The Commodore Programmer's Route Map for the Commodore 64 and VIC 20
Commodore Basic

Gordon Hill
ACKNOWLEDGEMENTS

I am indebted to my parents, Mr and Mrs L.G. Hill for checking my use or abuse of the English language and the readability of the book, and to John Spence for his detailed checking of the text for technical meaning, grammar and punctuation. With the help of Peter Jennings and my children Martin, Timothy and Susan, who ran all the programs or examples given in the book, a number of errors and omissions were rectified. I am very grateful to them for their help.

W. Foulsham & Company Limited
Yeovil Road, Slough, Berkshire, SL1 4JH


Copyright © 1984 W. Foulsham & Co. Ltd.

All rights reserved.
The Copyright Act (1956) prohibits (subject to certain very limited exceptions) the making of copies of any copyright work or of a substantial part of such a work, including the making of copies by photocopying or similar process. Written permission to make a copy or copies must therefore normally be obtained from the publisher in advance. It is advisable also to consult the publisher if in any doubt as to the legality of any copying which is to be undertaken.

Photoset in Great Britain by
M. B. Graphics (Typesetting) Services
and printed in Great Britain by
St Edmundsbury Press, Bury St Edmunds, Suffolk.
PREFACE

There is now a wide variety of books available for the CBM home computers, but few of them really get stuck into the practical problems that afflict any programmer once he is past the beginner stage. This book can't do the work for you, but it can help you get things working in several ways.

Right at the start when you begin to use your machine with the help of your manual, you will often find an explanation difficult to follow. Reference to this book may help by giving another way of looking at things.

In Section 1 it gives you an idea how good programmers – professionals and amateurs – go about designing programs, and the types of problems they tend to run into.

Section 2 will help you with BASIC language problems, how to get the best use from it and what to look for if it fails to work the way you wanted it to. Examples for most of the keywords help you to understand their purpose or can be used as part of your program.

Section 3 gives you a number of working routines that you can include in your programs, or you can use the ideas to write similar routines of your own.

The rest of the book contains useful reference information missing from or not easy to find in the manual that comes with your machine.

Commodore BASIC, as used in the VIC 20 and CBM 64 is not very friendly in comparison with some other dialects around and in comparison with the all-singing – all-dancing BASICS used by many home computers. For instance, while CBM BASIC has about 60 statements and functions, some other machines have 100 or more. Many regard it as primitive or even archaic (it dates from about 1977!). Where else do you find a computer like the CBM 64 that requires half a dozen five figure POKEs to play a single note? But that's not the whole story. The limited vocabulary means a more compact interpreter (meaning more memory for your program) and often more efficient code – provided you understand the subtleties and nuances of the language.

Programmers who have used other machines will find Section 2 a rapid reference to the particular qualities of this BASIC and the examples here and in Section 3 illustrate many ways in which the language can be used to overcome its apparent limitations.

Your computer is a very powerful and versatile machine and this book will help you to get the best results from it.
# CONTENTS

## Section 1 Introduction

| The Aim of the Book       | 6   |
| Writing in BASIC          | 7   |
| Programming Hints         | 10  |
| Arithmetic                | 10  |
| Comparisons               | 11  |
| File Handling             | 12  |
| Creating Serial and Random Access files | 13 |
| File Buffers              | 14  |

## Section 2 Basic Functions

| ABS         | 17  |
| AND         | 17  |
| ASC         | 19  |
| ATN         | 20  |
| CHR$        | 20  |
| CLOSE       | 20  |
| CLR         | 21  |
| CMD         | 22  |
| CONT        | 22  |
| COS         | 23  |
| DATA        | 23  |
| DEF FN      | 23  |
| DIM         | 24  |
| END         | 25  |
| EXP         | 25  |
| FN          | 25  |
| FOR         | 25  |
| FRE         | 28  |
| GET         | 29  |
| GET#        | 30  |
| GOSUB       | 30  |
| GOTO        | 31  |
| IF          | 32  |
Section 3 Useful Routines

Introduction 65
Sort 66
Numeric Check 68
Random Groups 69
Base Converter 70
Business Forms 71
Tape Handling 79
Printing 81
Game 81
Graph Plotting 83

Section 4 Glossary of Terms

Addressing 87
Algorithm 87
Array 87
ASCII 87
Assembler 87
Base 87
BASIC 87
Binary 88
Bit 88
Boolean Logic 88
Buffer 88
Bug , 88
Byte 88
Code 88
Compiler 88

Default 88
Device 88
Function 88
Garbage 88
Hex 89
I/O 89
Input 89
Interface 89
Keywords 89
Kilobyte 89
Logic 89
Logical line 89
Machine code 89
Nesting 89
Octal 90
Output 90
Peripherals 90
Port 90
RAM 90
Real 90
ROM 90
Software 90
Statement 90
Structured programs 90
Syntax 90
Transparent 90
Variable 90
Virtual 91

APPENDIX A CONTROL CHARACTERS USED IN THE PROGRAMS 91

APPENDIX B CHARACTERS PRINTED FOR ASCII VALUES 92

APPENDIX C SOME USEFUL MEMORY LOCATIONS 93

APPENDIX D ERROR CODES 95
SECTION 1 INTRODUCTION

The Aim of the Book

This book is intended for those users or owners of a Commodore VIC 20 or CBM 64 who aim to be serious programmers and would like a more in-depth guide than that provided by the user handbook. It can be used in conjunction with the handbook, or by itself for those who are already acquainted with BASIC. Those who have spent some time programming in PET or VIC BASIC may still find that this book allows them to make fuller use of the facilities of the language. Converts from other BASICs will find that the detailed descriptions of Commodore BASIC functions given here provide the means for a rapid transition.

The book contains detailed descriptions of all BASIC functions in Section 2, and some general information and useful routines in Section 3. The detailed descriptions are intended to overcome the frustration felt by programmers who know where a program is going wrong through normal debugging methods, but are unsure exactly how the BASIC functions operate. For instance, many programmers have been held up for hours on the bug in the INPUT (q.v.) function. Section 3 gives a collection of commonly required subroutines, mostly built into programs, which can speed up development of a program either by the direct incorporation of routines into your program or by showing ways in which problems can be tackled.

Hardware is not dealt with in this book. However, one of the major plus points for the VIC 20 and CBM 64 is the ease with which Commodore supplied or compatible accessories can be attached, and the way in which they operate without either affecting the operating system or taking away memory from the BASIC RAM area. Most users will therefore be able to attach and use these for simple applications such as LOADing, SAVEing and PRINTing without too much difficulty. Of course, for more sophisticated applications and many disk applications, a deeper understanding of the peripheral operations is required. These are all explained in the relevant manuals, but are not easy for many people to understand and require careful study. For this reason, I have explained certain aspects of file handling at some length at the end of this section.
Writing in BASIC

Writing short programs in BASIC is not difficult with the aid of the guide which comes with the computer, or other introductions to the language. However, writing more complex programs speedily and efficiently is more difficult than most people realise until they have tried it themselves. Hours can go by like minutes when debugging programs.

There is no easy solution to this (that’s why trained professional programmers are so well paid), but it is very easy to ignore the problem until much time has been wasted. Even professional programmers working on large software projects have fallen into the trap of believing that the programs they have undertaken would be easier to write than they in fact turned out to be, causing expensive disasters.

This problem of writing large and complex programs was not realised until about 15 years ago because before that, most computer hardware was not powerful enough to develop really sophisticated programs. At that time the largest computer available to me at Edinburgh was the University Atlas which cost (as I recall) several hundred thousand pounds and had only 64 kbytes of main memory! The only ‘micro’ we had was an Olivetti P101 which was driven by an electric motor like the old fashioned calculators you may have read about. Then, in the late sixties, as powerful computers began to become available to more and more people, it became possible to create complex systems which could only have been dreamed about previously. Few people foresaw the problems of complex software and this led to the software crisis of the sixties when many sophisticated systems proved much more difficult to create than expected, or failed altogether.

To understand why this happened and why it will also happen to your programs, unless you take due precautions, think of a program of 100 lines. You can probably completely understand what you have written and can go almost at once to the piece of code that carries out a given task. To write two similar programs of 100 lines each will obviously take about twice as long as writing one. To write a program of 200 lines combining two 100-line programs requires not only the writing of the two modules but also the process of joining up the code and ensuring that the two functions do not interfere. In addition, it becomes more difficult for you to remember exactly what you have done and where.

To put it another way, for every line of code you write there is a probability (perhaps 1% for each line in a 100-line program) of it interacting undesirably with another line of code in the program (i.e. causing a bug). The larger the program, the more chance there is of this (perhaps 3% for each line in a 200-line program etc), until one reaches the stage where the creation of a new line is almost bound to introduce a bug somewhere else. Worse still, the changing of a line somewhere to cure a bug will create one or more different bugs elsewhere in the system, leaving a situation where a program can never be made to work properly. This problem requires the use of special design methods to minimise the interaction between different parts of the program.
You will probably think that all this doesn't apply to you and that you will just be careful not to make mistakes. Well, don't forget I told you so!

To write really complex programs requires a knowledge of design methods beyond the scope of this book, but a methodical approach following a few common sense ground rules will serve well for the average 100- to 200-line program and will pave the way for more complex professional type of programming later.

Design and procedural suggestions are as follows.

1. Before switching on the machine or opening a BASIC manual, work out what you want to do and write it down in pencil on any handy piece of paper. Check it over to see if all your required features are present, but resist the temptation to add features that can be added later after the initial program has been tested out.

Then work out the steps you will need to take to do it, also in pencil. Unless you are possessed of a very logical mind and have a clear idea of what you want to do before you start, you will find yourself making much use of the rubber; hence the suggestion that you design in pencil. However keen you are to get on with the coding, remember that it is much easier to play about with your ideas in pencil than to amend reams of code, particularly if you don’t have a printer. (For meaning of code as used in computer terms see the Glossary in Section 4).

If you want to check out a particular machine function - perhaps to time nested FOR loops - then by all means write a program directly onto the machine, but don’t be tempted to do serious work this way.

2. When you have worked out your ideas, make up some code (i.e. a series of BASIC statements) to do it in the simplest possible way, allowing for non-essential options to be added later, and don’t worry too much about sound and colour. Take particular care to get the code right on paper before typing it in, particularly if you don’t have a printer.

3. Type the code into your machine, SAVE it and try it. If (or in most cases when!) it doesn’t work, first check that your program has been typed in correctly and then that it follows the logic of your design by checking it line by line. As you make corrections on the machine, mark them on your program sheet so that you know exactly where you stand, or periodically produce a listing on your printer. Check any BASIC statements you are doubtful about under the appropriate heading in this book.

4. If the program still doesn’t work and you cannot see anything obviously wrong with it, try a process of elimination by using a toolkit (see below) or by placing STOPs or PRINTs at various places in the program and using CONT to continue the program from STOP to STOP. This will allow you to follow the path the program has taken and at each STOP you can
check all variables to see if they have the values you expect. It is worthwhile performing a "dry run" to see exactly what the variables ought to be at any given point in the program, by working manually through the BASIC, otherwise it is easy to convince yourself that the variables are what you expect simply because they look reasonable.

For example:  

```
10 A=1: B=2
20 C=A+B
25 STOP
30 D=A*B
```

when STOP is reached:  

?A should give 1  
?B should give 2  
?C should give 3

and if the program has run for the first time:  

?D should give 0

D, of course, should be zero because we have not yet set it. However, a very common problem that arises in mid program is that variables that should have been, say, zero have been set to another value elsewhere in the program. So be careful not to assume it is zero until you have checked it.

Remember that as soon as you change the program, even just by placing the cursor on a program line and pressing <RETURN>, all values are lost (become 0), strings are cleared and CONT is disabled.

The use of a toolkit, mentioned above, can simplify coding and debugging in a number of ways such as automatic renumbering of lines, tracing the progress of a program by displaying the line numbers on the screen, explaining the 'syntax error' in more detail, single-stepping the program line by line and dumping all variables to the screen. Your local dealer will have details of toolkits available for program development. Although useful, a toolkit will not compensate for poor program design.

5. Once you have confidence that your program is working as it should, then proceed to add colours and sound as required and tidy up or rearrange the screen. A thorough program check is now in order, using the program in as many ways as you can imagine, including invalid commands or replies. If possible get someone else to run through it as well. This person need have no knowledge of programming and can be a friend or someone in the family who would like to try out your program in the same way that arcade games are used by people with no knowledge at all about programming. When he has finished, be prepared to accept constructive criticism with a good grace, even if some of it is not in your opinion valid, otherwise you may not get the same help again!

At this point you will have discovered the "wouldn't it be nice if" syndrome. Make a list of all the nice goodies you would like in your program and select the ones you will use. Don't choose any that involve much rewriting.
of code already produced – this probably means that your original design was incompatible with the items you are thinking of adding, and avoid those which seem complicated or vaguely defined. Also, if you intend to try to sell your software, be very careful before adding extra pieces of accessory hardware to your machine because your program requires them. These extras will limit your marketplace. If you decide that some of the features you want require extensive modification of the existing program, the best course is usually to throw the program away and start again, perhaps using some of the routines from your old program again. Your efforts will not have been wasted because you will have learned a lot about design and coding, and will be less likely to make the same mistakes again.

Programming Hints

1. If possible (i.e. if sufficient memory exists – see below) use any variable for one purpose only. This reduces the possibility of a variable being set to an unexpected value for one part of the program while being used for another. It also makes debugging a lot easier as values set up throughout the program are not overwritten by other routines.

- Space occupied by variables is as follows:
  - Integers e.g. X% . . . . 2 bytes each
  - Decimal numbers e.g.
    - X . . . . . . . . . . . . . . . 5 bytes each
  - Strings e.g. X$ . . . . . . 1 byte per char. + 3
  - Arrays . . . . . . . . as above for each element, + 5, +2 per dimension

- With only 2 bytes available for integers, the values are restricted to between 32767 and -32768. Attempting to exceed these limits will lead to an ILLEGAL QUANTITY ERROR.

2. Use REM statements wherever possible to make the program readable and to record the purpose of each variable.

3. Use only one or two character variable identifiers e.g. AB, A1, AB%, AB$, AB$(X), AC%(D1%,D2%). VIC64 BASIC recognises only the first two characters but does not give an error if more than two are used. Thus BLACK and BLUE will be treated by BASIC as if they were the same identifier BL. Unfortunately, this means that the variables cannot easily be given meaningful names, which makes the code difficult to read. Hence, it is all the more important to use REM statements to explain what each variable does. One particularly frustrating problem that can occur, happens when a BASIC keyword is accidentally incorporated in a variable name. NOTE for instance will be interpreted as NOT E, the result of which is usually a syntax error on what appears to be a perfectly good line.

4. Number lines in multiples of 10 and make extensive use of subroutines numbered well apart from each other. This makes changing the program much easier.

Arithmetic

Arithmetic in BASIC is very versatile and easy to use, following normal mathematical rules. The meanings of such expressions as A+B, A-B, A*B/C are obvious and for more complex expressions, the order of precedence of opera-
tors is \(^{(\text{power of})} * / + - \langle >\). As an example, the expression \(A + B \times C \times 2\) will be evaluated by squaring \(C\), multiplying that by \(B\) and then adding \(A\). Changing the order of precedence is easily achieved by using brackets e.g. \((A + B \times C) \times 2\) will be evaluated by multiplying \(B\) by \(C\), adding \(A\) and then squaring the result. For use of \(<\) and \(>\) as operators see below under ‘Comparisons’.

Conventional arithmetic can take place on one of two variable types. The default type – the one normally used – is an ordinary decimal floating point number. If a % sign is added to the variable name, this becomes an integer type with no figures after the decimal point (e.g. \(AB\)%). Note that in Commodore BASIC \(AB\)% is a totally different variable from \(AB\) or \(AB\$\). You can use all three in your program without any risk of them conflicting except through use of the wrong type! Although, ideally, you should use integers for indexing of arrays or FOR/NEXT loops etc, in practice it is not possible on the CBM 64 or VIC 20. Arithmetic is actually slower using integers, as the interpreter converts integers to floating point numbers before processing them and then truncates them to integer form again! All you save is variable space (see above). However, if you decide to use a compiler to speed things up, this will benefit greatly from the inclusion of integer (%) variables wherever possible.

Although not arithmetic, strings can be “added” which have the effect of concatenating them (running them together) e.g. “J” + “OHN” will give “JOHN”:

```
PRINT "J" + "OHN"
```

prints JOHN as does

```
PRINT A$ + B$
```

where \(A\$\) contains “J” and \(B\$\) “OHN”

### Comparisons

Six comparisons are available:

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

Their meaning is obvious for numerics, but they can also be used for strings which are compared character by character, the test being performed on the ASCII (see Glossary, Section 4) value of the character concerned, so that words, for instance, are compared as they might be when consulting a dictionary.

- e.g. ABC comes before ABCD
- ABCD comes before ABD

Unlike some versions of BASIC, Commodore BASIC supports boolean algebra beyond the usual AND OR NOT combinations. For the uninitiated this can lead to some strange looking program lines e.g.

```
C=(A > 1)
```

gives \(-1\) if \(A\) is greater than 1, or 0 if \(A\) is less than or equal to 1 and similarly for the other comparisons. In all cases \(-1\) represents “true” and 0 represents false.

```
A=A+(A > 10)
```

will reduce \(A\) by one unless it has reached 10. The brackets are important as otherwise the “+” will happen first, leaving \(A\) as \(-1\) or 0 after the first operation of the line, depending on \(A\)’s starting value.
FOR X=1 TO 20
  A$=LEFT$(A$,LEN(A$)+(LEN(A$)>10))
NEXT X

will truncate string A$ by removing 20 characters but will not allow the string to drop below 10 characters.

PRINT CHR$(NR+48-(NR>9)*7)

will print any number between 0 and 15 as its hex character (e.g. 1 prints “1” and 15 prints “F”).

File handling

One area of program writing that seems to cause more confusion than any other – at least for beginners – is that of file handling.

In essence, there are three types of file that you can use on most microcomputers. (There are more types on larger computers.) They are:
- Program files
- Serial data files
- Random Access data files

Program files are those you use every time you save a BASIC program to cassette tape or disk. All the information is saved in line number order, and any ancillary information you need to store – such as names and addresses – could be kept in DATA statements, often at the end of the program.

The great advantage of a program file is that it is taken care of automatically by the computer operating system, with the LOAD, SAVE and VERIFY functions. Once the program is working, there is very little that can go wrong.

On the other hand, there are two very serious limitations. Firstly, program data files are very inflexible; you can’t enter any of the stored information while the program is running. If you want to amend, for example, an address or telephone number, you have to STOP the program, LIST it to find the appropriate DATA statement (not always an easy task!) and edit the offending program line.

The problem is even worse if you need to add information to the DATA statements, because then you need also to change the part of the program that READs the DATA – often a FOR/NEXT loop.

The other limitation is memory size. You can include sufficient information to fill the memory space available between the end of your BASIC program and the top of RAM. This might represent a limit of perhaps 100-150 names, addresses and telephone numbers (on a CBM 64) – none of which, remember, can be changed other than by someone who knows a bit about programming!

Serial data files

A much better way to store data that might require up-dating or expanding, is to use a serial data file. If you only have a cassette system, this is in fact your only alternative. As with a program file, a serial data file stores the data in a continuous and contiguous stream from beginning to end, and so is subject to the same memory space limitations as a program file. The big difference is that data can be accessed, amended, deleted and added to while the program is running, then re-saved to tape (or disk) in its revised form before the program ends.

Once the program has been written and debugged, virtually anyone can use it to store,
retrieve and amend the data without any knowledge of how the program was written.

Random access data files The third type of data file – random access – frees you from almost all memory limitations, but can only be used with disk systems. The number of records you can store is limited only by the amount of room on the disk – or on several disks.

If an individual record consists of, say,

NAME
TITLE
ADDRESS
TELEPHONE NUMBER
DATE OF BIRTH
DATE OF JOINING
OFFICE HELD
SPECIAL INTERESTS
DATE SUBSCRIPTION DUE

you could quite easily maintain around 600 such records on a modest, unexpanded VIC 20, provided you had a disk drive.

There are snags, of course. Random access file handling programs can seem more difficult to write and each record must be a fixed length so that the computer can work out where to find it. It also takes slightly longer for the computer to access each record because it has first to start up the disk drive, locate and load the record and then display it on the screen.

Even these minor irritations can be largely overcome or minimised by techniques such as using a serial file, loaded in at the start of the program, as an index to the random file on disk. So, in practice, random access files though largely ignored by home computer users, are often the obvious choice for serious applications.

Creating serial and random access files

Serial data files are the ones most commonly used and they are the only data files that can be handled on cassette. A serial file consists of a continuous stream of information from beginning to end, with no organisation other than that which the programmer structures when writing his data.

In just the same way that you have to open a filing cabinet drawer before you can start work on a particular file, so you have to OPEN a serial data file (or random access file) before you can start reading information from it, or before writing information into it. When you have finished with it you must then neatly CLOSE the file. If you do not, the computer does not know that you have come to the end and will not mark the file end or put the last of the data away from its working area (see below). It can not be OPENed for reading and writing simultaneously and must be opened at the beginning and read or written from beginning to end by the program. Thus, it is not possible to go backwards, so once the file is closed it is necessary to start again.

As mentioned above, when opening the file it can only be opened for reading or opened for writing, not both. When writing, it starts from the beginning and writes until closed. This means that once a file has been written it cannot be updated except as follows:

(a) OPEN the serial file for reading.
(b) Copy the contents to a temporary serial file using GET and PRINT, CLOSE it or read (INPUT № or GET №) the specified variables into a suitable area such as a large array.
(c) CLOSE the serial file.
(d) OPEN the serial file for writing – the
computer now assumes it is empty.

(e) Write (PRINT#) back the contents from specified variables or from the temporary file created after opening it for reading. This overwrites the file that was previously held there.

(f) Add the new information to the end of the file, or combined with (e) above, insert new or modified information as required.

This may seem strange to someone unused to computing files, but if it were not so, then every time a file was opened for writing the cassette tape would have to start at the beginning and run right through to find the end. Don’t worry if you can’t immediately understand the procedures above at a first reading, it will become clear once you set about using serial files and work out what you want to do using the I/O (Input/Output) instructions (OPEN#, CLOSE#, INPUT#, GET#, PRINT#)

The data would then progressively overwrite any other files following on the tape.

Random files are only to be found on disk. The disk manual explains these, and how to use them to build indexed files. Here the file is set up with a fixed size and with records ordered in a particular fashion, gaps being left in the structure for future inserts. Each record has an address or record number so records can be easily inserted, deleted and replaced. The number of records is limited only by the size of the disk, but practically for the sort of information described earlier it would be one record per sector (each sector having 254 bytes available for information).

---

**File Buffers**

All files are addressed from the computer using buffers. For programming purposes these buffers are said to be transparent to the user, i.e. the programmer does not have to worry about handling them as this is done by the computer. However it helps to follow the way programs operate, to understand the buffer system. To take the cassette case, each time a character or a string is sent to a cassette file, it does not make sense to start the cassette motor, wait for it to reach speed, send a character, then stop, and repeat the process for each item of data it wants to save. So, by convention, characters are saved up in a buffer and then when the buffer is full (192 characters for a cassette) the entire buffer is written to the tape in one go.

When the program finishes using the file, any information left in the buffer will not have been put on to the tape because it won’t have reached the total of 192 characters needed to cause an automatic writing of the contents of the buffer to tape. So we must make sure that the information in the buffer is not lost, by ‘forcing’ it to be written to tape. This is done by using the CLOSE command which writes away the rest of the data to be stored. The input and output to disk and the output to the printer are handled in the same way. Reading works in the reverse manner.

A few routines to handle disk access are shown above opposite.
110 OPEN 15,8,15
130 PRINT "OPEN" POKE 53281,1 POKE 53280,3
140 PRINT "OPEN "

150 PRINT "New main menu"'
160 PRINT "NEW"

170 PRINT TAB(10); "D - directory"
190 PRINT TAB(10); "S - scratch"
200 PRINT TAB(10); "M - new disk"
205 PRINT TAB(10); "E - exit"
206 PRINT TAB(10); "E - error status"
210 PRINT TAB(10); "Enter the letter:

230 GET k1$: IF k1$ = "" THEN 230
240 PRINT " Enter the letter:

250 IF k1$ = "D" THEN 300
260 IF k1$ = "S" THEN 1000
270 IF k1$ = "M" THEN GOSUB 1200
280 IF k1$ = "E" THEN GOSUB 1400
284 IF k1$ = "X" THEN CLOSE 15: PRINT ":CLOSE": END
290 GOTO 140
300 REM ****************************
310 REM *** disk directory ***
320 REM **************************
330 OPEN 15,8,0,1;"D": REM for dir
333 f=0
340 GET#1,a$,b$
350 GET#1,a$,b$: GET#1,a$,b$
360 GOSUB 530: GOSUB 600: GOSUB 600
370 d$=""+fl$+""+tp$+""+d$
380 PRINT "Directory of ";d$
390 PRINT "D"

410 GET#1,a$,b$
420 GET#1,a$,b$
430 GOSUB 530: GOSUB 600: GOSUB 600
435 IF fl$="" THEN 470
440 bl$=RIGHT$(bl$,2): GOSUB 740: REM Printing
450 f=f+1: IF f>16 THEN f=0: GOSUB 510: GOTO 300
460 GOTO 410
470 REM ******* ending *******
474 PRINT "B: blocks free"
475 GOSUB 510: close 1
480 GOTO 140
490:
500:
510 GET k1$: IF k1$ = "" THEN 510
1410 Print "Rename"
1420 Print "---"
1430 Input "old file name": on$
1440 Input "new file name": nn$
1450 Print#15, "rename0": nn$="on$
1460 GoSub 2000
1470 Return
1480 :
1490 : REM routine for check
1500 REM *************
1510 REM ** check disk **
1520 REM ****************
1530 Print "Error status"
SECTION 2 BASIC FUNCTIONS

This section includes a detailed description of the BASIC functions of the CBM language as applied to the VIC 20 or CBM 64. The language was derived from the PET BASIC which means that many programs written on PETs from 1977 onwards are not difficult to convert for use on the newer machines.

Conventions used in the syntax are as follows:
< and > indicates an entry of the type described within the angle brackets.
[ and ] indicate an optional entry.
() and , are typed as shown.

The examples below the syntax statement under each BASIC word use the conventions mentioned above.

BASIC words can be abbreviated by using the table in the manual or by remembering that the abbreviation consists of sufficient letters to allow the word to be differentiated from other words, with the last letter shifted. PRINT is an exception, for this the abbreviation is just ‘?’

**ABS**

**Description**
Gives a number without its sign.

**Syntax**
ABS(<numeric>)
e.g. ABS(-1) gives 1.

**Function**
Very simply, ABS leaves a positive number as it is and removes the sign on a negative number making it positive. Not often used, two examples are shown below.

\[ X = \text{SQR(ABS(Y))} \]
will prevent a square root being attempted on a negative number.

\[ X = \text{ABS(A-B)} \]
finds the difference between two numbers when it is not known which is the larger of the two.

**AND**

**Description**
Logical AND operator.

**Syntax**
<operand> AND <operand>
e.g. IF (A=1) AND (B=1) THEN.....
(the brackets are optional)

**Function**
AND is a binary function i.e. it works between two operands. It has two main functions:
1. As part of a logical test in an IF statement where AND indicates that both conditions must be satisfied e.g.

\[
\text{IF A} \leq 3 \text{ AND A} \geq 1 \text{ THEN 50}
\]

will go to line 50 if A lies between 1 and 3 inclusive.

2. As a logical function on binary numbers with the following truth table which applies to each bit in a basic numeric variable:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X AND Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
so that \( A = 80 \) AND \( 48 \) will give 16 as follows:

\[
\begin{array}{cccccccc}
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\hline
80 & & & & 0 & 1 & 0 & 1 \\
48 & & & & 0 & 0 & 1 & 1 \\
\end{array}
\]

AND gives 16 as \( 0 0 0 1 0 0 0 0 \)

where 80 represents, say, menu items 2 and 3 chosen and 48 a check for items 3 or 4.

The maximum size of binary number which can be handled thus is 16 bits or two 8 bit bytes. This is the size of a basic integer (%) variable between \(-32768\) and \(32767\) in decimal (being \(10000000\) \(00000000\) and \(01111111\) \(11111111\) in binary with decimal zero being \(00000000\) \(00000000\)). Note that here the leftmost bit determines if the number is negative or positive. A zero means it is positive; a 1 that it is negative. All numbers must be expressed in decimal as the VIC and CBM 64 have no facilities for handling binary numbers.

As an example of type 1 we might wish to verify entries on cassette, (read by an INPUT #) against the original entries on paper (read by an INPUT)

```
10 INPUT "CAR AND COLOUR";A$,B$
20 INPUT #1,C$,D$
30 IF A$=C$ AND B$=D$ THEN PRINT "ENTRY VERIFIED"
40 GO TO 10
```

Type 2 can be used for pattern matching when looking at a byte which does not represent a number but a pattern. For instance there may be 8 items on a menu screen from which one wishes to select items 2 and 5. This could be stored as a bit pattern \(01001000\), which is later matched against \(10000000\) to see if the first item on the menu was selected, against \(01000000\) for the second item and so on. In the first case \(128\) AND \(72\) gives \(0\) (try it!), so item 1 on the menu was not specified, and in the second case \(64\) AND \(72\) gives \(64\), so item 2 on the menu was specified. A typical menu might appear on the screen as follows:

```
SELECT OPTIONS FOR TEST
(up to 3) Selected

1 History
2 Geography
3 English
4 French
5 German
6 Chemistry
7 Physics
8 Biology

SELECT? 1,4,8
1 selected
4 selected
8 selected
```

See also the other logical operators OR and NOT.

A small menu program for the CBM 64 using the AND function is given below. To maintain a simple presentation, checking of numbers input for range and other detailed validation is not included, but this program could form a basis for a multiple choice menu selection.

```
10 REM MENU SELECTION
20 REM SET UP SCREEN WITH MENU OPTIONS DISPLAYED
30 PRINT CHR$(14)"\t"TAB(15)"#1 - 8/.."
```
40 PRINT "\$L-1 /\"\"L-1 \"\"L-1 (UP TO 3)"
50 PRINT "\"\"L-1 \"\"L-1 \"\"L-1 (UP TO 3)"
60 FOR I=1 TO 8: READ IT$
70 PRINT I:IT$: NEXT I: PRINT:"PRINT
80 INPUT "\$L-1 \"\"L-1 \"\"L-1 \"\"L-1"; SL(1), SL(2), SL(3)
90 REM ALLOW THE 3 SELECTIONS
100 FOR J=1 TO 3: PRINT "\$L-1 \"\"L-1 \"\"L-1 \"\"L-1"; BP=1/2
110 IF SL(J)=0 THEN 140
120 FOR K=1 TO SL(J): PRINT:BP=BP*2: NEXT K: MS=MS OR BP
130 PRINT TAB(20)"**" 3 AS \$""L-1 \"\"L-1 \"\"L-1 \"\"L-1"
140 NEXT J
150 REM TEST FOR ITEM(S) SELECTED
160 PRINT "\$L-1 \"\"L-1 \"\"L-1 \"\"L-1"
170 IF MS AND 1 THEN PRINT "1 SELECTED"
180 IF MS AND 2 THEN PRINT "2 SELECTED"
190 IF MS AND 4 THEN PRINT "3 SELECTED"
200 IF MS AND 8 THEN PRINT "4 SELECTED"
210 IF MS AND 16 THEN PRINT "5 SELECTED"
220 IF MS AND 32 THEN PRINT "6 SELECTED"
230 IF MS AND 64 THEN PRINT "7 SELECTED"
240 IF MS AND 128 THEN PRINT "8 SELECTED"
250 STOP
260 DATA "History", "Geography", "English", "French", "German", "Chemistry"
270 DATA "Physics", "Biology"

Identical program with printer set to lower case mode.

10 REM menu selection
20 REM set up screen with menu options displayed
30 PRINT chr$(14) "TEST MENU"
40 PRINT "SELECT OPTIONS FOR TEST(up to 3)"
50 PRINT tab(15) "\$L-1 \"\"L-1 \"\"L-1 \"\"L-1"
60 FOR J=1 TO 8: READ IT$
70 PRINT I:IT$: NEXT J: PRINT:"PRINT
80 INPUT "\$L-1 \"\"L-1 \"\"L-1 \"\"L-1"; SL(1), SL(2), SL(3)
90 REM allow the 3 selections
100 FOR J=1 TO 3: PRINT "\$L-1 \"\"L-1 \"\"L-1 \"\"L-1"; BP=1/2
110 IF SL(J)=0 THEN 140
120 FOR K=1 TO SL(J): PRINT:BP=BP*2: NEXT K: MS=MS OR BP
130 PRINT TAB(20)"**" 3 AS \$""L-1 \"\"L-1 \"\"L-1 \"\"L-1"
140 NEXT J
150 REM TEST FOR ITEM(S) SELECTED
160 PRINT "\$L-1 \"\"L-1 \"\"L-1 \"\"L-1"
170 IF MS AND 1 THEN PRINT "1 SELECTED"
180 IF MS AND 2 THEN PRINT "2 SELECTED"
190 IF MS AND 4 THEN PRINT "3 SELECTED"
200 IF MS AND 8 THEN PRINT "4 SELECTED"
210 IF MS AND 16 THEN PRINT "5 SELECTED"
220 IF MS AND 32 THEN PRINT "6 SELECTED"

ASC

Description
Gives the ASCII value of the first character in a string.

Syntax
ASC(<string>)
  e.g. ASC ("2") is 50
  ASC ("A") is 65
  ASC ("B") is 66

Function
ASC is the reverse operation to CHR$ and gives the ASCII equivalent number of the first character in a string. If no character is present an "ILLEGAL QUANTITY" error occurs.

A character in a string is stored in memory as a number. This number is found in the ASCII conversion table in your user manual, for upper case mode. Lower case mode is stored in exactly the same way and using the same numbers as for the equivalent upper case. This is because the machine does not recognise any difference in storage between upper and lower case mode, and the character displayed by a PRINT statement or otherwise on the screen depends entirely on the mode that the computer is operating in at the time. For convenience the complete table (excluding control codes but including lower case) is shown in Appendix A.

As examples:
  PRINT ASC(X$)–48
will give the value of a single digit number stored as a string character (if X$ is 2 the answer will be 2).

\textbf{PRINT ASC("JONES")}

will give only the the value of “J” i.e. 74. If other values within a string are required this can be done using MID$ e.g.

\textbf{PRINT ASC(MID$("JONES",2,1))}

will give the ASCII value of the second character of the string “JONES”.

\textbf{PRINT ASC(A$+CHR$(0))}

will prevent an ILLEGAL QUANTITY error occurring if A$ contains no characters.

\underline{ATN}

\textbf{Description}

Trigonometric arctan.

\textbf{Syntax}

\texttt{ATN(<numeric>)}

e.g. \texttt{ATN(2+A)}

\textbf{Function}

This is the complement of TAN, turning a tangent back into an angle (in radians) e.g.

\texttt{A=ATN(B)*180/\pi}

converts a tangent to an angle in degrees.

Your machine has a $\pi$ function, so $\pi$ is available direct and there is no need to derive it from the trigonometric functions. For interest, on computers without a $\pi$ function, pi can be derived from ATN as $\pi=\text{ATN}(1)*4$. This is because the angle whose tangent is 1 is 45 degrees or $\pi/4$ radians.

\underline{CHR$\$

\textbf{Description}

Converts a number to its ASCII equivalent.

\textbf{Syntax}

\texttt{CHR$(<\text{numeric}>)$}

e.g. \texttt{CHR$(50)$} gives the string “2” in both modes

\texttt{CHR$(65)$} gives the string “A” or “a”

\textbf{Function}

This is the reverse instruction to ASC, and the concept is explained there. The number must be between 0 and 255 or an ILLEGAL QUANTITY error results. A non-integer is truncated e.g. \texttt{CHR$(50.89)$} still gives 2. It is very useful for storing codes which cannot be printed directly as, for example, the <RETURN> key which is \texttt{CHR$(13)$}. It is also useful for changing the printer modes (explained in the printer manual). \texttt{CHR$(34)$} encodes the ” sign, which is otherwise very difficult to handle as the computer enters ” (quote) mode every time you type it on the screen.

See also ASC and the table of CHR$\$ values and strings in Appendix A.

\underline{CLOSE}

\textbf{Description}

Closes a file.

\textbf{Syntax}

\texttt{CLOSE <file number>}

e.g. \texttt{CLOSE 4} closes file no.4
**Function**

CLOSE informs the computer that the program has finished processing a file or device (e.g. printer), either permanently or temporarily. CLOSE is necessary so that at the end of processing the file can be properly cleared from the system. Therefore when writing to a file, any remaining information waiting in the file buffer to be processed is sent to the disk, cassette or printer and the end of file marker is written, if requested. The input-output (I-O) channel is also freed in the case of the printer (but see also PRINT# and CMD), which ensures that disk information (sent along the same line) is not held up by the printer awaiting information.

If the file number is given as a zero, a message such as NOT INPUT FILE ERROR will appear. If the file number is not an integer it will be truncated to an integer, and if a file number is specified that had not been opened, no action is taken but no error messages are given. If a string is given (instead of a number) a TYPE MISMATCH ERROR occurs.

CLOSE can be used directly or within a program. It is worth noting that closing files can take some time especially when closing the cassette, as time has to be taken to write the cassette buffer. In the case of serial cassette files, remember to rewind before opening again if required in the same program.

```
100 FOR I=1 TO 10:CLOSE I:NEXT
```

closes any files with numbers between 1 and 10 that have been opened.

Note that CLOSE on its own does not work on CBM computers, although available on many others where it closes all open files.

See also Section 1 on file handling and I-O.

---

**CLR**

**Description**

Clears all BASIC variables.

**Syntax**

CLR

**Function**

CLR clears all BASIC from RAM except the screen RAM (which means the screen is not cleared) and the BASIC program itself which is there as if it had just been loaded. This is not a clever instruction, it doesn’t close files for instance, so it is necessary to make sure everything is closed before using it. Assembler (machine language) and other locations such as sound and colour are not affected.
CMD

Description
Alters the output from the screen to another device.

Syntax
CMD <file number>[, <string>]
e.g. CMD4 reallocates output to the printer which has been previously opened as file number 4

Function
This very useful command used mainly in direct mode, but also available for program use, requires an OPEN to have been executed for the file required and then sends all system or program output to that file. The file may, of course, be a device such as a printer, with a file number associated with it by the OPEN command. Any PRINT or LIST data will be sent to the file until the CMD statement is reversed.

A PRINT# sent to the output, followed by a CLOSE command to write the buffer away and revert to the screen, ends the sending of the data.

An error such as SYNTAX ERROR cancels the CMD and returns output to the screen, but a PRINT# should still be used in direct mode to clear any device or it may not respond when the next output is sent to it, and the system will hang up, waiting.

OPEN4,4:CMD4:LIST will list a program
PRINT# 4:CLOSE4 reverts to the screen.

CONT

Description
Continues a STOPped program.

Syntax
CONT.
Function
This is normally used after a STOP to resume the program. It can also be used after the <STOP> key has been depressed or after an END. It won't work (CAN'T CONTINUE) if any program editing has taken place - and this includes pressing <RETURN> in the middle of a BASIC line even if no changes have been incorporated. It also won't work if an error has occurred either to stop the program or while, say, looking at variables after the program has stopped. See also STOP and END.

COS

Description
Trigonometric cosine function, provided in a right angled triangle by the equation $\cos \theta = \frac{\text{adjacent side}}{\text{hypotenuse}}$.

Syntax
COS(<NUMERIC>) e.g. COS(0) gives 1

Function
The cosine of an angle (in radians) is produced. See also TAN, SIN and the table of other trigonometric functions in your user manual.

DATA

Description
Provides data embedded in the program.

Syntax
DATA <constant>[,<constant>]....
e.g. DATA 10,JAMES,12,MARGE,"AT HOME:456,AT WORK:2292"

Function
For the way DATA is used see READ. The constants in DATA statements must be enclosed in quotes if they contain commas, colons or cursor control characters.

DEF FN

Description
Defines a mathematical function.

Syntax
DEF FN <name>(<numeric variable>)=<numeric>
e.g. DEF FNA(N)=N+1/N

Function
If you have a complicated formula used in several places, it is best defined at the start of the program with the DEF FN statement to save space later. This is frequently used with RND. A DEF FN should appear near the start of the program amongst other non executable statements such as DIM.

Some examples:

DEF FNA(N)=INT(RND(1)*N+1)

gives a random integer number between 1 and N whenever the function is called, perhaps by A=FNA(5) giving 1,2,3,4 or 5. The DEF state-
ment is in an early line, perhaps 50 or so, directly after the REM statements describing the program, and the calls (e.g. A=FNA(5) or A=FNA(7) or A=FNA(X)) occur whenever it is required to perform the function defined. So A=FNA(7) would be equivalent to A=INT(RND(1)*7+1). The letters after the FN enable you to have as many such formulae defined at the start of your program as you wish. For example:

```
DEF FNB(R)=R^2
```
gives the area of a circle whenever called by say AR=FNB(10) the area of a circle of radius 10.

```
DEF FNRD(X)=INT(((10^-N)*X)+0.5)/(10^-N)
```
rounds off to N decimal places and

```
DEF FNM12(X)=12*(X/12-INT(X/12))
```
converts a 24 hour clock into a 12 hour clock.

Remember to check that the function you want is not already explicitly defined in BASIC e.g.

```
SIN(X)/COS(X)
```
is equivalent to TAN(X) so there is no need for

```
DEF FNA(X)=SIN(X)/COS(X).
```

**DIM**

**Description**
Dimensions an array

**Syntax**
DIM <array name> (<number>,

<number>])......

e.g. DIM A$(25) is an array of 26 elements (0-25 incl)
DIM B(11,11) is an array of 144 elements in two dimensions

**Function**
An array is a collection of variables ordered one after the other in list form or matrix form if greater then one dimension. This makes it easier to process a list within a FOR loop or wherever indexing is used:

```
100 FOR l=1 TO 10
110 INPUT "NAME";NA$(l)
120 NEXT l
```
or

```
100 FOR l=1 TO 10
110 INPUT "NAME";NA$(l)
120 FOR J=1 TO 4
130 INPUT "ADDRESS";AD$(l,J)
140 NEXT J,l
```
will fill a list of names or names and addresses. Every array must be dimensioned either by using a DIM statement or by letting the computer do it for you. If there is no DIM statement, your array will be dimensioned with 11 elements from 0 to 10 or 121 (11 * 11) elements if a two dimensional array, 11*11*11 for a three dimensional array and so on the first time the program encounters it while running. If you dimension an array twice (i.e. if the program runs through a DIM twice or does its own implicit DIM before finding yours) an error REDIM'D ARRAY will occur. Any DIM statements should therefore be put at the beginning of the program directly after any introductory REM statements. Note that as multi dimensional arrays use a lot of space, two
dimensional arrays should be used with care, three dimensional arrays only on very special applications and four dimensional arrays will prove almost impossible to use anyway.

See also FOR.

END

Description
Stops a BASIC program.

Syntax
END

Function
END has the same function as STOP except that the message BREAK IN LINE ... does not occur. See STOP for further details.

END is a neat way of ending programs and it is better than just allowing the program to run off the end of the code. It ensures that the program ends where you want it to, so that if you put a subroutine or error handling routine for instance, beyond the end of the main program code it won’t get run unintentionally:

170 FOR N=1 TO 10:
180 GOSUB 300
190 CLOSE1:CLOSE4
200 END

300 REM SUBROUTINE

More than one END can appear in a program.

EXP

Description
Calculates powers of the constant e.

Syntax
EXP (<numeric>)
e.g. EXP(1) is 2.7182818

Function
The mathematical exponential constant e is raised to the power given by <numeric>. It is related to the LOG function which uses a base of e, for example EXP(LOG(10)) IS 10.

Useful in many mathematical equations, the mathematician alone understands!

FN

See DEF FN.

FOR

Description
Control statement for the generation of a loop.

Syntax
FOR <variable>=<numeric1> TO <numeric2>[STEP<numeric3>]

NEXT [<variable>]
e.g. FOR X=1 TO 10: ............ :NEXT

Function
FOR allows the program to loop round a section of code enclosed by the FOR and NEXT statements a controlled number of times, usually (numeric2-numeric1)/numeric3. If the optional STEP is not entered, a step of 1 is assumed.

After each NEXT, the variable is increment interrupts a test is made to see if it is greater than the control end <numeric2>. If it is, the program continues with the statement after
NEXT, otherwise the loop is entered again. Note that this means that the FOR loop will always execute at least once. This differs from many other BASICs where the FOR loop is not performed at all unless the conditions for it are satisfied at the entry point.

There is nothing to prevent manipulation of numerics 1, 2 and 3 within the loop if they are variables, but obviously care must be taken doing this as it makes the logic of the program more difficult to follow, and it sometimes becomes quite difficult to calculate how many times the loop will actually execute for any given set of conditions. There is also nothing to prevent you jumping out of the FOR loop if some other condition such as end of line or end of file is encountered.

Note that STEP can be negative as well as positive, allowing you to step down an array from the top as well as up from the bottom. It is essential to specify STEP when negative.

At the end of the FOR loop, if it completes normally, the <variable> has the value it had for its last test at NEXT, which will be more/less than the maximum/minimum specified by TO and will be the value the next STEP has incremented or decremented it to when finding that it was outside the bounds of the loop.

10 FOR X=1 TO 10
20 PRINT X
30 NEXT
40:
X is always 11 on exit.

Jumping out of a loop

100 LN=1
110 FOR X=1 TO 10
120 PRINT Y$(X);
130 IF POS(0) > LN THEN 300
140 NEXT X
150 PRINT:PRINT "OK IN"LN"CHARS":END

300 REM LINE FULL ROUTINE
310 PRINT:REM SKIP A LINE
320 PRINT "TOO MANY WORDS TO FIT IN"LN"CHARS":PRINT:LN=LN+1
330 GOTO 110

The value of X on exit depends on the lengths of the strings in the array Y$(X) when the code is run and the program continues to look for another NEXT after line 300 so design and debugging becomes that bit more difficult. Furthermore a loop variable used internally by BASIC is left on the BASIC stack and will not be removed until a CLR or RUN. A better way might be as follows:

100 LN=1
110 FOR X=1 TO 10
120 PRINT Y$(X);
130 IF POS(0) > LN THEN GOSUB 300:X=11
140 NEXT X
150 IF X=12 THEN 110

300 REM LINE FULL SUBROUTINE
310 PRINT:REM SKIP A LINE
320 PRINT "TOO MANY WORDS TO FIT IN"LN"CHARS":PRINT:LN=LN+1
330 RETURN

X is 11 on exit if the string fitted or 12 if it did not.
In all cases the variable following NEXT is optional and is used for clarity only. If not used the speed of the loop is much increased.

If an attempt is made to change the STEP after the loop is running, nothing happens, as once the STEP has been set up it cannot be changed even if the variable used to set it is changed. Thus the code below will not work to increase the size of the step from within the loop:

```
100 Y=1
110 FOR X=1 TO 1000 STEP Y
120 PRINT X;
130 IF X>9 THEN Y=10
140 IF X>99 THEN Y=100
150 NEXT
```

X is 1001 on exit.

Lines 130 and 140 alter Y but have no effect on the step.

A **FOR** loop can also be used for a simple delay, perhaps to give people a chance to read a screen. Delays can be better handled in other ways in machine language, but for **BASIC** programmers this is an acceptable method unless the code is to be compiled, in which case the delay will be much reduced due to the greater efficiency of the compiled code. Another method is to use the internal clock, which will work unless the cassette is in operation actually reading or writing.

**FOR loop delay:**
```
FOR X=1 TO 1000: NEXT
```

**Time delay:**
```
100 T1$="000000": REM SET INTERNAL CLOCK
110 IF T1<(5*60)THEN
110 REM WAIT FOR CLOCK
120 : 
```

(see under **TIME** for explanation of T1$ and T1)

This gives a precisely timed 5 second delay, whereas the only way to determine the time in a **FOR** loop is by experiment. However the **FOR** loop is a little more compact in code. Compiled code is much more efficient and therefore much faster and will thus shorten considerably such timing loops.

A **FOR** loop should normally be used for stepping up or down an array:

```
FOR I=1 TO 16: AR(I)=0: NEXT: REM CLEARS AN ARRAY
```

or:
```
10 DIM AR(16)
: : 
100 FOR I=16 TO 1 STEP -1: IF AR(I)<>0 THEN PRINT I
110 NEXT
```

Nested loops are very powerful and you will soon find occasion to use them. A two dimensional array, perhaps a draughts board can be set up this way:

```
100 REM CLEAR ENTRIES ON DRAUGHTS OR CHESS BOARD
110 FOR I=1 TO 8
120 FOR J=1 TO 8
130 DB$(I,J)=""
140 NEXT J,I
```
The maximum number of FOR statements that can be nested is 10, which is more than adequate. Only a program error should give the condition OUT OF MEMORY. If using multiple nested FORs be careful not to use the same variable in any two loops. If you do the result will be a NEXT WITHOUT FOR error.

An example of FOR loops nested 3 deep is given below:

```
10 REM THREECOM
20 REM PRODUCES ALL Sums OF ANY 3 OF THE NUMBERS ENTERED
30 REM STOPS WHEN OPTONAL MATCH FOUND
40 OPEN 4, 4
50 INPUT "NUMBER TO MATCH"; NM
70 N=1
80 INPUT "NEXT NUMBER"; A(N): IF A(N)=0 THEN X=N-1: GO TO 100
90 N=N+1: GOTO 80
100 CMD4
105 FOR N=1 TO X-2
110 FOR A=N+1 TO X-1
120 FOR B=A+1 TO X
130 Z=A(N)+A(A)+A(B)
140 PRINT Z"."".; IF Z=NM THEN PRINT "NM"A(N);A(A);A
150 NEXT;PRINT;NEXT;PRINT;NEXT
160 PRINT"END
```

Some of the examples in Section 3 also use nested FOR loops. See also NEXT.

---

**Function**

In BASIC, strings and other variables are created dynamically and also deleted and extended dynamically, which makes it impossible to arrange all the data neatly in memory. Some gaps and thus wasted memory are bound to occur. For instance if a string A$ is first created by A$="LONG AND COMPLICATED ERROR MESSAGE" and B$ is then created as another string, A$ and B$ are created one after the other in memory (string space, to be precise). This means that if A$ becomes the shorter "SHORT AND SIMPLE ERROR MESSAGE" then a few spare bytes appear, which cannot be used unless another string is the same length or less than the hole created. In this manner, as a program goes on manipulating strings, the free memory gets scattered about within the computer and this will stop BASIC from time to time when it runs out of space and has to do a garbage collection.

Garbage collection is the computer jargon for repacking all these strings, recovering all the free space and packing it into one area. FRE carries out this garbage collection and gives the amount of memory free. If the amount of memory spare when your program is running is small, it is often advantageous to do a FRE from time to time to prevent the program having to do it at a time not of your choosing. For example you might use FRE to do a garbage collection while a menu or instructions are being displayed on the screen then the delay caused by the garbage collection will not normally be noticed.

There is a bug in FRE which shows up on the CBM 64 in that FRE gives a negative number if more than 32k of BASIC memory is available. The correct answer is given by adding 64k (65536) to the FRE number or by:

**FRE**

**Description**

Performs garbage collection and gives free space available.

**Syntax**

FRE(<dummy>)

e.g. FRE(8)
FRE(S) − (FRE(S) < 0) * 65536

which you will need in a program if checking to ensure that the user has sufficient space to continue.

GET

Description
Examines the keyboard for character entry.

Syntax
GET <variable>[,<variable> .....]  
e.g. GET A$

Function
GET picks up any keyboard entry. If nothing has been entered since the last GET, an empty string (or 0 if the variable is a numeric type) is returned. GET does not wait and the program proceeds immediately with the next statement, so the usual way to use it is to return to the GET until something is found e.g.

10 GET A$: IF A$ = "" THEN 10

GET is more flexible than INPUT and has many advantages, but is usually a bit more complicated to program.

Commas, RETURNs and any other characters can be used.

String length can be up to 196 characters.

Undesired characters (perhaps cursor controls) can be ignored by the program, or certain keys or combinations of keys only accepted e.g.

10 GET A$: IF VAL(A$) = 0 THEN 10

or:

10 GET A$: IF A$ = "" THEN 10
20 IF VAL(A$) <> 0 THEN 50: REM ACCEPT A NUMBER
30 IF ASC(A$) > 63 AND ASC(A$) < 71 THEN 50: REM HEX A TO F
40 GOTO 10
50 REM PROGRAM CONTINUES WITH A HEX CHARACTER IN A$

Unlike INPUT, GET does not echo the input characters on the screen, which gives you the flexibility of echoing just the ones you want, using PRINT or POKE statements.

The expanded syntax GET A$,B$,... is not often used because in normal use GET will only pick up one character. However if there is a delay before the GET, this gives time for keys to be pressed and stored internally in the keyboard buffer, which has the characteristics of a queue, until the GET is encountered which reads them:

110 FOR X=1 TO 10000:NEXT
120 GET A$,B$,C$,D$...

A maximum of 10 characters can be held in
the keyboard queue: any more than 10 are lost unless GET has taken some from the front of the queue. Unlike INPUT, if GET finds a mismatch when a numeric is given a non-numeric key, SYNTAX ERROR appears and the program stops. It is therefore always best to use a string variable (e.g. A$ and then convert to a numeric by using VAL(A$) with a program check for non-numerics if required). For example, if “A” is entered in error instead of a number when the program is at a line containing GET X or GET X%, there will be a SYNTAX ERROR. In addition, the normal check shown above (10 GET A$:IF A$="" THEN 10) cannot work and must be replaced by something like:

10 GET A: IF A=0 THEN 10

which means that 0 cannot be entered as a valid reply.

GET #

Description
Reads characters singly from a file or device.

Syntax
GET # <file number>,<variable>[,<variable>....
  e.g. GET # 1,A,B$,C

Function
GET # works the same way as GET except that the data comes from a file or device instead of the keyboard. No characters are specially treated, but are simply read one at a time and placed in successive variable names. This means that any data separation characters inserted in the file when writing (perhaps commas and RETURNs) must be analysed separately by the BASIC program. Before GET # can be used the file being accessed must have been OPENed. As an example:

10 OPEN 8,8,8,"0:FILE1"
20 GET # 8,A$
30 IF A$=CHR$(13) THEN 50:REM LOOK FOR RETURN
40 LNS=LNS+A$:GOTO 20
50 PRINT LNS:LNS="":GOTO 20

CLOSE 8

When using GET # on cassette, the characters are read out of the cassette buffer which is refilled as required (see File Handling in Section 1) from time to time. The program pauses and the clock stops while tape input or output is in progress.

GOSUB

Description
Calls a subroutine.

Syntax
GOSUB <line number>
  e.g. GOSUB 500

Function
GOSUB causes control to be transferred to the line number specified and stores the return address as the statement following the GOSUB. Control is passed back when a RETURN is encountered in the subroutine e.g.

10 REM SET UP SCREEN
20 GOSUB 1000
30 PRINT N
100 END
1000 REM CLEAR SCREEN AND SET PAGE
1010 PRINT <clr>
      " >>>>>>>>> >>>>>>>>>"
1020 PRINT "PAGE";
1030 RETURN

Lines executed will be: 10 20 1000 1010 1020 1030 30.....etc Notice the END at line 100. This is a useful safety precaution at the end of any program before subroutines start so as to prevent the program dropping through from the main program into the subroutine accidentally and causing a RETURN WITHOUT GOSUB error. Subroutines can equally well be placed at the front of the program before main code with a jump from say line 10 to the main code.

GOSUBs can be nested i.e. a routine can call another routine, which itself can call another. It is easy to nest subroutines accidentally by forgetting to put RETURN in. This can lead to most peculiar results as the code being executed is totally unexpected, and the return from any other GOSUB so entered will be to the wrong place. This is because the return line number is picked off the stack as the last one put there.

GOSUBs can be nested up to an unbelievable 24 deep, so always check your RETURNS particularly carefully in complicated code.

The purpose of GOSUB is to allow frequently repeated code to be put in one place and called from any part of the program. It can also be used to make the code more understandable, by taking out detail from the main code into subroutines thus allowing both the main code logic and the subroutine logic to be followed more easily. Note that a GOSUB can point to any valid line including a REM line, but cannot point to an undefined line. Calculated line numbers cannot be used:

100 A=10:GOSUB A

is invalid.
See also ON.

GOTO

Description
Jumps to another part of the program.

Syntax
GO TO <line number> or equivalently GOTO <line number> e.g. GOTO 70

Function
GOTO is an unconditional jump to another part of the program without returning. This should be avoided wherever possible except for short loops (as in GET handling), for ON (q.v.) or for simple skip overs (see IF) or errors. This is because unrestricted use of GOTOs make the logic of a program more difficult to follow, particularly if used indiscriminately or with flags (see design hints in Section 1). The intertwining logical paths created by large numbers of GOTO statements result in what is commonly known as ‘spaghetti software’. If you find that your program seems to need this kind of structure (or more accurately non-structure) it is wise to have another look at the logic of your design and see if it can be rearranged. Unfortunately BASIC was not designed as a structured language, and you will find in Section 3 fairly frequent use of GOTO in some examples. Note that these are not used to jump at will about the code.
Highly structured programs are undoubtedly slower than well written unstructured code. However, in practice it is so much easier to write structured code and thus it is usually more efficient than all but the best of unstructured code. If you find performance is a problem consider using short sections of machine code within loops where the BASIC program spends much of its time, or alternatively use a compiler. No attempt is made to explain either in this book but the Commodore Programmers' Reference Guide for your machine is a good starting point for machine code, and compiler manuals are available with the compiler software. Use GOTO in an error situation to display an error and terminate processing; use GOSUB if just displaying an error message with some of error processing and continuing:

```
100 GET #1,A$:REM GET A CHARACTER FROM TAPE
110 IF VAL(A$)=0 GOTO 1000:REM
    SHOULD ONLY CONTAIN NUMERIC
```

or equivalently
```
110 IF VAL(A$)=0 THEN 1000:REM....
```

(omitting the GOTO after the THEN if you wish)

```
1000 REM ERROR EXIT
1010 PRINT "NON NUMERIC ENTRY"
1020 PRINT "PROGRAM ERROR- ENDS"
1030 END
```

See also ON.

### IF

**Description**
Conditional statement allowing branching.

**Syntax**
1. IF <condition> THEN <line number>
   or equivalently IF <condition> GOTO <line number>
   e.g. IF X=0 THEN 1000
2. IF <condition> THEN <statement>......
   e.g. IF X=0 THEN PRINT "FUEL EXHAUSTED":Y=15:....

Note that <condition> can be a complex condition containing logical operators AND, OR or NOT (see below)

**Function**
IF is followed by a condition which can be numeric or string and can include strings, numbers and variables related by logical
operators and comparisons, but must avoid data mismatches e.g.

50 IF A=0 AND B$="ABC" THEN  
  GOSUB 1000  

No ELSE is available in this BASIC so to achieve the same effect a GOTO has to be used e.g.

50 IF A=0 AND B$="YES" THEN  
  C$=T1$:GOTO 70  
  60 C$="0"  
  70 :  

Statements following a THEN in the same BASIC line are not executed if the IF condition is not satisfied, so in line 50 above if A is 1 the program moves directly to line 60. If only a small number of statements is required after a THEN, this is satisfactory, otherwise a GOSUB is required e.g.

100 IF A=0 AND B$="YES" THEN  
  PRINT "THIS CONDITION IS OK":C$=T1$:PRINT "AT TIME"LEFT$(T1$,2)"MINUTES"....  

cannot be coded as it stands because there is no provision in BASIC for lines longer than about 80 characters, and while some juggling can take place removing spaces and using abbreviations, this can be more trouble than it is worth in loss of clarity and problems with later editing. It should therefore be written:

100 IF A=0 AND B$="YES" THEN  
  GOSUB 1000

### INPUT

**Description**

A simple method of acquiring information from the keyboard, already formatted as a string or number.

**Syntax**

INPUT ["<prompt>";] <variable>[, <variable>],

  e.g. INPUT "ENTER YOUR NAME, AGE";N$;A  

**Function**

When a program reaches the INPUT statement the prompt, if any, is produced and a "?" is printed on the screen, the latter being produced even if there is no prompt. The program stops and awaits input which is stored away in the variable list and may be

- a string
- a number
- a series of strings and/or numbers separated by commas

depending on the variables requested by the program. In any case, if nothing at all is entered (except <RETURN>) the contents of the program variables are unchanged. For example, if N$ contains JOHN and A...
contains 35, then a single press of the <RETURN> key will leave them unchanged.

If a response is entered, all the variables must be given values e.g. JOHN,36.

Note that the use of the prompt is just a shorthand way of writing PRINT "PROMPT";INPUT.....

If too many replies are entered the program resumes giving a warning message "EXTRA IGNORED", and leaving out the extra items. The same effect occurs if a comma is inadvertently included, of course, as this is recognised as a string separator.

If too few are entered, the program will not continue but will produce a double prompt "??" to indicate that more entries are required.

A couple of examples illustrate the use of INPUT:

10 INPUT "AM I A CLEVER VIC";IN$ 
20 PRINT "I'M INTERESTED THAT YOU SAY"IN$ 
30 PRINT "DO YOU FIND THIS CONVERSATION INTERESTING" 
40 INPUT IN$ 
50 PRINT "I THINK "IN$" TOO" 
   : 
   : 

5 FOR N=1 TO 5: A(N)=0:NEXT 
10 INPUT "HOW MANY CHILDREN ":NO 
20 ON NO GOSUB 90,80,70,60,50 
30 PRINT "THE AVERAGE AGE IS""(A(1)+A(2)+A(3)+A(4)+A(5))/NO 
40 GO TO 5 
50 INPUT A(5) 
60 INPUT A(4) 
70 INPUT A(3) 
80 INPUT A(2) 

90 INPUT A(1) 
100 RETURN

As can be seen above INPUT is easy to program within its limitations.

Some errors are checked for you and give the message REDO FROM START. These errors are mismatch errors such as entering non numeric data in an integer or numeric field. e.g. "ABC" in A or A%. However, no error message is given for a decimal number being entered into an integer field, but it will be truncated. So if 5.1 is entered into A% it becomes 5.

The insert/delete key and the cursor left/right control can be used to modify the input string without explicit programming, in just the same way that you would edit a line in the listing. The up/down cursor controls can also be used amongst others, but the use of these leads to unpredictable results.

No commas or RETURNs can be used in strings because these are separators and terminators. GET must be used in such cases. The string length is also limited to 80 characters. There is a known bug in Commodore BASIC where information displayed on the screen before an INPUT statement can become incorporated in the message picked up by INPUT e.g.

10 INPUT "DO YOU THINK I'M CLEVER";IN$ 
20 PRINT "I'M INTERESTED THAT YOU SAY"IN$ 

IN$ will pick up "DO YOU THINK I'M CLEVER" on the VIC as well as any message typed in. The problem is not so apparent on the CBM 64 as the screen line length is 40.
The problem occurs when a previously PRINTed or INPUT string covers more than one line on the screen, with the INPUT prompt occurring on the same logical line i.e. for any INPUT statement with a MESSAGE greater than 22 characters (40 for the CBM 64), or for any PRINT statement longer than this length and terminated with a semi-colon. All similar constructions have the same problem so there is no simple "fiddle" to get round the problem, which should be avoided by using GET if necessary.

A program cannot be interrupted using the <STOP> key when a reply to INPUT is awaited. It is then necessary to press <STOP> and <RESTORE> and start the program again.

Other considerations
The prompt and input can be made invisible on the screen by printing the background colour control character at the end of the MESSAGE. e.g. on a white screen:

10 INPUT "PASSWORD{white}";A$
20 PRINT "{blue}OK"

where {white} is obtained by pressing <CTRL> and 2 together and {blue} using <CTRL> and 7.

so that the letters are the same colour as the background and thus invisible. Of course the VIC, screen, cursors etc all continue to function normally – you just can't see anything. When (or before) the next PRINT or INPUT is executed the letters must be made visible again by a suitable control character.

GET is much more flexible than INPUT and can perform all the INPUT functions. GET should be the normal choice of the advanced programmer, but as it requires more complicated programming techniques, INPUT will normally be used by those with less BASIC experience.

INPUT #

Description
A simple method of acquiring information from an open file, the screen or other external device, in string or numeric form.

Syntax
INPUT # <file number>,<variable>, [<variable>.....
  e.g. INPUT # 1,NA$(I),AG(I)
as part of a program loading a list of names and ages in arrays NA$ and AG.

Function
This works in the same way as INPUT (q.v.) and has similar restrictions. The variables are separated by a <RETURN>, comma or colon and the required number of variables will be picked up from the file or device rather than from the screen (unless the screen has been designated as an input device with a CMD). Anything left over in the input after the last separator will be lost, and input will be ignored until the next <RETURN>, so it is important to ensure that input data is properly structured with exactly the right number of separators before the carriage return and exactly the right types of input used to match the syntax of the INPUT statement e.g.

10 INPUT # 1,A,NA$,B

must be matched on the cassette by a series of number/ comma/ string/ comma/ number/...
<RETURN>. Note particularly that no error message is given when data is lost and the program continues to read from the tape as if nothing were wrong, so particular care is required here. Separators can be commas, semi-colons or colons.

If you run off the end of the file while INPUTting or try to read a string longer than 80 characters a "STRING TOO LONG" error appears and the program stops with a BREAK ERROR. It is therefore best to place your own end of data marker, say ZZZ on the file and look for it in the program that reads the data.

Quite complicated data patterns can be read e.g.

```
10 INPUT #1,NO$,Q$,A$:IF NO$="ZZZ" THEN GOTO 400:REM READ COMPLETE
20 IF Q$="ZZZ" OR A$="ZZZ" THEN 300:REM ERROR
30 FOR I=1 TO VAL(N0$):
   INPUT #1,W$(I)
40 IF W$(I)="ZZZ" THEN 300:REM ERROR
50 NEXT I
60 GOTO 10
```

reads a question, answer and a variable number of wrong alternatives to construct a multiple choice type question. The routine expects a terminating ZZZ in NO$. If it finds it anywhere else it is an error.

GET# is more flexible than INPUT# and will perform the same functions. If the format of the incoming data is well defined in strings it is best to use INPUT# but if in doubt about unwanted characters such as commas, colons, semi-colons or <RETURN>’s (other than as separators) then use GET#.

---

**INT**

**Description**
A simple truncation function which returns an integer value of a number or expression.

**Syntax**
INT (<numeric>)
e.g. A=INT(B)

**Function**
INT reduces the expression to the next lower whole number, i.e. for a positive expression the decimal point and figures to the right are removed; for a negative expression the next lowest whole number is returned. If a number is already an integer it is unchanged. A simple check that an integer number has been used as a reply to an input statement might be

```
10 INPUT "NO OF CHILDREN",NO
20 IF NO=INT(NO) THEN 40
30 PRINT "HOW DID YOU MANAGE THE"NO-INT(NO): GOTO 10
```

INT is a simple and straightforward function and can be used for rounding decimal numbers more simply than say LEFT$ e.g. to round a money amount of £12.345 in A to £12.35 in B:

```
10 B=INT(A*100+0.5)/100
```

Note that as the next lower whole number is selected by INT, to get rounding 0.5 must be added to the number to be rounded, thus bringing it over the next number if it is closer to that number than the one below.

Truncation is carried out by multiplying the number by 100, taking its integer and then
dividing that by 100 with the effect that the decimal place is shifted two places into the number:

\[ B = \text{INT}(A \times 100) / 100 \] (try with A as 12.345)

If B is replaced by B% in the above examples the INT function is unnecessary so this should be borne in mind as a simpler alternative if the variable is never required to be fractional, e.g. B% = A instead of the above example.

**LEFT$**

**Description**
A string manipulation function extracting or replacing part of a string starting from the left.

**Syntax**
LEFT$(<\text{string}>,<\text{integer}>)
e.g. LEFT$("JOHN SMITH",4) gives JOHN

**Function**
LEFT$ takes the leftmost <integer> characters of the string. The integer must evaluate to between 0 and 255. Useful in truncation or as part of a string analysis routine.

\[ A$ = \text{LEFT$}(B$, 1) \]

takes the leftmost character of a string – useful for checking a Yes/No type answer by looking at the first character.

Quite complex syntax can be used here with good effect e.g.

\[ A$ = \text{LEFT$}(B$,\text{LEN}(B$-1)) \]

removes the last letter of a string.

\[ 10 \text{ IF } \text{LEN}(B$)<20 \text{ THEN } 50 \\
20 \text{ PRINT LEFT$}(B$,20) \\
30 B$=\text{RIGHT$}(B$,\text{LEN}(B$)-20) \\
40 \text{ GOTO } 10 \\
50 \text{ PRINT } B$ \]

prints the data in B$ in widths of 20.

Another example shows how to prevent words spilling over from one screen line to another on the VIC. (For the CBM 64 use 40 instead of 22.) Line 20 takes the leftmost 22 characters of the text in C$. Line 30 removes part words from the line by shortening to the first space. Line 50 replaces the original text string with the new version which has had removed the part already PRINTed. This process is repeated by returning to line 10.

\[ 10 \text{ IF } \text{LEN}(C$)<22 \text{ THEN PRINT } \\
C$:\text{GOTO } 70 \\
20 D$=\text{LEFT$}(C$,21) \\
30 \text{ IF } \text{RIGHT$}(D$,1)<>' THEN \\
D$=\text{LEFT$}(D$,\text{LEN}(D$)-1):\text{GOTO } 30 \\
40 \text{ PRINT } D$ \\
50 C$=\text{RIGHT$}(C$,\text{LEN}(C$)-(\text{LEN}(D$)+1))) \\
60 \text{ GOTO } 10 \\
70 \text{ END } \]

LEFT$, RIGHT$, MID$, and LEN make up the range of BASIC string handling functions and, as hinted above, can be programmed to a
quite sophisticated level without necessarily writing a lot of code. However, as can also be seen above, it can be quite difficult to follow the logic, so REMs or separate program notes are in order. Also note that after considerable string manipulation the memory can become quite cluttered with discarded bits of string and the occasional FRE can be useful (q.v.).

IF LEN(A$)=1 AND A$="X" THEN......

tests for the single character X.

110 FOR i=1 TO LEN(A$)
120 IF MID$(A$,i,1)="E" THEN
130 CN=CN+1
130 NEXT

counts the number of Es in string A$

### LEN

**Description**
Gives the length of a string.

**Syntax**
LEN (<string>)
e.g. A$="19 CHARACTER STRING"
LEN(A$) gives 19

**Function**
LEN counts all characters including spaces and non printing characters. See LEFT$ and RIGHT$ for some useful examples of string manipulation using LEN. Other uses are e.g.

### LET

**Description**
Assignment.

**Syntax**
[LET] <variable>=<expression>
e.g. LET A=5 or LET A$=B$+C$

**Function**
LET assigns a value or a string to a variable and is a very common statement. The word LET is optional and is therefore almost invariably left out to save typing and memory. This
is unfortunate, particularly for beginners as for instance A=A+5 is nonsense to the ordinary person, whereas the meaning of LET A=A+5 is much more obvious.

### LIST

**Description**
Lists all or part of a program.

**Syntax**
LIST [[<first line>]–[<last line>]]
e.g. LIST lists the entire program
LIST 100 lists line 100
LIST 100–200 lists lines 100 to 200 inclusive
LIST–100 lists lines up to 100

**Function**
This LISTS required lines on the requested device (usually screen but can be printer). If used within a program, LIST must be the last instruction, as after a LIST, control is returned to the BASIC system and READY is displayed.
LISTing about 10 lines at a time on the VIC and 20 on the CBM 64 is enough to fill the screen for a normal program.
Listing to a printer can be carried out as follows:

```basic
OPEN 4,4:REM OPENS CHANNEL TO PRINTER WITH ADDRESSING 4
CMD4:REM PASSES OUTPUT TO PRINTER
LIST
```
and then, when the listing is complete....

### PRINT

```basic
PRINT #4:REM PASSES OUTPUT BACK TO SCREEN
```
by terminating output to the printer using a null PRINT# statement. It is important not to forget the last line, as otherwise output, including the READY prompt, continues to go to the printer. The printer is normally device 4, but this can be changed to 5 by a switch on the rear, when the example above will contain 5 instead of 4.

### LOAD

**Description**
Loads and optionally RUNs a program.

**Syntax**
LOAD ["<filename>"][,< device>][,<address>]
e.g. LOAD loads the next program on cassette.

```
100 LOAD A$,8 loads the program from disk, the name of which is set up in A$ by the currently running program, and RUNs it.
```

**Function**
This loads a program from a program file on the specified device (usually 1 for cassette and 8 for disk). If executed from within a program, on completion of the LOAD, the newly LODed program will run, which does not of course apply in direct mode. This allows programs to be loaded serially and run one after the other, or even looped round to run one or a series of programs repeatedly. Obviously in the case of a loop the cassette has to be rewound.
and at some point the program must pause to allow this. On LOAD, any previous program in memory is lost, whether it is a direct LOAD or a LOAD from a program. As MERGE is not directly available, there is no built in single command way of adding one program to another already in memory (but see 'MERGE'). The LOADing of a menu program can allow the selection of other programs e.g.

```
110 IF CH=1 THEN
    A$="PROGRAM1"
120 IF CH=2 THEN
    A$="PROGRAM2"
130 IF CH=3 THEN
    A$="PROGRAM3"
140 LOAD A$,8
```

will load A$ from disk and run it. It is also possible to do this from cassette by omitting the ‘8’ but it may take some time if the required program is some distance down the tape.

Unfortunately if the second program is larger than the first, the system crashes! Yes, it goes horribly wrong because two locations in the BASIC area have not been set properly. These locations are the pointer to the start of BASIC variables at 45 and 46 in both the CBM 64 and VIC. These must therefore be reset by e.g.

```
POKE 45, <number1>:POKE 46, <number2>:CLR
```

where number1 and number2 are found by PEEKing 45 and 46 after writing the program. These numbers determine where the BASIC variable area ought to be. The line above is included as the first line of your program if it is chained (LOADed from another program) and must be changed whenever the program is modified.

If the filename is not entered, the first file on the cassette is LOADed or an error message given if the disk is addressed. If the filename is "*" the first file on the disk directory is loaded, with a FILE NOT FOUND error if this is not a program file. If the device is not entered, the cassette is assumed.

The secondary address is not normally used. An address of 1 will, on the CBM 64, cause the program to be loaded at the memory location from which it was saved, rather than automatically in the BASIC area. This could be useful if the BASIC pointers have been moved about to allow some machine code.

A load from cassette followed by an immediate run can be achieved by simply pressing <SHIFT> and <RUN> together.

---

### LOG

**Description**
Natural logarithm.

**Syntax**

```
LOG (<numeric>)
```

e.g. LOG (10) gives 2.30258509

**Function**

LOG finds the natural (base e) logarithm (usually abbreviated to log) of a number or expression. This is not the same as the logs shown in school log tables or most slide rules which are to the base 10 and can be obtained as follows:

```
LOG (A)/LOG (10) returns log base 10 of A
```

To return from a log to the original number (i.e. antilogarithm usually abbreviated to anti-
log) multiply by the base to the power given by the log. As an example using base 10 logs, if the log is 1 and the antilog is wanted this is $10^1$ (10 to the power of 1) which is 10. If the log is 2 the antilog is $10^2$ which is 100.

$$AL = BS^2$$

Where $AL$ is the antilog required
BS is the log base
LG is the log

(MERGE)

Description
Merges two programs.

Syntax
None.

Description
This facility does not exist in the CBM BASIC. However it is extremely useful to be able to write programs in chunks and then to MERGE them. The easy way of doing so is to buy a tool-kit such as the VIC 20 programmers’ aid cartridge, otherwise it can be done as follows:

Write your first program to cassette as a serial file e.g.

```
OPEN 1,1,1,"PROGRAM1":CMD1:
LIST
```

followed by

```
PRINT #1:CLOSE1
```

Then to merge with another program, rewind the cassette and load it using

```
POKE 19,1 setting the input prompt flag
```

and finally clear the screen and move down three cursor positions to where the cursor would be after a PRINT CHR$(19) and then

```
PRINT CHR$(19):POKE 198,1:
POKE 631,13:POKE 153,1
```

where

```
PRINT CHR$(19) clears the screen
POKE 198,1 sets 1 character in keyboard buffer queue
POKE 631,13 sets that character to <RETURN>
POKE 153,1 sets default input from cassette instead of keyboard
```

thus simulating someone typing into the machine the listing that was saved to cassette. End with a CLOSE 1 after ignoring the SYNTAX ERROR.

MIDS

Description
A string manipulation function extracting or replacing any contiguous part of a string.

Syntax
MIDS$(<string>, <numeric1>[,<numeric 2>])

e.g. MIDS$("JOHN SMITH ESQ",6,5) gives SMITH

Function
MIDS$ takes the first position of the string to be extracted from the left as numeric1 and extracts numeric2 characters beyond that. If numeric2 is greater than the remaining string length, or is not entered, the rest of the string is taken. Numeric2 must be between 0 and 255 and numeric1 between 1 and 255. No syntax
error is returned if numeric2 is 0 but obviously nothing will be returned by MID$.

MID$ can be usefully used in string indexing e.g.

```
10 A$="JANFEBMARAPRMAJUNJULAUGSEPOCTNOVDEC"
50 INPUT "MONTH NUMBER";C
100 B$=MID$(A$,C*3-2,3)
110 REM EXTRACTS THE MONTH USING THE MONTH NO. IN C
```

See also LEFT$ and RIGHT$

### NEW

**Description**

Clears program and deletes variables.

**Syntax**

```
NEW.
```

**Function**

NEW can be used directly or within a program, in which case it clears everything BASIC including itself. It can therefore be used as a rather primitive security device by clearing a program from memory under certain conditions, although additional line blanking (so that the line containing the NEW is not visible) is also required, if the program is not compiled e.g.

```
100 INPUT "DATE (DDMMYY)"; DAS
110 IF VAL(RIGHT$(DA$,2))>84 THEN NEW
```

Programs containing these statements will automatically delete if used beyond 1984. The cassette or disk copy is, of course, not affected.

CBM 64 sprites, machine code and high resolution graphics are not cleared and of course POKEs that have been performed remain in effect, so in many cases where these have been used it may be necessary to press <STOP> and <RESTORE> or even switch the machine off and on again.
**NEXT**

**Description**
Gives the repeat point in a FOR loop.

**Syntax**
NEXT [<counter>],[<counter>]......

**Function**
NEXT returns the program to the most recent FOR statement that has not already been matched with a NEXT. If this does not correspond to the FOR identifier, a NEXT WITHOUT FOR error occurs i.e. NEXT on its own cannot cause this error if there is a FOR current as it will automatically match that FOR and assume that the identifier in the FOR statement is the one intended in the NEXT. However, it is safer (but much slower) to use an identifier (e.g. NEXT I rather than just NEXT) so that the BASIC interpreter can check the syntax and thus your logic.

If the exit condition has been reached, the program will continue by returning to the FOR to test it and then skipping to the code directly after the NEXT.

See FOR.

**NOT**

**Description**
Logical NOT.

**Syntax**
NOT <operand>
e.g. A=NOT B

**Function**
NOT is a unary logical operator, i.e. it works on just one operand, and has two functions.

1. As an operator on an item, in which case the item is converted to an integer between -32768 and 32767 and the bit pattern reversed e.g.

   A=NOT 32767 gives A as -32768.
   32767 binary is 01111111 11111111
   -32768 binary is 10000000 00000000

   The machine interprets the left hand (most significant) bit in an integer as a sign bit. If set, the integer is negative, otherwise positive.

2. As part of an IF statement, in effect IF NOT...THEN...

   e.g.

   100 IF NOT A$="YES" THEN PRINT "A$ IS NOT YES"
   or
   10 IF NOT (A=B) THEN A=A+1: GOTO 10

   is equivalent to

   10 IF A=B THEN 30
   20 A=A+1: GOTO 10
   30 ....

   In line 10 (A=B) evaluates as -1 if true and 0 if false – see Comparisons in Section 1.

   NOT is best used purely as a logical operator in the two senses explained above, and not as shorthand for arithmetic operations except on integers, as ILLEGAL QUANTITY errors will ensue if numbers above 32767 are used and decimal numbers will be truncated. NOT X% is equivalent to -X%-1 between 32767 and -32768 because the bit pattern in the 16 bit integer is reversed. (The same applies to other
numeric variable types provided they contain only integers.)

### Description
A multiple switch.

### Syntax

```
ON <variable> {GO TO} <line number> {GOSUB}
[,]<line number> ........
```

**e.g.** ON A GOSUB 1000,1100,1200,1300,1000,1200

### Function

**ON** is a software switch that allows a multiple branch in one instruction and is very useful for table or data driven programs where the path is not determined by program logic but by a table or by data read from a file or input from the screen; e.g. if a screen is displayed as

```
MAIN MENU
```

<table>
<thead>
<tr>
<th>1. LOAD STOCK FILE FROM CASSETTE/DISK</th>
</tr>
</thead>
</table>

```plaintext
2. SAVE AMENDED FILE TO CASSETTE/DISK
3. ENTER NEW STOCK ITEMS
4. AMEND STOCK AMOUNTS
5. REVIEW
6. REPORTS
```

**SELECT ITEM REQUIRED?**

then code to perform the relevant functions could be selected as below (repeating the code that produces the last line above)

```plaintext
10 INPUT "SELECT FUNCTION REQUIRED";MN$
20 ON VAL(MN$) GOTO 100,200,300,400,500,600
30 PRINT "INVALID CHOICE": GOTO 10
```

or from a file...

```plaintext
10 GET #1,MN
20 ON MN GOTO 100,200,500,100,300,400,100
30 PRINT "ERROR ON INPUT TAPE"
```

```plaintext
40 END
```

If the variable (e.g. MN) is 0 or greater than the highest number in the list, the program will continue with the next statement which displays an error message and stops. (It could return to the selection with a GOTO.) If it is
greater than 255 or negative an ILLEGAL QUANTITY error occurs. If it is not an integer
the fractional part is ignored.

ON replaces a whole series of IF statements
provided that the selection criteria can be
converted to a sequential series of numbers e.g. if the reply to a menu was A B C or D in MN$.

MAIN MENU

A. LOAD STOCK FILE FROM CASSETTE/DISK
B. SAVE AMENDED FILE TO CASSETTE/DISK
C. ENTER NEW STOCK ITEMS
D. AMEND STOCK AMOUNTS
etc

110 N=ASC(MN$)-64
120 ON N GOTO 200,300,400,500
130 PRINT "INVALID REPLY" .......

or even

FILE MENU

C. CREATE FILE
K. KILL FILE
A. AMEND FILE

110 N=ASC(MN$)-64
120 ON N GOTO 300,130,130,
130,100,130,130,130,130,130,200
130 PRINT "INVALID REPLY" .......

will handle a reply of C, K or A

Unrelated variables and complex conditions cannot be handled and in these cases IF must
be used or a table of values set up.

Although in general programming terms the
use of GOTO is not recommended, in the case
of ON it should normally be used rather than
GOSUB as otherwise invalid entries cannot
easily be checked in the statement following
the ON statement.

OPEN

Description
Opens files or channels to peripherals.

Syntax
OPEN <file number>,[<device>]
[, <operation >][,"< file name>
[, <file type >][, <mode >]"]

  e.g. OPEN 1,1,0,"DATA" opens a file on
cassette named DATA in READ mode and
gives it a number 1.

Function
OPEN sets up a channel for the transfer of
information to or from the computer. This can
then be used by CMD, GET#, INPUT# and
PRINT# statements until it is closed with a
CLOSE.
The file number is the number that all other file handling statements, such as the ones above, will use to identify it. It should be between 1 and 127. Numbers over 127 can be used but were intended for I/O use and not as files. A good convention is to use the same file numbers for devices as the device number for cassette screen and printer and then numbers from 8 upwards for disk files.

The device can be 1 for the supplied cassette, 2 for a second cassette (which must be Commodore or Commodore compatible as unlike other home computers ordinary cassette players are not usable), 4 or 5 for the printer depending on a selector switch on the printer, and 8 upwards for disk drives. No entry defaults to 1 (the cassette).

The operation indicates what the file will be used for:

0 = OPEN for reading from cassette – this is the value if nothing is entered.

1 = OPEN for writing to cassette without end of tape (or file) marker.

2 = OPEN for writing with end of tape marker.

(The end of tape (or file) marker can be used to prevent accidentally reading past the end of the data on a file on a subsequent read.)

2–14 = secondary addresses for use with the disk unit.

The file name is a string of between 1 and 16 characters which is the name that will be set up on the disk or cassette when writing, or searched for when reading.

The file type is sequential (SEQ) if not entered. The other types are relative (REL), random which is only applicable to disk files or program (PRG) which is used by the system during LOAD and SAVE etc.

The file mode is only used for disk and can be R for read (the default if no mode is entered) or W for write.

All this may seem rather complicated, and for someone unused to microcomputer files on disk it can be rather difficult. Careful study of the disk manual and an ability to understand file handling concepts is required and is beyond the scope of this book. It is in fact all there in the disk manual but will require some perseverance. However for cassette and printer use it is not too much of a problem:

OPEN 1,1,0,“DATA” for read and OPEN 1,1,1,“DATA” for write are all that is required for the cassette. OPEN 4,4 opens a normal printer after which information can be sent to the screen using CMD or PRINT# 4. See LIST for the procedure needed to list a program to the printer.

See also CLOSE, CMD, PRINT#, GET#, INPUT#.

Description
Logical OR operator.

Syntax
<operand> OR <operand>
  e.g. IF (A=1) OR (A=2) THEN...(brackets are optional)
Function
OR is a binary function, i.e. it works between two operands and has two functions:

1. As part of a logical test in an IF statement where OR indicates that either condition can be satisfied to satisfy the IF e.g.

   IF A=B OR A=C THEN 50.

   IF A%=1 OR A%=2 OR A%=3 THEN 50

will go to line 50 for any integer between 1 and 3 inclusive.

10 INPUT "MENU NUMBER";MN%
20 IF MN%<0 OR MN%>5 THEN 100

will check that a menu entry lies between 1 and 5. Line 100 is the start of the error routine. See ON for a description of menu handling and some examples. ON can be clumsy where the spread of possible replies is too great and OR is therefore a possible alternative as is a series of IFs. Beware that occasionally this OR test does not work due to a bug in BASIC

2. As a logical function on binary numbers with the following truth table which applies to each bit in the integer part of a BASIC variable:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X OR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

so that A=80 OR 48 will give 64+32+16 = 112

128 64 32 16 8 4 2 1 - decimal values of bits

The maximum size of binary number which can be handled in this way is 16 bits or two 8 bit bytes. This is the size of a BASIC integer variable and in decimal is between -32768 and 32767 (being 10000000 00000000 and 01111111 11111111 with decimal zero being 00000000 00000000 of course). All numbers have to be expressed in decimal as unfortunately the VIC and CBM 64 (in common with most BASICSs) have no handling of binary numbers except through decimal numbers.

There are many different ways of carrying out branching comparisons and binary logical functions. Once your design has clearly stated the logic required, a truth table should be set up to show what is required. This is then implemented with OR NOT AND statements as required, in the simplest fashion. Equivalent statements abound:

(a) IF A%=0 OR A%=1 OR A%=2 OR A%=3 THEN...
(b) IF A%<=0 AND A%>=3 THEN...
(c) IF NOT(A%<0 OR A%>3) THEN...

are all equivalent, and the one used should be the one that is clearest to understand and reflects the design. A choice of several unrelated numbers (or strings) requires format (a), a range requires format (b). It is difficult to think of circumstances in which format (c) would be preferable.

It is also easy to write incorrect logic or even meaningless logic:
IF A <> 4 OR A <> 5 THEN....

is always true but the computer is not smart enough to recognise this and give an error message, so you have to do it by checking your code by hand or when testing the flow of logic during debugging.

See also AND, NOT.

PEEK

Description
Examines a memory location.

Syntax
PEEK(<location>)

E.g. PEEK(1024) gets the contents of the first screen location on a CBM 64.

PEEK(7680) gets the same for an unexpanded VIC.

Function
If PEEK is used on a valid memory location, the value of the bit pattern of the 8 bit byte is found and expressed as a decimal number. A location of less than 0 or more than 65535 causes the error ILLEGAL QUANTITY to appear. If on the VIC the memory location is not valid (no memory exists at this location) an undefined value will be returned, but no error message, so it is important to check that the memory you have and your PEEKs and POKEs are consistent. PEEK and POKE are the BASIC way of direct communication with machine memory and therefore they are extremely varied in usage, including being able to get into and modify the BASIC operating system. This is beyond the scope of this book. Common uses will be found in your user manual, so they are not repeated here. Remember that <location> can be an expression, so indexing and logical operations are possible:

PEEK(1024+(V-1)*40+(H-1))

gives the screen contents at location H position across the screen and V vertical lines down the screen for the CBM 64.

It is sometimes convenient to show location numbers in 2 bytes as this shows more clearly where they are in memory. For instance upper case characters on the CBM 64 start at 53248 or 208*256+0, this latter being easier to remember and use for offsets to the various characters in the character set. The formula for PEEKing a 16 bit address is

\[ AD = PEEK(X) + PEEK(X+1) \times 256 \]

for two adjacent locations with the least significant or low order byte first.

See also GET# which can read from the screen, and WAIT which waits for a location to change.
POKE

Description
Sets the contents of a memory location.

Syntax
POKE <location>,<integer>
e.g. POKE 1024,1 sets the character A into the top left hand corner of the CBM 64 screen.
POKE 7680,1 sets the character A into the top left hand corner of the VIC screen.

Function
POKE is the complement of PEEK, setting any 8 bit location in memory. The location follows the same rules as for PEEK. The integer must evaluate to between 0 and 255 or the ILLEGAL QUANTITY error appears. This is because an 8 bit location can only contain numbers between 0 and 255.

POKEing values into ROM or non-existent memory locations will obviously not work, but there’s no error message so take care! Also as POKE is not restricted to any part of memory, an ill advised POKE or one in a runaway loop (see FOR) can crash the operating system or cause other unpredictable effects, with possible loss of program. If your program contains POKEs, it is therefore prudent to SAVE it after typing it in and each time substantial modifications are made.

POKEs can be used to set up screens (normal or graphic), operate the sound system or even to set up a few lines of machine code. Most uses of POKE are described adequately in your CBM 64 or VIC manual, but for machine code a good understanding of the machine hardware and operating system is required to achieve anything significant, together with some programming expertise of course. The CBM 64 and VIC are a little unusual in the number of intrinsic functions carried out by POKE instead of say SOUND or COLOUR commands which do not exist in this BASIC.

To set up machine code in BASIC, work out a start address and the code statements required and then load them with a routine such as

```
100 FOR I=1 TO 10
110 READ N:POKE (BS+I),N:NEXT I
120 DATA ........
```

where 10 code statements held in DATA are to be entered, starting at address BS. The DATA statements have to be decimal which takes some working out!

POKE can be used in place of PRINT, if you don’t want to disturb the cursor position while putting something on the screen. If V is the vertical and H is the horizontal position required a formula to do this would be

```
POKE (1024+(V-1)*40+(H-1)),1 for the CBM 64
POKE (7680+(V-1)*22+(H-1)),1 for the unexpanded VIC
```

for the character A in this example, and a similar expression for the colour. The expression could of course be set up by, say A=V*22+H on the previous line.

Try working out the same in PRINT – much more complicated for just one character. Normal text and header output is better handled by PRINT.

The general formula for POKEing in a 16 bit number to two 8 bit bytes is

```
POKE X,NO−INT(NO/256)*256:
```
POKE X+1, INT(NO/256)

where X is the least significant part of the word.

POS

Description
Gives the cursor position.

Syntax
POS(<dummy>)
   e.g. POS(0)

Function
POS returns the position of the cursor in a given logical line, i.e. between 0 and 79, where 0 to 39 is on the first screen line (64), 40–79 is on the second screen line, and similarly for the VIC. The dummy is ignored. As an example POS(0) returning a value of 20 is directly above a value of 60 in a line of information that spreads over 2 lines of the screen:

110 IF POS(0) + LEN(WDS) < 39 THEN
   PRINT WDS$ "":INPUT #1,
   WDS$:GOTO 110
120 PRINT:GOTO110

will prevent words being read from cassette file from being split between lines.

PRINT

Description
Outputs information to a specified device.

Syntax
PRINT [<variable>][{;}]......

   e.g. PRINT A$B$;
   concatenates and prints A$ and B$ and does not move to the next line.

Function
PRINT is an extremely versatile statement, so its function can be broken down as follows:

1. Detailed Syntax
   PRINT on its own produces a new line.
   PRINT <variable> causes the contents of the variable to be displayed on the output device, followed by a new line. The variable can be any string or expression. Some variables are specific to PRINT. These are TAB and SPC.

   A comma causes formatting similar to tabulation on a typewriter, the tab positions being 10 apart from 10 to 70. A semi-colon causes the next item or PRINT statement to follow directly on the same line.

   If neither a semi-colon nor a comma is used, items within this PRINT statement are concatenated on the same line, but the next PRINT statement follows on the next line.

2. Formatting of Output
   This varies slightly depending on the device to which the output is being sent. This device is not specified in the PRINT statement, and will be the screen unless a CMD statement has been executed directing all PRINT statements to another device.
Numeric items are preceded by a space or minus sign and followed by a space. Trailing zeros are removed e.g. 25.10 becomes 25.1 and 25.00 becomes 25. Literal strings (e.g. "STRING") are printed exactly as shown with a few special exceptions, but note that SPC(X) is not quite the same as printing X spaces since SPC does not delete information previously present in the spaced out area. The exceptions above are " (quotes) which terminate a string, <RETURN> and <SHIFT/RETURN>. Cursor control characters can be put inside quotes and appear as shown in Appendix B when typed in and will cause the relevant cursor actions when PRINTed.

Colour controls within strings cause the colour of the succeeding character to change when PRINTed. Within the quotes they appear as various graphic characters as shown in your VIC/CBM 64 manual and also in Appendix B (for the main colours).

The <INST/DEL> keys perform their normal functions within quotes, so that the reverse T for DEL following INST will appear and will have that effect in printing e.g.

PRINT "FORMING AT"

will appear as FORMAT. It is best to avoid this feature if possible as LISTing will not show the full line and thus editing is tricky.

Some other fiddling about with the information within quotes is possible but equally tricky. For instance spaces are left for special characters, the line is completed with <RETURN> and the cursor controls are used to get back into the line.

Experimentation is the only way to really understand these features.

To avoid all the above drama, use should be made of CHR$ to encode special characters, e.g. to switch the display to lower case use

PRINT CHR$(14)......

Note that if the output is to the printer CHR$(17) must be used instead and has to be repeated at the start of every line or it goes back into upper case mode. CHR$(14) on the printer is double width mode. Other CHR$ characters on the printer retain their effect until cancelled e.g. by PRINT CHR$(15).

POKE may also be used occasionally (q.v.) and PRINT # may be used instead of CMD and PRINT when writing to the printer. This is often the preferred mode as then repeated use of the CMD statement to switch control is avoided.

<table>
<thead>
<tr>
<th>PRINT #</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Prints directly to a file or device.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td>PRINT # &lt;file number&gt;, [&lt;variable&gt;[&lt;,&gt;]]</td>
</tr>
<tr>
<td>e.g. PRINT # 1,A$</td>
</tr>
<tr>
<td><strong>Function</strong></td>
</tr>
<tr>
<td>PRINT # is very similar to PRINT, but is used to write data to a logical file (which may be output to the printer direct).</td>
</tr>
<tr>
<td>The comma, semi-colon or blank are all interpreted as input separators but are not passed through as characters or separators to the file unless within quotes when they are treated like any other character; i.e. items are written one after the other without separators. Commas cause spaces to be written to the file</td>
</tr>
</tbody>
</table>
after each variable or group of variables between commas to make its total length up to 10. At the end of a PRINT # statement, if no punctuation exists, a carriage return and line feed are output. Any punctuation by comma or semi-colon causes the next PRINT statement to put characters in the next character position, again without separators. A line can be up to 255 characters long. To separate the variables on file (cassette or disk), CHRS$ (13) should be used instead of <RETURN>:

110 CR$=CHR$(13)
120 PRINT #1, A$CR$B$CR$C$CR$...

Before a PRINT # can be used an OPEN statement must have been executed for the file or device. It is possible to OPEN the screen for output as with any other device, enabling the same code to be used for printing both to screen and printer. A CLOSE must be executed before program end or the file will be left incomplete.

Example – to compact and store a list of names:

10 OPEN1,1,'"NAME"':CR$=
CHR$(13)
20 PRINT "ENTER NAMES; ENTER *
TO COMPLETE FORENAMES IF < 3"
30 PRINT "SURNAME,
FORENAMES, TO FINISH ENTER *
IN SURNAME"
40 INPUT S$,F1$,F2$,F3$
50 IF S$="**"THEN CLOSE1:END
60 PRINT #1,S$CR$F1$
70 IF F1$="**"THEN 30
80 PRINT #1,F2$
90 IF F2$="**"THEN 30
100 PRINT #1,F3$:GOTO 30

READ

Description
Reads from a DATA statement.

Syntax
READ <variable>[,<variable>.....

Function
Information is transferred from the DATA statements as if they were a sequential file. The first READ picks up the first DATA line and then works its way through the DATA items until all variables specified have been filled. The next READ follows on where the previous one left off until a RESTORE statement resets the data pointer to the start.

OUT OF DATA appears if the READ tries to go beyond the end of the DATA and SYNTAX ERROR appears if there is a type mis-match. The line identified is the line containing the DATA item. READ can be used to simulate data input from a file when developing a program, or for fixed patterns such as characters or music. It can also be used as a fixed table of values to control a program e.g.

110 READ CL%
120 ON CL% GOTO 100,200,
300,400,500

DATA has the advantage over files in that it is fast, with no waiting for file buffers to be loaded, and convenient, as it is already in the program. However, for large amounts of DATA, memory becomes a problem and of course there is no easy way of reorganising DATA, which can only be done by careful editing of the program.

Field structures cannot easily be shown in
complex cases and the omission of one item in a sequence will throw out the organisation of the data. This can be avoided in a file by writing a 'data take-on program' which guards against the user making such errors:

```
110 READ DT1: IF DT1 > 127 OR DT1=0 THEN 300:REM ERROR
110 READ DT2: IF DT2=0 THEN 300
```

in a loop reading a series of values of character codes (DT1) and their positions on a line (DT2).

`INPUT # OR GET #` should almost always be used instead of `DATA`. (In most other computer languages `DATA` or its equivalent does not exist.) The effect of `READ` can better be achieved for tables by `DIMensioning` a number of arrays and reading information into them from files by `GET #` statements. A "data" pointer then can be set to anywhere within the arrays and not just to the beginning with `RESTORE`. Initial values can be set with data statements, and read into an array after which they can be processed or modified at will e.g.

```
110 FOR I=1 TO 10
120 READ DT: DT(I)=DT
130 NEXT
```
or from a file:

```
110 FOR I=1 TO 10
120 GET #1, DT: DT(I)=DT
130 NEXT
```

not forgetting to open and close the file as required.

---

**REM**

**Description**
Comment statement.

**Syntax**
`REM [<text>]`

**Function**
The `REM` statement itself has no function as far as the computer is concerned, although its line number can be used as target for a `GOTO` or `GOSUB` statement. Its purpose is to allow the programmer to note important details of the program, or headings for subroutines or other distinctive sections of code. In any program, `REM` is a very useful reminder of what has been done and why, or as a heading to identify a section of the program. In Commodore BASIC `REM` is particularly important as it is difficult to attach meanings to variables which can only (safely) be two characters. Thus without `REM`, programs tend to resemble a jungle of indecipherable code.
The text of a REM statement can be any character except <RETURN> which terminates the line.

If only blank lines are required for formatting a listing, a lone colon e.g. 110: will serve instead of REM, and will not cause a SYNTAX ERROR! The same applies after a GOTO as anything thereafter is never accessed by the program, but this is not recommended as changes to the program (GOSUB instead of GOTO) may cause problems.

REM statements should be used at the beginning of a program to give general information such as title and date last amended and a section should then be included to identify the variables e.g.

```
REM L=LENGTH W=WIDTH
H=HEIGHT V=VOLUME.......
```

Unfortunately REMs do occupy space and marginally slow a program down, so if you are short of space transfer your REM statements to a piece of paper kept with the program. Beware of long programs without reasonable use of REMs which are often impossible to finish. Large well written commercial programs are heavily commented, and it is worth noting that if you are writing large programs it is worthwhile getting a compiler. This will compact your code and remove all REM statements from the running version while leaving your original ('source') code intact.

**Syntax**

**RESTORE**

**Function**

Each READ processes further DATA statements until a RESTORE occurs, after which the READ picks up the first DATA statement again. RESTORE might be used as part of a loop to allow a musical program to be played repeatedly or repeat information on a screen.

Unlike many dialects of BASIC, CBM computers do not have a RESTORE <line number> function, so to return to a given DATA statement (not being the first one) requires a combination of RESTORE and dummy READS:

```
110 RESTORE
120 FOR N=1 TO 100:READ DM$: NEXTN
```

to reach item no. 101

See also READ, DATA
RETURN

Description
Terminates a subroutine.

Syntax
RETURN

Function
Every subroutine must be terminated by a RETURN which transfers control to the statement following the GOSUB statement. Note that failure to put a RETURN at the right place will simply cause the program to run wild through any code following that point. No SYNTAX ERROR can appear as the computer does not know when to expect a RETURN. One should be vigilant over this problem to avoid having painful debugging sessions. See also GOSUB.

RIGHT$

Description
A string manipulation function which extracts part of a string starting from the right.

Syntax
RIGHT$(<string>,<integer>)
e.g. RIGHT$("JOHN SMITH",5) GIVES SMITH

Function
RIGHT$ takes the rightmost <integer> characters of the string. The integer must evaluate to between 0 and 255. This is useful in truncation or as part of a string analysis routine and quite complex syntax can be used here with good effect. For examples see LEFT$ and MID$ and also INT which is valuable for rounding and truncating numerics and is easier to use than the string functions under these circumstances.

RND

Description
A pseudo random number generator.

Syntax
RND (<numeric>)
e.g. RND(1)

Function
The RND function is useful to create a floating point number between 0 and 1. The numeric is a dummy when positive and generates the numbers in a fixed repeatable sequence. To start such a sequence a negative number should be used to provide the seed for the generator, in other words to restart the generator at the beginning of a known pseudo random sequence. The sequence of random numbers generated depends on the negative numeric used to set up the sequence. The negative number RND does not itself generate a random
number that can be used in a sequence. If the numeric is zero there is no fixed repeatable sequence. If a known sequence is not required, it is possible to use RND without seeding e.g.

```
10 GET A$: IF A$="" THEN 10
20 PRINT ""
30 FOR N=1 TO 4: PRINT
   INT(RND(1)*6+1): NEXT
40 GOTO 10
```
simulates throwing 4 dice on pressing any key.

As can be seen from the example, to get a maximum random number of 6 requires that 1 is added to RND(1)*6 as the random number generator will never produce a value of exactly 1 and so INT(RND(1)*6) will be between 0 and 5 instead of 1 and 6 as required.

Function

RUN is normally used directly from the keyboard to start a program but can be run from within a program, when its effect is rather like a GOTO except that it implies a CLR statement clearing not only variables and arrays, but loop counters and subroutine pointers, just as if the program had been loaded and started for the first time. If a line number is specified it starts the program from that line which must exist even if only as a REM or lone colon, otherwise the message UNDEF'D STATEMENT appears.

```
60 IF CU$="NEW" THEN RUN 100
70 IF CU$<>"NEW" THEN RUN 200
```
re-initialises from one of two starting points. Be careful of DIMensioned arrays as any array dimension statements are cleared by typing RUN.

Don't use RUN when debugging using STOP statements, CONT should be used.
**SAVE**

**Description**
Saves a program.

**Syntax**
SAVE [ [@0:]program name > ][, <device number >][, <address >]

- e.g. SAVE "PROGRAM", 8

saves a program named "PROGRAM" to disk, PROGRAM being currently not on disk.

**Function**
SAVE stores a program to cassette or disk, giving it any name up to 10 characters long. The '@0' is not part of the name but is required if the original program on disk under the same name is to be overwritten. The device number is 1 (cassette) if nothing is entered. If 8 is entered, the program is saved to the disk designated on device 8 (the usual device number). The address, if used, has only 3 values which are as follows:

1. The cassette/disk copy being made will be loaded back at the same place in memory where it currently resides, when LOADed again.

2. An end of tape marker is put on tape.

3. Both functions 1 and 2 are carried out.

If a disk is used, a filename must be given, as the disk is not a serial device like a cassette. The SAVEed program is put on to a convenient space on the disk and the disk programs are not in any particular order. On cassette a filename need not be given, when the program will be saved without a name. However, it is sensible to give it a name so that you can check when loading that you have the right one. Beware of the fact that the cassette can load on a part match, so LOAD P will load the first program whose name begins with P.

Programs on cassette are SAVEed one after the other in the order of the SAVEs. SAVE can be used from within a program but if in main code this can occupy a lot of time during debugging if long programs need to be SAVEed to tape every time a small bug is corrected. SAVEing a long program can take over 10 minutes. An exception might be made for a program containing POKEs or machine code calls SYS or USR which might wipe out or corrupt the operating system, e.g.

```
10 SAVE "POKEPROG"
20 DEF ..........
:
```
saves the program before any code is executed.
A better method would be:

```
1 GOTO 10
2 SAVE "PROGRAM"
3 STOP
4 VERIFY "PROGRAM"
5 STOP
10 REM START OF PROGRAM
```

where RUN 2 will SAVE the program, and, after the cassette has been rewound, CONT will VERIFY it and CONT again will run it.

A good discipline is to mark all changes on your listing as you go along, or on a piece of paper if you have no printer, so that in case of disaster you know exactly what has been done, and also to save when the listing becomes difficult to read or you stop for a break.
SAVE all important or lengthy programs on alternate cassettes so that damage or corruption on one cassette does not cause loss of all your work.

**SGN**

**Description**
Logical test of the sign of a number.

**Syntax**
SGN(<numeric>)
e.g. SGN(A-1)

**Function**
If the numeric evaluates to a positive number the SGN is 1, if it is 0, 0 and if negative then -1. An example is

```plaintext
50 ON SGN(A)+2 GOSUB1000, 2000,3000
```

which is a neat way of branching three ways, especially as ELSE does not exist. This could be used where a money amount is printed in the debit column if negative, the credit column if positive or not printed at all if zero.

**SIN**

**Description**
Trigonometric sine function, giving the ratio of the side opposite and the hypotenuse in a right angled triangle.

**Syntax**
SIN(<numeric>)
e.g. SIN(1) gives 0.84147

**Function**
The sine of an angle (in radians) is produced. See also TAN, COS and the table of other trigonometric functions in your user manual. Example:

```plaintext
SIN(π/4)
gives the sine of 45 degrees = 0.707106781
```

**SPC**

**Description**
Produces spaces in a PRINT statement.

**Syntax**
SPC(<numeric>)
e.g. SPC(5) moves the cursor 5 spaces to the right
Function
SPC steps the cursor along, without printing, by the number of spaces specified. On the screen this leaves anything previously on the screen unchanged, but obviously on other devices it writes spaces. Line overflow is prevented in all cases by automatic <RETURN>s where required. The numeric must evaluate to between 1 and 255 (or 254 for disk files) e.g. in,

PRINT SPC(I*40+J)

will move the PRINT position down I lines and along J spaces if the expression is less than 255 (CBM 64).

PRINT " " actually prints spaces rather than skipping them and TAB gets to a predicted position beyond the existing cursor position thus often avoiding the need to recount spaces every time a program changes slightly. See also POKE.

SQR

Description
Square root function.

Syntax
SQR (<numeric>)
e.g. SQR(4) gives 2

Function
Gives the square root of a number or expression. The computer cannot handle imaginary numbers, so the numeric must not be negative (error ILLEGAL QUANTITY appears).

STATUS

Description
Variable holding the status of the file most recently operated on.

Syntax
ST

Function
ST is zero for an ordinary I/O operation. Bits are set in the 8 bit ST word for any conditions that arise, as follows:

<table>
<thead>
<tr>
<th>ST (Bit)</th>
<th>Cassette</th>
<th>Serial I/O</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Timeout - write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Timeout - read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Short block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Long block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Non-recoverable read error/ mismatch on LOAD/ VERIFY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Checksum error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>End of file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Device not present</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>End of tape</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

e.g.

100 IF ST=64 THEN 1000

: : : 

1000 PRINT "END OF FILE - NO ERRORS"
tests for end of file without other error conditions. 64 is binary 01000000 representing the end of file condition.

**STEP**

**Description**
Defines amount of step in a FOR loop.

**Syntax**
See FOR.

**Function**
See FOR. Note that if STEP is not explicitly coded BASIC assumes a STEP of 1. In backward counting loops therefore, STEP has to be specified.

**STOP**

**Description**
Halts a BASIC program.

**Syntax**
STOP.

**Function**
STOP simulates a depression of the <STOP> key. The BASIC program waits and all its variables are available to examine or change. The message given by the system is BREAK IN LINE... and the line number followed by READY as the machine returns to command mode. To restart type CONT or to skip to another part of the program type GOTO <line number>. The program then resumes without resetting stacks or variables.

The STOP function is extremely useful in finding the bugs in a program. You can check if a program follows a given path by placing STOPS in it and can check the variables within the program to see if they are at the expected values.

This function is very similar to END except that the END does not produce a BREAK IN LINE .... error message.

**STR$**

**Description**
Converts a numeric to a string.

**Syntax**
STR$ (<numeric>)

**Function**
Evaluates a numeric or numeric expression as if it were to be PRINTed and converts each numeric character to its ASCII character equivalent. The first character is always a space or negative sign e.g.

PRINT STR$(4.5E5)
gives "450000" (length 7)
This function is not often used except during disk handling as the work is considerable, chiefly because one often needs to get an exact match between the strings and the expected results when these are converted back to numerics. For example 45.00 is converted to 45 by **VAL**, while 45.01 remains as 45.01. However see the number checking routine in Section 3.

**SYS**

**Description**

Starts a machine code subroutine.

**Syntax**

SYS <location>

e.g. SYS 828 starts a machine code program in the cassette buffer, starting at decimal 828.

**Function**

The safety of BASIC is left behind, all commands must be programmed in machine language using an assembler or a series of **POKE**s in the BASIC program. Good luck, but watch out as any error may sink you! Don’t forget to end your machine code on an **RTS** if you want it to come back to the BASIC program.

It is beyond the scope of this book to describe machine level programming, usually required to obtain speed or special features of the hardware. It is worth considering that a BASIC compiler may achieve the same effect, with far less effort.

SYS should be used in preference to the other BASIC machine code call, **USR**, which is more difficult to use without any real advantages.

---

**TAB**

**Description**

Moves the cursor to a defined position in the logical line.

**Syntax**

TAB (<numeric>)

e.g. TAB (35)

**Function**

TAB acts in the same way as a typewriter TAB but can run on beyond the logical line length of 80 characters, up to 255 positions in total (from the start of the line). E.g.

```plaintext
110 PRINT "MAR"TAB(10)"APR"
TAB(20)"MAY"TAB(30)"JUNE"
120 FOR N=1 TO 5
130 PRINT S$(N,1)TAB(10)
S$(N,2)TAB(20)S$(N,3)TAB(30)
S$(N,4)
150 NEXT
```

prints 5 sets of (say) monthly sales figures (held
in a two dimensional string array S$ as equal length strings) in neat columns.
Or on the printer:

```
110 PRINT "MAR"TAB(7)"APR"
   TAB(7)"MAY"TAB(7)......etc
```

produces the same effect, from which can be seen that TAB works like SPC for the printer.

### TAN

**Description**
Trigonometric tangent function being the result of the side opposite to the angle divided by the side adjacent to the angle in any right angled triangle.

**Syntax**
TAN (<numeric>)
e.g. TAN (π/4)=1 (actually comes out as .9 recurring)

**Function**
The tangent of an angle (in radians) is produced. See also SIN, COS and the table of other trigonometric functions given in your user manual. As the tangent, unlike SIN or COS can approach infinity, an error DIVISION BY ZERO will appear if the tangent is too close to π/2 or a multiple thereof. (i.e. BASIC cannot handle infinite numbers, so a test must be incorporated to bypass TAN and give an answer of INFINITE if the tangent of π/2, is expected to be requested).

### TIME

**Description**
Reads an internal clock.

**Syntax**
TI or TI$.

**Function**
The function TI starts at 0 on power up and can be reset at any time using TI$ (trying to reset with TI=0 gives a syntax error). e.g.

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

It counts time accurately (even if it's on a 50 Hz supply) in 1/60 second units (except when cassette Input or Output is taking place) as a pure count in TI but as a string of six numerics of the format “HHMMSS”in TI$. It can usefully be used as a keyboard timer e.g.

```
10 TI$="000000"
20 GET A$:IF TI<10*60 AND A$=""THEN 20
30 IF A$=""THEN PRINT "MISSING - 10 SECONDS UP";END
40 PRINT "YOU TOOK "TI/60
   "SECONDS";GOTO 10
```

As an illustrative program the CLOCK program below shows how to set up and display hours minutes and seconds. METRONOME gives a method of creating a fixed time interval.
```
1 REM CLOCK
2 POKE 53281,1
3 PRINT "....................TAB(15):";-----
5 PRINT "....................TAB(15):":!
7 PRINT "....................TAB(15):";-----
10 PRINT "....................TAB(15):";AB(16):LEFT$(TI$,2):";:
   MID$(TI$,3,2):";RIGHT$(TI$,2)
12 POKE 53280,INT(RND(1)*16)+1
20 TI$=TI$
30 IF TI$=TI$ THEN 30
40 GOTO 10

Printer in lower case mode.

100 rem metronome - vic 20
110 vl=36878:s1=36874:POKE vl,15:ti$="000000"
120 input "beats/min":bt$
130 print "Enter c to change e to end"
140 bt=val(bt$):if bt=0 then print "0 is too slow!":GOTO 110
150 mt%=60*60/bt:rem Pulses/sec * secs / no. beats per min
160 if k$="e" then 210
170 if k$="c" then 120
180 if mt%>ti then 160:rem wait for clock tick or command
190 POKE s1,128:POKE s1,0
200 ti$="000000":GOTO 160
210 POKE s1,0:POKE vl,0:end
ready.
```

**USR**

**Description**
Starts a machine code routine.

**Syntax**
USR(<numeric>)
   e.g. X=USR(A*2+4)

**Function**
USR calls a subroutine in the location pointed to by memory locations 1 and 2 (VIC) or 785 and 786 (CBM 64). This must be set up by POKEs before USR is called. The numeric is stored in the floating point accumulator at the start of the subroutine and the result of the machine code routine is left in 97 for the BASIC program when control is returned with an RTS machine language statement. See also SYS which is easier to use and more flexible.

**VAL**

**Description**
Extracts the numeric content of a string.

**Syntax**
VAL(<string>)
   e.g. VAL("1.01A") gives 1.01

**Function**
Starting from the left, VAL extracts first a sign, if present, then numerics and one decimal full stop, until a non-numeric or second full stop is reached. The number is then stripped of leading and trailing zeros to be held as a conventional numeric. e.g.

```
VAL("-1.0.01A") gives -1
VAL("ABC") gives 0
VAL("ABC123") gives 0
VAL("-25.43") gives -25.43
```

VAL is the converse operation to STR$, and has rather more use. When using INPUT for numerics, an error occurs and the program terminates if a non-numeric is entered in a numeric field. It is therefore advisable to use a string variable and VAL e.g.

```
10 INPUT "NUMBER 1 TO 9":A$
20 IF VAL(A$)<1 OR VAL(A$)>9 GOTO 10
```
30 PRINT "OK"

See also the number checking routine in Section 3 and INPUT.

VERIFY

Description
Checks a SAVEd program.

Syntax
VERIFY ["<program name>"][,< device>]
e.g. VERIFY "PROGRAM"

Function
The program name is read from the device (defaults to cassette if not specified) and checks against the contents of memory. A VERIFY ERROR message is produced if the title and data contents do not exactly match. VERIFY can also be used within a program e.g.

5010 SAVE "@0:PROGRAM",8
5020 VERIFY "PROGRAM",8
5030 PRINT "END":END

Note that this cannot be used on cassette without a WAIT at 5015 or

5015 PRINT "REWIND CASSETTE, PRESS PLAY THEN RETURN"
5017 INPUT A$ 

otherwise the program goes straight into VERIFY as the tape is still running.

WAIT

Description
The program waits for an external event.

Syntax
WAIT <location>,<mask1>,[<mask2>]
e.g. WAIT 1,48 waits for cassette to be stopped on CBM 64.
(48 is the bit pattern 00110000 set in location 1 when the tape starts, and cleared when it stops)

Function
WAIT halts the BASIC program until an external event such as the pressing of a key on the cassette recorder or the expiry of time occurs. It does this by watching the location specified for a defined pattern. Normally the location will be one of the I-O registers or a related position. This location need not be a number, an expression will be evaluated to an integer even if it is not, in fact, an integer.

To define the pattern being WAITed for, the contents of the location are ANDed with mask1, complemented and ANDed with mask2, if present. In other words, if any location bits are 1 in positions corresponding to 1s in mask1 or 0 in positions corresponding to 1s in mask2, the wait is over and the program proceeds.
SECTION 3 USEFUL ROUTINES

Introduction

The examples set out in this section of the book are intended to illustrate the way programs can be built up using the BASIC described in detail in Section 2. The BASIC syntax (i.e. language construction) is very flexible, and commands can be combined or nested in countless different ways and to a complex level before the interpreter can no longer cope. In fact the usual limitation is the ability of the human brain to easily understand complex constructs that limits the complexity of code, rather than BASIC. Complex solutions are therefore better implemented by carefully constructed combinations of simple easy to understand code, than by convoluted code nested deep in FOR loops and GOSUBs with a sprinkling of GOTOs.

To get started on the computer is extremely easy, but many people don’t realise this. After a little while playing with the machine, reading from the manual, most children can produce a simple program like PROTEST below, which apart from line 5 is exactly as it was written by my 9 year old daughter.

5 REM PROTEST
10 PRINT"BOOOOOOO HOOOOOOOO"
20 PRINT"I CAN'T PLAY FOOTBALL"
30 PRINT"MUM WON'T LET ME."
40 PRINT"HELP ME"
50 PRINT"
60 PRINT"HELP ME"
70 GOTO 10

Another slightly longer program which requires little knowledge of BASIC, but an understanding of the way sound and colour is generated (in your VIC 20 manual) is as follows.

10 REM MUSIC
20 SC=53281:PRINT "J":POKE SC,1
30 CL=55296:S=54272:FOR N=S TO S+24:POKE N,0:NEXT
40 POKE SD+5,9:POKE SD+9,0:POKE SD+24,15
50 READ D,T,TA,LN
60 REM RANDOM COLOURED SQUARES ON THE SCREEN
70 RD=INT(RND(1)*1000)
80 POKE RD+1024,32+128
90 CO=INT(RND(1)*17):IF CO=1 THEN 90
100 IF DA=999 THEN RESTORE:GOTO 50
110 POKE RD+CL,CO
120 :
130 REM PLAY NOTE
140 POKE SD+1,DA:POKE SD,TA:POKE SD+4,33
150 FOR N=1 TO LN*2:NEXT
160 POKE SD+4,32
170 GOTO 50
180 REM ************DATA***************** **
190 DATA 14,107,150,19,63,300
200 DATA 21,154,150,22,227,300
210 DATA 28,214,150,25,177,300
220 DATA 19,63,150,19,63,300
230 DATA 38,126,150,38,126,300
240 DATA 38,126,150,34,75,300
250 DATA 28,214,150,28,214,600
260 DATA 38,126,75,38,126,75
270 DATA 38,126,300,21,154,150
280 DATA 22,227,300,28,214,150
290 DATA 25,177,150,19,63,150
300 DATA 19,63,150
310 DATA 19,63,300
320 DATA 19,63,150
330 DATA 19,63,300,19,63,150
340 DATA 18,42,300,18,42,150
350 DATA 19,63,300
360 DATA 999,999,999

READY.
The first few lines of this program generate random colours in random positions on the screen, and the last few lines generate the music from the data read in line 190. The data is used exclusively to generate the music.

The examples that follow are in some cases quite complex and although all have been tested or used for various applications, the sharp programmers amongst you will very probably feel there are better ways of carrying out some of these functions. If so then this book has served its purpose and of course I would be pleased to hear of any improvements.

**SORT**

The two programs below both work on the principle of a bubble sort. Starting with the first two items in the list of numbers or words, the program compares them and swaps them if the order is wrong. It then goes on to the second and third items, third and fourth items and so on to the bottom of the list. Having passed the list once it then repeats the process until the items are all in the right order, i.e. no further swaps are carried out during a pass of the list. Thus an item at the bottom of the list will rise one position at a time until it arrives under the item next to which it belongs.

```
10 S1=36874:POKE 36879,15:POKE 36879,30
20 GOSUB 300
30 PRINT"ENTER EACH NUMBER AND WHEN YOU HAVE FINISHED, PRESS 'RETURN'."
40 PRINT"UP TO ONE HUNDRED NUMBERS MAY BE ENTERED"
50 PRINT"SHIT ANY KEY":GOSUB 290
60 PRINT"AM=100:DIM NO(AM)"
70 FOR N=1 TO AM
80 PRINT"ENTER NUMBER$:N""$N"
90 INPUT NO$
100 IF NO$="" THEN 150
110 NO(N)=VAL(NO$):NO$=""
120 X=X+1
130 FOR L=1 TO AM-1:POKE S1+1,240:NEXT:POKE S1+1,0
140 NEXT N
150 REM*************** ** SORTING **
160 GOSUB 300:PRINT"BUBBLE SORTING.."
170 AM=X:X=0:N=1
180 PRINT"PLEASE WAIT..."
190 IF NO(N)>NO(N+1) THEN NO=NO(N):NO(N)=NO(N+1):N(N+1)=NO:X=0
200 X=X+1:N=N+1:IF N=AM THEN N=1
210 IF X=AM THEN 190
220 REM*************** ** PRINTING **
230 GOSUB 300:PRINT"X"
240 FOR N=1 TO AM
250 P=P+1:IF P/18=INT(P/18) THEN GOSUB 280:PRINT":0.1"
260 IF NO(N)=0 THEN PRINT "NO(N)"
270 NEXT N:PRINT "END OF LIST":END
280 PRINT":SHIT ANY KEY"
290 GET KY$:IF KY$="" THEN 290
300 PRINT"READY.
```

**NO SORT**

```
10 REM NO SORT
20 REM SORTS NUMBERS INTO ASCENDING ORDER
30 VL=54296:BD=53280:SC=53281:SD=54272:REM COLOUR & SOUND
40 POKE VL,15:REM VOLUME
50 POKE BD,6:POKE SC2,1:REM SCREEN COLOURS
60 POKE SD+6,248:REM SOUND
70 GOSUB 490
80 PRINT"ENTER EACH NUMBER AND"
90 PRINT TAB(8)"WHEN YOU HAVE FINISHED"
100 PRINT TAB(11)"PRESS 'RETURN'."'
110 PRINT TAB(10)"UP TO ONE THOUSAND NUMBERS MAY BE ENTERED"
120 PRINT"SHIT ANY KEY":GOSUB 4
80
130 AM=1000:IM=NO(AM)
140 FOR N=1 TO AM
150 PRINT"ENTER NUMBER$:N""$N"
160 INPUT ":NO$
170 IF NO$="" THEN 290
180 NO(N)=VAL(NO$):NO$=""
190 X=X+1
200 POKE SD+6,190
210 POKE SD+6,248
```
220 POKE SD+1,20+INT(RND(1)*20):POKE SD,37
230 POKE SD+4,17
240 FOR L=1 TO 100:NEXT L
250 POKE SD+4,0
260 POKE SD+5,0:POKE SD+6,0
270 NEXT N
280
290 REM********** SORTING **********
300
310 TI$="000000"
320 GOSUB 490:PRINT"*************** SORTING ***************"
330 AM=X:X=0:N=1
340 PRINT"PLEASE WAIT..."
350 IF NO(N)<NO(N+1) THEN NO=NO(N):NO=NO(N+1):N=N+1:IF N=AM THEN N=1
360 IF X=AM THEN 350
370 T$t=TI$
380
390 REM******** Printing Numbers *********
400
410 GOSUB 490:PRINT"W"
420 FOR N=1 TO AM
430 P=P+1:IF P/18=INT(P/18) THEN GOSUB 510
440 IF NO(N)<>"" THEN PRINT TAB(TB):"W"NO(N)
450 NEXT N:PRINT TAB(TB):"END OF LIST":PRINT "END
470 PRINT"PLEASE WAIT..."
480 GET KY$:IF KY$="" THEN 480
490 PRINT"W":RETURN
500 RETURN
520 PRINT"000000":RETURN

5 REM********** WORD SORT ** * **
6 ***************
7 10 SI=36874:POKE 36878,15:POKE 36879,30
8 20 GOSUB 300
9 30 PRINT"ENTER EACH WORD AND WHEN YOU HAVE FINISHED PRESS 'RETURN'."
10 PRINT"UP TO ONE HUNDRED WORDS MAY BE ENTERED"
11 POKE"W":GOSUB 290
12 60 PRINT":":AM=100:DIM NO$(AM)
13 70 FOR N=1TOAM
14 80 PRINT"ENTER WORDS":NO$(N)
15 90 INPUT NO$
The bubble sort is easy to understand but very inefficient under most circumstances. Many better sorts exist; most are difficult to understand but usually fairly easy to use by incorporating them as subroutines in your program. One such subroutine is shown below.

1 REM ** J IS NO OF WORDS TO GO
2 REM ** S IS NO OF WORDS DONE, OR THE
The string manipulation in the above routine may also be of some interest. It provides a method of reducing a string until unwanted filler characters have been entirely removed.

**RANDOM GROUPS**

The dice rolling and holding program below illustrates the use of random numbers in a selection from 1 to 6, then incorporated in pictures of dice. The sound effects are also randomly generated. Note the use of the screen as an input device.
BASE CONVERTER

The base converter program below provides a useful utility for hexadecimal (hex) to decimal and the reverse conversion. Once in hex it is easy to produce binary, as each hex character maps directly on to 4 binary bits. See AND, and OR for the methods that can be used for bit manipulation.

10 SC=53281:POKE SC,1:REM SET SCREEN TO WHITE
20 REM*************** ** CONVERTER **
30 GOSUB 610:PRINT"******** HEX-DEC OR DEC-HEX ?"

40 PRINT"*** FOR HEX-DEC"
50 PRINT"*** FOR DEC-HEX"
60 PRINT"*** FOR MENDING"
70 PRINT"*** FOR STOPPING WHILE ENTERING NUMBERS"

80 GET KY$: IF KY$="H" AND KY$="D" AND KY$="E" THEN 80
90 PRINT"J"
100 IF KY$="E" THEN END
110 IF KY$="H" THEN 370
120 REM*************** ** VARIABLES **
130 REM DE$ IS DECIMAL NUMBER ENTERED
140 REM DE IS ITS NUMERIC Equivalent
150 REM DS$ = STRING OF NOS FROM DE WITHOUT LEADING SPACE - FOR NO. VALIDATION
160 REM D AND E ARE VARIABLES FOR NUMBER MANIPULATION
170 REM RES() IS HEX REMAINDER AFTER N+1TH DIVISION
180 REM************** DECIMAL-HEX**
190 GOSUB 610:PRINT"DECIMAL HEXADECIMAL"
210 DS$=RIGHT$(STR$(DE),LEN(STR$(DE))-1):REM REMOVES SPACE BEFORE NO.
220 IF DS$=DE THEN PRINT"NOT DECIMAL":GOTO 200
230 E=INT(D/16)
240:
250 RES(N)+STR$(D-E*16):D=E
260 IF VAL(RES(N))>9 THEN RES(N)=CHR$(VAL(RES(N))+55)
270:
280 IF D<10 THEN 330
290:
300 N=N+1:GOTO 230
310 REM** BUILD UP HEX NO. FROM ARRAY.
320 REM HE$=HEX NO.
330 HE$="":IF D>0 THEN HE$=HE$+STR$(D)
340 FOR L=N TO 0 STEP-1:HE$=HE$+RIGHT$(RES(L),1):N=EXT
350 PRINT ":"TAB(18));HE$
360 GOTO 200
370 REM*************** **HEXADEC-DEC**
380 GOSUB 610:PRINT"HEXADECIMAL DECIMAL"
390 HE$="":INPUT HE$:H=LEN(HE$)+1:IF HE$="S" THEN 20
400 IF H>12 THEN PRINT"SHORTER NUMBER PLEASE!":
GOTO 390
410:
420 REM CREATE AND CHECK ARRAY
In this program the colon has been used to make the code more readable by separating blocks of code. In other machine BASICs the apostrophe (') would be used as shorthand for REM, but this is not available in VIC/CBM 64 BASIC, so the dummy statement represented by colon can be used in a line on its own.
the entry of information a much more routine affair that can be left to non-computer trained staff who are guided through the entries by the form layout.

The first program listed is a screen form creation program that allows a programmer or non-programmer to lay out a screen and then store it on tape. This saves a lot of coding in screen handling programs which simply pick up and use the appropriate form from tape (or disk). In this program, the screen is set up by moving the cursor around and setting up strings or entry fields wherever required. The program is then run and picks up and compresses the data on the screen (first 20 lines) before writing it to tape. Brackets () are used to delineate the areas where entries are to be made. The program stores only the start position and contents of strings (defined as containing no more than one contiguous space) and the start position and length of the spaces to be reserved for entries (‘permitted fields’). The rest of the screen (empty space and the last two lines) are ignored. Note the use of function keys to allow rapid entry.

5 REM SCRGEN - GENERATES A SCREEN FORMAT ON A TAPE ALLOWING TEXT ENTRY AND...
6 REM BRACKETS TO INDICATE WHERE FIELDS WILL BE ENTERED BY A PROGRAM USING...
7 THE STORED SCREEN FORMAT
8 REM RUN BY SETTING UP THE REQUIRED SCREEN AND THEN PRESSING RUN
9 REM SA#: STR# ADDRESS OF STRING
10 OPEN1,1:"SCREEN"
20 SC=1023:PK=32
30 REM *** ONLY FIRST 20 LINES OF SCREEN ARE CONSIDERED ***
40 FOROF=1 TO 800:REM WORK THROUGH 20 LINES OF SCREEN (800 POSITIONS)
50 IFPK=32 AND PK=32 THEN 105
60 IFPK<32 THEN PRINT#1,999:PRINT#1,"END"
70 IFPK=32 THEN PRINT#1,"SCREEN STORED TO TAPE"
80 END
90 REM TEST ROUTINE TO CHECK THE SCREEN GENERATED ABOVE BY DISPLAYING IT
100 OPEN1,1,"SCREEN":J=1:OF=1023:CF=55295
110 PRINT"J"
120 INPUT A,A$:IFA=A THEN PRINT"SCREEN PRINTED":END
130 RT=ASC(MID$(A$,J,1)):PK=PRT
140 IFRT>63 THEN PRT=RT-32
150 POKEOF+A,J,PK
160 POKECF+A,J,1
170 IFJ=LEN(A$) THEN J=1:GOTO140
180 J=J+1:GOTO142
190 READY.

The next program will appear at first sight to have nothing whatever to do with the first program or with business forms, but it is an essential tool for building forms programs and has uses elsewhere. This second program is an extremely primitive word processor/line editor. When the program is running, a line can be typed and characters can be changed or deleted just as if in the screen editor excluding the vertical cursor movement, working on one line at a time. When <RETURN> is pressed, the line is typed out to the printer. As most of this program simulates the operating characteristics of the machine with no program in memory, there is probably a rather neat way of doing this in machine code using calls to BASIC ROM subroutines, but if we stick to BASIC, then this is the way it must be done.

10 REM TYP464
20 BD=53288:SC=53281
Although the VIC is not really suitable as a business machine for most applications, this particular routine is useful on its own, so below is a VIC version of the same program.
N	130 IF LC>I THEN FL$(2)=LI$(I+1):IF ASC(LI$(I+1))=34 THEN FL$(2)="\""
140 IF LC=I THEN FL$(2)="
150 IF LC>79 THEN CH$=CHR$(13):GO TO 350
160 FOR L=1 TO 2
170 FOR K=1 TO 30
180 GET CH$:PRINT FL$(L)="\"
190 IF CH$="" THEN 240
200 NEXT K
210 NEXT L
220 GOTO 120
230 IF I=LC THEN PRINT"\""
240 PRINT FL$(2)="\"
250 IF ASC(CH$)=20 THEN 430
260 IF ASC(CH$)=148 THEN 500
270 IF ASC(CH$)=157 THEN 590
280 IF ASC(CH$)=29 THEN 640
290 IF ASC(CH$)=133 THEN CLOSE:END
300 IF ASC(CH$)=13 OR ASC(CH$)=141 THEN 350
310 CA$=CH$
320 IF ASC(CH$)=34 THEN CA$="\"
330 I=I+1:PRINT CA$:LI$(I)=CH$:GO TO 110
340 U/I<$>=CH$
350 REM*************** ** PRINTING **
360 PRINT:SN$=""
370 FOR J=1 TO LC+1
380 SN$=SN$+LI$(J):U$(J)=""
390 NEXT J:
400 PRINT\n410 REM*************** ** DELETE **
420 I=I-1:IF I<0 THEN I=0:GO TO 110
430 GOTO 110
440 REM*************** ** INSERT **
450 PRINT \n460 FOR N=I+1 TO LC+1:LI$(N)=LI$(N+1)
470 NEXT N
480 GO TO 110
490 REM*************** ** CURSOR <- **
500 REM*************** ** CURSOR -> **
510 I=I-1:IF I<0 THEN I=0:GO TO 110
520 PRINT "\"
530 GO TO 110
540 REM*************** ** SCREEN USE **
550 DIM LI$(80),EN$(20),EX$(20),ST$(20)
560 GO SUB 4000
570 REM*************** ** BUILD SCREEN SUBROUTINE **
580 INPUT "SCREEN NAME"; SC$:
590 R=1:T=1:OF=1023:CF=~5$95
600 FOR N
610 IF LEN(A$)=1 THEN T=1-EN<T>:T=T+1
620 PRINT\n630 REM*************** ** ACCEPT INPUT SUBROUTINE **
640 IF LEN(A$)=1 AND A$="A" THEN R=+1:GO TO 4040
650 IF A$="" THEN R=0:GO TO 4040
660 IF A$="A" THEN R=1:GO TO 4040
670 PRINT FL$(2):GO TO 110

Now defining all the fields in TYPER64 variably instead of as a standard 80 characters long, and if the output is written to an array rather than the printer, then the combination of this program with a routine to read the screen format created earlier, will provide a business entry form:
The next stage is to include a read of the file into the machine from cassette, validation of the fields on the screen according to the requirement, using the numeric check routine described earlier where appropriate, and storing the information collected back onto cassette. The same principle will work for a diskette serial file.

10 REM CUSTOMER
20 OPEN 4,4
30 REM DIMENSIONS FOR MAIN PROGRAM - CUSTOMER UPDATE
40 DIM CH$(99),A$(99,4),P1$(99),D$(99,4),P2$(99),BL$(99)
50 REM DIMENSIONS FOR SCREEN HANDLING ROUTINE
60 DIM LI$(91),EN(20),EX(20),ST$(20)
72 TP=13
80 INPUT "LOAD CUSTOMER DATA":AN$$
90 IFAN$=="Y" THEN GOSUB 430
100 GOSUB 890: CLOSE 1
110 GOSUB 1130: IFASC(CH$)=1 THEN 140
120 IFP=1 AND DE=0 THEN NO=VAUSN$:130 GOSUB 980=CLOSE 1
140 GOSUB 760: CLOSE 4: END
150 REM FUNCTION
170 "REM FUNCTION"
180 CN$(S1)=ST$(2)
190 FOR Q=1 TO 4: A$(S1,Q)=ST$(2+Q): NEXT
200 P1$(S1)=ST$(7)
210 FOR Q=1 TO 4: D$(S1,Q)=ST$(7+Q): NEXT
220 P2$(S1)=ST$(12)
230 I=1: GOSUB 270
240 BL$(S1)=ST$(13)
250 FOR V=1 TO 20=ST$(V)="": NEXT
260 RETURN
270 REM NUMERIC CHECK SUBROUTINE
280 REM IF VALID, NO IS RETURNED AS "NO"
290 NO=VAL(ST$(N0)): ST$=STR$(NO): TS$=ST$(NO)
300 IFNO=0 AND LEN(TS$)=0 THEN 420
310 IFS$=ST$: THEN 420
320 IF LEFT$(TS$, 1)="0" OR TS$="0" THEN 420
340 IF TS$=ST$: THEN 420
350 IF TS$=ST$: THEN 420
360 IF TS$=ST$: THEN 420
370 IF TS$=ST$: THEN 420
380 PRINT "BAD DATA"
390 GOSUB 560
400 IF TS$=ST$: THEN 420
410 IF TS$=ST$: THEN PRINT "LOAD CUSTOMER TAPE": ST$(N0)=E=1: GOTO 110
420 E=0: RETURN
430 REM ROUTINE TO LOAD CUSTOMER DATA
440 PRINT "REMOVE PROGRAM & LOAD CUSTOMER TAPE"
450 OPEN 1, 1, "CUSTOMER"
460 INPUT 1, NO 1: IFNO=999 THEN 530
470 ST$(1)=ST$(NO)
480 FOR W=2 TO 8
490 INPUT 1, ST$(W)
500 IF W=8 THEN 510
510 NEXT
520 GOSUB 150: GOTO 460
530 PRINT "REMOVE CUSTOMER & LOAD SCREEN TAPE"
540 INPUT "TYPE RETURN WHEN READY": A$:
550 CLOSE 1: RETURN
560 REM LINE CLEAR ON SCREEN SUBROUTINE
570 FOR T=1 TO 4: PRINT "": NEXT
580 RETURN
590 REM CLEAR SCREEN WITH EXISTING INFO
600 CH$=""
610 W=1: GOSUB 270
620 S1=NO
630 ST$(1)=ST$(S1)
640 ST$(2)=CN$(S1)
650 FOR Q=1 TO 4: A$(S1,Q)=ST$(2+Q): NEXT
660 ST$(7)=P1$(S1)
670 FOR Q=1 TO 4: D$(S1,Q)=ST$(7+Q): NEXT
680 ST$(12)=P2$(S1)
690 ST$(13)=BL$(S1)
700 PRINT "" + SP$(LEN(TS$)-1)
710 FOR W=1 TO 4: PRINT "": NEXT
720 PRINTST*: THEN IFNO(T)+1=999 THEN 530
730 PRINTSP$(LEN(T)+1)-LEN(TS$)
740 NEXT
750 RETURN
760 REM WRITE AWAY DATA SUBROUTINE
770 PRINT "LOAD CUSTOMER TAPE"
780 OPEN 1, 1, "CUSTOMER"
790 FOR NO=1 TO 99
800 IF NO= "NO" THEN 940
810 PRINT NO: PRINT NO.. CN$(NO)
820 FOR Q=1 TO 4: IF A$(NO,Q)="" THEN PRINT LA$(NO,Q): NEXT
830 PRINT "NO"
840 NEXT
850 IF P1$(NO)="" THEN PRINT P1$(NO): GO TO 940
860 PRINT "NO"
870 NEXT
880 IF P2$(NO)="" THEN PRINT P2$(NO): GO TO 940
890 PRINT "NO"
900 IF BL$(NO)="" THEN PRINT BL$(NO): GO TO 940
910 PRINT "NO"
920 NEXT
930 PRINT "NO"
940 NEXT
950 PRINT "NO"
960 CLOSE 1
970 PRINT "CUSTOMER TAPE UPDATED": RETURN
980 REM BUILD SCREEN SUBROUTINE
990 R=1: T=1: OF=1023: CF=55295: SC$="SCREEN"
1000 PRINT "LOAD SCREEN TAPE"
1010 OPEN 1, 1, 0, SC$="SCREEN"
1020 INPUT 1, A$: IF A$=999 THEN EN(T)=A: RETURN
1030 AC=ASC(MID$(A$,R,1)):PK=AC
1040 IF AC>63 THEN PK=AC-64
1050 IF AC>95 THEN PK=AC-32
1060 POKE OF+$A+R,PK
1070 POKE CF+$A+R,1
1080 !F LEN(A$)=1 AND A$="(" THEN EN(T)=A+1
1090 IF LEN(A$)=1 AND A$=")" THEN EN(T)=A-EN(T): T=T+1
1100 R=R+1:GOTO 1010
1110 RETURN
1120 REM ACCEPT INPUT SUBROUTINE
1130 REM 'P' IS THE START POSITION OF THE CURRENT LINE.
1140 REM 'I' IS THE CURRENT POSITION IN THE LINE.
1150 REM 'J' IS THE CURRENT POSITION IN THE PRINT LINE WHEN PRINTING.
1160 REM 'LC' IS THE LAST CHARACTER POSITION IN THE CURRENT LINE.
1170 REM 'N' IS A POSITION IN THE CURRENT LINE USED DURING LINE MANIPULATION.
1180 REM 'LI' IS AN ARRAY OF 80 SINGLE CHARACTERS, EACH REPRESENTING ONE LINE PSN.
1190 REM 'FL' IS THE CURSOR.
1200 IF I=0 THEN P=1:CH$=" "
1210 PRINT "":
1220 FOR O=1 TO EN(P):PRINT "" : NEXT
1230 FL$(1)="":FL$(2)=""
1240 REM NEXT FIELD ENTRY POINT
1250 FOR J=0 TO LC:LI$(J)=LI$(J+1)
1260 FOR J=1 TO EN<P>:LI$(J)=LI$(J-1)
1270 !FASC(CH$)<32 THEN 1250
1280 IFASC(CH$)=133 THEN 1550
1290 REM NEXT CHAR ENTRY POINT
1300 IFASC(CH$)=130 THEN 1410
1310 IFASC(CH$)=141 THEN 1490
1320 IFASC(CH$)=138 THEN 1410
1330 IFASC(CH$)=139 THEN 1410
1340 FOR L=1 TO 2:PRINT "": NEXT
1350 FOR K=1 TO 30:PRINT "": NEXT
1360 GETCH$:PRINT FL$(1):PRINT "":
1370 IF CH$="" THEN 1410
1380 NEXT
1390 NEXT
1400 GOTO 1300
1410 PRINT FL$(2):!FASC(CH$)=20 THEN 1660
1420 IFASC(CH$)=148 THEN 1750
1430 IFASC(CH$)=167 THEN 1860
1440 IFASC(CH$)=29 THEN 1900
1450 IFASC(CH$)=133 THEN 1550
1460 IFASC(CH$)=134 THEN 1590
1470 IFASC(CH$)=135 THEN 1580
1480 IFASC(CH$)=130 THEN 1500
1490 IFASC(CH$)<32 THEN 1290
1500 IFASC(CH$)>127 AND ASC(CH$)<130 THEN 1290
1510 CA=CH$
1520 IFASC(CH$)=34 THEN CA="**"
1530 I=I+1:PRINT CA$:LI$(I)=CH$:GOTO 1290
1540 LI$(I)=CH$
1550 REM STOREING
1560 SN$=""
1570 IFEN(P+1)<999 THEN FOR O=EN<P>+1 TO EN<P+1>-1:PRINT "": NEXT
1580 P=P+1
1590 I=0:LC=0
1600 IF EN<P>=999 THEN RETURN
1610 GOTO 1290
1620 LC=LC-1
1630 I=I-1:IFI=0 THEN RETURN
1640 IFASC(CH$)=20 THEN 1290
1650 IFASC(CH$)=14 THEN 1290
1660 IFASC(CH$)=15 THEN 1290
1670 IFASC(CH$)=29 THEN 1290
1680 IFASC(CH$)=34 THEN 1290
1690 IFASC(CH$)=128 THEN 1290
1700 FOR N=1 TO EN<P>-I:PRINT "": NEXT
1710 FOR N=I TO EN<P>:PRINT "": NEXT
1720 FORM=I+1:ITOLC+1:LI$(N)=LI$(N+1)
1730 NEXT
1740 GOTO 1290
1750 REM INSERT
1760 IFI=0 THEN RETURN
1770 FORM=I+1:ITOLC
1780 IFASC(LI$(N))=34 THEN PRINT "" : NEXT
1790 IFASC(LI$(N))=34 THEN PRINT LI$(N)
1800 NEXT
1810 FORM=I+1:PRINT "" : NEXT
1820 FORM=LC+1+STEP-I:LI$(N)=LI$(N-1)
1830 NEXT
1840 LI$(I+1)=""
1850 GOTO 1290
1860 REM CURSOR <-
1870 IFEN<P>=999 THEN RETURN
1880 PRINT "": FOR L=1 TO 80:PRINT "": NEXT
1890 FOR J=I TO LC:LI$(J)=LI$(J-1)
1900 NEXT
1910 IFI=0 THEN RETURN
1920 PRINT "": FOR L=1 TO 80:PRINT "": NEXT
1930 REM CLEAR ENTRIES SUBROUTINE
1940 PRINT "": FOR N=1 TO EN<P>:PRINT "": NEXT
1950 FOR J=1 TO 80:PRINT "": NEXT
1960 IFEX(U)<999 THEN FOR R=1+EN(U)+EX(U):TOEN(U+1):PRINT "": NEXT
1970 IFEN(U+1)<999 THEN FOR R=1+EN(U)+EX(U):TOEN(U+1):PRINT "": NEXT
1980 RETURN
1990 REM CURSOR ->
2000 I=I+1:IF I=EN<P> THEN RETURN
2010 PRINT FL$(2):GOTO 1290
2020 IFEX(U)<999 THEN FOR R=1+EN(U)+EX(U):TOEN(U+1):PRINT "": NEXT
2030 IFEN(U+1)<999 THEN FOR R=1+EN(U)+EX(U):TOEN(U+1):PRINT "": NEXT
2040 RETURN
Another useful little routine for business programs is a date routine for displaying today's date in the top right hand corner of the screen. The parts you want can be selected from the program below which operates on a VIC 20.

```basic
100 REM ***************
110 REM **
120 REM **date-vic20**
130 REM **
140 REM ***************
150 vi=36878: sb=36879: s3=36826: REM define colour & sound
160 PRINT "":POKE sb,186
170 PRINT "":POKE sb,186
180 PRINT "":POKE sb,186
190 PRINT "":POKE sb,186
200 DIM m$(12)
210 FOR mn=1 TO 12: read m$(mn): NEXT
220 DATA JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
230 :  
240 INPUT "day"; a$: a$=VAL(a$)
250 INPUT "month"; b$: b=VAL(b$)
260 IF a=0 THEN PRINT "invalid day"; GOTO 420
270 IF b=0 THEN PRINT "invalid month"; GOTO 420
280 ON b GOTO 310,300,310,300,310,300,310,300,310,300,310,300,310,300,310
290 :
300 IF a>29 THEN PRINT "invalid day"; GOTO 420
310 IF a>31 THEN PRINT "invalid day"; GOTO 420
320 GOTO 350
330 IF a>30 THEN PRINT "invalid day"; GOTO 420
340 GOTO 350
350 IF b>12 THEN PRINT "invalid month"; GOTO 420
360 INPUT "year"; c$: c$=VAL(c$)
370 IF c$<0 OR c$>40 THEN PRINT "invalid day"; GOTO 420
380 IF LEFT$(c$, 2)<>"19" THEN c$="19" + c$ : c$=1900+c$ 
390 IF 1984>c$ THEN PRINT "invalid year"; GOTO 420
400 GOTO 530
410 :
420 REM ***************
430 REM * bleep **
440 REM ***************
450 POKE v1,15
460 POKE 36876,225
470 FOR n=1 TO 500
480 NEXT
490 POKE v1,0
500 POKE s3,0
510 GOTO 240
520 :
530 PRINT "":POKE sb,27: REM restore to normal colours
540 PRINT a$"/"b$"/"c$
550 PRINT 
```

Many people use spreadsheet programs and you can buy one of the standards on the market. However, if you wish to play around with the idea, this program may help:

```
5 REM DATA
10 ND=10
20 SC=53281: BD=53280: POKE SC, 1
30 POKE 650, 128: REM ** REPEAT KEYS **
40 DIM DT$(12,10), MN$(12): REM *' REPEAT KEYS '
50 FOR MN=1 TO 12: READ MN$(MN): NEXT
60 MN$(MN)="Jbr"+MN$(MN)+" : NEXT
70 DATA JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
80 OPEN 1.0:"DATA": FOR MN=1 TO 10: FOR DT=1 TO 10: INPUT UT#2, DT$(MN, DT)
90 FOR DT=1 TO 10: IF DT$(MN, DT)=" " THEN 170
100 PRINT" NGO GET DATA FROM TAPE OR ENTER?"
110 PRINT "ENTER NO. MONTH TO START ENTERING AT"
120 INPUT#1, TE$: PRINT"PRINT"
130 MB=VAL(TE$): IF MB=0 OR MB>12 THEN MB=1
140 IF LEFT$(TE$, 1)<>"T" THEN 190
150 OPEN 2.1.0."DATA": FOR MN=1 TO 12: FOR DT=1 TO 10: INPUT UT#2, DT$(MN, DT)
160 IF DT$(MN, DT)=" " THEN 170
170 FOR DT=1 TO 10: IF DT$(MN, DT)=" " THEN 190
180 FOR MN=1 TO 12
190 PRINT MN$(MN)
```
The two programs below are a pair for programmed learning. The first, TEACHER allows questions and multiple choice answers to be set up on a tape. The second, PUPIL allows the pupil to be tested and scored against the data set up on tape.

10 REM TEACHER
20 OPEN 4,4
30 DIM L$(19),R$(1)
40 SB=36870:FL=1
50 POKE SB.26
60 PRINTCHR$(14)CHR$(13)
70 PRINT"READY.
80 REM TEST FOR END OF PROGRAM.
90 IF L$(1)="THEN400
100 REM SET UP CORRECT ANSWER.
110 PRINT"RANSW€R"
120 PRINT" ":
130 GOSUB 470
140 IF R$(1)="THEN290
150 NEXT M
160 M=M-1
170 IF M=1 THEN PRINT"ALTERNATIVES!":GOTO 220
180 PRINT#:100
190 PRINT#:1,CHR$(13)
200 IF L$(1)="THEN 490
210 NEXT DT,MN
220 IF N=0 TO 490
230 IF N=2 AND MN=13 THEN MN=1
240 IF MN=13 THEN MB=0:MN=MB+N
250 PRINT "":TAB((N)*13):MN
260 FOR DT=1 TO 10
270 PRINT:PRINT:PRINT
280 FOR P=1 TO M:PRINT#:P:PRINT#:1:PRINT#:100:PRINT#:1,CHR$(13)
290 IF R$(1)="THEN 380
300 =A+1
310 GOTO 120
320 PRINT#:1,CHR$(13)
330 IF R$(1)="THEN 410
340 PRINT"PROGRAM COMPLETE."
430 CLOSE1
440 CLOSE4
450 PRINT"*TOP TAPE: REWIND AND LABEL"
460 END
470 REM LINE SETTING UP SUBROUTINE
480 I=0:LC=0
490 FL$(1)="~":FL$(2)="\n"
500 REM MAIN LOOP
510 IFLC<13 THEN LC=I
520 IFLC=13 THEN FL$(2)=LI$(I+1):IFASC(LI$(I+1))=34 THEN
530 IFLC=13 THEN FL$(2)=LI$(I+1):IFASC(LI$(I+1))=34 THEN
540 IFLC=13 THEN LI$(I+1)=IFASC(LI$(I+1))=34 THEN
550 FORL=1 TO 2
560 FORK=1 TO 30
570 OCTM$:PRINTFL$(L)"\n"
580 IFCH$="\n" THEN 620
590 NEXTK
600 NEXTL
610 GOTO510
620 PRINTFL$(2)"\n":IFASC(CH$)=20 THEN 780
630 IFASC(CH$)=148 THEN 590
640 IFASC(CH$)=157 THEN 580
650 IFASC(CH$)=29 THEN 570
660 IFASC(CH$)=130 THEN 560
670 IFASC(CH$)=33 THEN 550
680 IFASC(CH$)=127 THEN 540
690 CH$="\n"
700 IFASC(CH$)=34 THEN CA$="~"!!"
710 I=I+1:PRINTCA$:IFASC(CA$)=34 THEN 730
720 REM************ ** LINE SET **
730 PRINT:SR$=""
740 FORJ=1 TO I+1
750 SR$=SR$+LI$(J):LI$(J)="\n":NEXT J:
760 PRINT
770 RETURN
780 REM************ ** DELETE **
790 I=I-1:IFICTHENI=0:Goto500
800 LC=LC-1
810 PRINTCH$;
820 FORN=1+I TO I+1:LI$(N)=LI$(N+1)
830 NEXTN
840 GOTO500
850 REM************ ** INSERT **
860 PRINT" \n":LC=LC+1:LI$(LC)=" \n"
870 FORN=I TO I+1
880 IFASC(LI$(N))=34 THEN PRINT"\n"
890 IFASC(LI$(N))=34 THEN PRINTLI$(N):NEXTN
900 REM************ ** DELET **
910 FORN=ITOLC:PRINT":NEXTN
920 FORN=LCTO1+1STEP-1:LI$(N)=LI$(N-1)
930 NEXTN
940 LI$(I+1)="\n"
950 GOTO500
960 REM************ ** DELET **
970 I=I-1:IFICTHENI=0:Goto500
980 PRINT":NEXTN
990 GOTO500
1000 REM************ ** DELET **
1010 I=I+1:IFICTHENLI$(I)="\n"
1020 PRINTFL$(2):GOTO500
1030 READY.

10 REM PUPIL
20 SB=36879:N=1
30 POKE SB,29
40 PRINTCHR$(14)" I=P "
50 PRINT"
60 PRINT"LOAD QUESTION TAPE & PRESS "L"I"
70 WAIT37151,64,64
80 OPEN1,1,0:"DATA"
90 REM GET INITIAL DATA
100 GOSUB500
110 REM GET THE QUESTION
120 QUS=RI$(1)
130 IFLEFT$(QUS,1)="\n" THEN 430
140 ANS=RI$(2)
150 FORJ=1 TO 2
160 WR$(J)="\n"
170 NEXTJ
180 WR$(Q-1)=AN$;
190 FORN=1 TO Q-2
200 WR$(N)=RI$(N+2)
210 NEXTN
220 PRINT"J QUESTION"
230 PRINT"
240 PRINT"Q="
250 REM RANDOMISE THE ANSWER POSITION IN THE LIST
260 A=INT(RND(1)*(Q-1)+1)
270 CR=Q-A
280 FORK=1 TO Q-1
290 PRINT"\n"WR$(K):WR$(A)
300 A=A+1:IFACR=Q-1 THEN 400
310 NEXTK
320 REM LOAD NEXT FROM TAPE BEFORE ACCEPTING ANSWER
330 GOSUB500
A screen dump program is always useful to produce on the printer a copy of what is at present on the screen.

10 REM PRINT SCREEN (TOP 22 LINES)
20 SC=1024:OPEN 4, 4:CMD4=PRINT CHR$(14):
30 FOR Y=0 TO 22
40 FOR X=0 TO 39
50 PK=PEEK(X+(Y*40)+SC); LE=PK
55 IF PK>127 THEN PK=PK-128:PRINT CHR$(18):
60 IF PK=96 THEN LE=32:GOTO 150
70 IF PK<32 OR PK>96 THEN LE=PK+64:GOTO 150
80 IF PK<36 AND PK>63 THEN LE=PK+32:GOTO 150
90 IF PK>96 THEN LE=PK+64
150 PRINT CHR$(LE)CHR$(146);
340 PRINT "----START-----":NEXT
360 GET K$:IF K$="":GOTO 360
370 FOR N=1 TO 9:PRINT:PRINT RIGHT$(A$:LEN(A$)-25):NEXT
380 IF CH=2 THEN CH=0:GOTO 410
390 IF CH=2 THEN CH=0:GOTO 410
400 REM *****************************************
410 REM ****** THE PROG ITSELF ******
420 REM *****************************************
430:
440 GOSUB 670:REM *** PRINT ROAD ***
450 IF CH=2 THEN CH=0:GOTO 510
460 :
470 GOSUB 770:REM *** MOVE CAR ***
480 :
490 GOTO 410
500:
510 REM ****** ENDING ******
520 PRINT "TOMTWIN!!
530 SC=VAL(TI*)*(10-L)
540 PRINT "E";SC IF SC=HI THEN W=SC
550 FOR N=1 TO 2000:NEXT
560 X=0:G=1
570 GOTO 270
580:
590 REM ****** SHAPES ON ROAD ******
600 B$="*
610 B$="O
620 B$="B
630 B$="*
640 B$="+
650 B$="X
660 :
670 REM ****** PRINTING ROAD ******
680 X=X+1
690 IF X<L THEN PRINT:GOTO 730
700 RD=INT(RND(1)*6)+1
710 ON RD GOSUB 600,610,620,630,640,650
720 PRINT TAB(6+INT(RND(5)*9));B$;X=0
730 :
740 PRINT A$
750 RETURN
760:
770 REM ****** MOVE CAR ******
780 IF PEEK(CR)<32 AND PEEK(CR)<V THEN CH=2:RETURN
790 GET K$:
800 IF K$="Z" THEN CR=CR-1
810 IF K$="M" THEN CR=CR+1
40 PRINTD1$: D$: A$: "\": "RETURN
41 POKESC, 24: POKES 1, 128: PRINTD1$: A$: \"W0\": FOR N=1 TO 10
42 FOR X=1 TO 10: NEXT X: PRINTD1$: A$: \"I\"MQt---.0/\"": PRINTD1$: D$: A$: \"J\": FOR N=1 TO 10
43 FOR X=1 TO 10: NEXT X: PRINTD1$: D$: A$: \"I\"MQt---.O/\"": PRINTD1$: D$: A$: \"J\": FOR N=1 TO 10
44 PRINTD1$: FOR N=1 TO 10: NEXT N: PRINTD1$: A$: \"T$; A$: \"*: \"
45 PRINTD1$: A$: \"MELL\": \"DONE!*";
47 PRINT"YOU MUST DECIPHER THE WORD IN THE BLOCK OF \"M\"S. IF YOU FAIL,"
FOR J=1 TO 20
90 IF Y(J)=-999 OR Y(J)=199 THEN NA=J-1:GOTO 150:
REM END OF ENTRIES IN X OR Y
100 IF X(J)>MX THEN MX=X(J):REM PUSH UP MAX IF REQUED
110 IF Y(J)>MY THEN MY=Y(J):REM PUSH DOWN LOWER LIMIT IF REQUIRED
120 IF Y(J)<NY THEN NY=Y(J):REM Ditto
130 NEXT J
140 PRINT "J"
150 FOR I=1 TO 20=PRINT " J":NEXT=REM SET UP VERT ICAL LEFT LINE FOR Y AXIS
160 PRINT "Y"
170 FOR I=1 TO 35=PRINT "\":=NEXT=PRINT=REM SET UP HORIZONTAL LINE FOR X AXIS
180 REM X-AXIS
190 PRINT FNA(NX):
200 PRINT TAB(35)FNA(MX)
210 REM Y-AXIS
220 PRINT "Y"
230 PRINT FNA(NY):
240 FOR K=LEN(NY)-1 TO 15=PRINT:NEXT K
250 NY=STR$(FNA(NY)):FOR I=1 TO LEN(NY)$=PRINT M:
260 FOR K=LEN(MY)-1 TO 15=PRINT:NEXT K
270 MY=STR$(FNA(MY)):FOR I=1 TO LEN(MY$)=PRINT M:
280 FOR I=1 TO 15=PRINT=".":=NEXT=PRINT=REM CLEAR SCREEN?
290 IF K$="n" or bS="N" then 300
300 FOR I=-10 TO 10: X=I:9osub 520=next
310 REM CHANGE 290 TO EXIT TO NEXT PART OF YOUR PROGRAM
320 READY.

High resolution graphics can be slow but obviously will give a more precise picture than anything constructed above. One of the restrictions of CBM BASIC is that there are no language commands for graphics, sound or colour and all this manipulation has to be done by use of POKE commands. This is illus-
The above program can be easily adapted, using different locations for the POKEs, for the VIC 20, although you will be tight for space if you try to do anything useful on an unexpanded VIC 20. Sprites, however, are only available on the 64 and as the manual does not give a routine for generating them, one is included here. The numbers to be used for POKEing into the sprite data area are listed on the printer at the end of the program.
Finally, you may be interested in the little routine used to print these listings in 50 column format for convenience of printing in this book. First a listing must be made to the tape by loading any program say PROGRAM for which the listing is required, and typing in direct mode

OPEN 1,1,"LISTING"

...press play/record and wait for READY.

CMD1: LIST

...wait for the listing to finish, then

PRINT #1 followed by CLOSE 1

Then load the program below, set up the printer and run.

10 OPEN 1,1,0,"listing": OPEN 4,4: CT=0
20 GET#1, A$; IF ST<>B THEN 50
25 LN$=LN$+A$   
30 CT=CT+1: IF A$=CHRS(13) THEN CT=0: GOSUB 60 
40 IF CT=50 THEN CT=0: GOSUB 60: PRINT#4: PRINT#4,"" 
45 GOTO 20
50 PRINT#4: CLOSE1: CLOSE4
55 END
60 PRINT#4, LN$; LN$=CHR$(17): RETURN

ready.

790 POKE V+5, Y: REM *** loc of sprite 3
800 GOTO 710

340 IF KY$=" " OR ASC(KY$)=160 THEN N=1: IF YN=1 THEN N=0
350 YN=N
360 IF KY$="e" THEN 400: REM *** ending
370:
380:
390 IF Peek(ob)<80 AND Peek(ob)>96+128 THEN ob=b:
400 POKE ob, 80, PK ob+cl, 5, PK ob, 96+128, PK ob 
410 POKE ob, 80, PK ob+cl, 5
420 GOTO 200
430:
440:
450 REM *******************************************
460 REM *** working out sprite ***
470 REM *******************************************
480 POKE sc, 6, dim SP(65): REM move dim to start if 
490 for n=1 to 21
500 for n=1 to 3
510 no=1023+((12-1)*N)+(11*40)
520 for n=1+no to 8+no
530 IF Peek(13+45)=224 THEN n=n+2+(8-(13-no))
540 NEXT 13
550 SP(ri)=m: ri=ri+1: m=0
560 NEXT 12
570 NEXT 11
580:
590 REM *** Printout ***
600 OPEN 4,4: CMD4: PRINT"": FOR n=0 TO 62: PRINT SP(n);: NEXT n: PRINT#4
610 :
620 :
630 :
640 :
650 :
660 PRINT"":
670 V=53248: POKE V+21, 4: REM sprite 3
680 POKE 2042, 13: REM ** data loc
690 FOR n=0 TO 62: POKE 632+n, SP(n): NEXT: REM *** p 
700 y=100: x=y: POKE v+41, 1
710 GET KYS: IF KYS=" " THEN 710
720 XX=x: YY=y
730 IF KY$=" " THEN y=y+3: REM down
740 IF KY$=" " THEN y=y-3: REM up
750 IF KY$=" " THEN x=x+3: REM right
760 IF KY$=" " THEN x=x-3: REM left
770 IF X>255 OR x<0 OR y>255 OR y<0 THEN X=XX: Y=YY
780 POKE v+4, x: REM *** x loc of sprite 3
SECTION 4 Glossary of Common Terms

Addressing
The computer needs to talk to its memory and to various peripherals. Each memory location and peripheral has an address, usually unique, which the computer uses to get it to accept or send information. For non unique addresses, the computer must have only one item at a time responding or there will be trouble. It is therefore important to ensure that a cartridge is not fitted while extra RAM memory occupying the same address is switched in.

Algorithm
The method used for solving a problem. As the designer and programmer cannot solve the actual problems the computer will meet, all they can do is to provide it with the means or algorithm required to solve these problems.

Array
A set of variables held as a single list e.g. A(1), A(2).... or as a multi dimensional array or matrix e.g. B$(1,1), B$(1,2).... when variables are related such as a set of cars which may be identified by serial numbers 1 to 20. Then the drivers' names would be held as say NAS$(N) where N is the car number. Two dimensional arrays are useful when comparing month-by-month expenses for the 20 cars when EX(15,3) could represent the expenses in March for car number 15.

ASCII
American Standard Code for Information Interchange. This is one of the standard conventions for converting characters to numbers as stored on a computer or sent along a communications link. The other main standard is EBCDIC which differs only slightly.

Assembler
An aid to writing programs in machine code. Instead of writing down the actual machine code entries as must be done in POKE statements, this allows the use of mnemonics such as LDA (load A register). Not covered by this book.

Base
Normal decimal numbers are to the base 10. Other bases used are binary (2), octal (8) and hexadecimal (16). Binary digits can only be 1 or 0, octal digits are between 0 and 7, and hexadecimal digits are between 0 and 15 (i.e. 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

Whereas people are familiar with decimal, computers work in binary. Three binary digits can be displayed as a number between 0 and 7 (octal) and 4 binary digits can be expressed as hexadecimal, so making these two particularly useful in writing down long strings of binary digits without making it difficult for the human observer to convert them back again, as each digit is individually converted to 3 or 4 binary digits. For example decimal 63 is binary 111111 or OCT 77 or HEX 3F.

BASIC
Beginners All-purpose Symbolic Instruction Code - the language supplied with almost all microcomputers, both business and home, the high level language described in this book. High level means that it does the work of converting your wishes from an English-like language to machine code and memory storage without troubling you with the details. COBOL, FORTRAN and RPG are also high level languages. Low level languages like Assembler require you to know a great deal
about how the processor and some other chips operate.

**Binary**
See Base.

**Bit**
An element of a computer. In almost all computers today this element can be in one of only two states represented by 1 and 0 i.e. as a binary digit.

**Boolean Logic**
The logic used in AND OR NOT statements (q.v.).

**Buffer**
An area of memory held for temporary storage before moving data around in larger chunks – usually for writing out to cassette, which requires 192 bytes, or to disk or printer. These buffers are handled by the computer and do not normally need any direct intervention by the programmer. They can be in the computer memory itself or in the peripherals.

**Bug**
An error, usually in software but sometimes used to refer to problems in the design of a chip or other hardware.

**Byte**
A contiguous set of bits, usually 8 (always 8 on these machines) in number, occupying one memory location. This width is sufficient to hold an ASCII character, converted to a number between 0 and 255, and this therefore often represents one character, e.g. 65 is ASCII code for ‘A’, 66 is ASCII code for ‘B’.

**Compiler**
A compiler scans the BASIC code before it is run and sets up the machine code ready for the run. The BASIC interpreter normally used (the one you get with the system) deals with every line as it reaches it whereas the compiler has scanned them all to produce more efficient code. Unfortunately a true high efficiency compiler does not to my knowledge exist at the time of writing, but there are intermediate compilers which can still improve speed performance by about a factor of 5, and save some space.

**Default**
This describes what will happen if you make no positive decision in a particular case. For instance, if you do not DIMension an array before the program comes across it, its dimensions will default to 11.

**Device**
Any piece of hardware e.g. DEVICE NOT PRESENT as a message when addressing, say the printer, means that the computer is not getting the right response and therefore does not recognise it as present. See Peripherals.

**Function**
A mathematical formula that can be defined at the start of a program and then used as shorthand within the computer (see FN), or its ordinary English meaning as in ‘the function of the keyboard is to allow data entry’.

**Garbage**
A term referring to the small pieces of memory left after string manipulation has altered lengths of strings and left spaces in between too small to be useful. Garbage collection takes place when the strings are shuffled about to create one large usable space. See FRE.
Hex
See Base.

I/O
The standard abbreviation for Input/output to or from the computer from its peripherals or other items in the outside world.

Input
Input is information coming into the computer from outside.

Interface
The contact between a computer and its peripherals or its contact with the outside world through the RS232 interface or the user ports. The RS232 interface is a standard hardware socket, but the VIC/CBM 64 only has 5 volt output and must be converted to 12 volt standard for compatibility with most other RS232 devices.

Keywords
Words having a specific meaning to BASIC such as GOTO. Using long BASIC variable names, there is always a chance that variables may be confused (by the Interpreter) with Keywords. TOTAL is confused with the keyword TO and will give a SYNTAX ERROR.

Kilobyte (k)
One thousand bytes, or to be exact 1024 bytes of memory.

Logic
Logic is the structure on which we try to build our computer systems. The computer is a hard taskmaster and reveals the flaws in our logical powers! (see also AND NOT OR boolean logic)

Logical line
A logical line is up to 80 characters long. Because the screen cannot hold that number of characters, there can be more than one screen line per logical line. When entering BASIC lines these are only terminated by a <RETURN>. The screen lines have little significance to the program except when working on screen displays.

Machine code
The natural language of any computer, consisting, fundamentally, of a series of ON-OFF switches, represented by the binary numbers 0 and 1.

In an 8 bit computer, each machine code instruction is provided by a block of eight binary digits, which for convenience, we normally convert to either Hexadecimal or decimal numbers, as long strings of zeros and ones can be unwieldy and difficult to remember.

Short lengths of machine code can be created by POKEing numbers into RAM locations. This can be done only for a few locations, without becoming thoroughly confused, and it is better to use an Assembler for any significant lengths of code. Remember that machine code does not insulate you from crashing the system, like BASIC does, so frequent SAVEs to disk of any BASIC using POKEs and SYS statements is desirable. It is essential to SAVE any machine code program before running it for the first time.

Nesting
Nesting is the very useful technique of placing loops inside each other in order to process arrays or other large structured data. FOR ...NEXT loops can be nested, as can subroutines (GOSUB). Examples abound in the main text (FOR,GOSUB) and in Section 3.
Octal
Not often used now – see Base.

Output
The transfer of information from the computer to its peripherals or the outside environment (e.g. network).

Peripherals
The devices attached to the computer to allow it to interact effectively with the outside world. Examples are monitor, printer, tape cassette unit and disk drive. See Device.

Port
A plug connection for peripherals. The user ports are plug connections for anything you care to connect. However as these ports connect directly to the works of the computer caution is advised!

RAM
Random Access Memory is memory on chips within the machine, the contents of which can be changed as required by the programmer or the interpreter.

Real
Means that the facility (e.g. memory) you are using is actually present on the machine. Not relevant to most micros – all the memory you can access by POKE and PEEK is really there – see Virtual.

Real numbers
Numbers expressed to one or more decimal places. The opposite of integers.

ROM
Read Only Memory is memory with a fixed pattern burnt into the chips which the computer can interpret as either data or programs. The BASIC interpreter is held in this way, and read into RAM on starting the CBM 64.

Software
Programs used to make the computer operate.

Statement
A statement is BASIC code terminated by a colon or a <RETURN>, whichever is sooner.

Structured Programs
These are built up of subroutines (or the equivalent in other languages) in a systematic and ordered manner. GOTOS are rare as each subroutine returns only to the subroutine that called it, or to the main program code which consists mainly of a series of GOSUBs.

Syntax
The structure or ‘grammar’ of a language, whether it is English, German or BASIC is known as the syntax. Bad BASIC syntax will produce a SYNTAX ERROR and no further progress will be possible until this is corrected. In other words, syntax is the rule book for the language.

Transparent
The facility you are using is not visible to you. Loading and use of the BASIC interpreter is transparent because it all happens without you having to know any more than the rules of the BASIC language. See also Virtual.

Variable
A variable is the name given by you to a location or locations in memory where information is held. BASIC decides where to put the variable and how much space to allow for it from the way you use it and from its name terminator if any (i.e. $ for strings and % for integers). For example if you say A=2 in your program, BASIC will check to see if A exists and if not will create a location for it. It will then put the value 2 in that location. If A already exists, the new value you are assigning to it will overwrite
Virtual
This term is used when your machine pretends it has facilities that it does not have. Some machines have only 64 kbytes of memory but if the user wants to use more than that, some of the 64k can be stored on disk and replaced by disk memory which is given addresses above 64k, so appearing to the user (you, that is) as if it had, say, 128k. Not a facility provided by CBM machines, but you can use it in your own programs to store part of a spreadsheet on disk if it's too large to go in memory.

Remember:
If you can see it and it's there, it's real
If you can see it but it's not there, it's virtual
If you can't see it but it's there, it's transparent
If you can't see it and it's not there, it's gone!

Appendix A – Control Characters used in the Programs

- "\s" - HOME
- "\t" - CLR
- "\n" - CURSOR DOWN
- "\r" - CURSOR UP
- "\d" - CURSOR RIGHT
- "\l" - CURSOR LEFT
- "\f" - REVERSE ON
- "\v" - REVERSE OFF

- "\b" - BLACK - CTRL & 1
- "\w" - WHITE - ETC
- "\e" - RED
- "\c" - CYAN
- "\p" - PURPLE
- "\g" - GREEN
- "\x" - BLUE
- "\y" - YELLOW
- "\m" - F1 - FUNCTION KEYS
- "\h" - F2
- "\i" - F3
- "\j" - F4
- "\k" - F5
- "\l" - F6
- "\m" - F7
- "\n" - F8

READY.

control characters used in the programs

- "\h" - home
- "\i" - clr
- "\j" - cursor down
- "\k" - cursor up
- "\m" - cursor right
- "\l" - cursor left
- "\f" - reverse on
- "\e" - reverse off
- "\b" - black - ctrl & 1
- "\w" - white - etc
- "\e" - red
- "\c" - cyan
- "\p" - purple
- "\g" - green
- "\x" - blue
- "\y" - yellow
- "\m" - f1 - function keys
- "\h" - f2
- "\i" - f3
- "\j" - f4
- "\k" - f5
- "\l" - f6
- "\m" - f7
- "\n" - f8

READY.
Appendix B – Characters Printed for ASCII Values

<table>
<thead>
<tr>
<th>ASCII</th>
<th>UPPER</th>
<th>LOWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>65</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>66</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>67</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>68</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>69</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>70</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>71</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>72</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>73</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>74</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>75</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>76</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>77</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>78</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>79</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>80</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>81</td>
<td>q</td>
<td>q</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII</th>
<th>UPPER</th>
<th>LOWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>83</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>84</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>85</td>
<td>u</td>
<td>u</td>
</tr>
<tr>
<td>86</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>87</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>88</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>89</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>90</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>91</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>92</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>93</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>94</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>95</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>96</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>97</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>98</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>99</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>100</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>101</td>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>102</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>103</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>104</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>105</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>106</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>107</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>108</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>109</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>110</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>111</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>112</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>113</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>114</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>115</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>116</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>117</td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td>118</td>
<td>]</td>
<td>]</td>
</tr>
<tr>
<td>119</td>
<td>^</td>
<td>^</td>
</tr>
<tr>
<td>120</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>121</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>122</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>123</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>124</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>125</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>126</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>127</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>128</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>129</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>130</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>131</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>132</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>133</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>134</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>135</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>136</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>137</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>138</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>139</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>140</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>141</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>142</td>
<td>u</td>
<td>u</td>
</tr>
<tr>
<td>143</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>144</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>145</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>146</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>147</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>148</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>149</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>150</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>151</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>152</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>153</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>154</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>155</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>156</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>157</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>158</td>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>159</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>160</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>161</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>162</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>163</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>164</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>165</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>166</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>167</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>168</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>169</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>170</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>171</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>172</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>173</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>174</td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td>175</td>
<td>]</td>
<td>]</td>
</tr>
<tr>
<td>176</td>
<td>^</td>
<td>^</td>
</tr>
<tr>
<td>177</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>178</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>179</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>180</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>181</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>182</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>183</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>184</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>185</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>186</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>187</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>188</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>189</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>190</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>191</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>192</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>193</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>194</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>195</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>196</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>197</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>198</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>199</td>
<td>u</td>
<td>u</td>
</tr>
<tr>
<td>200</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>201</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>202</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>203</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>204</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>205</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>206</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>207</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>208</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>209</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>210</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>211</td>
<td>G</td>
<td>G</td>
</tr>
</tbody>
</table>

...
### Appendix C Some Useful Memory Locations

<table>
<thead>
<tr>
<th>HEX ADDRESS</th>
<th>DECIMAL LOCATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001-0002</td>
<td>1-2</td>
<td>User jump to location (high byte – low byte)</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>6510 On-Chip 8-Bit I/O Register</td>
</tr>
<tr>
<td>0013</td>
<td>19</td>
<td>Flag: INPUT Prompt POKE 19, 1 suppresses the ? but also affects PRINTING, so must be restored to 0 immediately afterwards</td>
</tr>
<tr>
<td>0016</td>
<td>22</td>
<td>Pointer to temporary string stack –35 in here supresses line numbers in listings</td>
</tr>
<tr>
<td>0026-002A</td>
<td>38-42</td>
<td>Floating Point Product Of Multiply</td>
</tr>
<tr>
<td>002B-002C</td>
<td>43-44</td>
<td>Pointer to start of BASIC program</td>
</tr>
<tr>
<td>002D-002E</td>
<td>45-46</td>
<td>Pointer to start of variables (following program)</td>
</tr>
<tr>
<td>002F-0030</td>
<td>47-48</td>
<td>Pointer to start of arrays (following variables)</td>
</tr>
<tr>
<td>0031-0032</td>
<td>49-50</td>
<td>Pointer to position after end of arrays</td>
</tr>
<tr>
<td>0033-0034</td>
<td>51-52</td>
<td>Pointer to bottom of string storage (moving down)</td>
</tr>
<tr>
<td>0037-0038</td>
<td>55-56</td>
<td>Highest address used by BASIC</td>
</tr>
</tbody>
</table>

### Current BASIC line number

<table>
<thead>
<tr>
<th>HEX ADDRESS</th>
<th>DECIMAL LOCATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0039-003A</td>
<td>57-58</td>
<td>Current BASIC line number</td>
</tr>
<tr>
<td>003B-003C</td>
<td>59-60</td>
<td>Previous BASIC line number</td>
</tr>
<tr>
<td>003D-003E</td>
<td>61-62</td>
<td>Pointer to BASIC statement for CONT</td>
</tr>
<tr>
<td>003F-0040</td>
<td>63-64</td>
<td>Current DATA line number</td>
</tr>
<tr>
<td>0041-0042</td>
<td>65-66</td>
<td>Pointer to current DATA item</td>
</tr>
<tr>
<td>0043-0044</td>
<td>67-68</td>
<td>Vector for input routine</td>
</tr>
<tr>
<td>0045-0046</td>
<td>69-70</td>
<td>Current BASIC variable name pointer</td>
</tr>
<tr>
<td>0049-004A</td>
<td>73-74</td>
<td>Current BASIC variable data pointer</td>
</tr>
<tr>
<td>007A-007B</td>
<td>122-123</td>
<td>Index variable pointer for FOR/NEXT</td>
</tr>
<tr>
<td>008B-008F</td>
<td>139-143</td>
<td>Pointer to current byte of BASIC text</td>
</tr>
<tr>
<td>0090</td>
<td>144</td>
<td>Floating RND function Seed Value</td>
</tr>
<tr>
<td>0091</td>
<td>145</td>
<td>Status word ST</td>
</tr>
<tr>
<td>0092</td>
<td>146</td>
<td>Flag: STOP key/RVS key</td>
</tr>
<tr>
<td>0098</td>
<td>152</td>
<td>Timing constant for tape</td>
</tr>
<tr>
<td>0099</td>
<td>153</td>
<td>Number of open files or pointer to file table</td>
</tr>
<tr>
<td>009A</td>
<td>154</td>
<td>Default input device(0)</td>
</tr>
<tr>
<td>009B</td>
<td>155</td>
<td>Default output device(3)</td>
</tr>
<tr>
<td>00A0-00A2</td>
<td>160-162</td>
<td>Tape character parity</td>
</tr>
<tr>
<td>00B2-00B3</td>
<td>178-179</td>
<td>Clock</td>
</tr>
<tr>
<td>00B8</td>
<td>184</td>
<td>Pointer to start of tape buffer</td>
</tr>
<tr>
<td>00B9</td>
<td>185</td>
<td>Current logical file number</td>
</tr>
<tr>
<td>00BA</td>
<td>186</td>
<td>Current secondary address</td>
</tr>
<tr>
<td>00BB-00BC</td>
<td>187-188</td>
<td>Current device number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Current file name pointer</td>
</tr>
<tr>
<td>Address</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>00C5</td>
<td>Current key pressed</td>
<td>197</td>
</tr>
<tr>
<td>00C6</td>
<td>Number of characters in keyboard buffer</td>
<td>198</td>
</tr>
<tr>
<td>00C7</td>
<td>Inverse video on or off: 0=Off, 1=On</td>
<td>199</td>
</tr>
<tr>
<td>00C8</td>
<td>Pointer to end of logical line for INPUT</td>
<td>200</td>
</tr>
<tr>
<td>00C9-00CA</td>
<td>Cursor X-Y pos, start of INPUT</td>
<td>201-202</td>
</tr>
<tr>
<td>00CC</td>
<td>Cursor blink enable: 0 = flash cursor</td>
<td>204</td>
</tr>
<tr>
<td>00CD</td>
<td>Timer; Countdown to toggle (switch on or off) cursor</td>
<td>205</td>
</tr>
<tr>
<td>00CE</td>
<td>Character under cursor</td>
<td>206</td>
</tr>
<tr>
<td>00CF</td>
<td>Flag: Last cursor blink On/ff</td>
<td>207</td>
</tr>
<tr>
<td>00D0</td>
<td>Flag: INPUT or GET from keyboard</td>
<td>208</td>
</tr>
<tr>
<td>00D1-00D2</td>
<td>Current screen address</td>
<td>209-210</td>
</tr>
<tr>
<td>00D3</td>
<td>Cursor position on line</td>
<td>211</td>
</tr>
<tr>
<td>00D4</td>
<td>Flag: Editor in quote mode OFF: 0; ON: 1</td>
<td>212</td>
</tr>
<tr>
<td>00D5</td>
<td>Length of screen line (physical)</td>
<td>213</td>
</tr>
<tr>
<td>00D6</td>
<td>Screen row where cursor is</td>
<td>214</td>
</tr>
<tr>
<td>00D8</td>
<td>Flag: Insert mode &gt; 0 = INST</td>
<td>216</td>
</tr>
<tr>
<td>00F3-00F4</td>
<td>Pointer to current area of colour</td>
<td>243-244</td>
</tr>
<tr>
<td>0277-0280</td>
<td>Keyboard buffer queue</td>
<td>631-640</td>
</tr>
<tr>
<td>0281-0282</td>
<td>Pointer to start of memory</td>
<td>641-642</td>
</tr>
<tr>
<td>0283-0284</td>
<td>Pointer to top of memory</td>
<td>643-644</td>
</tr>
<tr>
<td>0286</td>
<td>Current character colour in range 0-15</td>
<td>646</td>
</tr>
</tbody>
</table>
Appendix D Error Codes

BAD DATA indicates that string data has been received from a file where the program expected numbers, i.e. the program was reading the received information into numeric variables.

BAD SUBSCRIPT occurs when an array is being used and the subscript (or for a multi-dimensioned array, one of the subscripts) is out of the range specified in a DIM statement. If no DIM statement has been entered by the programmer, the subscript is assumed to be between 0 and 10. For example `120 A(X)=25` will give a BAD SUBSCRIPT report if `X` is not within the DIMensions set in the DIM A () statement at the time that line 120 above is executed.

BREAK is not really an error message. It occurs whenever the program was stopped by the <STOP> key or a STOP in the program. Variables can be altered before continuing.

CAN'T CONTINUE will be displayed if a CONT is typed when the program is unable to resume where it left off. This will occur for a syntax error because the statement cannot be understood and will also occur if the program has been edited, as it has then been rearranged in memory, and all variables cleared. If the program has stopped because of a syntax or similar error, it is possible to continue running the program by GOTO a valid line. This can be useful if the error has occurred in a line such as a PRINT statement which does not affect the logic of the program. GOTO the line logically following the error line will allow the program to continue while still retaining the values of all variables set. Variables can be altered at this point if desired, before resuming.

DEVICE NOT PRESENT usually appears if the device referred to is not present – exactly as the message says! You may have forgotten to plug it in or switch it on or initialise it, or you may have it on the wrong channel number. (Disk is usually 8 and printer 4, but they can be changed – see the manuals.) Sometimes after a read or write error while using disks the error will appear on trying to SAVE a program. This can be cleared without losing information by a VERIFY.

DIVISION BY ZERO. As you will remember from school this is not allowed in the real world.

EXTRA IGNORED. Too many items in response to an INPUT. This usually occurs because a comma has been unintentionally included in response to an input. The last item(s) entered are rejected.

FILE NOT FOUND means what it says – i.e. no file of this name on disk.

FILE NOT OPEN means that you have tried to use a file not yet opened. Check your program logic.

FILE OPEN. Once a file has been opened it cannot be opened again, nor can the same number be used to open any other file. Close it before trying to continue.

FORMULA TOO COMPLEX. You have to be pretty smart to get this one! Break it down into simpler expressions until both you and the computer can understand it.

ILLEGAL DIRECT occurs when using statements such as INPUT in direct mode. Such statements are only valid within a program.

ILLEGAL QUANTITY. A number is out of range. This can happen for a variety of reasons and is explained within the main body of the book in the places it can occur. It can also occur anywhere where an integer is > 32767 or < -32768 or in extreme circumstances where ordinary numerics are greater than about 10 to the power of 38 or less than 10 to the power -39 using the E notation. It is difficult to imagine what you would be doing to get either of these last two errors.

LOAD (ERROR) means a corrupt tape, dirty heads or perhaps an incorrect disk initialisation causing the disk directory (BAM) to become corrupt. Always give your disks different identifiers and ID numbers and then this problem should not occur.

NEXT WITHOUT FOR is usually obvious but watch out for nested loops where you have inadvertently used the same variable in two FOR statements.

NOT INPUT FILE. You told the computer it was an output file so you can't now go reading from it.
NOT OUTPUT FILE. No writing to an input file!

OUT OF DATA means your DATA is finished and you are trying to go on READING it without having first done a RESTORE. Have you missed out a DATA item?

OUT OF MEMORY. This can occur for two reasons.
   (a) The RAM is full because you have written a large program or have put a large number of strings in memory during program execution. Check the memory with a FRE before running the program, and another FRE after the program has carried out a few manipulations. A clue to string problems is that the program will stop for a few moments to a few minutes while the poor thing rushes around memory tidying up strings to make room for more, a process known as garbage collection. The cure is to tidy up your program or data strings (keep some on file perhaps) or to expand your VIC, or to buy a compiler for your 64.

   (b) There are too many nested GOSUBs or FOR loops. This is indicated by the fact that a FRE reveals plenty of memory. It is rare for this problem to occur, except as a result of a program error where incorrect logic has caused FOR loops or GOSUBs to be called repeatedly from within themselves. See FOR and GOSUB.

OVERFLOW means that a calculation gives a result which is too large for the numeric variable to handle i.e. greater than 1.70141884E+38. Note that this error does not occur for integers which always give ILLEGAL QUANTITY as error.

REDIM'D ARRAY occurs when an array has been dimensioned twice or used (and thus dimensioned by the computer) before the DIM statement. Note that this means that DIM statements at the start of your program should not be included in any loop, but only executed once at the start of your program.

REDO FROM START means that letters instead of numbers have been typed in response to an INPUT. The problem can also occur because of the INPUT (q.v.) bug.

RETURN WITHOUT GOSUB means what is says. It usually arises because you have forgotten to include an END or STOP between the main program and the subroutines, or a RETURN is missing at the end of a previous subroutine. This allows the program to continue into the wrong area of code where it meets a RETURN without having been sent there.

STRING TOO LONG – only 255 characters maximum are allowed in a string.

SYNTAX indicates that the computer cannot understand the intention of the programmer in writing the BASIC line. Usually the problem is a missing comma or THEN or mis-spelt BASIC word. The machine actually does know a bit more than it tells you, as it may have got half way along a line before discovering a problem, but it only actually gives you the line number. There are two solutions to a SYNTAX ERROR that cannot be easily found. One is to get a programmers toolkit which has a facility to point to the position where the problem was found, and the other is to break down your statement line until the syntax problem has become apparent. Look particularly for peculiarities like BASIC keywords embedded in variable names if your names are longer than 2 characters.

TYPE MISMATCH appears when a numeric type is assigned to a string variable or vice versa. Note that this error does not appear for assignments of integers to numerics and vice versa. Truncation can occur in these circumstances (see Introduction).

UNDEF'D FUNCTION. User defined functions must be defined in a DEF FN statement before use.

UNDEF'D STATEMENT means that a GOTO or similar statement refers to a line number which does not exist in this program.

VERIFY occurs if the information read off the tape or disk is not the same as that present in the computer. Check that the tape heads are clean and that the drive/tape unit is away from possible sources of interference including the TV set. Then check that you are verifying the correct item and try the SAVE and VERIFY again.