,4)
70 REM.
80 REM.
90 REM ** INPUT DATA LOOP **
100 FOR N=1 TO 6:Z=0
110 INPUT "ENTER PRISM NUMBER INPUT 0 TO
FINISH";P(N)
120 PRINT "PRISM NUMBER";P(N)
130 GOSUB 640
140 IF A$<>"C" THEN 110
145 IF P(N)=0 THEN 460
150 REM...
160 REM..
170 REM ** START READINGS LOOP **
180 FOR M=1 TO 4
190 PRINT "INPUT PRISM ANGLE IN DEGREES
INPUT 0 TO FINISH"
200 INPUT A(N,M)
210 PRINT "ANGLE OF PRISM";A(N,M)
220 GOSUB 640
230 IF A$<>"C" THEN 190
235 IF A(N,M)=0 THEN 390
240 INPUT "ENTER MIN.DEV. IN DEGREES";D(
N,M)
250 PRINT "MIN.DEV.:";D(N,M)
260 GOSUB 640
270 IF A$<>"C" THEN 190
280 REM ** CONVERT DEGREES TO **
      ** RADIANS **
300 A=A(N,M)* $\pi$ /180:D=D(N,M)* $\pi$ /180
310 REM ** REFRACTIVE INDEX **
320 N(N,M)=(SIN((A+D)/2))/SIN(A/2)
330 N(N,M)=INT(1000*N(N,M)+.5)/1000:Z=Z+
N(N,M)
340 NEXT M
350 REM ** END READING LOOP **
360 REM ** AVERAGE TO 2 DEC. PLACES **=
380 REM..
390 Z=Z/(M-1):Z(N)=INT(100*Z+.5)/100:NEX
T N
400 REM ** END INPUT LOOP **
420 REM
430 REM..
440 REM ** PRINT OUT **

```

```

450 REM..
460 PRINT "PRISM * ANGLE * MIN.DEV*RF. IN
. ****"
470 FOR Y=1 TO N-1
480 FOR X=1 TO 4
490 IF A(X,Y)=0 THEN 510
500 PRINT P(Y);TAB(6);"*";A(X,Y);TAB(14)
; "*" ; D(X,Y);TAB(24); "*" ; N(X,Y)
510 NEXT X,Y
520 PRINT "*****"
530 PRINT "AVERAGE REFRACTIVE INDICES"
540 PRINT "*****"
545 REM *** NUMBERS SUB ***
550 PRINT "PRISM REF. IND. "
560 FOR X=1 TO N-1
570 PRINT P(X),Z(X)
580 NEXT X
590 PRINT "*****"
600 GOTO 700
610 REM..
640 REM ** CORRECT ERROR SUB **
650 PRINT "INPUT C IF CORRECT OTHERWISE
PRESS ENTER"
660 INPUT A#
670 PRINT " "
680 RETURN
690 REM ..
695 REM ** END SUBROUTINE **
700 END

```

Sample printout:

```

PRISM * ANGLE * MIN.DEV * RF.IN.
*****
1 * 59.8 * 40.5 * 1.54
2 * 60.1 * 40.2 * 1.533
2 * 61.2 * 45.3 * 1.574
2 * 62 * 45.1 * 1.562
2 * 62.5 * 45.4 * 1.558
*****

```

```

AVERAGE REFRACTIVE INDICES
*****
PRISM REF.IND.
1 1.54
2 1.56
*****

```

Comments: Note the importance of a check subroutine in this type of multiple-entry program. It gives the user the chance to verify that the input data is correct. It might have been better to have written a program to deal with any number of prisms and any number of readings. Consider what changes this would make to the program.

There are many other physics experiments which may be treated in a similar way. For example:

- The value of the gravitational constant, g , by simple pendulum, using the time period equation

$$T = 2\pi\sqrt{l/g}$$

for experiments with pendula of various lengths l to determine the average value of g resulting.

- The determination of the velocity of sound, using a resonance tube to find the 1st and 2nd resonance lengths (L_1 and L_2 respectively) and using the equation

$$V = 2f(L_2 - L_1)$$

where f = frequency of the sound, and V the velocity.

"SERIES"

Problem: to sum the series that gives the value of e raised to the power x . This is:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$$

for any value of x , with an accuracy of 1 part in 100,000 (ie. 10^{-5} , .00001).

Research the problem: this is the EXP function on the computer, evaluated as a series. The resulting value for example, of e^3 , would be approximately the value of EXP(3) if you used it on the computer. This is a *convergent* series; the value of each term gets smaller. For example, if we take

$$2, \frac{x^2}{2!} \text{ is } \frac{4}{2 \times 1} = 2 \text{ while } \frac{x^3}{3!} = \frac{8}{3 \times 2 \times 1} = 1.333\dots$$

At a certain point, the effect of adding the value of another term is to increase the sum by less than the accuracy required. The summing is then finished.

Procedure: this requires a procedure which allows any term in the series to be calculated from the previous term. This is the basic algorithm for summing many series.

At this stage you should consider whether this is a specific problem or whether it could be extended to deal with other, similar series-sum problems. It is important to make this decision before the program is written, as it is time-consuming to modify a program at a later stage. The answer is yes. We can then restate the problem as: a program is required which will sum a convergent series to any desired accuracy (subject to the limitations of the computer's arithmetic), provided that any term may be expressed as a function of the previous term, with the information needed being the first term and the common ratio. The common ratio is the equation that enables us to calculate any term from the previous one.

For your problem; take the exponential series:

$$e^x = 1 + x + \frac{x^2}{2!} + \dots + \frac{x^{n-1}}{(n-1)!} + \dots$$

nth term

In this series the first term is 1. The common ratio =

$\frac{n^{\text{th}} \text{ term}}{(n-1)^{\text{th}} \text{ term}}$. The n^{th} term is $\frac{x^{n-1}}{(n-1)!}$ and the
 $(n-1)^{\text{th}}$ term is $\frac{x^{n-2}}{(n-2)!}$. The ratio is $\frac{x^{n-1}}{(n-1)!} \cdot \frac{(n-2)!}{x^{n-2}} = \frac{x}{n-1}$.

An accuracy is required of one part in 100,000 (10^{-5}). When the effect of adding another term increases the sum by less than this, the program should stop processing and output the result.

Outline procedure

- 1 Input necessary information
- 2 Sum series term by term
- 3 Compare sum with previous value
- 4 Print result when sums differ by less than required value

Detailed procedure

Input

- 1.1 Common ratio – may be easiest to have a replaceable line in program.
- 1.2 First term – easily input
- 1.3 Accuracy – easily input
- 1.4 Value of X – easily input

Sum

- 2.1 Initialisation – set 1st term equal to given value which in turn is the sum of the series at this stage.
- 2.2 Next term – may be calculated by multiplying first term by common ratio.
- 2.3 Sum – may be calculated by adding this term to previous sum.

Comparison

- 3.1 Compare this sum with previous value, then EITHER go to 4.1 if the difference is *less* than that required OR go back to 2.2 if difference is *more* than that required.

Output results

- 4.1 Name of series
- 4.2 Accuracy
- 4.3 First term
- 4.4 Value of X
- 4.5 Sum of series
- 4.6 Number of terms

Variables table

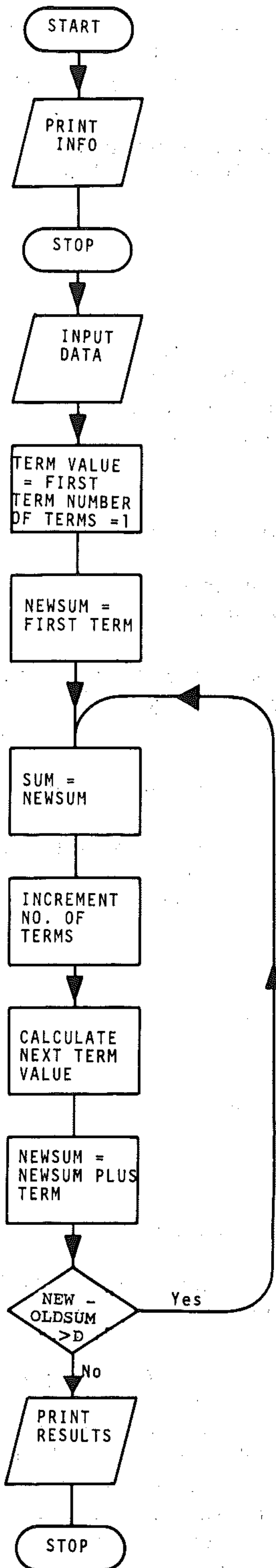
Input

- A\$ Name of Series
A Value of first term in series
X Value of X
D Accuracy required (difference between terms, such that program terminates when difference smaller than this).

Processing and output

- T Value of current term being processed
N Number of terms
S1 Value of Sum of series of N-1 terms
S Value of Sum of series to N terms
S1-S Difference between the sums of the series to the Nth and (N-1)th terms, checked against value of D
T=T*X/(N-1) Calculates value of next term in series from current term.

The common ratio is set in line 260. This line must be changed for different series.



```

10 PRINT ""
14 X$=""
18 Y$=""
20 PRINT TAB(14);  "**SERIES  **"
24 PRINT"THIS PROGRAM SUMS ANY SERIES WH
ICH IS CONVERGENT AND WHERE ANY TERM MAY

40 INPUT "ENTER NAME OF THE SERIES";N$
50 PRINT ""
60 INPUT "ENTER THE VALUE OF FIRST TERM"
;TM
70 INPUT "ENTER VALUE OF X";X
80 INPUT "ENTER ACCURACY REQUIRED";AC
90 T=TM
100 N=1
110 S1=TM
120 S=S1
130 REM *** CALCULATE NEXT TERM ***
140 N=N+1:T=T*X/(N-1)
150 REM ** CALCULATE NEW SUM **
160 S1=S1+T
170 REM ** COMPARE S AND S1 **
180 IF ABS(S1-S)>AC THEN 120
185 PRINT""SUM OF ";N$;" SERIES"
190 PRINT "*****"
*****"
195 PRINT"TO AN ACCURACY OF";AC:PRINT
200 PRINT"FIRST TERM=";TM;"VALUE OF X=";
X
210 PRINT "SUM"S1;"NO. OF TERMS";N

```

Sample printout:

```

SUM OF EXPONENTIAL SERIES
*****

TO AN ACCURACY OF .00001
FIRST TERM = 1
VALUE OF X = 1
SUM 2.7182815
NO. OF TERMS 10

```

Now consider how you might improve upon this program. (Ideally, this should have been considered at the planning stage, not *after* coding.) We should perhaps include a subroutine to correct the sum to the appropriate number of decimal places or routines to enable the user to check that input data is correct.

"ELEMENT"

Problem: to calculate for a chemical compound:

- a percentage elemental composition and molecular weight, or
- b molecular formula

from inputs of, for a), number of atoms each element and for b) percentages of each element. These are complementary calculations.

Research problem: A program is required that performs two separate operations. The common requirements will be the element names, symbols and molecular weights. These will be input and stored as data in arrays on the computer. This will require an input routine that is not used every time the program is run, and could be edited out. The elements involved should be capable of being changed to facilitate different analyses, and this must be allowed for in the program. This is an advantage over the alternatives of either defining all data with LET statements or the use of DATA and READ.

The program must be split into two processing sections. The first of these (percentage elemental composition and molecular weight) requires an input for the number of atoms of each element. A loop can be used for this, using the loop variable to access the stored element names. Molecular weight equals the number of atoms of each compound multiplied by the atomic weight, and this can be calculated within the input loop. Molecular formulae are then derived from the atomic symbol, plus the number of atoms, for each of the elements concerned.

Percentage composition for each element is the number of atoms of the element, times the atomic weight of the element, divided by the molecular weight.

The second section of the program requires an input loop for the percentage of each element. The proportion of atoms of each element will then be the percentage divided by atomic weight. This can be calculated within the input loop. To calculate the molecular formula, it is necessary to know the minimum (to give the smallest number of atoms of any of the elements) elemental proportion. This can be checked through another loop. The number of atoms of each element is then the proportion of each element divided by the smallest proportion. This could be rounded to integers, but will be done to two decimal places because the actual molecular formula may be a multiple of the derived formula. This possibility should also be indicated to the user. Zero inputs will be required when elements do not occur in the compound concerned.

For a common organic analysis the following data will be stored via the input routine:

HYDROGEN	H	1.008
CARBON	C	12.01
NITROGEN	N	14.008
OXYGEN	O	16.00
PHOSPHOROUS	P	30.98
SULPHUR	S	32.06
CHLORINE	CL	35.457
BROMINE	BR	79.916

The data entry module is keyed in and the data input as above. The procedure for the program is given below.

Detailed procedure:

- 1 Data entry module
 - 1.1 Dimension arrays for data: element names, element symbols, atomic weights
 - 1.2 FOR each input (1 to 8)
 - 1.3 Input name, symbol, atomic weight
 - 1.4 GOTO subroutine to justify atomic weight (5 below)
 - 1.5 NEXT input

- 2 *Menu*
- 2.1 Print instructions and menu
- 2.2 If second calculation required, GOTO 4
- 2.3 If first calculation required, proceed to 3

- 3 *Percentage element calculation*
- 3.1 Input:
- 3.1.1 Reference for compound
- 3.1.2 Initialise arrays, molecular weight variable
- 3.1.3 FOR each element (1 to 8)
- 3.1.4 Input number of atoms present
- 3.1.5 Calculate total molecular weight (molecular weight plus (number of atoms * atomic weight))
- 3.1.6 Next element
- 3.2 Processing/output:
- 3.2.1 Print reference, molecular weight
- 3.2.2 FOR each element (1 to 8)
- 3.2.3 Print symbol, number of atoms
- 3.2.4 NEXT element
- 3.2.5 FOR each element (1 to 8)
- 3.2.6 If no atoms present, GOTO 3.2.10
- 3.2.7 Calculate percentage as (number of items atomic weight) divided by molecular weight of compound, times 100
- 3.2.8 GOSUB (5) for rounding and justification of results
- 3.2.9 Print element symbol, percentage
- 3.2.10 NEXT element

- 4 *Molecular formula calculation*
- 4.1 Input:
- 4.1.1 Reference for compound
- 4.1.2 Initialise arrays
- 4.1.3 FOR each element (1 to 8)
- 4.1.4 Input percentage present
- 4.1.5 Calculate proportion of element as percentage present divided by atomic weight
- 4.1.6 NEXT element
- 4.2 Processing/output
- 4.2.1 Print reference
- 4.2.2 Set C = 100 for percentage calculation
- 4.2.3 FOR each element (1 to 8)
- 4.2.4 If proportion is zero, GOTO 4.2.6
- 4.2.5 If proportion less than current value of C, then let C equal proportion of element
- 4.2.6 NEXT element
- 4.2.7 FOR each element
- 4.2.8 If proportion is zero, GOTO 4.2.12
- 4.2.9 Number of atoms equals proportion divided by smallest proportion present, C
- 4.2.10 GOSUB (5) for rounding and justification to 2 decimal points.
- 4.2.11 Print symbol, number of atoms
- 4.2.12 NEXT element
- 4.2.13 Print user warning that multiples of this calculated formula may be the actual formula

- 5 *Subroutine to round and justify*
 (Input of number of decimal places (P) and number to be justified (N)
 from data defined in main program modules)
- 5.1 X\$ set to contain P zeros
 5.2 Integer value set as XN
 5.3 Decimal value set as XD
 5.4 Define X\$ as rounded number string
 5.5 X\$ returned to main modules for printing
- 6 *Auto;run.*
 An automatic RUN routine using GOTO 200 is used to prevent the user
 entering RUN accidentally after LOADING

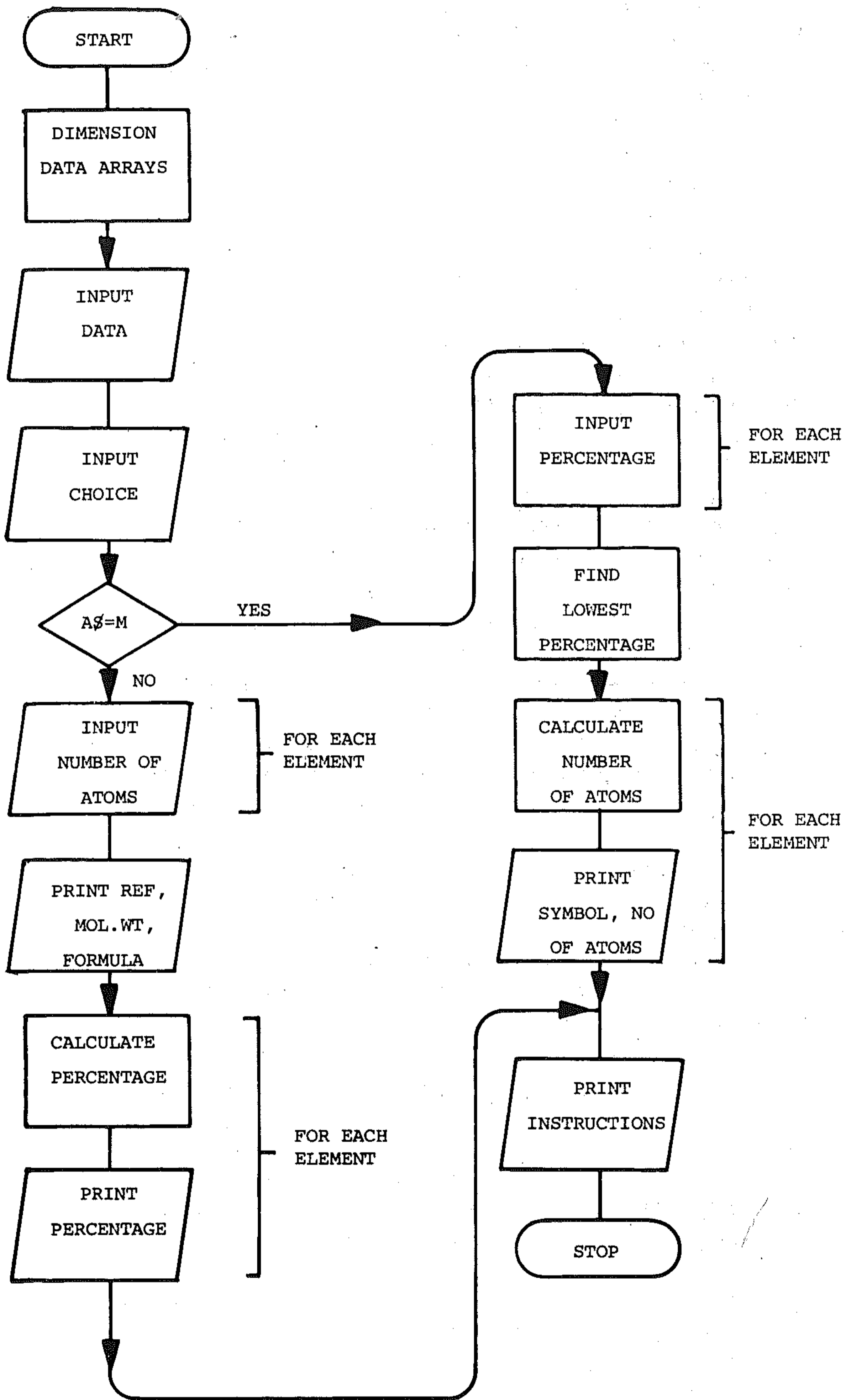
Variables table

Input and main program sections:

E\$(8,10)	Holds element names
S\$(8,2)	Holds element symbol
M(8)	Holds atomic weight
E	Loop variable in main program
X	Loop variable in input routine
A\$	Menu choice
R\$	Reference for compound
MOLWT	Molecular weight
A(8)	Holds input of number of atoms of element
P(8)	Holds input or calculated percentage of element
N(8)	Holds calculated number of atoms of element
C	Constant for calculating percentages

Round and justify subroutine:

N	Holds number for operation of subroutine
X\$	Holds string form of number returned by subroutine
XN	Holds integer value of N
XD	Holds decimal value of N
F	Loop variable in subroutine



```

5 PRINT " "
14 REM *****
15 REM 1. DATA ENTRY. ROUTINE CAN BE
16 REM DELETED WHEN ENTRY COMPLETE.
17 REM *****
20 DIM E$(8)
30 DIM S$(8)
40 DIM M(8)
55 PRINT
60 FOR X=1 TO 8
65 REM ** ELEMENT NAME **
70 INPUT "ELEMENT NAME";E$(X)
75 REM ** ELEMENT SYMBOL **
80 INPUT "ELEMENT SYMBOL";S$(X)
85 REM ** ATOMIC WEIGHT **
90 INPUT "ATOMIC MASS";M(X)
91 N=M(X):P=3:GOSUB 2000:X$(X)=X$:NEXT
125 PRINT "  ELEMENT      SYMBOL MOL.WT."
126 FOR X=1 TO 8
130 PRINT X; TAB(3);E$(X); TAB(14);S$(X)
;TAB(27-LEN(X$(X)));X$(X)
150 NEXT
151 PRINT"  PRESS ANY KEY TO CONTINUE  "
152 GET Z$:IF Z$="" THEN 152
159 REM *****
160 REM LINES 15 TO 160 CAN BE DELETED
161 REM AFTER DATA ENTRY AND REST OF
162 REM PROGRAM ENTERED MUST USE #GOTO
163 REM 200* TO USE PROGRAM, NOT RUN, TO
164 REM PRESERVE DATA
165 REM *****
170 REM          **END 1.**
180 REM
199 REM *****
201 REM 2. PROGRAM MENU TO CHOOSE
202 REM      CALCULATION.
203 REM *****
210 PRINT "CHOOSE CALCULATION REQUIRED:-
"
220 PRINT
230 PRINT"TO INPUT NUMBER OF ATOMS AND G
ET PRECENTAGE ELEMENT COMPOSITION AND";
232 PRINT" MOLECULAR WEIGHT INPUT E"
240 PRINT
250 PRINT "TO INPUT PRECENTAGE ELEMENT A
NALYSIS AND GET MOLECULAR FORMULA";
255 PRINT"INPUT M"
260 INPUT A$
270 IF A$<>"E" AND A$<>"M" THEN GOTO 260
280 PRINT " "

```



```

290 IF A$="M" THEN GOTO 1000
292 REM          *****
295 REM          ** END 2. **
296 REM          *****
297 REM *****
298 REM 3. PRECENT ELEMENT
299 REM *****
300 PRINT "ENTER REFERENCE FOR THE COMPO
UND"
310 INPUT R$
320 REM ** INITIALISE **
330 DIM A(8)
340 MOLWT=0
350 DIM P(8)
360 FOR E=1 TO 8
370 PRINT "NUMBER OF ATOMS OF ";E$(E);"?
"
380 INPUT A(E)
390 MOLWT=MOLWT+(A(E)*M(E))
400 NEXT E
410 PRINT ""
420 REM *CALCULATION AND PRINT RESULTS*
430 PRINT "MOLECULAR WEIGHT AND COMPOSIT
ION"
440 PRINT
450 PRINT "REF: ";R$
460 PRINT
470 PRINT "MOL.WEIGHT= ";MOLWT
480 PRINT
490 PRINT "MOL.FORMULA: ";
500 FOR E=1 TO 8
510 IF A(E)>1 THEN PRINT S$(E);A(E);
520 IF A(E)=1 THEN PRINT S$(E);" ";
530 NEXT E
540 PRINT ".",,,
550 FOR E=1 TO 8
560 IF A(E)=0 THEN GOTO 620
565 REM *CALCULATE PERCENTAGES*
570 P(E)=A(E)*M(E)/MOLWT*100
580 N=P(E)
590 P=2
600 GOSUB 2000
610 PRINT "PERCENT ";S$(E);TAB(18-LEN(X$
));X$
620 NEXT E
625 REM *COPY HERE FOR PRINTED RESULTS*
630 GOTO 1800
635 REM          *****
640 REM          ** END 3. **
645 REM          *****

```

```

650 REM
993 REM *****
994 REM 4. MOLECULAR FORMULA
995 REM *****
1000 PRINT "ENTER REFERENCE FOR COMPOUND
"
1010 INPUT R$
1020 REM **DIM ARRAYS**
1030 DIM F(8)
1040 DIM N(8)
1050 PRINT
1060 REM **INPUT PERCENTAGES**
1070 FOR E=1 TO 8
1080 PRINT "PERCENT OF ";E$(E); "?"
1090 INPUT P(E)
1095 REM **CALCULATE PROPORTION*
1100 N(E)=P(E)/M(E)
1110 NEXT E
1120 PRINT "[]"
1130 REM **CALCULATE AND PRINT RESULTS**

1140 PRINT "*MOLECULAR FORMULA*";
1145 PRINT TAB(0); "*****"
1150 PRINT
1160 PRINT "REF: ";R$
1170 PRINT
1175 REM **CALCULATE MINIMUM**
1176 REM **PROPORTION ELEMENT*
1180 C=100
1190 FOR E=1 TO 8
1200 IF N(E)=0 THEN GOTO 1220
1210 IF N(E)<C THEN C=N(E)
1220 NEXT E
1225 REM **DERIVE FORMULA**
1230 FOR E=1 TO 8
1240 IF N(E)=0 THEN GOTO 1290
1250 P=2
1260 N=N(E)/C
1270 GOSUB 2000
1280 PRINT S$(E); TAB(8-LEN(X$));X$
1290 NEXT E
1295 REM *COPY HERE FOR PRINTED RESULTS*

1300 PRINT "MULTIPLES OF THIS FORMULA MA
Y BE THE";
1302 PRINT "ACTUAL FORMULA IF AT LEAST 2
ATOMS OF";
1304 PRINT "EACH ELEMENT PRESENT"
1320 REM
1325 REM *****

```

```

1330 REM          **END 4. **
1340 REM          ****
1780 REM
1790 REM
1795 REM  ** ENDRUN INSTRUCTIONS**
1800 PRINT LEFT$(Y$,0);LEFT$(X$,20);"USE
      GOTO 200 TO RUN AGAIN.",
1802 PRINT "USE GOTO 3000 TO SAVE"
1810 STOP
1820 REM
1825 REM  *** END PROGRAM ***
1830 REM
2000 REM  ****
2002 REM 5. SUBROUTINE TO ROUND
2004 REM AND JUSTIFY.
2006 REM  ****
2010 X$=""
2020 FOR F=1 TO P
2030 X$=X$+"0"
2040 NEXT F
2050 XN= INT(N)
2060 XD= INT(10↑P*(N-XN)+0.5)
2070 X$=STR$(XN)+". "+LEFT$(X$,P+1-LEN(ST
R$(XD)))+LEFT$(STR$(XD)+X$,P)
2080 RETURN
2090 REM
2095 REM  ** END SUB **
2889 REM  ****
2980 REM
2990 REM AUTO-RUN ROUTINE
2995 REM  ****
3000 SAVE "ELEMENT"
3010 GOTO 200
3020 REM ** END PROGRAM LIST **

```

V4: Games programming

Games are applications programs which are not of a type which fulfils a specific purpose in a functional context; that is, they are not written to do a specific scientific, educational or data-manipulation task. In other words, they are only games; But this does not mean that, as programming tasks, they are frivolous. The enjoyment of playing the game on or with the computer is the application for which the program is written, but the task of programming a game is often difficult. Games programming is good practice for finding, deriving and coding algorithms and producing efficient and user-friendly programs. Graphics manipulation plays a larger part in games programming than in most applications programs, and such programs are also more interactive, requiring repeated inputs and outputs.


```

95 REM ** START OF LOOP **
100 V=0
110 IF H<600 THEN PRINT "▢";LEFT$(Y$,20)
;LEFT$(X$,20-H/30);"  "
120 PRINT"▢FUEL ";F
121 PRINT"HEIGHT ";H
125 PRINT "SPEED:";S;" "
135 REM ** CHECK ROCKETS **
140 GET A1$:IF A1$="" THEN 140
145 IF A1$="R" AND F>=5 THEN V=5
150 IF V THEN PRINT "▢";LEFT$(Y$,20);LEF
T$(X$,21-H/30);"V V"
160 IF F<5 THEN PRINT"▢";LEFT$(Y$,7);LEF
T$(X$,1);"*EMPTY*"
170 F=F-V
180 S=S+2-V
190 PRINT "▢";LEFT$(Y$,20);LEFT$(X$,20-H
/30);"  ";TAB(20);"  "
195 REM ** CHECK IF LANDED **
200 IF H<30 THEN GOTO 230
210 H=H-S
220 GOTO 100
225 REM ** LANDING RESULT **
226 PRINT"▢"
230 PRINT"▢"
235 IF S<4 THEN PRINT "▢"LEFT$(Y$,10);LE
FT$(X$,21);"PERFECT LANDING"
240 IF S<4 THEN PRINT "▢"LEFT$(Y$,10);LE
FT$(X$,21);"BUMPY BUT SAFE"
250 IF S>=8 THEN PRINT "▢"LEFT$(Y$,10);L
EFT$(X$,21);"CRASHED AND SMASHED"
260 PRINT"▢"LEFT$(Y$,0);LEFT$(X$,12);"AN
OTHER GAME?(INPUT Y OR N)"
270 INPUT A2$
280 IF A2$="Y" THEN GOTO 45
290 REM ** E N D **
1210 PRINT"▢FUEL ";F

```

Lines 10 to 30 print instructions, and line 40 stops the program until a key is pressed. Lines 50 to 90 clear the screen, print the 'lunar surface' and set the variables: F is fuel units, H is height above surface, S is speed of descent.

The main program is in the loop between lines 100 and 220. V is set to zero as a flag, and the craft is printed by line 110 if the scale set allows it to be on screen. The craft disappears off the top of the screen if the height is greater than 600. Line 120 prints the current data. Note the spaces after the variables to overprint if the values decrease, or in the case of speed becoming positive after being negative.

Line 140 checks if the R (for rocket) key is being pressed. If it is, V is set to 5. Line 150 prints rocket exhausts below the craft if the R key was pressed (evaluated by $V = 0 = \text{False}$ if not pressed, $V = 5 = \text{True}$ if pressed). The fuel is checked (line 160) and reduced by the value of V if not empty. The speed is adjusted by increasing it, then reducing it by the value of V if the rockets have been fired. Line 190 overprints the craft and rockets, and 200 checks if the surface is near enough for landing to be assumed. If it is, control is transferred to the landing message section. If not, the height is adjusted and the program loops back to repeat the process.




















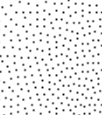
Notice that the variable V is used in three ways within the loop, and that the loop structure, using the GET instruction to see if the player has input instructions, is common in interactive games. It provides a simulation of a real-time process. In this game, the speed is assumed to be metres per second (hence the simple $\text{LET } H = H - S$ of line 210). It is actually nominal 'metres' per program loop! Other games can wait for inputs, but the use of a loop allows the inexorable attraction of gravity to go on its way unless the player does something.








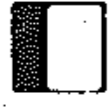

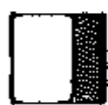







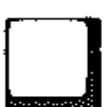














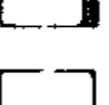









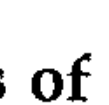
Programming for this type of game can show the programmer that certain structures of programs are inefficient in program execution, since conditional branches to routines requiring calculations will noticeably slow the loop. In the interests of a good game, structured programming practice may be set aside and speed of execution can become a goal in itself. However, remember not to transfer these techniques to serious programs!

Games programming can become extremely complex, considering the strategy and tactics which must be built into the response from the computer. Lander, however, is only the simplest type of game. If this area interests you, you can put to work the techniques you have learned in this text to analyse some of the tactical games in the popular computing magazines. In order to appreciate the problems involved, you could start by writing a program to play noughts and crosses. You may think it is a simple game, but it is surprisingly difficult to program!

APPENDIX I: CHR\$ CODES

This appendix shows you what characters will appear if you PRINT CHR\$(X), for all possible values of X. It will also show the values obtained by typing PRINT ASC("X"), where X is any character you can type. This is useful in evaluating the character received in a GET statement, converting upper/lower case, and printing character based commands (like switch to upper/lower case) that could not be enclosed in quotes.

PRINT\$	CHR\$	PRINT\$ CHR\$	PRINT\$ CHR\$	PRNT\$CHR\$
		& 38	@ 64	Z 90
	0 - 4	' 39	A 65	[91
WHITE	5	(40	B 66	# 92
	6 - 7) 41	C 67] 93
<i>Disables SHIFT C=</i>	8	* 42	D 68	↑ 94
<i>Enables SHIFT C=</i>	9	+ 43	E 69	← 95
	10 - 12	, 44	F 70	 96
RETURN	13	- 45	G 71	 97
SWITCH TO LOWER CASE	14	. 46	H 72	 98
	15	/ 47	I 73	 99
	16	0 48	J 74	 100
MOVES CURSOR DOWN	17	1 49	K 75	 101
INVERSE VIDEO	18	2 50	L 76	 102
HOME CURSOR	19	3 51	M 77	 103
DELETE CHARACTER	20	4 52	N 78	 104
	21 - 27	5 53	O 79	 105
RED	28	6 54	P 80	 106
CURSOR RIGHT	29	7 55	Q 81	 107
GREEN	30	8 56	R 82	 108
BLUE	31	9 57	S 83	 109
SPACE	32	: 58	T 84	 110
!	33	; 59	U 85	 111
"	34	< 60	V 86	 112
#	35	= 61	W 87	 113
\$	36	> 62	X 88	 114
%	37	? 63	Y 89	 115

PRINT\$	CHRS\$	PRINT\$	CHRS\$	PRINT\$	CHRS\$
	116	CLR HOME	147		178
	117	INST OFF	148		179
	118	Brown	149		180
	119	Lt. Red	150		181
	120	Grey 1	151		182
	121	Grey 2	152		183
	122	Lt. Green	153		184
	123	Lt. Blue	154		185
	124	Grey 3	155		186
	125		156		187
	126	CRSR	157		188
	127		158		189
	128		159		190
Orange	129	SPACE	160		191
	130		161		
	131		162		
	132		163		
f1	133		164		
f3	134		165		
f5	135		166		
f7	136		167		
f2	137		168		
f4	138		169		
f6	139		170		
f8	140		171		
SHIFT RETURN	141		172		
SWITCH TO UPPER CASE	142		173		
	143		174		
BRK	144		175		
CRSR	145		176		
RVS OFF	146		177		

Codes from 128 to 255 are the reversed images of codes 0 to 127

APPENDIX II: SCREEN DISPLAY CHARACTERS AND CODES

The following chart lists all of the characters built into the Commodore 64 character sets. It shows which numbers should be POKEd into screen memory (locations 1024–2023) to get a desired character. Also shown is which character corresponds to a number PEEKed from the screen.

Two character sets are available, but only one set at a time. This means that you cannot have characters from one set on the screen at the same time you have characters from the other set displayed. The sets are switched by holding down the **SHIFT** and **C=** keys simultaneously.

Codes from 128–255 are reversed images of codes 0–127

POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2
0	@	@	21	U	u	42	*	:
1	A	a	22	V	v	43	+	+
2	B	b	23	W	w	44	,	,
3	C	c	24	X	x	45	-	-
4	D	d	25	Y	y	46	.	.
5	E	e	26	Z	z	47	/	/
6	F	f	27	[[48	0	0
7	G	g	28	£	£	49	1	1
8	H	h	29	I	I	50	2	2
9	I	i	30	↑	↑	51	3	3
10	J	J	31	←	←	52	4	4
11	K	k	32	SPACE	SPACE	53	5	5
12	L	l	33	!	!	54	6	6
13	M	m	34	"	"	55	7	.7
14	N	n	35	#	#	56	8	8
15	O	o	36	\$	\$	57	9	9
16	P	p	37	%	%	58	:	:
17	Q	q	38	&	&	59	;	;
18	R	r	39	'	'	60	<	<
19	S	s	40	((61	=	=
20	T	t	41))	62	>	>

POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2
63	?	?	85		U	107		
64			86		V	108		
65		A	87		W	109		
66		B	88		X	110		
67		C	89		Y	111		
68		D	90		Z	112		
69		E	91			113		
70		F	92			114		
71		G	93			115		
72		H	94			116		
73		I	95			117		
74		J	96	SPACE	SPACE	118		
75		K	97			119		
76		L	98			120		
77		M	99			121		
78		N	100			122		
79		O	101			123		
80		P	102			124		
81		Q	103			125		
82		/R	104			126		
83		S	105			127		
84		T	106					

APPENDIX III: ABBREVIATIONS FOR BASIC KEYWORDS

Commodore 64 BASIC allows you to abbreviate keywords by typing the first one or two letters of a word followed by the **SHIFT**ed next letter. Words will be listed in their full form whether entered as abbreviations or in full.

Com- mand	Abbrev- iation	Appearance on screen	Com- mand	Abbrev- iation	Appearance on screen
ABS	A SHIFT B	A <input type="checkbox"/>	INT	NONE	INT
AND	A SHIFT N	A <input checked="" type="checkbox"/>	LEFT\$	LE SHIFT F	LE <input type="checkbox"/>
ASC	A SHIFT S	A <input checked="" type="checkbox"/>	LEN	NONE	LEN
ATN	A SHIFT T	A <input type="checkbox"/>	LET	L SHIFT E	L <input type="checkbox"/>
CHR\$	C SHIFT H	C <input type="checkbox"/>	LIST	L SHIFT I	L <input type="checkbox"/>
CLS	CL SHIFT O	CL <input type="checkbox"/>	LOAD	L SHIFT O	L <input type="checkbox"/>
CLR	C SHIFT L	C <input type="checkbox"/>	LOG	NONE	LOG
CMD	C SHIFT M	C <input checked="" type="checkbox"/>	MID\$	M SHIFT I	M <input type="checkbox"/>
CONT	C SHIFT O	C <input type="checkbox"/>	NEW	NONE	NEW
COS	NONE	COS	NEXT	N SHIFT E	N <input type="checkbox"/>
DATA	D SHIFT A	D <input checked="" type="checkbox"/>	NOT	N SHIFT O	N <input type="checkbox"/>
DEF	D SHIFT E	D <input type="checkbox"/>	ON	NONE	ON
DIM	D SHIFT I	D <input type="checkbox"/>	OPEN	O SHIFT P	O <input type="checkbox"/>
END	E SHIFT N	E <input checked="" type="checkbox"/>	OR	NONE	OR
EXP	E SHIFT X	E <input checked="" type="checkbox"/>	PEEK	P SHIFT E	P <input type="checkbox"/>
FN	NONE	FN	POKE	P SHIFT O	P <input type="checkbox"/>
FOR	F SHIFT O	F <input type="checkbox"/>	POS	NONE	POS
FRE	F SHIFT R	F <input type="checkbox"/>	PRINT	?	?
GET	G SHIFT E	G <input type="checkbox"/>	PRINT#	P SHIFT R	P <input type="checkbox"/>
GET#	NONE	GET#	READ	R SHIFT E	R <input type="checkbox"/>
GOSUB	GO SHIFT S	GO <input checked="" type="checkbox"/>	REM	NONE	REM
GOTO	GO SHIFT O	G <input type="checkbox"/>	RESTORE	RE SHIFT S	RE <input checked="" type="checkbox"/>
IF	NONE	IF	RETURN	RE SHIFT T	RE <input type="checkbox"/>
INPUT	NONE	INPUT	RIGHT\$	R SHIFT I	R <input type="checkbox"/>
INPUT#	I SHIFT N	I <input checked="" type="checkbox"/>	RND	R SHIFT N	R <input checked="" type="checkbox"/>

Com- mand	Abbrev- iation	Appearance on screen	Com- mand	Abbrev- iation	Appearance on screen
RUN	R SHIFT U	R	SYS	S SHIFT Y	S
SAVE	S SHIFT A	S	TAB(T SHIFT A	T
SGN	S SHIFT G	S	TAN	NONE	TAN
SIN	S SHIFT I	S	THEN	T SHIFT H	T []
SPC(S SHIFT P	S	TIME	TI	TI
SQR	S SHIFT Q	S	TIMES	TI\$	TI\$
STATUS	ST	ST	USR	U SHIFT S	U [♥]
STEP	ST SHIFT E	ST	VAL	V SHIFT A	V
STOP	S SHIFT T	S	VERIFY	V SHIFT E	V
STR\$	ST SHIFT R	ST	WAIT	W SHIFT A	W

APPENDIX IV: ERROR MESSAGES

BAD DATA When the program was expecting numeric data, string data was received from an open file.

BAD SUBSCRIPT An attempt was made to reference an element of an array whose number is outside the range specified with DIM.

BREAK Program execution has stopped as a result of hitting the **STOP** key.

CAN'T CONTINUE The CONT command will not respond: either the program has not been run, or a line has been edited, or an error has been encountered.

DEVICE NOT PRESENT One of the commands OPEN, CLOSE, CMD, PRINT#, INPUT# or GET# has attempted to access an unavailable I/O device.

DIVISION BY ZERO BASIC will not divide a number by zero.

EXTRA IGNORED An attempt has been made to INPUT too many items of data.

FILE NOT FOUND *Tape:* no End Of Tape marker found. *Disk:* no file with the requested name exists on disk.

FILE NOT OPEN The file specified in a OPEN, CLOSE, CMD, PRINT#, INPUT# or GET# has not been OPENed.

FILE OPEN You have attempted to OPEN a file with a file number which is already in use.

FORMULA TO COMPLEX It is sometimes necessary to split complex string expressions and formulae which use many layers of brackets to allow BASIC to process them correctly.

ILLEGAL DIRECT You have tried to use an INPUT statement in direct mode. This is not allowed.

ILLEGAL QUANTITY The value of an argument is outside the allowed range.

LOAD Unspecified tape error.

NEXT WITHOUT FOR Loops incorrectly nested or FOR...NEXT statements incorrectly matched.

NOT INPUT FILE INPUT or GET used with a file specified as output only.

NOT OUTPUT FILE PRINT used with a file specified as input only.

OUT OF DATA Insufficient DATA items for the current READ statement.

OUT OF MEMORY No RAM available for program or variables. Usually either: program too large, too many FOR loops or too many GOSUBs.

OVERFLOW BASIC cannot handle numbers larger than 1.7014188^{38} .

REDIM'D ARRAY An array may only be DIMensioned once within a program.

REDO FROM START You have tried to type characters into a numeric variable using INPUT. Retype the entry correctly.

RETURN WITHOUT GOSUB RETURN statements must always be matched with GOSUB statements.

STRING TOO LONG 255 characters is the maximum allowed in a string.

?SYNTAX ERROR Spelling, punctuation or other similar problems have produced an unrecognisable statement.

TYPE MISMATCH You have confused numbers and strings within a statement.

UNDEF'D FUNCTION You have tried to use DEF FN to access a user-defined function which you have not yet defined.

UNDEF'D STATEMENT You have tried to use RUN, GOTO or GOSUB with a non-existent line number.

VERIFY The program on tape or disk is not identical to the program currently held in memory.

APPENDIX V: MUSIC NOTE VALUES

MUSICAL NOTE			OSCILLATOR FREQ	
NOTE	OCTAVE	DECIMAL	HI	LOW
0	C-0	268	1	12
1	C#-0	284	1	28
2	D-0	301	1	45
3	D#-0	318	1	62
4	E-0	337	1	81
5	F-0	358	1	102
6	F#-0	379	1	123
7	G-0	401	1	145
8	G#-0	425	1	169
9	A-0	451	1	195
10	A#-0	477	1	221
11	B-0	506	1	250
16	C-1	536	2	24
17	C#-1	568	2	56
18	D-1	602	2	90
19	D#-1	637	2	125
20	E-1	675	2	163
21	F-1	716	2	204
22	F#-1	758	2	246
23	G-1	803	3	35
24	G#-1	851	3	83
25	A-1	902	3	134
26	A#-1	955	3	187
27	B-1	1012	3	244
32	C-2	1072	4	48
33	C#-2	1136	4	112
34	D-2	1204	4	180
35	D#-2	1275	4	251
36	E-2	1351	5	71
37	F-2	1432	5	152
38	F#-2	1517	5	237
39	G-2	1607	6	71
40	G#-2	1703	6	167
41	A-2	1804	7	12
42	A#-2	1911	7	119
43	B-2	2025	7	233
48	C-3	2145	8	97
49	C#-3	2273	8	225
50	D-3	2408	9	104
51	D#-3	2551	9	247
52	E-3	2703	10	143
53	F-3	2864	11	48
54	F#-3	3034	11	218
55	G-3	3215	12	143
56	G#-3	3406	13	78
57	A-3	3608	14	24
58	A#-3	3823	14	239
59	B-3	4050	15	210

MUSICAL NOTE

OSCILLATOR FREQ

NOTE	OCTAVE	DECIMAL	HI	LOW
64	C-4	4291	16	195
65	C#-4	4547	17	195
66	D-4	4817	18	209
67	D#-4	5103	19	239
68	E-4	5407	21	31
69	F-4	5728	22	96
70	F#-4	6069	23	181
71	G-4	6430	25	30
72	G#-4	6812	26	156
73	A-4	7217	28	49
74	A#-4	7647	29	223
75	B-4	8101	31	165
80	C-5	8583	33	135
81	C#-5	9094	35	134
82	D-5	9634	37	162
83	D#-5	10207	39	223
84	E-5	10814	42	62
85	F-5	11457	44	193
86	F#-5	12139	47	107
87	G-5	12860	50	60
88	G#-5	13625	53	57
89	A-5	14435	56	99
90	A#-5	15294	59	190
91	B-5	16203	63	75
96	C-6	17167	67	15
97	C#-6	18188	71	12
98	D-6	19269	75	69
99	D#-6	20415	79	191
100	E-6	21629	84	125
101	F-6	22915	89	131
102	F#-6	24278	94	214
103	G-6	25721	100	121
104	G#-6	27251	106	115
105	A-6	28871	112	199
106	A#-6	30588	119	124
107	B-6	32407	126	151
112	C-7	34334	134	30
113	C#-7	36376	142	24
114	D-7	38539	150	139
115	D#-7	40830	159	126
116	E-7	43258	168	250
117	F-7	45830	179	6
118	F#-7	48556	189	172
119	G-7	51443	200	243
120	G#-7	54502	212	230
121	A-7	57743	225	143
122	A#-7	61176	238	248
123	B-7	64814	253	46

Appendix VI: Program Library

This appendix contains applications and utility programs and routines, and games. Some of these have been referred to in the main text. Due to lack of space, the programs are not fully documented.

"CUBE"

This program gives the computer the capacity to draw a line between specified plot co-ordinates. Commodore BASIC does not provide us with a LINE instruction that does this automatically, but you may be interested in the method, which is the way the LINE instruction automatically calculates the points to plot. As it stands, the program gets the values of X and Y, using a READ statement. It is possible to INPUT for two sets of X, Y points, giving an error message if the points are out of range. Line 110 calculates the X and Y axis differences between the specified points. Line 90 defines the variable A as the greater of these. In the loop (I = 1 to ABS(A), since A may be negative) XA and YA are decremented or incremented (as X and Y are positive or negative) by the distance to be covered between points, divided by the number of steps needed.

```
10 POKE53272,8
20 POKE56576,PEEK(56576) AND 254
30 POKE53272,8
40 POKE53265,PEEK(53265) OR 32
50 FOR I=0 TO 999:POKE16384+I,230:NEXT
60 FOR I=0 TO 8000:POKE24576+I,0:NEXT
70 FOR J=1 TO 12
80 READ XS,YS,XE,YE
90 XA=XE-XS:YA=YE-YS:A=(XA AND (ABS(XA) >=
  (ABS(YA))))+(YA AND (ABS(XA) < (ABS(YA))))
100 X=0:Y=0
110 X=X+XS:Y=Y+YS
120 FOR I=1 TO ABS(A)
130 X1=INT(X/8):B=7-(X AND 7)
140 Y1=INT(Y/8):L=Y AND 7
150 C=Y1*320+X1*8+L
160 POKE24576+C,PEEK(24576+C) OR 2+B
170 X=X+XA/ABS(A):Y=Y+YA/ABS(A)
180 NEXT I,J
190 DATA 132,80,172,80,172,80,172,120
200 DATA 172,120,132,120,132,120,132,80
210 DATA 152,60,192,60,192,60,192,100
220 DATA 192,100,152,100,152,100,152,60
230 DATA 132,80,152,60,172,80,192,60
240 DATA 132,120,152,100,172,120,192,100
```

"LISSAJOUS"

A program to produce intricate and interesting patterns, named after the mathematician who discovered the equation that produces them. You merely enter the values of A, B and C in response to the prompts and watch the patterns develop.

```
10 INPUT"INPUT A1(INTEGER 1 TO 10) ";A1
20 INPUT"INPUT B1(INTEGER 1 TO 10) ";B1
30 INPUT"INPUT C1(0 TO 3) ";C1
40 PI=4*ATN(1)
50 POKE56576,PEEK(56576) AND 254
60 POKE53272,8
70 POKE53265,PEEK(53265) OR 32
80 FOR I=0 TO 999:POKE16384+I,230:NEXT
90 FOR I=0 TO 8000:POKE24576+I,0:NEXT
100 FOR I=0 TO 300 STEP 0.1
110 Y=100+50*SIN(B1*PI*I/100)
120 X=160+80*SIN(C1+A1*PI*I/100)
130 X1=INT(X/8):B=7-(X AND 7)
140 Y1=INT(Y/8):L=Y AND 7
150 C=Y1*320+X1*8+L
160 POKE24576+C,PEEK(24576+C) OR 2+B
170 NEXT
```

"CODER"

The computer chooses a four-digit code sequence comprised of the digits 1 to 6. You input your guess for this code sequence. The computer prints your guess, checks for the number of digits in the correct place which correspond to the code, storing this as AST (for asterisk), and then checks through the remaining digits for numbers which occur in the code sequence, but are not in the correct place (DOLLAR). These values are then printed. This information helps you to refine your next guess. 15 goes are allowed, and if you haven't got the code in 15 tries, it is printed out for you. The program is structured with a sequence of calls to subroutines. Note that the code can include repeated digits, and analyse the checking procedures to see how this is dealt with.

```
5 REM**CODER**
6 D$=""
10 GOSUB 1000
20 REM**INITIALISE AND CHOOSE CODE**
30 GUESS=0
40 C$=""
50 N$="123456"
60 FOR F=1 TO 4
70 C$=C$+MID$(N$, (INT(RND(1)*5+1)), 1)
```

```

80 NEXT
90 INPUT "ENTER YOUR GUESS"; G$
100 PRINT "GUESS      +      $"
110 MARK=0
120 GOSUB 400
130 IF MARK=1 THEN 90
140 GUESS=GUESS+1
150 PRINT " "; LEFT$(D$, GUESS+2); " "; G$; TAB(7);
160 GOSUB 500
170 GOSUB 600
180 PRINT AST; TAB(9); DOLLAR
200 REM**SEE IF 15 TRIES OR CODE CRACKED
**
210 IF AST=4 OR GUESS=15 THEN 2000
220 REM**LOOP TO NEXT GUESS**
230 GOTO 90
400 REM**CHECK INPUT**
410 FOR F=1 TO LEN(G$)
420 IF ASC(MID$(G$, F, 1)) > 54 OR ASC(MID$(G$, F, 1)) < 49 THEN MARK=1
430 NEXT
440 IF LEN(G$) <> 4 THEN MARK=1
450 IF MARK=0 THEN RETURN
460 PRINT " "; LEFT$(D$, GUESS+3); "WHAT "; G$; " TRY AGAIN"
470 FOR I=1 TO 100:NEXT
480 PRINT " "; LEFT$(D$, GUESS+3); "
"

490 RETURN
500 REM** FIND + NUMBER**
510 A$=C$:AST=0
520 FOR F=1 TO 4
530 IF MID$(A$, F, 1) <> MID$(G$, F, 1) THEN 570
540 AST=AST+1
550 G$=MID$(G$, 1, F-1)+"0"+MID$(G$, F+1, 4-F)
560 A$=MID$(A$, 1, F-1)+"X"+MID$(A$, F+1, 4-F)
570 NEXT
580 RETURN
600 REM**FIND $ NUMBER**
610 REM
620 DOLLAR=0
630 FOR F=1 TO 4
640 FOR N=1 TO 4
650 IF MID$(A$, F, 1) <> MID$(G$, N, 1) THEN 690

```

```

660 DOLLAR=DOLLAR+1
670 A$=MID$(A$,1,F-1)+"X"+MID$(A$,F+1,4-
F)
680 G$=MID$(G$,1,N-1)+"O"+MID$(G$,N+1,4-
N)
690 NEXT N
695 NEXT F
700 RETURN
1000 PRINT "*****CODE
R"
1010 PRINT "*****"
1020 PRINT "COMPUTER CHOOSES 4 NUM
BERES BETWEEN"
1030 PRINT "1 TO 6 AND AT RANDOM, REPAT
ITIONS BEEING "
1040 PRINT "ALLOWED, FOR EXAMPLE 1232."
"
1050 PRINT "YOU ENTER A SEQUENCE OF 4 N
UMBERES"
1060 PRINT "THE COMPUTER TELLS YOU HOW
MANY "
1070 PRINT "ARE RIGHT AND IN THEIR CORR
ECT "
1080 PRINT "POSITION, DENOTED BY + OR
"
1090 PRINT " DENOTING RIGHT NUMBER B
UT IN"
1100 PRINT "WRONG POSITION. YOU HAVE UPT
O 15"
1110 PRINT "GUESSES"
1120 PRINT "PRESS A KEY WHEN READY"
1130 GET A$: IF A$="" THEN 1130
1140 PRINT "": RETURN
2000 REM**END ROUTINE**
2010 IF AST=4 THEN 2050
2020 PRINT "15 TRIES AND NO SUCESS. COD
E WAS ";C$; "": END
2030 PRINT "PRESS A KEY TO END"
2040 GET A$: IF A$="" THEN 2040
2045 PRINT "": END
2050 PRINT "SUCESS IN "; GUESS; "
TRIES"
2060 END

```

"GRIDHUNT"

The computer hides itself on an 8 by 8 grid, which is displayed on the screen. You input your guesses of the co-ordinates, the guessed square is marked, and if not correct the computer gives a prompt for its direction from this square, using compass directions.

```
1 REM**GRIDHUNT**
10 PRINT "GRIDHUNT"
15 D$=" "
17 L$=" "
20 FOR X=2 TO 16 STEP 2
30 PRINT " "; " "; LEFT$(L$, X+1); X/2; " "; L
LEFT$(D$, X+2); " "; X/2
40 PRINT " "; LEFT$(L$, X); "+"; " "; LEFT$(
(D$, 19); LEFT$(L$, X); "+"
50 PRINT " "; LEFT$(D$, X+1); " "+"; " "; LEFT
$(D$, X+1); LEFT$(L$, 18); "+"
60 NEXT
70 E=INT(RND(1)*8)+1
75 N=INT(RND(1)*8)+1
80 G=(E-1)*8+N
90 M=0
100 PRINT " "; " "; LEFT$(L$, 20); "YOUR GU
ESS: -"; " "; LEFT$(L$, 20); "ACROSS?"
110 INPUT " "; GUESS
"; A
120 PRINT " "; LEFT$(L$, 26); " "; A; " ";
"; LEFT$(L$, 20); "DOWN?"
130 INPUT " "; GUESS
"; D
131 FOR X=1 TO 5
132 PRINT " "; LEFT$(D$, 2+D*2); LEFT$(L$, 1
+A*2); " "
133 PRINT " "; LEFT$(D$, 2+D*2); LEFT$(L$, 1
+A*2); " "
134 NEXT X
135 PRINT " "; LEFT$(D$, 2+D*2); LEFT$(L$, 1
+A*2); "*"
140 PRINT " "; LEFT$(D$, 7); LEFT$(L$, 24); "
"; D; " "; LEFT$(D$, 9); LEFT$(L$, 20);
145 M=M+1
150 C=(A-1)*8+D
160 IF C=G THEN 300
165 PRINT "I AM ";
170 IF N=D THEN 210
190 IF N>D THEN PRINT "S";
200 IF N<D THEN PRINT "N";
210 IF E>A THEN PRINT "E";
220 IF E<A THEN PRINT "W";
```

```

230 PRINT " "; TAB(62); "OF YOU"
240 FOR I=1 TO 5000:NEXT
250 FOR X=3 TO 10
260 PRINT " "; LEFT$(D$, X); LEFT$(L$, 20); "
      "
270 NEXT
280 GOTO 100
290 END
300 PRINT "GOT ME IN "; M; " MOVES"
310 PRINT "PRESS A KEY TO END"
320 GET A$: IF A$="" THEN 320
330 PRINT " ": END

```

"MATTMULT"

The program multiplies two square matrices. A two-dimensional matrix is stored as a two-dimensional array, with the size input. Matrix multiplication requires the number of columns in one matrix to be equal to the number of rows in the other. The matrices are set up as square arrays of equal size in this program, and nonsquare matrices may be multiplied by entering 0 for the elements of a row or column which is unused. Users familiar with matrix arithmetic will be able to derive from this program the routines to handle other matrix operations. The method involves nested FOR...NEXT loops, in conjunction with three arrays in this program, the third array holding the resultant matrix.

Other points to be noted are the input and error routines. The input routine prompts for inputs by row and column number, and when all elements have been entered the error-check subroutine is called, so that the user can check the whole matrix at once. This avoids the possibility of confusion over row/column numbers.

```

5 REM**MATTMULT**
10 PRINT "*****2D MATRIX MULTIPL
ICATION*****"
15 PRINT "*****"
20 PRINT "MULTIPLIES SQUARE MAT
RICES TO USE FOR"
30 PRINT "NON-SQUARE MATRICES ENTER MAT
RIX SIZE"
40 PRINT "TO ACCOMADATE, AND ENTER ZERO
ES "
50 PRINT "FOR EXAMPLE TO MULTIPLY A (1)
(2) (3) "
60 PRINT " BY A (4) (5) (6) ENT
ER 3 AS MATRIX SIZE, "
70 PRINT "ENETR 1 COLUMN AND
ONE ROW"
80 PRINT " , REST 0. ENTER MATRI
CES ROW"

```

```

90 PRINT "#####BY ROW
"
95 A$="
"
100 PRINT "PRESS A KEY TO CONTINUE"
105 GET Z$: IF Z$="" THEN 105
110 D$="#####
#####"
111 L$="#####
#####"
112 PRINT "#####LEFT$(D$,21); " **ENTER MATRI
X SIZE"
113 INPUT "#####"; S
114 DIM A(S,S),B(S,S),C(S,S)
120 PRINT "#####ENTER MATRIX ONE ENTER # OF
OR "
130 PRINT "UNUSED ELEMENTS"
135 PRINT "#####LEFT$(D$,21); A$
140 FOR F=1 TO S
150 FOR N=1 TO S
160 PRINT "#####LEFT$(D$,21); "ROW "; F; " CO
LUMN "; N; " ?";
170 INPUT A(F,N)
180 PRINT "#####LEFT$(D$,F*3); LEFT$(L$,N*6
-6); A(F,N) "
190 NEXT N,F
200 M=1
210 GOSUB 500
220 PRINT "#####MATRIX 2"
230 FOR F=1 TO S
240 FOR N=1 TO S
250 PRINT "#####LEFT$(D$,21); "ROW "; F; " CO
LUMN "; N; " ?";
260 INPUT B(F,N)
270 PRINT "#####LEFT$(D$,F*3); LEFT$(L$,N*6
-6); B(F,N) "
280 NEXT N,F
290 M=2
300 GOSUB 500
310 PRINT "#####MATRIX 1 * MATRIX 2"
320 FOR F=1 TO S
330 FOR N=1 TO S
340 FOR L=1 TO S
350 C(F,N)=C(F,N)+A(F,L)*B(L,N)
360 C(F,N)=INT(C(F,N)*1E5+.5)/1E5
370 PRINT "#####LEFT$(D$,F*3); LEFT$(L$,N*6-
6); "#####C(F,N); "#####"
380 NEXT L,N,F
390 END
500 PRINT "#####LEFT$(D$,21); A$

```

```

510 PRINT "ARE ALL ENTRIES CORRECT (Y/N) ";
520 INPUT B$
530 IF B$="Y" THEN RETURN
540 PRINT "HOW MANY INCORRECT ENTRIES ? ";
550 INPUT EN
560 FOR F=1 TO EN
570 PRINT "LEFT (D$,20); A$; "LEFT (D$,21); A$; "ERROR "; F;
571 PRINT " ROW ";
580 INPUT R
590 PRINT "LEFT (D$,21); LEFT (L$,7); "
    COLUMN ";
600 INPUT C
610 PRINT "LEFT (D$,21); A$; "LEFT (D$,21); "ENTER CORECT NUMBER ";
620 INPUT N
630 IF M=1 THEN A(R,C)=N
640 IF M=2 THEN B(R,C)=N
650 PRINT "LEFT (D$,R*3); LEFT (L$,C*6-6); "
    "LEFT (D$,R*3);
660 PRINT LEFT (L$,C*6-6); N
670 NEXT F
680 PRINT "LEFT (D$,21); A$
690 GOTO 510
700 REM**END**

```

Example print-out:

```

ENTER MATRIX 1
ENTER 0 FOR UNUSED ELEMENTS

```

```

1    2    3
0    0    0
0    0    0

```

```

MATRIX 2

```

```

0    0    4
3    0    5
0    0    6

```

```

MATRIX 1 * MATRIX 2 GIVES:-

```

```

0    0    32
0    0    0
0    0    0

```

```

INPUT R(RUN) OR E(END)

```


"HEXDEC"

This converts hexadecimal numbers up to FFFF (65534 decimal) into decimal.

```
1 REM**HEXDEC**
10 DIM A(4)
20 N=0
30 PRINT"HEXADICIMAL TO DECIMAL"
40 PRINT
50 PRINT"ENTER HEXADECIMAL NUMBER 4 CHAR
ACTERS LONG ";
60 INPUT"LETTERS MUST BE CAPITOLS";H$
65 G=4
70 FOR F=1 TO LEN(H$)
80 A(F)=ASC(MID$(H$,F,1))
90 IF A(F)>=65 AND A(F)<=70 THEN A(F)=A(
F)-55:GOTO 110
100 A(F)=A(F)-48
110 G=G-1
120 A(F)=A(F)*16+G
140 N=N+A(F)
150 NEXT
160 PRINT"DECIMAL NUMBER IS ";N
```

"DECHEX"

This program converts decimal (base 10) numbers up to 65534 to their four-figure hexadecimal/(base 16) equivalents. Hexadecimal numbers use the digits 0 to 9 plus the letters A to F. This requires a means of deciding which character is to be printed, after the decimal number has been broken down.

```
1 REM**DECHEX**
10 DIM A(4)
20 PRINT"DECIMAL TO HEXADECIMAL CONVERS
ION"
30 PRINT"NUMBER MUST BE POSITIVE AND LES
S THAN 65535"
40 INPUT"ENTER DECIMAL NUMBER";N
50 A(1)=INT(N/4096)
60 B=N-A(1)*4096
70 A(2)=INT(B/256)
80 C=B-A(2)*256
90 A(3)=INT(C/16)
100 A(4)=C-A(3)*16
110 FOR F=1 TO 4
120 X=48
130 FOR Y=0 TO 15
```

```

140 IF X=58 THEN X=65
150 IF A(F)=Y THEN A$=A$+CHR$(X)
160 X=X+1
170 NEXT Y,F
180 PRINT N$;" IS ";A$;" IN HEX"
190 END

```

"RESCODE"

This program calculates resistor values from inputs of the colour bands on the resistor. Three bands are input, end band first, using the abbreviations given. The first two bands define the basic value and the third the multiplier.

```

1 REM**RESCODE**
5 DIM A$(3),Z(3)
10 PRINT "-----"
20 PRINT "CODE | COLOUR |"
30 PRINT "-----"
40 FOR A=1 TO 12
50 READ A$,B$,Z
60 PRINT " | ";A$;" | ";B$;" |"
70 NEXT
80 PRINT "-----"
90 PRINT"ENTER THREE CODES FOR RESITOR"
100 RESTORE:FOR A=1 TO 3
110 INPUT A$
140 FOR B=1 TO 12
150 READ C$,D$,Z
160 IF A$=C$ THEN 190
170 NEXT B
180 PRINT CHR$(13);"INCORECT CODE REENT
ER":A$="":RESTORE:GOTO 110
190 A$(A)=A$:Z(A)=Z:RESTORE
200 NEXT A
210 PRINT""
220 FOR A=1 TO 3
230 PRINT "CODE ";A;" IS ";A$(A)
240 NEXT A
250 N=Z(1)
260 N=N*10+Z(2)
265 IF Z(3)>9 THEN 330
270 PRINT"RESISTANCE IS ";N;
280 FOR A=1 TO Z(3)
290 PRINT"0";
300 NEXT A

```

```

310 PRINT " OHMS"
320 END
330 PRINT N/Z(3); " OHMS"
340 END
1000 DATA RE,"RED  ",2,BL,"BLACK ",0,BR
,"BROWN ",1,OR,"ORANGE",3,YE,"YELLOW",4
1010 DATA GR,"GREEN ",5,BL,"BLUE  ",6,VT
,"VIOLET",7,GY,"GREY  ",8,WH,"WHITE ",9
1020 DATA GO,"GOLD  ",10,SI,"SILVER",100

```

"FRUIT"

The program simulates a fruit machine. It allows you to continue playing until your money runs out (which it will eventually) and you can then 'borrow' more. Points to be noted in the program are the overprinting of a string to simulate the spinning of the wheels, and the logic used to check wins and amount (if any) won. The program loops back unless the money has all gone.

```

10 REM**FRUIT**
20 PRINT "##### ONE ARMED
  BANDIT###"
30 PRINT "##### YOU HAVE £2## TO GAMBLE
  "
40 PRINT "##### EACH ROLL COSTS £10 P £"
50 PRINT "##### PAYOUTS: ## TWO THE SAME PAYS
  £20 P"
60 PRINT TAB(10); "## THREE THE SAME PAYS ##
  40 P"
70 PRINT TAB(10); "## EXCEPT ### WHICH
  PAYS £1.00"
80 PRINT "##### PRESS KEY 5 TO STA
  RT##"
90 GET Z$: IF Z$="" THEN 90
100 PRINT "##"
110 C$="£2.00": A$="*#####"
120 PRINT "##### ONE ARMED BANDIT"

131 PRINT "#####
  "
132 PRINT "##### |##### |"
133 PRINT "##### |##### THE RIPPER |##### |
  "
134 PRINT "##### |##### |"
135 PRINT "##### |##### |"
136 PRINT "##### |##### |"
137 PRINT "##### |##### |"
138 PRINT "##### |##### |"

```

```

139 PRINT "##### |##### |"
140 PRINT "##### |##### |##### |"
141 PRINT "##### |##### |##### |"
142 PRINT "##### |##### |##### |"
143 PRINT "##### |##### |##### |"
144 PRINT "##### |##### |##### |"
145 PRINT "##### |##### |##### |"
146 PRINT "##### |##### |##### |"
147 PRINT "##### |##### |##### |"
";CHR$(13);TAB(12);
149 PRINT "##### |##### |##### |";CHR$(13);TAB(
12);"##### |"
150 GET Z$:IF Z$="" THEN 150
155 IF Z$<>"5" THEN 150
160 B$="":FOR F=1 TO 3:A=INT(RND(O)*6)+1

170 B$=B$+"##### |" +MID$(A$,A,1)
180 NEXT
190 F$="##### |" + $ * "##### |"
200 FOR F=1 TO 10
210 PRINT "##### |";LEFT$(
F$,7)
220 F$=RIGHT$(F$,4)+LEFT$(F$,3)
230 NEXT
240 PRINT "##### |";B$
245 FOR Z=1 TO 1000:NEXT
250 W=(MID$(B$,4,1)=MID$(B$,8,1))+(MID$(
B$,4,1)=MID$(B$,12,1))
255 W=W+(MID$(B$,8,1)=MID$(B$,12,1))
260 IF W=-1 THEN C=.20
265 IF W=0 THEN C=0
266 IF W=-3 AND MID$(B$,8,1)="*" THEN C=
1.0
270 C$=MID$(C$,1,1)+MID$(STR$(VAL(RIGHT$(
C$,4))+C-.10),2)
271 IF LEN(C$)=4 THEN C$=C$+"0"
272 IF LEN(C$)=2 THEN C$=C$+".00"
273 IF LEN(C$)=3 THEN C$=LEFT$(C$,1)+"0"
+MID$(C$,2,2)+"0"
290 IF VAL(RIGHT$(C$,4))>=.10 THEN 131
300 PRINT "##### |*YOU ARE BROKE.*"
310 INPUT "##### |BORROW $2.00 ?(Y/N)";M$
320 IF M$="Y" THEN PRINT "##### |":GOTO 110
330 PRINT "##### |BETTER LUCK NEXT TIME##### |"
340 END

```

"ASTEROIDS"

The program puts you at the helm of a Mars shuttle disguised as an asterisk. Avoiding the Nova Heat you have to weave through the strangely square low albedo asteroids that look surprisingly like inverse squares. Your controls are fairly minimal – not much money on the Mars run smuggling algae these days, so you have a button marked 1 to go left and one marked 0 to go right.

```
5 REM**ASTEROIDS**
10 PRINT "*****ASTEROIDS*"
20 PRINT "*****AVOID BLACK ASTEROIDS."

30 PRINT "*****YOU STEER YOUR SHIP (*) BY
"
40 PRINT "*****PRESSING 1 TO GO LEFT AND 0"
50 PRINT "*****TO GO RIGHT*****"
60 FOR I=1 TO 1000:NEXT
65 PRINT "*****"
70 S=0
80 C=10
90 U$="*****"
*****"
100 R$="*****"
*****"
110 PRINT:PRINT LEFT$(U$,6);LEFT$(R$,C);
"*"
115 IF PEEK((6*40)+C+1024)=160 THEN 200
120 L=C
130 GET A$:IF A$="" THEN 150
140 C=C-(C>1 AND A$="1")+(C<39 AND A$="0
")
145 IF C<=0 THEN C=2
146 IF C>=39 THEN C=38
150 PRINT LEFT$(U$,RND(1)*24);LEFT$(R$,R
ND(1)*39);"  ";LEFT$(U$,25);" "
160 S=S+1
170 PRINT LEFT$(U$,6);LEFT$(R$,L);" "
175 FOR I=2 TO 6
176 PRINT" ";LEFT$(U$,I);LEFT$(R$,L);" "

177 NEXT
180 GOTO 110
200 PRINT"*****"
210 PRINT"*****YOU CRASHED"
220 INPUT"ENTER Y TO PLAY AGAIN OR N TO
STOP";A$
230 IF A$="Y" THEN PRINT"*****":GOTO 65
240 IF A$="N" THEN END
250 GOTO 200
```

"SOCCER"

A program to create a football league table of results and points. The results are first entered one at a time and stored in a file (cassette) called FOOTBALL. When the 'display results' option is selected from the menu, the stored results are analysed and points are allocated to each team. This produces the points table, which is sorted in order of points after each results entry.

```
10 DIM TM$(22), PL(22), PT(22), WN(22), DW(2
2), LS(22), FO(22), CO(22)
20 PRINT "MENU": PRINT TAB(17); "MENUE"
30 POKE 781, 4: POKE 782, 5: SYS 65520: PRINT
"1.....INPUT RESULTS"
40 POKE 781, 6: POKE 782, 5: SYS 65520: PRINT
"2.....PRINT RESULTS"
50 POKE 781, 8: POKE 782, 5: SYS 65520: PRINT
"3.....GET TABLE"
60 POKE 781, 10: POKE 782, 5: SYS 65520: PRINT
"4.....UPDATE TABLE"
70 POKE 781, 12: POKE 782, 5: SYS 65520: PRINT
"5.....PRINT TABLE"
80 POKE 781, 14: POKE 782, 5: SYS 65520: PRINT
"6.....STORE TABLE"
90 POKE 781, 16: POKE 782, 5: SYS 65520: PRINT
"7.....INITIALISE TABLE"
100 POKE 781, 18: POKE 782, 5: SYS 65520: PRINT
"8.....END"
110 GET KY$: IF KY$="" THEN 100
120 IF KY$="8" THEN END
130 ON VAL(KY$) GOSUB 200, 400, 1000, 1200, 1
500, 1800, 2000
140 GOTO 20
150 REM
160 REM
170 REM
200 REM***** INPUT SUBROUTINE *****
205 CR$=CHR$(13)
210 OPEN 1, 1, 1, "FOOTBALL"
220 PRINT "MENU": PRINT TAB(5); "ENTER XXX TO E
ND"
230 POKE 781, 10: POKE 782, 5: SYS 65520: INPUT "
HOME TEAM "; HT$
240 IF HT$="XXX" THEN PRINT #1, HT$: GOTO 3
10
250 POKE 781, 10: POKE 782, 30: SYS 65520: INPUT
HS
260 POKE 781, 14: POKE 782, 5: SYS 65520: INPUT "
GUEST TEAM "; GT$
270 POKE 781, 14: POKE 782, 30: SYS 65520: INPUT
AS
```

```

280 PRINT#1,HT$;CR$;GT$;CR$;HS;CR$;AS;CR
$
290 POKE781,10:POKE782,17:SYS65520:PRINT
" ";
300 POKE781,14:POKE782,17:SYS65520:PRINT
" ";:GOTO 230
310 CLOSE 1
320 RETURN
330 REM
340 REM
350 REM
400 REM**** PRINT RESULTS SUB ***
410 PRINT"":L=3:M=1
420 POKE781,1:POKE 782,10:SYS65520:PRINT
"FOOTBALL RESULTS"
430 OPEN1,1,0,"FOOTBALL"
440 INPUT#1,HT$,GT$,HS,AS
450 IF HT$="XXX" THEN 670
460 POKE 781,L:POKE 782,0:SYS65520:PRINT
M
470 POKE 781,L:POKE 782,4:SYS65520:PRINT
HT$
480 POKE 781,L:POKE782,15:SYS65520:PRINT
"V ";GT$
490 POKE 781,L:POKE782,32:SYS65520:PRINT
HS;"-";AS
500 FOR I=1 TO 22
510 IF HT$=TM$(I) THEN 530
520 NEXT I:GOTO 580
530 PL(I)=PL(I)+1:FO(I)=FO(I)+HS:CO(I)=C
O(I)+AS:IF HS<=AS THEN 550
540 PT(I)=PT(I)+3:WN(I)=WN(I)+1:GOTO580
550 IF HS<AS THEN 570
560 PT(I)=PT(I)+1:DW(I)=DW(I)+1:GOTO580
570 LS(I)=LS(I)+1
580 FOR I=1 TO 22
590 IF GT$=TM$(I) THEN 610
600 NEXT I:GOTO 660
610 PL(I)=PL(I)+1:FO(I)=FO(I)+AS:CO(I)=C
O(I)+HS:IF AS<=HS THEN 630
620 PT(I)=PT(I)+3:WN(I)=WN(I)+1:GOTO660
630 IF AS<HS THEN 650
640 PT(I)=PT(I)+1:DW(I)=DW(I)+1:GOTO660
650 LS(I)=LS(I)+1
660 L=L+2:M=M+1:GOTO 440
670 CLOSE 1
680 GET AA$:IF AA$="" THEN 680
690 RETURN
700 REM
710 REM

```

```

720 REM
1000 REM*****      GET TABLE SUB      ****
1010 PRINT" "
1020 OPEN1,1,0,"POINTTAB"
1030 FOR I=1 TO 22
1040 INPUT#1, TM$(I), PL(I), WN(I), DW(I), LS
(I), FO(I), CO(I), PT(I)
1050 NEXT I
1060 CLOSE 1
1070 RETURN
1080 REM
1090 REM
1100 REM
1200 REM*****      UPDATE TABLE SUB  ****
1210 PRINT" ":POKE781,12:POKE782,15:SYS6
5520:PRINT"UPDATING TABLE"
1220 FOR I=1 TO 21
1230 FOR J=1 TO 22-I
1240 IF PT(J+1)<PT(J) THEN 1330
1250 PP=PT(J+1):PT(J+1)=PT(J):PT(J)=PP
1260 PP#=TM$(J+1):TM$(J+1)=TM$(J):TM$(J)
=PP#
1270 PP=FO(J+1):FO(J+1)=FO(J):FO(J)=PP
1280 PP=CO(J+1):CO(J+1)=CO(J):CO(J)=PP
1290 PP=PL(J+1):PL(J+1)=PL(J):PL(J)=PP
1300 PP=WN(J+1):WN(J+1)=WN(J):WN(J)=PP
1310 PP=DW(J+1):DW(J+1)=DW(J):DW(J)=PP
1320 PP=LS(J+1):LS(J+1)=LS(J):LS(J)=PP
1330 NEXT J,I
1340 RETURN
1500 REM*****      PRINT TABLE SUB  ****
1510 PRINT" ":LI=0
1520 POKE781,LI:POKE782,0:SYS65520:PRINT
"TEAM      PL      W      D      L      F";
1530 PRINT "      A      PT":LI=2
1540 FOR I=1 TO 22
1550 POKE 781,LI:POKE 782,0:SYS 65520:PR
INT TM$(I);TAB(9);PL(I);TAB(14);
1560 PRINT WN(I);TAB(18);DW(I);TAB(22);L
S(I);TAB(26);FO(I);
1570 PRINT TAB(30);CO(I);TAB(35);PT(I)
1580 LI=LI+1:NEXT I
1590 GET AA$:IF AA$="" THEN 1590
1600 RETURN
1770 REM
1780 REM
1790 REM
1800 REM*****      STORE TABLE SUB  ****
1810 PRINT" ":CR#=CHR$(13)
1820 OPEN1,1,1,"POINTTAB"

```



```

1830 FOR I=1 TO 22
1840 PRINT#1, TM$(I); CR$; PL(I); CR$; WN(I);
CR$; DW(I); CR$; LS(I); CR$;
1850 PRINT#1, FO(I); CR$; CO(I); CR$; FT(I)
1860 NEXT I
1870 CLOSE 1
1890 RETURN
1970 REM
1980 REM
1990 REM
2000 REM***** DIVISION ONE CLUBS *****
2010 FOR I= 1 TO 22
2020 READ TM$(I)
2030 NEXT I
2040 DATA ARSENAL, A VILLA, BIRMINGHAM, COVENTRY,
EVERTON, IPSWICH, LEICESTER, LIVERPOOL
2050 DATA LUTON T, MAN UTD, NORWICH, NOTTDM
F, NOTTS C, RANGERS, SHEFFIELD, STOKE C
2060 DATA SUNDERLAND, TOTENHAM, WATFORD, W BROM
, WESTHAM, WOLVES.
2070 RETURN

```

"WP"

The program produces a word-processing effect by shifting the entered text so that it fits a specified margin.

```

10 INPUT "SPECIFY LEFT MARGIN: "; L
20 INPUT "SPECIFY RIGHT MARGIN: "; R
30 PRINT ""
35 LI=0
40 IF L>R THEN 20
50 POKE 781, LI: POKE 782, L-1: SYS 65520: PR
INT L
60 POKE 781, LI: POKE 782, R-2: SYS 65520: PR
INT R
70 PRINT TAB(L); : FOR I=1 TO R-L: PRINT "-"
; : NEXT I
80 S$="": B$="": PRINT: PRINT TAB(L);
90 GET A$: IF A$="" THEN 90
100 IF ASC(A$)=13 THEN PRINT: PRINT TAB(L)
; : S$="": LI=LI+1: GOTO 90
110 X$=A$
120 IF ASC(A$)<> 20 THEN 150
130 IF LEN(S$)>0 THEN S$=LEFT$(S$, LEN(S$)
-1): GOTO 160
140 GOTO 90
150 S$=S$+X$

```

```

160 IF LEN(S$)<(R-L+1) THEN PRINTX$;:GOTO
 90
170 GOSUB 1000:POKE 781,LI+2:POKE 782,L:
SYS65520:PRINT TAB(L);S$:S$=B$:B$=""
180 LI=LI+1:POKE 781,LI+2:POKE 782,L:SYS
65520:PRINTTAB(L);S$;:GOTO 90
1000 IF RIGHT$(S$,1)=" " THEN S$=LEFT$(S$
,LEN(S$)-1):RETURN
1010 I=R-L+1
1020 I=I-1
1030 IF MID$(S$,I,1)=" " THEN 1040
1035 GOTO 1020
1040 SP=R-L+1-I
1050 B$=RIGHT$(S$,SP)
1060 I=I-1
1070 S$=LEFT$(S$,I)
1080 I=I-1
1090 IF MID$(S$,I,1)<>" " THEN 1110
1100 IFMID$(S$,I-1,1)<>" " THENS$=LEFT$(S
$,I)+" "+RIGHT$(S$,LEN(S$)-I):SP=SP-1
1110 IF SP=0 OR I=1 THEN 1120
1115 GOTO 1080
1120 IF SP>0 THEN I=LEN(S$):GOTO 1080
1130 RETURN

```


**THE CENTURY COMPUTER PROGRAMMING COURSE
FOR THE COMMODORE 64**



The Century Computer Programming Course provides the Commodore 64 with a manual worthy of the machine. Designed to provide the Commodore 64 owner with a full course in Commodore BASIC, it includes nearly 200 programs, plus subroutines, and hundreds of hints and tips on getting the most from your micro and its peripherals.

All you need is an hour a day and your Commodore 64. **The Century Computer Programming Course** provides the rest:

- ★ Step-by-step through the BASIC language
- ★ Understanding how a program works
- ★ How to design and structure programs
- ★ Debugging and tracing programs
- ★ Understanding how to use discs and printers
- ★ Each stage illustrated with example programs and exercises
- ★ A comprehensive program library, games to play while you learn, useful programs for home, school or college, subroutines to write into your own programs – all tested and ready to key in.

Designed by a team of educationalists, **The Century Computer Programming Course** is the ideal manual for the beginner, although the advanced programming techniques will equally appeal to the experienced Commodore 64 user.



ISBN 0 7126 0383 2
CENTURY COMMUNICATIONS LTD

£10.95