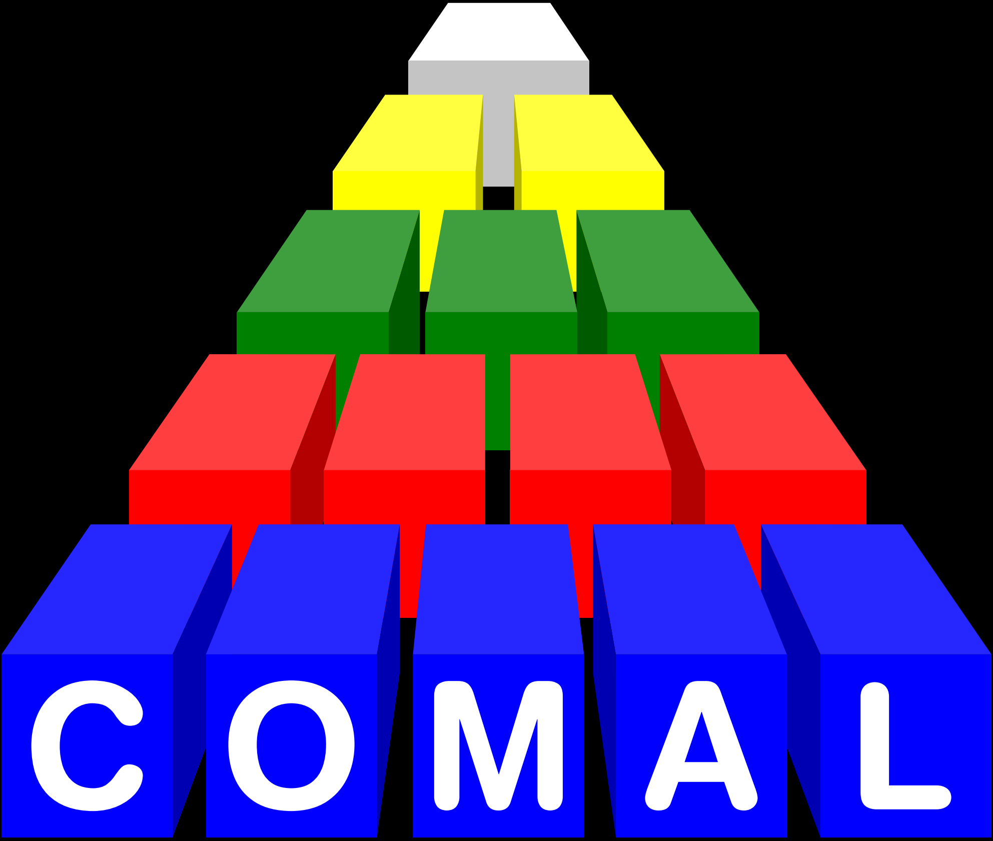
ACORNSOFT

ROY THORNTON and
PAUL CHRISTENSEN



on the BBC Microcomputer and Acorn Electron

COMAL

on the BBC Microcomputer and Acorn Electron

ROY THORNTON and
PAUL CHRISTENSEN

ACORNSOFT

Acknowledgements

The production of this manual, in the time scale required could only have been achieved with the support of our families, the encouragement of friends including Mark Taylor, Andy Tonks and Graham Noble, and with the extensive assistance of the implementers of Acornsoft COMAL.

Thanks are also due to the staff of Optimus Graphic Design, Cambridge, for their invaluable efforts.

The Acornsoft implementation of COMAL was written by David Christensen, James Warwick and David Evers who at the time were all studying at Portsmouth Grammar School. David Christensen and James Warwick are now at Cambridge University. The implementers and authors have been actively involved in the international development and standardisation of the COMAL language.

ISBN 0 907876 90 0

FIRST EDITION

Re-mastered by dv8 in 2020
First revision, October 2020
For the latest revision of this book go to:
stardot.org.uk/forums/viewforum.php?f=42

Note: Within this publication the term ‘BBC’ is used as an abbreviation for ‘British Broadcasting Corporation’.

First published by Acornsoft Limited, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, England. Telephone (0223) 316039

Contents

Part IV

Introduction

COMAL is modern, as computer languages go. It has its ancestry in Pascal and BASIC, combining the simplicity of the operating environment of BASIC with the power of Pascal. COMAL has been developed considerably in recent years and now has an elegance and structure to rival any language available for a microcomputer. COMAL is a general purpose language encouraging the writing of well structured programs. It should be possible for a program or procedure written by one author to be understood and used by another.

About this manual

Part I of this manual has been designed as a steady introduction for the absolute beginner. End of section questions and summaries are provided to aid consolidation. If the reader finds that he is unable to give the correct answer to a significant proportion of the questions then he is advised to reread the appropriate parts of the chapter.

The experienced BBC BASIC user should be able to skim through the summaries at the end of each section of Part I. Points that are specific to Acornsoft COMAL are printed on a grey background. Part II contains many topics that will be new to BASIC users, as well as a greater coverage of some of the ideas introduced in Part I.

Part III contains essential reference material for the program writer who wishes to check a point of detail. All commands, statements, functions and operators are defined and demonstrated.

Part IV contains some COMAL programs, a history of the development of the language and suggested answers to the end of section exercises in Part I.

Throughout this manual Acorn Electron users should take 'BBC Microcomputer' as referring to the 'Acorn Electron' and *BBC Microcomputer System User Guide* as referring to the *Acorn Electron User Guide* unless the Acorn Electron is mentioned specifically.

Note that this manual is primarily about COMAL itself, and therefore contains few details about setting up the BBC Microcomputer and using the operating system. Such information may be found in the *BBC Microcomputer System User Guide*.

Part I – for the absolute beginner

1 Learning a new language

There are three requirements for learning to use a new language.

– *Vocabulary:* new names for quite familiar things. The computer has a limited vocabulary of what are known as ‘keywords’.

– *Grammar:* ways of linking words together to give understandable statements and commands. The computer grammar is known as the ‘syntax’ of the language.

– *Ideas:* something to be expressed in the new language. The ideas come from us but they may be affected by our ability to use the vocabulary and grammar.

2 Keyboard introduction

In this section we shall learn to use some of the keyboard controls and the keyword `PRINT`.

2.1 Getting going

Connect up the computer and switch on (refer to the *BBC Microcomputer System User Guide* if you are not sure how to do this). You should see a screen message like one of those shown below. The other words on the screen may vary and so dots have been used in our diagram.

```
BBC Computer ...          or    Acorn Electron ...
.............                   .............
......                          ......
._                              ._
```

There is a symbol pointing to the right. This is called the 'prompt' and means that the computer is waiting for you to do something. Alongside the prompt is a short flashing line, which is called the 'cursor'. The shape of the prompt may be one of the following.

`>_` or `→_` or `]_`

The prompt and cursor indicate that the computer is waiting for an instruction. For simplicity, the prompt shaped > will be shown in the first few examples.

You will not do any harm to the microcomputer by typing at the keyboard. Don't worry if you make a typing error: it's easy to make a correction.

The keyboard is similar to that of a typewriter, but with some extra features. Find the key at the bottom right of the keyboard marked with the word DELETE. If you make a typing mistake, then press this key to remove the error.

Now find the letter G (near the middle of the keyboard) and lightly press it down a few times. Notice its appearance on the screen, with the cursor always moving on to the next place.

```
>GGGGG_
```

Find the DELETE key and press it lightly to delete one of the Gs. The cursor moves back as well.

```
>GGGG_
```

The long unmarked key nearest you is called the ‘Space Bar’. Press it once. An ordinary typewriter moves across when the Space Bar is pressed, but on a computer a blank appears.

```
>GGGG _
```

If you now hold down the key G for about two seconds you will find that the letter is repeated automatically whilst the key is pressed. Try it.

```
>GGGG GGGGGGGGGGGGGGGGGGGGGGGG_
```

On a simple typewriter when you come to the end of a line, you hear a bell and the carriage stops. However, when we come to the edge of the screen the letters typed will appear at the beginning of the next line.

Keep pressing the G key until the Gs appear on the next line.

```
>GGGG GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGG_
```

A similar effect happens with the DELETE key, which will move back into the previous row. Try holding it down for a while.

```
>GGGG GGGGGGGGGGGGGGGGGGGGG_
```

You can use the DELETE key whenever you want to remove one or more letters. It may be pressed once or kept pressed down to repeat the deletion.

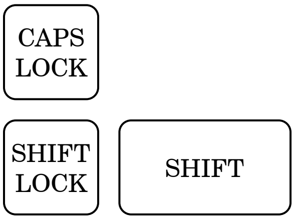
Hold it down and continue to delete all the Gs. If you continue to hold down the DELETE key you will find that you cannot delete the prompt.

```
>_
```

2.2 Caps and shift

Note for Acorn Electron users: the Electron has a slightly different keyboard from the BBC Microcomputer version described here. There is no SHIFT LOCK key and the only indicator is a yellow light beside the CAPS LK key. References in the text to the SHIFT LOCK key should be ignored or read as SHIFT as appropriate. References to the caps lock indicator still apply to the yellow light on the Electron. The facility for generating keywords using FUNC and the letter keys is available in Acornsoft COMAL as in BBC BASIC. The only keywords which are not valid in COMAL are `LOCAL` and `CHAIN`.

At the front left of the keyboard there are three keys that we need to use.



Just below them are three red indicators labelled

cassette motor	caps lock	shift lock
o	o	o

The caps lock indicator should already be lit.

Press the CAPS LOCK key a few times and notice that the matching caps lock light goes on and off.

Similarly try pressing the SHIFT LOCK key a few times.

Press the SHIFT LOCK key until both lights are off.

If you now press the key G a few times you should find that you get a line in lower case (or small) g.

```
>gggggggggggggggg_
```

Now press the key marked 6 with an & sign above it. You should find that you just get a figure 6.

```
>gggggggggggggggg6666_
```

To get capital letters (or the upper symbols), you can hold down the SHIFT key (rather than the SHIFT LOCK) with one finger and press a key with another finger. Try to get a capital H using the SHIFT and H key.

```
>gggggggggggggggg6666HHHHH_
```

This is just like using an ordinary typewriter. Try to get the * by using the SHIFT and colon key marked **:**

```
>gggggggggggggggg6666HHHHH******_
```

Use the DELETE key to remove all the symbols that you have typed, right the way back to the prompt.

```
>_
```

2.3 Into COMAL

Type `*COMAL` and then press the key on the right, which is marked RETURN. The word `COMAL` should now have been printed on the next line to confirm that the computer is ready in the language we are going to use.

```
COMAL    or    COMAL

→_             ]_
```

Notice the shape of the prompt symbol, which is a distinctive feature that identifies COMAL. Whenever you switch on there will be a message on the screen which identifies the language you are using. If the language is not COMAL then type `*COMAL` and press the RETURN key.

Upper or lower case

When using Acornsoft COMAL you may type in capitals or small letters.

If you want to type in capital letters then you may press the SHIFT LOCK key, switching on the matching light. All the letters of the alphabet will appear in upper case and the upper symbols on the number and punctuation keys will be used.

The CAPS LOCK key can be used to give you a mixture of upper case letters and the lower case (unshifted) character for the other keys. Press it to switch on the matching caps lock light.

Press the key G a few times.

```
>GGGGGG_
```

Then press the key marked 6 a few times.

```
>GGGGGG66666_
```

The G will be in capitals, but the 6, which is the lower symbol on the key will appear. CAPS stands for CAPITALS. The capital letters, only, are locked ON.

To get the upper symbols on all the other keys, one of the SHIFT keys will need to be pressed. Try getting the !, which is on the 1 key.

```
>GGGGGG66666!_
```

2.4 PRINT, addition and subtraction

`PRINT` is the keyword for displaying on the screen. A keyword is a word that forms part of the computer vocabulary. `PRINT` should be followed by a space, then the item which is to be printed (displayed).

Complicated calculations can be carried out on a computer, but let us start with something simple that we can check: such as adding 2 and 2.

We will need the plus sign, `+`. To get it we shall use the SHIFT key and press the `;` key, but first delete all the symbols back to the prompt.

```
→_
```

The normal pocket calculator has only one thing it can do with its answers, that is to display them, whereas a computer has many options. We have to tell it to display the answers by using the keyword `PRINT`. When you type the command that follows, do not leave out the word `PRINT` and make sure you type a space after the word `PRINT` and before the 2. The prompt sign is shown for you.

```
→PRINT 2 + 2_
```

So far, nothing should have happened because the computer doesn't know that you have finished giving it all of the calculation. We have to be able to say 'Now, over to you!'. We do that by pressing the RETURN key which is on the right of the keyboard. Press the RETURN key.

In the instructions that follow, the prompt symbol and the cursor will not be shown.

Now type the following command after the prompt symbol and don't forget the space after the word `PRINT`.

```
PRINT 2 + 2 + 3
```

then press the RETURN key.

Be careful about the difference between the figure for nought, 0 and the letter O. They are almost under each other. The nought is next to the 9 key and has a line through it to distinguish it from the letter O.

Type

`PRINT 10 + 10` and press the RETURN key.

2.5 Syntax errors

If you type a letter O where you intend a figure nought the computer will produce an error message. Type the same command with an incorrect entry using the letter O instead of a figure nought and press the RETURN key.

The result is:

```
Syntax error
PRINT 1o + 1o
      ^
```

A `Syntax error` occurs when the computer cannot make any sense out of your command.

If you do make an error in the way you give a command then you will find that the command is reprinted for you with an arrowhead pointing to the place where the mistake was discovered. You may then try to enter the correct command.

To make the computer do subtraction we must use the minus sign `-` which is next to the 0 in the top line. Beware of the longer line _ which is just above the RETURN key. It is an underlining character and does not represent subtraction.

Type

`PRINT 56 - 45` and press the RETURN key,

then

`PRINT 56 - 45 - 2` and press RETURN.

If you got a syntax error, did you use the wrong sign?

2.6 Enter

So that you don't need to be told to press the RETURN key after every instruction, we shall use the word 'enter' to mean: type the instruction and press the RETURN key. If you forget to press the RETURN key there will be an embarrassing pause while you look at the computer and wonder why nothing is happening.

2.7 More arithmetic

Multiplying and dividing

Look for the `:` and `*` key which is on the right.

Enter

```
PRINT 3 * 4
```

Did you remember the RETURN key?

Then

```
PRINT 4 * 5 * 6
```

The symbol * represents multiplication.

On the right hand side at the front of the keyboard you will find the symbol /. Don't get mixed up with \ which is near the top, on the right and also don't forget the space after `PRINT`.

Enter

```
PRINT 8 / 2
```

then

```
PRINT 15 / 3
```

The symbol / represents division.

We can mix mathematical symbols within a calculation. Enter

```
PRINT 4 * 5 / 2
```

Precedence in calculation

When there are several mathematical operations in a calculation the multiplications and divisions are carried out before the additions and subtractions unless brackets have been used.

Enter

```
PRINT 3 + 4 * 5
```

The result is 23 because the multiplication 4 times 5 is done before the addition of the 3.

If we want addition to be done before multiplication then we must use round brackets (the shifted 8 and 9 keys).

Enter

```
PRINT (3 + 4) * 5
```

Using the square brackets `[` and `]` or the curly brackets `{` and `}` will not have the same effect.

Decimals

The symbol `.` (which is next to `/` at the front of the keyboard) is used for both a decimal point and a full stop.

Enter

```
PRINT 1.23 * 4.56
```

then

```
PRINT 9.87 / 0.654
```

When dividing by a decimal value which is between 0 and 1 it makes the command clearer to include the 0 before the decimal point.

Powers

Powers of a number can be calculated using the symbol `^` which is next to the `-` in the top row. Don't be surprised, when you press this key, if you find a slightly different symbol appearing on the screen.

Enter

```
PRINT 2 ^ 3
```

You should get 8 which is 2 to the power 3 or 2*2*2.

Raising to a power is done before multiplication or division unless brackets are used.

Enter

```
PRINT 2 * 5 ^ 3
```

The computer has multiplied the cube of 5 (5*5*5) by 2.

Enter

```
PRINT (2 * 5) ^ 3
```

The computer has multiplied 2 by 5 and then taken the result, 10, to the power 3 (ie 10*10*10).

The computer will accept any proper calculation with `+`, `-`, `*` or `/` together with `^` and round brackets `( )` provided that the numbers used are not too big or too small. (The range is ample for any normal calculation.)

There should always be a closing bracket to match each opening bracket. If not, a syntax error will result.

2.8 Printing numbers

Did you notice that our answers were not printed at the beginning of a line, but seemed to always end at the tenth place across the screen? The screen is normally divided up into 'zones' of ten places.

Numbers are normally printed to fit as far right as possible in the current zone.

If you want the answer to be printed as far left as possible then the symbol `;` (semi-colon) must be used.

Enter

```
PRINT ;2 + 2
```

and then

```
PRINT ;3 * 4
```

The semi-colon `;` is used in a `PRINT` command to cause a number to be printed as far left as possible in the space available in the current zone.

2.9 Printing messages

Look for the quotation mark symbol `"` which is the upper symbol on the 2 key.

Don't forget to use closing quotation marks as well as opening ones or you will get a syntax error.

Enter

```
PRINT "HELLO"
```

and then

```
PRINT "HOW DO YOU DO ?"
```

The computer is not being polite. It isn't maintaining a conversation but doing exactly as it is told. The symbols in quotations will be printed exactly as they appear inside the quotations.

A set of symbols placed in quotation marks after a `PRINT` command will be printed as far left as possible.

Enter

```
PRINT "TWO AND TWO MAKE FIVE"
```

The computer will `PRINT` what is in the quotations without seeing if it makes sense.

It will produce nonsense if the person at the keyboard instructs it to.

2.10 End of section exercise

Try to make the computer display the answers to the following calculations. Make the computer do all the calculations (including all the steps of the working). Note down your commands and then check them in chapter 43.

1. Add 7 to 6.

2. Add up the whole numbers from 1 to 5.

3. Multiply 4 by 5 by 9 and divide the result by 6.

4. From 7 multiplied by 7 take 6 multiplied by 6.

5. Divide 24 by the result of adding 3 to 5.

6. Multiply 12345679 by 7 and then by 9.

7. Divide 1 by 0.11.

8. Multiply 3.142 by the square of 3.5 and divide the result by 3.

9. If one pound can be exchanged for 120 cents, how many cents would you get for five pounds 40 pence?

10. Using the same rate of exchange as in question 9 how many pounds is 450 cents worth?

2.11 End of section summary

1. It doesn't matter whether you type COMAL commands in upper or lower case letters.

2. After a keyword such as `PRINT` you should leave a space.

3. Messages inside quotation marks are printed exactly as typed and as far left as possible. Example:

```
PRINT "LIKE THIS"
```

produces

```
LIKE THIS
```

4. The symbol for raising to a power is `^`.

5. Unless round brackets are used to change the order of calculation, raising to a power will be done before any multiplication or division, which in turn will be done before any addition or subtraction.

6. Numbers are normally printed on the right of the current ten-place zone, unless a semi-colon is used before the number, in which case the number is printed starting at the current position of the cursor.

3 Variables and assignment

In this section we shall learn how to store numbers in the computer memory. We shall use the keywords `INT` *and* `FREE`*, and meet* `PI`.

In the last section we typed in capital letters (upper case). However, in COMAL we could just as easily use small letters. So, for a change, press the CAPS LOCK key until all the lights are switched off.

3.1 Variables

The computer can store numbers ready for use when required. Each number that we store in this way must have a name to identify it and, conversely, each name will refer to a number (once it has been introduced).

We may change the number value associated with each name as and when we wish, ie the value may be varied. The name associated with each number is known as a 'variable' and a value may be 'assigned' to that 'variable'.

3.2 Assignment

Look for the `:` symbol, which is on the same key as `*` and look for the = symbol, which is next to the nought.

Enter

```
a := 5
```

Nothing will appear to happen. However, the value 5 has been 'assigned' to the label 'a'.

The symbol for assignment is `:=`

`a := 5` was a direct command to the computer. It means: let the variable called 'a' become equal to 5.

We can show that the computer has remembered by asking for the value of 'a' to be printed out. Don't forget the space after the `print` as you enter:

```
print a
```

The figure 5 is printed in the tenth place across.

To get the number as far left as possible, enter

```
print ;a
```

Upper and lower case

No distinction is made between `a := 5` and `A := 5`.

We can instruct the computer to print the value of `A`.

Enter

```
print ;A
```

and we still get 5. However, as we have seen, if we place the `a` in quotation marks then it will be printed as just the letter `a`.

Enter

```
print "a"
```

3.3 Recognising and finding

The computer recognises keywords of the language and operations such as `+` and `*`. Any other symbols are examined to see if they have been introduced and if not, then an error message is given. Don't forget the space after `print` as you enter:

```
print b
```

The computer has searched its memory for the value of 'b' and cannot find it. Not too surprising really! We haven't told it about 'b' yet!

Enter

```
b := 6
```

and

```
print b
```

This time the computer can oblige because it has been introduced to 'b' and can find it in its memory. We must 'assign' a value to a 'variable' before it can be used.

3.4 Changing and calculating

We can change the values that are given to `a` and `b`.

Enter

```
a := 7
```

and

```
b := 8
```

then

```
print a
```

and

```
print b
```

The computer can do arithmetic with the variables. The calculation is done and the result printed out.

Enter

```
print a + b
```

and

```
print a * b
```

We can also introduce new variables based on existing ones. For example, the values of 'a' and 'b' could be added together and the total assigned to the variable 'c'.

Enter

```
c := a + b
```

and

```
print c
```

The assignment symbol is correctly `:=` but if we wish we may type only the = sign.

Enter

```
c = a * b
```

and

```
print c
```

The = sign may be used where we wish to assign a value.

The variable to which we are making the assignment must always be on the left hand side of the `:=` (or =).

If we try to reverse the number and variable then we shall receive an error message.

Enter

```
2 := c
```

We can, however, assign a new value to a variable based on its old value.

Enter

```
c := c + 2
```

ie take the value of ‘c’, add on 2, then assign this new value to the variable ‘c’.

Enter

```
print c
```

3.5 Mixing messages and numbers

We can print a mixture of messages and numbers, but to do it we have to be careful about our grammar (or syntax). Don’t miss out the quotation marks or the `;`.

Enter

```
print "The value is ";c
```

The semi-colon `;` causes ‘c’ to be printed as far left as possible, in its zone (ie next to the message).

3.6 More about variables

We can use any letter of the alphabet to be a variable. In fact, we can use whole words to be variables, such as ‘price’ and ‘total’, enabling us to make assignments in the form:

```
price := total + vat
```

Errors

The only words that we must not use are the keywords that the computer recognises as part of COMAL. They are reserved for the computer and we cannot use them as variables. Let’s see what happens if we try.

Enter

```
print := 7
```

The computer is trying to carry out the `print` instruction. It cannot cope with the `:` sign because it is not a number, variable or anything allowable.

```
Syntax error
PRINT := 7
      ^
```

The arrowhead points to the place where the computer has realised there is an error. This may not be the exact position of the syntax error.

Try leaving out the final quotation mark in a command to print a message.

Enter

```
print "Testing
```

The arrowhead is pointing at the first quotation mark even though it is the second one that is missing. Leaving out a bracket can have a similar result.

Enter

```
print (3 + 4 * (5 - 2)
```

The arrowhead is pointing at the first bracket. This is because the computer cannot know where we intended the matching closing bracket to go.

Acceptable variables

Apart from using keywords, any word or collection of letters and numbers can be used as a variable, provided that they start with a letter.

Any of the following would be valid variable names: `superman`, `k9`, `rocky1`, `rocky2`, `rocky3`, `r2d2`, `c3po`. However, because they begin with a number the following may not be used: `3d`, `2001`, `9lives`.

After every variable name it is a good idea to type a space to separate the variable from what follows, so as to make the command easily readable.

Although we cannot use a keyword as a variable we can use variable names that contain a keyword.

For example

```
printer := 2

sprint := 99
```

This gives us plenty of scope to use variable names that describe what the number represents.

eg `price`, `cost`, `amount`, `total`, `vat`, `discount`.

If we want a keyword to be recognised then we must type a space after the keyword.

Enter

```
number := 42
```

and then enter the incorrect command with no space after the word print.

```
printnumber
```

The error message produced will look like this

```
Not found
 EXEC printnumber
→_     ^
```

`EXEC` is a command that we haven't met yet. It is a command to carry out a procedure. (Procedures are introduced in a later section.)

If you do get this sort of message then go back, check the syntax of your command and try entering it again with a space if required.

Using a space between `print` and `number` enter

```
print number
```

and all should be well.

Longer variables

There is another symbol that we can use as part of a variable name. It is the underline symbol _ which is next to the upward pointing arrow. It is not the same as the minus sign, which is shorter.

The underline symbol _ can be used to create longer variable names consisting of several words.
eg `number_on_dice := 6` or `moves_made := 23`

If we use a space instead of _, then the computer considers that it has come to the end of a variable name when it reaches the space and doesn't know what to do with the extra letters.

Try entering

```
number on dice := 6
```

then enter

```
number_on_dice := 6
```

3.7 Little and large

Since we have a powerful computer at our fingertips we can use it to perform complicated calculations. Are there any snags with using large numbers?

Commas are not used when entering numbers. For example: do not place commas in 1000 or 1000000.

Enter

```
print 1000 * 1000
```

then

```
million := 1000000
```

and

```
print million * million
```

The result is surprising and is the computer's way of representing very large numbers. 1E12 means:

1 times 10^12 or 1 times 1000000000000

Very large numbers can be written in a compact way using this method, but we won't normally meet it .

5.39E15 means 5.39 times 10^15 or 5390000000000000.

What about small numbers?

Enter

```
print 1 / million
```

The result is given in the same format.

1E−6 means 1 divided by 10^6 or 0.000001

5.34E−8 means 5.34 divided by 10^8 or 0.0000000534.

3.8 Accuracy

Numbers are printed in Acornsoft COMAL accurate to the ninth figure. The number may be stored in the same way that we have just seen: as two parts – a number from 1 up to 10 together with a power of 10.

If the result of a calculation is neither very large nor very small then it is printed out in the usual way.

3.9 Integers and reals

If we know that we are only using whole numbers, such as for the days of the month, then we can save some of the computer's memory by storing the numbers in a different format in which it will always be taken to be an 'integer' (whole number).

A symbol is placed at the end of the variable name to indicate that it is an integer. In COMAL we use the # which is a shifted 3 key. The symbol # is usually read aloud as 'hash' and must be placed after the variable name *without* a space between.

An integer variable is indicated by a # sign at the end of its name.

The word 'real' is used to refer to numbers that may have fractional parts.

Integer and real numbers may be positive, negative or zero.

Enter

```
five# := 5
```

Enter

```
print five#
```

then

```
print five# / 2
```

The result is just the same as if any other variable had been used. The only difference comes if you try to assign a decimal value to an integer variable.

Enter

```
five# := 5.899
```

and

```
print five#
```

The part after the decimal point is removed and `five#` takes only the whole number part. This process is called ‘truncation’.

Enter

```
ten# := 10
```

then

```
print ten# / 3
```

which produces the value to 9 figures.

Enter

```
number# := ten# / 3
```

and

```
print number#
```

If a non-integral number is assigned to an integer variable then only the whole number part is taken.
Integer variables use less memory than real variables.

Resident integer variables

Now enter

```
print a#
```

A value is printed out for us even though we have not assigned `a#`. The integer variables `a#` through to `z#` are always resident in the computer. We may assign values to them if we wish.

3.10 Functions

We shall come across many functions. A function uses one or more values, called the ‘arguments’, to calculate another (related) value. We shall always use the word ‘return’ for the process of producing a value from a function.

3.11 INT

`INT` is an example of a function.

Enter

```
number = 5.678
```

and

```
print int(number)
```

`number` is the ‘argument’ and the value 5 was ‘returned’.

Enter

```
print int(-number)
```

then

```
print int(number*10)
```

The value inside the bracket, the argument, is calculated first and if it is not an integer then the integer immediately below it is returned. If the argument is already an integer then it is returned.

`INT(number)` is a function that returns the integer below or equal to the value of the argument.

3.12 System functions and FREE

A system function does not need an argument. It will return a value that depends on the state of the computer’s memory. For example, as we introduce more variables we use up more of the computer’s memory. We can find out how much memory is left by entering the command

```
print free
```

`FREE` is a system function. We can print the value of the variable `FREE`, but we cannot assign values to it. We can’t give our computer more or less memory using `FREE`. If we assign a new variable then we should see that `FREE` returns a different value.

Enter

```
d := 123456789
```

and

```
print free
```

`FREE` is a system function that returns the remaining amount of available memory.

3.13 PI

`PI` is a constant used in many mathematical calculations for example, in finding the area enclosed by a circle or the volume taken up by a sphere. It is usually represented by the Greek symbol π. To find out what value is given to `PI` enter

```
print pi
```

PI has a constant value and may be used whenever required. It isn't a variable and so cannot be assigned. The computer has a value for PI correct to ten significant figures. It will normally print the value correct to nine significant figures.

Enter

```
radius := 7

circumference := 2 * pi * radius

print "Circumference of circle is ";circumference
```

and

```
area := pi * radius * radius

print "Area of circle is ";area
```

Then

```
volume := 4/3 * pi * radius ^ 3

print "Volume of sphere is ";volume
```

PI is a constant with the value 3.141592653.

3.14 End of section exercise

Now try the following questions which will help you find out if you have grasped the ideas in this chapter. Make a note of your answers and check them in chapter 43.

1. Which of the following are not suitable variable names?
`u`, `u2`, `2u`, `account_number`, `tax paid`, `print`, `free`, `pint`

2. Which of the following represents 9000000000?
`1E9`, `1E10`, `9E9`, `9E10`

3. Which of the following represents 0.00876?
`8.76E-2`, `8.76E-3`, `8.76E-4`, `8.76E-5`

4. If `out := 3`, which of the following will produce a figure 3 on the screen?
`printout`, `print out`, `PRINTout`, `PRINT OUT`, `printOUT`

5. If `lower := 3` what is the value of `LOWER`?

6. If `number# := 9` what is the value of `number#/2`?

7. What is the value of `int(7/2)`?

8. What is the value of `int(pi)`?

3.15 End of section summary

1. It doesn't matter whether you type a variable name in upper or lower case letters. No distinction is made between `six := 6` and `SIX := 6`.

2. Variable names cannot begin with a number, but otherwise can be of any length and can contain numbers.

3. The underline symbol `_` can be used as part of a variable name, eg `price_of_fish` (but not as the first character).

4. Variable names can contain keywords anywhere inside them as part of the complete name.
Examples: `printout`, `sprint`, `pick`.

5. A space must be typed after a keyword so that it is clear to the computer that a variable is not being used.

6. The variable being assigned is on the left of `:=`

7. Commas must not be used when entering numbers.

8. `7.5E9` means 7.5 multiplied by 10^9 ie 7500000000

`7.5E-9` means 7.5 divided by 10^9 ie 0.0000000075

9. Integer variables are indicated by a `#` (hash) symbol.

10. `FREE` is a system function which returns the remaining amount of available computer memory.

11. `INT(x)` returns the integer immediately below or equal to `x`.

12. The integer variables `a#` to `z#` are always resident in the computer even if they have not been assigned.

13. `PI` is a system function which returns the value 3.141592653.

4 Strings and CLEAR

In this section we shall learn about strings of symbols, how to handle them, and how to use the keywords `LEN`, `IN`, `DIM` *and* `CLEAR`.

Let us again use lower case letters for our typing. Press the CAPS LOCK key, if necessary, to turn off the caps lock and shift lock lights.

4.1 Strings of symbols

COMAL can work with sets of letters and symbols just as easily as with numbers. When a set of symbols is used it is known as a 'string' and can include figures and punctuation.

To indicate that the symbols form a string, a `$` sign is placed at the end of the variable name (in much the same way that `#` was placed at the end of an integer variable). The `$` is the upper symbol on the 4 key. Examples are: `letter$`, `key1$`, `reply2$`, `day_of_week$`.

The symbols we assign to the string are placed in quotation marks. Use the underline symbol and both quotation marks as you enter

```
any_old$ := "ABCDEFG"
```

The quotation marks are essential to mark the beginning and end of the string. We may read the assignment as:

'any old string becomes quote ABCDEFG unquote'

Enter

```
print any_old$
```

We can alter the contents of the string, if we like.

Enter

```
any_old$ := "xyz123*"
```

and

```
print any_old$
```

Because we can vary the symbols that we place inside the string, it is known as a string variable. Its name must obey the same rules as for numeric variables.

It can contain a keyword as part of the name. For example, `printout$` is acceptable, but `print$` is not.

A string variable is indicated by a `$` symbol at the end of the variable name. A set of symbols may be assigned to a string variable using quotation marks around the symbols.

Upper and lower case

Inside the quotation marks it does matter whether you have used capitals or small letters. They are different symbols and are considered to be distinct elements of the string.

4.2 How long is a string?

That's very easy to answer if we make use of a new keyword, which is an abbreviation of the word length.

Enter

```
this$ := "pqrst"
```

and

```
print len(this$)
```

We should have got the value 5. It isn't too hard to think of `LEN( )` as standing for length.

`LEN(a$)` is a function that returns the number of characters in `a$`.

What happens if we place spaces inside the string?

Enter

```
try$ = "1 2 3 4 5 "
```

and

```
print len(try$)
```

A space counts as a character in a string, just as much as any letter or figure.

Null string

In the following line the two quotation marks are placed next to each other, with no space between.

Enter

```
blank$ = ""
```

and

```
print len(blank$)
```

The computer can cope with a string that has no symbols inside it. Such a string is called a ‘null’ string.

A ‘null’ string has no symbols inside it.

4.3 IN

Here’s a good bit of typing practice.

Enter

```
alpha$ = "abcdefghijklmnopqrstuvwxyz"
```

Making sure you leave a space after `print`, use both quotation marks and leave a space after `in`:

enter

```
print "c" in alpha$
```

and

```
print "z" in alpha$
```

Enter

```
test$ = "m"
```

and

```
print test$ in alpha$
```

and

```
print test$ in "lmn"
```

`IN` tells us where the first string starts to appear in the second string.

Enter

```
print "+" in alpha$
```

If the part in quotes is not in the string then the value 0 is returned.

Enter

```
print "cdef" in alpha$
```

and

```
print "cdefgh" in alpha$
```

4.4 Binary operators

`IN` is an example of a ‘binary operator’.

A binary operator is a function that needs two ‘arguments’. It uses the arguments to calculate a resulting value which it ‘returns’. One argument is placed before a binary operator and one after it.

The arithmetic symbols `+`, `-`, `*`, `/` and `^` are other examples of binary operators, eg `5 * 2` returns `10`.

4.5 DIMensioning strings

When we first assign a string, 40 memory places are reserved (one place for one symbol), unless we indicate that we need more or less places.

Long strings

If we want to use a string containing more symbols (up to 253 are allowed) then we must have a way of reserving more room. The process is called dimensioning.

`DIM` is the keyword for reserving the space we need for a string.

Enter

```
dim long$ of 200
```

The keyword `DIM` is followed by a space, the string name, a space, the keyword `OF`, a space and then the number of places that are to be reserved for the string.

Now assign `long$` using

```
long$ := "...............................
............"
```

Show that `long$` has a length greater than 40 by entering

```
print len(long$)
```

Short strings

If the computer is likely to run out of available memory and we don't need as many as 40 places for the string then we can reserve less.

Enter

```
dim short$ of 5
```

and

```
short$ := "exact"
```

We must think ahead when dimensioning a string and anticipate the greatest length we shall need.

Enter

```
short$ := "bigger"
```

We receive the error message that the string is too long for the space reserved.

Normally, we don't need to dimension a string, unless we know that it will be more than 40 characters long or we wish to reserve less than 40 places.

Having assigned a string, we cannot change the amount of memory that is reserved by just giving a revised dimension statement.

Enter

```
dim short$ of 10
```

The error message indicates that the variable has already had a space reserved for it. We are not allowed to redimension that space.

4.6 CLEARing memory

We can instruct the computer to forget about the strings which we have assigned (and also all of the numeric variables except the resident integer variables `a#` to `z#` by using the keyword `CLEAR`.

Enter

```
clear
```

and now we should find that the computer does not recognise the existence of `short$`.

Enter

```
print short$
```

and we receive the `Not found` message.

Enter

```
print a#
```

and a value is printed because the resident integer variables `a#` to `z#` are not cleared.

Enter

```
dim short$ of 10
```

Once `CLEAR` has been used a string may be dimensioned afresh to the length required (up to 253).

4.7 Printing formats

Now enter

```
a$ = "TOM"
b$ = "DICK"
```

and

```
c$ = "HARRY"
```

When we printed out numbers they appeared in zones of ten places, either on the left or right depending on whether we used a semi-colon before the number.

There is a way in which we can make the strings appear in these same zones, though they will be as far left as they can in each zone.

Using the comma

Look for the comma, which is at the front of the keyboard, next to the letter M.

Enter

```
print a$,b$,c$
```

Having printed `a$` the comma is the element of syntax that tells the computer to move over to the next zone and then do the next bit of printing there. The `b$` is then printed as far left as possible in that zone. The following comma indicates that the next item, `c$`, is to appear in the next zone.

The comma is the instruction to `PRINT` in the next available zone. If the print position is already at the beginning of a zone then a comma will not cause the print position to change to another zone.

Using the semi-colon

Enter

```
print a$;b$;c$
```

The semi-colon has the same effect for strings as for numbers. The item that follows is printed as far left as possible. In this case `DICK` follows `TOM` with no gap. Similarly, `HARRY` follows `DICK`.

The semi-colon is the instruction to `PRINT` in the next available print position.

We can mix the use of commas and semi-colons.

Enter

```
print a$,b$;c$
```

One important point to note is that after `c$` there is no comma, to say move across to the next zone; and there is no semi-colon to say print in the next space.

If there is no special symbol at the end of a `PRINT` command then the next print position is at the beginning of the next line.

Using the apostrophe

Look for the apostrophe symbol `'` which is the upper symbol on the 7 key.

Enter

```
print a$'b$'c$
```

In a `PRINT` command the `'` causes the print position to move to the beginning of the next line.

Enter

```
print a$,b$'c$;
```

The symbols `,` `;` `'` are called print 'separators' and one of them must be used between items that are to be printed.

Notice where the prompt and cursor are flashing. Having printed `c$` the semi-colon is the instruction to carry out the next print instruction in the next available print position (ie alongside the end of `c$`). The next thing to be printed is the prompt and then the cursor. If you want the prompt and the cursor on the next line, all you have to do is press the RETURN key.

Making space

Meanwhile, we haven't printed out `TOM DICK HARRY` in the way that we might have expected with a space between their names. A space can be placed inside a set of quotation marks and then be treated as a string. The semi-colons are used as print separators.

Enter

```
print a$;" ";b$;" ";c$
```

4.8 Adding strings

There is also a way in which we can join strings together using the addition symbol.

Enter

```
print a$ + a$ + a$
```

Each string is added on to the end of the previous string and then the result is printed. Adding strings in this way is called 'concatenation'.

The `+` sign can be used to join strings together.

4.9 Using spaces inside a string

The addition of strings has a similar effect to that of using the semi-colon. We can define a new string from the ones we have already introduced to the computer.

Enter

```
d$ = a$ + " " + b$ + " " + c$
```

and

```
print d$
```

to produce a very tidy effect.

We could have found another way to produce the result we wanted. In the assignments for `a$` and `b$` we could have entered spaces inside the strings.

Enter

```
a$ := "TOM "

b$ := "DICK "

c$ := "HARRY."
```

Now enter

```
print a$;b$;c$
```

and

```
print a$ + b$ + c$
```

In computing there are often many ways of achieving the same result. There is not necessarily a right or a wrong way.

4.10 A quotation mark inside a string

If we try to assign a quotation mark as part of a string then we shall cause a syntax error.

Try entering

```
quote$ := "this quote " doesn't work"
```

We can place a quotation mark inside a string by using a special notation. Inside the string we use a `""` where we want the string to contain a single `"`.

Enter

```
quote$ := "this quote "" does work"
```

and

```
print quote$
```

4.11 End of section exercise

Make a note of your answers and check them in chapter 43.

In the following questions assume that the assignments `a$ := "ACORN"`, `b$ := "Micro"`, `c$ := "computer"` have been made.

What commands are needed to print

1. `ACORN      Micro      computer`

2. `ACORN Microcomputer`

3.
   ```
   ACORN
   Micro
   computer
   ```

4. `Micro Micro Micro`

In the following questions `reply$ := "Y"`

What would be the result of the following commands?

5. `print reply$ IN "YyNn"`

6. `print reply$ IN "Nn"`

7. `print reply$ in "RGYBMCW"`

8. `print reply$ in "YYYYYYY"`

9. `print LEN(reply$)`

What dimension statements would be needed to reserve

10. 100 places for `ton$`?

11. 40 places for `test$`?

12. 10 places for `ten$`?

4.12 End of section summary

1. A string contains symbols which may include numbers, spaces and any other character which the computer can generate.

2. The symbol `$` is placed at the end of a variable name to indicate that it is a string.

3. `LEN(a$)` is a function returning the length of `a$`.

4. `IN` is a binary operator which calculates the position at which one string starts to occur inside another, eg `"AB" IN "abcABC"` is 4.

5. Forty places are reserved for a string when first assigned, unless it has previously been `DIM`ensioned.

6. To reserve a space of 100 characters for `k$` a `DIM k$ OF 100` statement is required.

7. An attempt to assign more characters to a string than have been reserved for it will produce an error.

8. `CLEAR` is the keyword to wipe the memory of all the string and numeric variables except the resident integer variables `a#` to `z#`.

9. The symbols `, ; '` are used to format the printing at the beginning of the next available zone, as far left as possible and on the next line respectively.

10. Strings may be combined together (concatenated) with the `+` symbol.

11. A quotation mark may be made part of a string by using a double quote `""` inside the normal quotes when assigning the string.

5 Programs

In this section we shall learn how to construct a simple program and extend our vocabulary of keywords.

So far we have given ‘direct’ commands, ie commands which are entered at the keyboard and carried out immediately. To be useful, the computer needs to be able to store a series of commands, and then to carry them out when we tell it to.

The series of commands are stored as ‘program’ statements. (That is the correct spelling for computer program.)

5.1 Preparing for a program

Before we start creating a program we should prepare the computer to make a fresh start.

Enter

```
new
```

`NEW` is the keyword to instruct the computer to prepare for a new program to be entered.

5.2 Program lines

A program consists of a set of statements, each of which begins with a whole number (called the line number).

If you forget to start with a line number then the computer will take your instruction as a ‘direct’ command and try to carry it out. That’s enough of the theory! Let’s see how it works.

Use the CAPS LOCK key to switch off all the lights.

Now enter

```
    1 print "I think"
```

Nothing should have happened unless you made an error in typing, in which case, check what you should have entered and try again until the entry is accepted.

The computer has remembered the statement and is waiting to see if there are any more.

5.3 RUN

Enter

```
run
```

To make the computer carry out the instructions which it has stored, we give it the direct command `RUN`.

It doesn't matter whether the statements are in lower case or upper case.

Now enter

```
    2 print "I am"
```

5.4 LIST

If we want to see all the lines of the program then we may use the direct command `LIST`.

Enter

```
list
```

Notice that the keyword `PRINT` is shown in capital letters.

Enter

```
run
```

Program lines must begin with a whole number, so if we want to add extra statements into the middle of a program then there must be room for them.

It was a bad idea to number the program lines 1 and 2, because there is now no space left to place another program line between them.

5.5 RENUMBER

Enter

```
renumber
```

and

```
list
```

The command `RENUMBER` causes the lines to be renumbered in sequence 10, 20, 30, 40, …

We can now add an extra program line between our existing lines.

Enter

```
   15 print "therefore"
```

and

```
renumber
```

We can list only one line of a program if we want.

Enter

```
list 20
```

Now list and run the program.

5.6 Order of execution

Did you notice that the computer carried out the statements in the sequence of the line numbers and not in the order in which we entered them?

The computer follows the sequence of line numbers unless it is instructed to do otherwise (by repeating some lines or carrying out only selected lines).

Enter

```
   40 print "intelligent"
```

list and run the program.

If we want to list just some of the lines, such as from 20 to 40 then we may indicate the range of lines.

Enter

```
list 20,40
```

5.7 Editing

Sometimes we want to remove one of the lines from a program.

To remove a line from a program, type its line number and press RETURN.

Type

```
20
```

and press RETURN.

Then list to check that line 20 has been removed.

Run the program, but don't renumber it.

If we want to replace one line by another then all we need do is to type the new line.

Enter

```
    40 print "here"
```

and list again to see the new line 40.

Copy cursor

If we want to make a small change to an existing line then we can use an editing facility. On the right there are four arrow keys and at the bottom right of the keyboard there is a key marked COPY.

Press the upward pointing arrow once.

A marker block is left behind where the cursor used to be.

Use the arrow keys to move the cursor up (and down, if necessary) until it is positioned in front of line 10 as shown below.

```
_   10 PRINT "I think"
```

Now use the right (and left, if necessary) pointing arrows to move the cursor under the 1 of the 10.

```
    10 PRINT "I think"
```

Press the COPY key once and notice the effect.

The 1 should have been copied where the marker block was and the marker block and cursor have both moved across one place.

Use the COPY key to copy up to and including the first quote, but do not press RETURN yet.

```
→10 PRINT "
```

Using the keyboard, type

```
I know"
```

not forgetting the closing quote, and press RETURN.

The complete line

```
    10 PRINT "I know"
```

should now have been entered and the cursor should be back in its normal place.

List and run the program to check it.

Now enter `new` and list the program.

You should find that nothing happens. The command `NEW` has made the computer prepare for a new program. It makes it appear that you have lost your previous work.

5.8 OLD

If you change your mind and want the previous program back again then you may enter `OLD`.

Enter

```
old
```

and

```
list
```

The direct command `OLD` may restore the previous program provided that a new program has not been started and that no new variables have been introduced. `OLD` may also be used when the BREAK key has been pressed. Provided that the program has not been corrupted it may be listed and run.

5.9 Abbreviations

Enter

```
    20 p."that"
```

and

```
list
```

In line 20 the `p.` has become `PRINT`.

The abbreviation `P` . may be used for `PRINT` if you wish.

The abbreviation for `LIST` is just the full stop.

Enter

```
.
```

and you should get a listing of the program. This abbreviation certainly makes for increased speed. There are short forms of most keywords. In the keyword summaries given in Part III the abbreviations are shown at the top right of each page.

5.10 Deleting several lines

If we want to remove a set of program lines then we use the keyword `DEL`.

Enter

```
del 10,30
```

and list (or enter `.`)

The lines from 10 to 30 inclusive have been deleted.

A continuous sequence of program lines can be removed from a program using the `DEL` command. For example: `DEL 80,120` would delete the lines from 80 to 120 inclusive. A single line may be deleted, eg `DEL 80`.

Enter

```
new
```

5.11 AUTO

Enter

```
auto
```

producing

```
→auto
   10 _
```

The `10` is a line number that has been displayed for us.

Enter

```
x := 3
```

and we find that another line number has appeared.

```
→auto
   10 x := 3
   20 _
```

AUTO is the direct command to produce line numbers in the sequence 10, 20, 30, …

Enter

```
y := 4
```

and the sequence continues. How do we stop it?

ESCAPE

At the top left of the keyboard you will find the ESCAPE you are looking for. Press the key.

Press the ESCAPE key to get out of AUTO numbering.

Taking care to leave a space after the keyword AUTO

enter

```
auto 100,5
```

producing

```
  100 _
```

The automatic numbering has restarted at 100 and will continue at intervals of 5. Complete the line by entering

```
print x*x + y*y
```

5.12 Comments

There is a way in which we can leave useful comments in a program by using the keyword symbol // consisting of the division sign used twice. A long program can be made much easier to understand by using comments.

Anything after // in a program line is ignored.

Complete line 105 by entering

```
// sum of squares
```

Remarks or notes may be stored at the end of almost all program lines, or on a line by themselves, using the keyword //.

Enter

```
  110 print x + y // sum of values
```

Press ESCAPE to break out of the numbering sequence and then list and run the program.

5.13 Modifying a program

Enter

```
new
```

and

```
auto 100
```

which will start numbering at 100 and go up in tens.

Enter the following program lines:

```
  100 first := 99
  110 print first
  120 second := 88
  130 print second
  140 print "Total"
  150 print first + second
  160 end
```

Use the ESCAPE key to end the numbering sequence.

`END` is a keyword that usually marks the physical end of a program. Its use is optional and has no effect other than that no more program lines will be executed once the `END` command has been encountered.

Run the program.

Now remove line 140 by entering

```
DEL 140
```

List the program.

Use the arrow keys and the COPY key to change line 100.

Copy

```
  100 first
```

and then type

```
$ := "ACORNSOFT "
```

and press RETURN, producing

```
  100 first$ := "ACORNSOFT "
```

Similarly using the COPY key amend:

line 110 to

```
  110 PRINT first$
```

line 120 to

```
  120 second$ := "COMAL"
```

line 130 to

```
  130 PRINT second$
```

Line 150 is a little harder to edit.

Copy to get

```
  150 PRINT first
```

and then type

```
$
```

producing

```
  150 PRINT first$
```

then copy

```
+ second
```

producing

```
  150 PRINT first$ + second
```

then type

```
$
```

and press RETURN, producing

```
  150 PRINT first$ + second$
```

List the program

and run it.

Now enter

```
renumber 200,20
```

and list the program.

The `renumber` command has caused the program listing to start at line 200 and go up in twenties.

Enter

```
renumber 200
```

and list to see that the line numbers go up in tens.

5.14 Variables in listings

User variables, ie those introduced by the computer user, will always list in lower case. A system function or a system variable will list in upper case.

User variables will always `LIST` in small letters.

5.15 End of section exercise

Make a note of your answers and check them against those suggested in chapter 43.

1. What command would give you automatic line numbering starting at 200 and going up in fives?

2. How do you get out of automatic line numbering?

3. What command would change program line numbers to the sequence starting at 100 and going up in tens?

4. What command prepares for a fresh program?

5. How might you restore the previous program?

6. Use program lines that assign `a$ := "SUPER"`, `b$ := "CALI"`, `c$ := "FRAGI"`, `d$ := "LISTIC"` and try to create a program to produce the following:

```
SUPER     CALI      FRAGI     LISTIC
SUPER     CALI      FRAGILISTIC
SUPER     CALIFRAGILISTIC
SUPERCALIFRAGILISTIC
SUPER
CALI
FRAGI
LISTIC
```

7. Use a program line to assign the value 111 to the variable `x`, and then print out the values of `x`, its square and its cube in zones across the screen.

5.16 End of section summary

1. A program is a stored set of statements that are carried out when the direct command `RUN` is given.

2. Each program line consists of a whole number followed by a statement or comment.

3. `AUTO` is a direct command to display line numbers in the sequence 10, 20, 30, 40 … waiting after each line number for a statement and the press of the RETURN key before showing the next line number. `AUTO 200` would start the numbering at 200 going up by tens. `AUTO 100,5` would produce the sequence 100, 105, 110, 115 etc.

4. ESCAPE can be used to break out of `AUTO`.

5. `LIST` is the direct command to display the program in the sequence of its line numbers. An individual line of the program, such as line 80, can be shown by using `LIST 80`. A continuous sequence of lines, such as from line 80 to 120, can be displayed by the command `LIST 80,120`.

6. When a COMAL program is `LIST`ed the keywords are shown in upper case and the user variables in lower case. The abbreviation for `LIST` is a full stop.

7. `P.` is the abbreviation for `PRINT`.

8. `//` indicates that what follows is not a statement to be carried out by the computer. Comments or remarks may follow the `//` symbol.

9. `RENUMBER` is the direct command to take the stored statements in the sequence of their existing line numbers and renumber the lines in the sequence 10, 20, 30, 40 etc. `RENUMBER 200,5` would start at 200 and go up in fives.

10. A line can be removed from the program by typing its line number and pressing the RETURN key or by entering `DEL` and the line number.

11. A continuous sequence of program lines, such as from 50 to 100 can be removed by using `DEL 50,100`.

12. The direct command `NEW` prepares for a new program to be entered.

13. A previous program may be recovered by the direct command `OLD`, provided that a new program has not been started and no new variables have been assigned.

14. `END` may be used to mark the physical end of a program. No more program lines are carried out once `END` is encountered. Its use is optional.

6 Simple loops

In this section we shall learn how to repeat a set of statements using the keywords `FOR` *and* `NEXT`.

6.1 Repeating statements

Program lines are normally carried out in their numbered sequence. However, if we want the same statements to be carried out several times then it is wasteful to keep entering the same set of instructions, line after line. We need a way of repeating a set of statements a precise number of times.

Enter

```
new
```

and

```
auto
```

then complete the following program lines, taking care to use spaces where shown.

```
10 print "Start here"
20 for number := 1 to 5 do
30 print "***************"
40 print "At line 40"
50 print "***************"
60 next number
70 print "All done"
80 end
```

and press ESCAPE to get out of the auto sequence.

Run the program

and then list it.

Indentation

When a program is `LIST`ed the statements between the lines containing `FOR` and `NEXT` are 'indented' by two places. These are the statements that are repeated.

6.2 Control of the loop

Notice that the variable `number` is in lower case, whereas `FOR` and `NEXT` are in upper case.

In line 20 the value 1 is assigned to the variable `number` and then lines 30, 40 and 50 are carried out.

When line 60 is reached, the `NEXT` number has the effect of adding 1 on to `number`, making it become 2.

Since 2 is in the range `1 TO 5` (as given in line 20), the lines 30, 40 and 50 are carried out again.

The `NEXT` number is 3 and the loop is repeated; and so on until `number` is set to 5, which is at the limit of the range. This value is allowed and the print statements are carried out for the fifth time.

The `NEXT number` statement then sets `number` to 6, which is outside the range `1 TO 5` and so the loop is finished and the computer continues into line 70.

Confirm this by entering the direct command

```
print number
```

After a program has run, the values of the variables are still stored and can be used if required.

Loop variables

The variable that we use to count our way round the loop is called the ‘loop variable’.

We can make use of a loop variable inside the loop.

Enter

```
   40 print "This is loop ";number
```

Run the program.

List it and follow through the logic.

6.3 Helpful additions

Enter

```
new
```

then

```
auto 100
```

and add lines

```
  100 for number = 10 to 20
  110 print ;number
  120 next number
```

ESCAPE and list.

Look carefully at line 100.

If the `:` or `DO` are left out of a `FOR...` statement then they will be inserted automatically.

They are both, technically, proper parts of a COMAL `FOR...` statement.

Run the program and then list it.

Use the arrow keys and the copy cursor to copy all of line 100 except the `DO`, but *before* you press the RETURN key, type a space and then type `step 2` then press RETURN.

List and run the program.

6.4 STEP

`STEP` is the keyword to use in a `FOR...` statement to indicate the amount by which the loop variable is to be changed.

Use the COPY and arrow keys to change line 100 to

```
  100 FOR number := 10 TO 14 STEP 0.5
```

List and run the program.

If there is no `STEP` at the end of a `FOR...` statement then the step size is assumed to be 1. (This is known as its ‘default’ value – the value which it takes if no other is specified.)

Negative STEP

We can produce a decreasing series of numbers if we use a minus or negative `STEP`.

Change line 100 to

```
  100 FOR number := 10 TO 1 STEP -2
```

List and run the program.

The value 1 is not printed. The series moves from 2, which is printed, to 0 which brings the loop to an end.

6.5 Impossible loops

An impossible `FOR... NEXT...` loop is ignored.

Change line 100 to

```
  100 FOR number := 10 TO 100 STEP -1
```

Then enter

```
   90 print "Start"
```

and

```
  130 print "Finish"
```

List to check that your program is:

```
   90 PRINT "Start"
  100 FOR number := 10 TO 100 STEP -1 DO
  110   PRINT ;number
  120 NEXT number
  130 PRINT "Finish"
```

Run the program.

No part of the `FOR... NEXT...` loop has been carried out. It is as if the lines from 90 to 130 didn't exist. If we did start at 10 and went in steps of −1 through 9,8,7,6,... we should not reach 100.

In COMAL, the computer will bypass any `FOR... NEXT...` loop which cannot be carried out properly and will carry on with the rest of the program.

6.6 Introduce variables first

Enter `new` and then the following program

```
   10 total := 0
   20 for number := 1 to 5
   30 print number
   40 total := total + number
   50 next number
   60 print "The total is ";total
   70 end
```

and run it.

In line 40 we need to know the value of `total` before we can add on the value of `number` and then reassign the new value back to `total`. That is why line 10 is needed: it assigns a value to `total` before we start the loop.

Any user variable appearing on the right of an assignment, must have been introduced previously.

6.7 The :+ notation

Line 40 is a bit of a mouthful! It says: take the `total` and add on `number`, then assign the result to the variable `total`. In effect we only want to add `number` on to `total`. There is a short way of saying this. Be careful not to leave a space between the `:` and `+` as you enter

```
40 total :+ number
```

We may read this as ‘`total` increases by `number`’.

Run the program to check that the result is still the same.

`:+` is an abbreviated assignment which adds an amount on to the previous value of the variable on the left of `:+`.

String use of :+ notation

The `:+` notation can also be used with strings.

Enter `new` and then the following program:

```
10 basic$ := "That"
20 print basic$
30 extra$ := " and this" // note space
40 for go = 1 to 3
50 basic$ :+ extra$
60 print basic$
70 next go
80 end
```

List and run the program.

Line 50 has the effect of `basic$ := basic$ + extra$` but is considerably shorter.

6.8 The :– notation

A command such as

```
money := money - cost
```

may also be shortened to

```
money :- cost
```

`:-` is an abbreviated assignment which takes an amount off the previous value of the variable to the left of `:-`.

6.9 Doubling up

Enter `new`, `auto` and then the following:

```
    10 dim star$ of 130 // space needed
    20 star$ := "*"
    30 print star$'
    40 for go := 1 to 7
    50 star$ :+ star$
    60 print star$'
    70 next go
    80 end
```

ESCAPE and list, but don't run the program yet.

In lines 30 and 60 the `'` will have the effect of moving on an extra line before carrying out the next `print` statement.

Look carefully at the program and try to deduce what it will do, then run the program.

Because line 50 adds `star$` to itself, its length doubles each time.

Each new assignment of `star$` changes the length of `star$`. Now you can see why we needed line 10 to allocate so many places for `star$`.

Now move on and try the end of section exercise.

6.10 End of section exercise

Make a note of your answers and check them against those given in chapter 43.

What programs will produce the following sets of numbers?

1.	21	2.	10	3.	1
	22		20		3
	23		30		5
	24		40		7
	25		50		9
	26		60		11
	27		70		13
	28		80		15
	29		90		17
	30		100		19

4.	2	5.	99	6.	2
	4		88		2.5
	6		77		3
	8		66		3.5
	10		55		4
	12		44		4.5
	14		33		5
	16		22		5.5
	18		11		6
	20		0		

What programs will produce the following?

7.
```
*
*!
*!!
*!!!
*!!!!
*!!!!!
*!!!!!!
*!!!!!!!
```

8.
```
ABAD
ABADABAD
ABADABADABADABAD
ABADABADABADABADABADABADABADABAD
```

6.11 End of section summary

1. A `FOR... NEXT...` loop allows a set of statements to be repeated a precise number of times.

2. When a program is `LIST`ed the statements inside a `FOR... NEXT...` loop are indented by two places.

3. `STEP` is used if the increment in a `FOR... NEXT...` loop is not 1. Non-integral and negative steps may be used.

4. An impossible `FOR... NEXT...` loop is ignored and the program bypasses it.

5. Any variable appearing on the right of an assignment must have been introduced previously.

6. `:+` is an abbreviation for adding on to an existing variable.
Examples: `a$ :+ b$` is short for `a$ := a$ + b$`
`x :+ y` is short for `x := x + y`

7. `:-` is an abbreviation for subtracting from an existing number variable.
Example: `x :- y` is short for `x := x - y`
`:-` may not be used with string variables.

8. After a program has been `RUN` the variables are still stored and may be printed if required.

7 More about PRINT

In this section we shall learn how to print in various places on the screen.

7.1 Clearing the screen

It would be pleasing to the eye to start with an empty screen, on which to display our results.

Enter the direct command `CLS`.

`CLS` is the command to CLear the Screen.

`CLS` can be used at any stage of a program where we want to remove printing from the area in which we display our typing (or text).

Enter `new` and then the following short program:

```
10 cls
20 for go := 1 to 16
30 print "*"
40 next go
```

Run and list the program.

So far we have used only the simpler versions of the `PRINT` statement. There are several interesting ways in which we can control where our printing takes place on the screen.

7.2 Tabulating across the screen

Use the COPY key to help change line 30 to

```
30 PRINT tab(15);"*"
```

Look closely to see that there is no space between the `B` of `TAB` and the open bracket `(`. There should be a semi-colon after the close bracket.

List the program and notice that a space has been inserted between the `TAB(` and the 15. This space will always appear in a listing.

Run the program.

Change line 30 to

```
    30 PRINT TAB( 30);"*"
```

Run and list the program.

Our screen has room, at the moment, for 40 characters across each row. The positions are numbered 0 to 39.

`PRINT TAB( 15);` tells the computer to move to the position numbered 15 in the row and start printing there. Since the row starts numbering at 0, this means moving past 15 places (numbered 0 to 14) before starting to print. Similarly `TAB( 30);` means move past 30 places (numbered from 0 to 29) and start printing at the place which is numbered 30.

`PRINT TAB( a);` is the statement to start printing at the position numbered 'a' in the row. If the print position is already greater than 'a' printing occurs in the next available position.

Variables in tabulation

We can get some patterns by using the loop variable inside the `TAB(  );` command.

Change line 30 to

```
    30 PRINT TAB( go);"*"
```

Run and list. Examine the listing carefully and think why the asterisks appeared in a diagonal line.

Change line 20 to

```
    20 FOR go := 16 TO 1 STEP -1
```

Run and list then check the program to see how a negative `STEP` changed the direction of the asterisks.

7.3 Structural errors

Delete line 40 by typing 40 and pressing RETURN, then try to run the program.

Error messages such as `Unclosed at END` are extremely helpful to us when we write a program. The computer has done a test (before the program is run) to see if there are any obvious errors. One of the tests is to see that a `FOR...` statement has got a matching `NEXT` statement. The computer searched through to the `END` of the program and didn't find a `NEXT go`. Hence the error message.

A program that is structurally unsound will not run and instead a list of the errors will be displayed.

This same feature may also be obtained by using the direct command `DEBUG`, which causes a test of the structure of a program and a report of any errors.

Enter

```
debug
```

and the same error message should be obtained. To repair the program we must put back the `NEXT` statement. We can now discover another feature of Acornsoft COMAL.

Completing the NEXT statement

Enter

```
   40 next
```

Did you notice that we didn't put the variable `go` at the end of the statement?

Now list the program.

The loop variable `go` has been inserted in line 40. If the computer can tell that the structure is unsound, then it can also work out which variable should follow the `NEXT` statement and add it for us.

The loop variable need not be stated after a `NEXT` statement. The computer will deduce the variable and will insert it automatically.

Enter

```
debug
```

and there should be no errors to report.

`DEBUG` reports structural errors in a program. If there are no errors then there is no report.

7.4 Tabulating down the screen

Give the direct command `CLS`

and enter

```
print tab(10,15);"Here"
```

Notice that a prompt and the cursor appear in the line immediately after the one in which `Here` was printed.

Now enter

```
print tab(10,8);"There"
```

The cursor now appears in the line following `There`.

The first number in the `TAB( , );` command is the place across the screen and the second number is the line down the screen at which the printing is to start.

Our screen, at the moment, has 25 lines numbered from 0 to 24, starting with row 0 at the top.

When told to `PRINT TAB( 10,8);` the computer moves to position numbered 10 in the row which is numbered 8, to start printing there. This is actually the eleventh element in the ninth row from the top.

`PRINT TAB( a,d);` is the command to start printing at place numbered `a` across and `d` down the screen.

7.5 Codes

The computer displays each symbol on the screen by keeping in its permanent memory a record of the shapes. The computer uses numbers to refer to each symbol. Since `A` and `a` have different shapes the computer refers to them by different numbers. We can get the computer to tell us what number it uses.

Enter

```
cls
```

and

```
print ord("A")
```

The `ORD` stands for ‘Ordinal’ which is a mathematical word for the number value. 65 is the code number for the letter A. `ORD` must be given a string value to work on. That is why quotation marks are needed.

Enter

```
print ord("B")
```

`ORD` returns a number, which will be printed to the right of the zone, unless we use a semi-colon. If `ORD` is applied to a string then the code given is that of the first element of the string.

Enter

```
x$ := "ZEBRA"
```

and

```
print ord(x$) // no quotation marks.
```

The set of capital letters is represented by the series of numbers starting at 65 and going up to 90. We can find the codes for the small (lower case) letters by entering

```
print ord("a")
```

and

```
print ord("z")
```

ASCII code

These numbers are part of an internationally known code for the characters used on computers. This ensures that computers can communicate with each other in the same number language. The code is called ASCII, short for American Standard Code for Information Interchange.

`ORD` is the function that returns the ASCII code of a string element.

7.6 Characters

We can reverse the process just as easily. We can get the computer to tell us the character it associates with a particular number.

Enter

```
print chr$(90)
```

The `chr$` stands for 'the character whose string is' and may be read as 'character string'.

`CHR$` is a function that returns the symbol associated with a given ASCII code.

Even a digit like 5 has an ASCII code. Enter

```
print ord("5")
```

and

```
print chr$(49)
```

and

```
print chr$(48)
```

Notice that `CHR$` returns a string. The number symbols are strings rather than pure numbers. They are printed at the left of the zone.

Enter `new` and then the following program.

```
    10 cls
    20 for code := 65 to 90
    30 print chr$(code), // note comma
    40 next code
```

Run and then list the program.

Lower case codes

Change line 20 to

```
    20 FOR code := 97 TO 122
```

Run and then list the program.

Number codes

Change line 20 to

```
    20 FOR code := 48 TO 57
```

and change 30 to

```
    30 PRINT ;code,CHR$(code)
```

Run and list the program.

When you are sure that you know how the program has worked, enter `new` and `cls`.

7.7 End of section exercise

Make a note of your answers to the following questions and check them in chapter 43.

1. What command would print `here` in the third row with four spaces before it?

2. What command would print `now` in the fifteenth row from the top with eight spaces before it?

3. What command would print out the ASCII code for G?

4. What command would print out the character whose ASCII code is 70?

Create programs to:

5. Display on the top nine rows, the word `here` with four spaces in front of it.

6. Display on the top nine rows, the integers from 1 to 9 with four spaces in front of the numbers.

7. Display on the top nine rows, the integers from 1 to 9 with the first number having one space in front of it and so on until the ninth number has nine spaces in front of it.

8. Display at the beginning of rows 11 to 19 the characters whose ASCII codes are 81 to 89.

7.8 End of section summary

1. `CLS` is the command to clear the screen of all text.

2. `PRINT TAB(across);` is the command to print in the same line at the place given by the value of `across`. If the print position is already greater than `across`, the printing occurs in the next available position.

3. A program that is structurally unsound will not run but will display the errors detected in a pre-run test.

4. The loop variable need not be stated after a `NEXT` command. The computer will deduce the variable and will insert it automatically, when the program is listed or run.

5. `PRINT TAB(across,down);` is the command to print at place `across` in the line given by the value of `down`.

6. `CHR$` is a function which returns the character whose ASCII code is given.

7. `ORD` is a function which returns the ASCII code of the first element of its string argument.

8 Modes and text colours

In this chapter we shall use some of the modes available and print in colour.

8.1 Choosing the mode

The BBC Microcomputer can display printing in several different ways. We can choose which `MODE` to use.

`MODE` is a system variable used to select the type of screen display. It may be assigned a value from 0 to 7 (although 7 will give the same result as 6 on the Acorn Electron). We can find its value as well as assign it. In a listing `MODE` will be in capitals.

Let's find out which mode we are using at present.

Enter

```
print mode
```

You should get 7 if you are using a BBC Microcomputer or 6 if you are using an Acorn Electron. The range of modes available is from 0 to 7 (0 to 6 on the Acorn Electron) though the BBC Microcomputer Model A uses only modes 4 to 7 unless it has had extra memory added.

Features of each mode

- The number of places in each row can be 20, 40 or 80.
- The number of rows can be 25 or 32.
- The number of colours you can place on a contrast background can be 1, 3 or 7.
- In some modes you can only have text displayed, whilst in others graphs or pictures can be drawn as well.

Mode information table

MODE	0	1	2	3	4	5	6	7
Places per line	80	40	20	80	40	20	40	40
Number of lines	32	32	32	25	32	32	25	25
Text colours	2	4	8	2	2	4	2	8
Graphics	Yes	Yes	Yes	No	Yes	Yes	No	Ltd

Changing mode

Enter

```
mode := 5
```

The screen is cleared when the mode is changed. The prompt may change shape and printing starts at the top left.

In mode 5 we have four colours available of which we are using two already. The background is black and the prompt and cursor are shown in white. Notice that they are much larger than before. The other colours that we can use easily in this mode are red and yellow.

If you are using a black and white television or a monochrome display then the 'colours' will appear as shades of the display colour.

Enter the following program, but don't worry if the typing is continued on into the next screen line. The line and its command will still be accepted when you press RETURN.

```
  100 mode := 5
  110 colour 1
  120 print "RED"
  130 colour 2
  140 print "YELLOW"
  150 colour 3
  160 print "WHITE"
  170 end
```

Run and list the program.

It is not easy to read a program in mode 5 so enter:

```
mode := 6
```

and list.

The program is not lost when we change mode. Line 100 changes mode which also clears the screen.

8.2 Function keys

At the top of the BBC Microcomputer keyboard there are some brightly coloured keys marked f0 up to f9 (on the Acorn Electron the symbols f1 to f0 appear on the front of the number keys 1 to 0). We may use these keys to store commands.

Enter the direct command

```
*key 0 MODE := 6
```

When we enter this command the statement `MODE := 6` is stored and is displayed whenever key f0 is pressed.

On the BBC Microcomputer press the coloured key f0. On the Acorn Electron, find the FUNC key on the left hand side and whilst holding it down also press the key 0.

You should find that `MODE := 6` has appeared on the screen with the cursor alongside. The statement has been recalled from the memory and displayed.

Press the RETURN key to enter this command.

We can also store a symbol which represents the pressing of the RETURN key. Then we only need to press one key to carry out the change of mode.

On a BBC Microcomputer you will find that next to the left-pointing cursor key there is a key with a reverse diagonal and two short lines for the shifted symbol. (On an Acorn Electron the two short lines are a shifted right-pointing cursor key.)

When the shifted key is pressed you may find that on the screen the two lines are placed beside each other rather than in line.

In this text take the symbol `|` to mean the shifted key with the two lines. Make sure there is no space after `|` and before the `m` as you enter the direct command:

```
*key 0 MODE := 6 |m
```

The symbol `|` is read as ‘control’ and when followed by an `m` or `M` has the same meaning as pressing the RETURN key. There are other ‘control codes’ available, and a full list is given in section 40.2.

Press the key f0 (on the Acorn Electron use FUNC key and 0).

The screen should have been cleared as the `MODE := 6` command was briefly displayed and then carried out.

Now enter

```
*key 0 MODE := 6 |m LIST |m
```

and press key f0.

The screen should have cleared and the program listed. We have now stored two commands on key f0. Several commands can be stored using the function keys and brought into action when required. `*key` statements can be part of a program.

Enter

```
   90 *key 0 MODE := 6 |m LIST |m
```

8.3 COLOUR numbers

`COLOUR` is the keyword used to select the colour in which printing is to be carried out (except in mode 7).

In a two-colour mode (0, 3, 4 or 6) the colours are numbered 0 and 1.

0 is black and 1 is white.

In a four-colour mode (1 or 5) the colours are numbered 0 to 3.

0 is black, 1 is red, 2 is yellow and 3 is white.

(The colours associated with each number can be changed. See the entry for `VDU` in Part III for details of using the `VDU 19` command.)

The program is easier to read if we assign these colour numbers to variables. Enter the extra program lines shown below:

```
   10 red := 1
   20 yellow := 2
   30 white := 3
```

Instead of selecting `COLOUR 1` we can now change line 110 to

```
  110 COLOUR red
```

Run the program to see that this works.

Now press function key f0 to see a clearer listing.

Change 130 to

```
  130 COLOUR yellow
```

and 150 to

```
  150 COLOUR white
```

Now add

```
   40 black := 0
  170 colour black
  180 print "BLACK"
  190 colour white
  200 end
```

producing the program

```
   10 red := 1
   20 yellow := 2
   30 white := 3
   40 black := 0
   90 *key 0 MODE := 6 |m LIST |m
  100 MODE := 5
  110 COLOUR red
  120 PRINT "RED"
  130 COLOUR yellow
  140 PRINT "YELLOW"
  150 COLOUR white
  160 PRINT "WHITE"
  170 COLOUR black
  180 PRINT "BLACK"
  190 COLOUR white
  200 END
```

Run the program.

The printing in black doesn't show up because the background colour is also black!

Press the f0 key to change mode and list.

Background colours

Add the line

```
  105 colour 129
```

and run the program again.

Now we can see the black lettering but the red print on a red background is a bit difficult! The command `colour 129` has set the background of our printing to red instead of the more normal black.

It would be neater if the whole screen could have a red background instead of just the part on which we have printed. This is easily achieved if we clear the screen, because the colour used to wipe the screen is always the background colour.

Press the f0 key then enter

```
  107 cls
```

and run again.

Selecting a number for `COLOUR` which is 128 greater than its normal value will cause the background colour to change. This only works for printing that is to follow. A `CLS` command will fill the screen with that colour.

Change line 105 to

```
  105 colour 128 + black
```

and run again.

We could select yellow or white as background colours by commands

```
colour 128 + yellow
```

or

```
colour 128 + white
```

However, be warned. It isn't easy to read yellow printing on white or white printing on yellow.

Reverting to normal

If you want to get back to the normal colouring of white on black then a change of mode will always reset the colours and clear the screen.

Enter

```
mode := 6
```

Using eight colours

Try the command

```
mode := 2
```

If you get back the message `Bad Mode` then your computer hasn't enough memory for the next program. (You can try it if you use `MODE := 5` in line 10.)

Enter

```
new
```

and then change mode to 6 to enter this program.

```
    10 mode := 2
    20 *key 0 mode := 6 |m list |m
    30 for choice := 0 to 7
    40 colour choice
    50 print "Colour ";choice
    60 next choice
```

Run the program.

In mode 2 the colour numbers and names are:

0	Black	(Bk)
1	Red	(R)
2	Green	(G)
3	Yellow	(Y)
4	Blue	(B)
5	Magenta	(M)
6	Cyan	(C)
7	White	(W)

Notice that the numbers for yellow and white are not the same as in a four-colour mode.

Press the f0 function key to change mode and list.

If you entered line 20 in lower case then the keywords are unchanged. This exception occurs with * commands.

Change line 50 to the following which has 20 stars.

```
    50 print "********************"
```

Run the program.

Why do we get a line spare between each row of stars?

Having printed 20 stars the cursor moves to the next line. But, having carried out its print instruction, there is no comma or semi-colon, therefore the cursor moves on to the next line again.

Press the f0 function key then place a semi-colon at the end of line 50 obtaining

```
    50 PRINT "********************";
```

and run the program again.

8.4 No comment

After a program line which contains a `*` command you may not add a comment using `//`. Try adding a comment to line 20 and then running the program.

Remove the comment from the end of line 20.

8.5 TAB and COLOUR

Press f0 and change line 50 to

```
    50 print tab(8,choice);"*"
```

then run the program and press f0.

Change line 50 to

```
    50 print tab(choice,11);"*"
```

then run the program and press f0.

Change line 50 to

```
    50 print tab(choice,choice);"*"
```

then run the program and press f0.

8.6 End of section exercise

Suggested answers may be found in chapter 43.

Assuming that you are using mode 5:

1. What command would be needed to set the print colour to red?

2. Create a program using a `FOR... NEXT...` loop to print out the letters of the alphabet in capitals across the screen.

3. What command would reset the print colour to white?

4. Combine 1, 2 and 3 to create a program to print the alphabet in red and then reset the print colour to white.

5. What extra statements would be needed to carry out question 4 but with the background in yellow?

Assuming that you are using mode 2:

6. What command would be needed to set the print colour to green?

7. What commands would set the background colour to cyan across the whole screen?

8. Create a program using a `FOR... NEXT...` loop to print the numbers 1 to 9 in green on a cyan background. The numbers should be printed in a column at the end of the first zone. The print colour should then be reset to white.

9. What command should be entered so that when function key f2 was pressed `RENUMBER` appeared on the screen?

10. What command should be entered so that when function key f2 is pressed a program in the computer is renumbered?

11. What command should be entered so that when function key f2 is pressed a program in the computer is renumbered and then lists in mode 6?

8.7 End of section summary

1. `MODE` is the system variable that controls the screen mode. It may be assigned a value from 0 to 7. The value may be printed like any other variable. Example: `PRINT MODE`.

2. `COLOUR number` is the command to select the colour in which printing will take place. The value of `number` can be chosen from the list of colours available in the particular mode being used.

3. `COLOUR number + 128` will set the background colour of future printing to the colour given by the value of number in the list of colours for the mode in use.

4. The `CLS` command will fill the printing area using the background colour currently selected.

5. `*key` commands may be used to store text or control codes so that when a function key is pressed this text is printed as if it were input from the keyboard.

6. The key marked ¦ may be used with the letter `m` in a `*key` instruction to give the effect of pressing the RETURN key.

7. A comment may not be placed at the end of a program line containing a `*` command.

9 Getting information from the keyboard

In this section we shall learn to take in values from the keyboard using the keywords `INPUT`*,* `GET` *and* `GET$`*.*

Enter the following program, which is intended to calculate the area of a rectangle.

```
100 mode := 6
110 length := 10
120 breadth := 5
130 print tab(5,8);"Area of a Rectangle"
140 print tab(5,12);"If the length is ";
150 print ;length;" cm"
160 print tab(5,14);"and the breadth is ";
170 print ;breadth;" cm"
180 area := length * breadth
190 print tab(5,16);"then the AREA is ";
200 print ;area;" sq cm"
```

and run it.

The values for the length and breadth are assigned in the program. New values could be assigned by changing lines 110 and 120. However, we don't want to rewrite the program every time we want to work out the answer to a new question.

9.1 Using INPUT

List the program and then change line 110 to

```
110 input length
```

line 120 to

```
120 input breadth
```

and add

```
102 print tab(0,5);
```

These statements will allow us to put in the numbers we choose for `length` and `breadth`, then work out the area.

Start the program running and then read on.

A question mark is printed to let us know that an entry is expected. Type `12` and press RETURN.

The number 12 has been assigned to `length`.

Another question mark is printed to tell us that another entry is expected. Type `6` and press RETURN.

The 6 is assigned to `breadth` and the program is completed.

Run the program again entering different values for `length` and `breadth`.

`INPUT` is the keyword for entering the value of a variable using the keyboard. A question mark is printed to indicate that a reply is expected.

The program works very well, but we should need to be very inspired if we were to guess what was wanted by the question mark when we first saw it.

9.2 Prompting an INPUT

Enter

```
  105 print tab(5,3);"Length ";
```

and

```
  115 print tab(5,5);"Breadth ";
```

List and run the program again.

This does seem to make more sense. The words `Length` and `Breadth` act as useful prompt messages.

Now delete lines 105 and 115.

The `INPUT` and the prompt message can be combined together into one command.

The colon is an essential part of the ‘syntax’ of the commands given below.

Change lines 110 and 120 to

```
  110 INPUT "Length ":length
  120 INPUT "Breadth ":breadth
```

Run the program again using new values for `length` and `breadth`.

Did you notice that this time there was no question mark? If you want a prompt symbol such as ? or > then type it inside the prompt message.

If there is a prompt message in the middle of an `INPUT` statement then no question mark is printed. A colon must be placed between the message and the variable.

Enter `new` and then the following program.

```
 10 total := 0
 20 mode := 6
 30 print tab(8,1);"Average of 5 numbers"
 40 for row := 1 to 5
 50 print tab(10, row + 4);
 60 input "Enter a number > ":number
 70 total :+ number
 80 next row
 90 print tab(8,16);"The average is ";
100 print ;total / 5
```

Run the program a few times with, at first, very easy numbers that you can check. It is important that you test that a computer program is behaving as you were anticipating. Sometimes an error of typing or logic may give rise to the unexpected. The computer will always do what you have told it, as far as it is able.

Line 50 ensures that the message appears in the middle of the screen.

9.3 Using the wrong type of input

Let's see what happens if we enter a letter instead of a number. Run the program again and this time press a letter key when length is asked for.

The `INPUT` command is expecting a number and rejects our offer of a letter. An error message is displayed and the entry format is spoiled, but we are given another chance. Enter a number and complete the program.

9.4 String INPUT

We can `INPUT` strings if we declare in the statement that we expect a string.

Enter `new` and then this program.

```
10 mode := 6
20 print tab(0,5);
30 input "Please enter your name > ":name$
40 print tab(5,15);"Thank you ";name$
```

This program will run satisfactorily provided that `name$` uses no more than 40 symbols. Try it.

If we want to use more than 40 symbols then we must use a `DIM`ension statement such as:

```
    5 dim name$ of 60
```

9.5 GETting one code

Sometimes we need only one key to be pressed to give us the information we need. The keywords `GET` and `GET$` are available for this case.

`x := GET` is the instruction to wait until a key is pressed. The ASCII code for that key is then assigned to `x` (provided that we do not press the ESCAPE key).

Enter `new` and then

```
   10 mode := 6
   20 print tab(5,5);"Press a key, please"
   30 x := get
   40 print tab(5,7);"The ASCII code is ";x
```

Run this program a few times. Notice that the RETURN key is not needed and that the key symbol is not shown.

To decide what key has been pressed we can use `CHR$( )`.

Enter

```
   50 print tab(5,9);"The key was ";chr$(x)
```

Run the amended program to test it.

Now list the program.

9.6 GETting one symbol

`x$ := GET$` is a similar looking instruction, to wait for a key to be pressed. The key pressed is then assigned to `x$` (unless the ESCAPE key is pressed).

We can convert our last program to use `GET$`.

Change line 30 to

```
   30 x$ := get$
```

Enter

```
   40 print tab(5,7);"The key was ";x$
```

and

```
   50 print tab(5,9);"Its code is ";ord(x$)
```

List then run the program a few times.

9.7 End of section exercise

Write programs to do the following:

1. Input two numbers and print out their sum and product.

2. Take in the length, breadth and height of a rectangular block and print out its volume.

3. Adjust program 2 to also print out the total surface area.

4. Take in values for `p`, `r` and `t` then use the formula

```
i = p * r * t / 100
```

to calculate and print out the value of `i`. This is the way of calculating simple interest on a principal `p`, at a rate `r`% per annum for a time `t` years.

5. What amendments would you make to the average program in section 9.2 so that you are asked ‘How many numbers are there?’ and the computer takes in the number of values, the values and then averages them.

Check your solutions with those shown in chapter 43.

9.8 End of section summary

1. `INPUT x` will cause the computer to print a question mark and wait for a number to be entered. The number is assigned to the variable `x`, which need not have been assigned previously.

2. `INPUT x$` will cause the computer to print a question mark and wait for a string to be entered. The string is assigned to the variable `x$`, which need not have been assigned previously (unless the length of the string will be greater than 40 symbols, in which case a previous `DIM`ension statement is then required).

3. A prompt message may be placed in quotations after the `INPUT` and before the variable. Example: `INPUT "Postcode > ":p$`.

A colon must be placed between the prompt and the variable.

No question mark will be shown when the command is carried out.

4. `TAB( );` and `INPUT` cannot be combined in the same statement. If you want to `INPUT` at a particular place on the screen then a `PRINT TAB( a,d);` and an `INPUT` statement will achieve the required result.

5. When input is being taken from the keyboard, then `x := GET` assigns to the variable `x` the ASCII code for the symbol which is next pressed on the keyboard (provided that the ESCAPE key is not pressed).

6. When input is being taken from the keyboard, then `x$ := GET$` assigns to the variable `x$` the symbol of the key which is next pressed (provided that the ESCAPE key is not pressed).

10 Creating random numbers

In this chapter we shall learn how to create random numbers and to make use of `TIME` *and* `NULL`.

Computer games usually have an element of chance in them. Events occur irregularly and unpredictably yet under control. How can this happen with a computer that only carries out the commands it is given?

The answer is that one of the computer's keywords can be used to produce 'random numbers'. These should be in an unpredictable sequence and should be equally likely to occur anywhere in the range of numbers.

10.1 RND

Enter and run the following program, taking care not to type a space between the `rnd` and the bracket.

```
10 for go := 1 to 10
20 print rnd(999)
30 next go
```

The values given are not the same every time. `RND( )` is a keyword for the process of finding a random number.

`RND( 999)` will produce a whole number from 1 to 999 inclusive.

`RND( n)` is a function that returns a whole number from 1 to `n` (provided that `n` is greater than or equal to 2).

10.2 Random decimal values

Change 20 to

```
20 print rnd(1)
```

and run the program again.

`RND( 1)` will produce a random value between 0 and 1.

Decimal values are only produced by `RND( 1)`.

To produce a decimal random number between 0 and 100 we need only multiply `RND( 1)` by 100.

Change 20 to

```
   20 print 100 * rnd(1)
```

10.3 Example program

Enter `new` and then the following program

```
   10 mode := 6
   20 for shake := 1 to 24
   30 number := rnd(6)
   40 print number
   50 next shake
```

and run it.

Displaying randomness

We can make it easier to see the number of ones, twos, threes etc if we use a `TAB( );` command.

Change line 40 to

```
   40 print tab(5 * number);number
```

and run again several times.

There does seem to be a good spread of all values and they occur apparently unpredictably.

Combining random values

If we wanted to simulate the throwing of two dice then we could do that quite easily. Enter `new` and then

```
   10 mode := 6
   20 print "Die 1","Die 2","Total"
   30 for throw := 1 to 20
   40 die_1 := rnd(6)
   50 die_2 := rnd(6)
   60 score := die_1 + die_2
   70 print ;die_1,;die_2,;score
   80 next throw
```

Run the program.

Uneven spread

That looks reasonable, but delete line 20 and change 70 to

```
    70 print tab(2 * score);score
```

Run again to see something interesting.

The values 2 and 12 don't occur very often, whereas 5, 6 and 7 are much more frequent, because there are more ways that the dice can fall to give a total of 7 than there are to give 2.

10.4 Random numbers from here to there

Delete lines 40 and 50 and change line 60 to

```
    60 score := rnd(2,12)
```

`RND( 2,12)` produces a random number from 2 to 12.

Run this program a few times.

This time the values are more evenly spread.

`RND( a#,b#)` gives a random whole number in the range `a#` to `b#` inclusive (# indicates an integer). `a#` should be lower than `b#`. `a#` (or both `a#` and `b#`) may be negative.

Change line 60 to

```
    60 score := rnd(6,12)
```

and run again.

10.5 TIME

Enter the direct command

```
print time
```

and then enter the same command again.

`TIME` is a system variable that measures the passage of time. It increases every hundredth of a second.

That is why the second value of `TIME` was slightly greater than the first.

Now enter the direct commands

```
time := 0
```

and

```
print time
```

Setting the internal clock

`TIME` can be assigned a value. When the computer is switched on `TIME` is assigned the value 0.

This ‘clock’ inside the computer can be used to ‘time’ events.

Enter `new` and then

```
   10 mode := 6
   20 time := 0
   30 for measure := 1 to 24
   40 print time
   50 next measure
```

Run the program and notice how quickly it has been carried out. If you run the program again you may not get exactly the same values. There may be small differences depending on the exact moment at which the program starts and what the computer was doing at that time.

10.6 Doing nothing

We can make the computer wait by sending it into a loop in which nothing happens.

`NULL` is the keyword for the computer to do nothing.

Enter `new` and

```
   10 print "Start"
   20 time := 0
   30 for pause := 1 to rnd(1000,3000)
   40 null
   50 next pause
   60 delay := time
   70 print "That took ";delay/100;" secs"
```

then run the program.

The main reason for using `NULL` is clarity. If at a particular stage of a program you want the computer to do nothing, then you may use the keyword `NULL`.

We shall discover a neater way of pausing in a later section.

10.7 End of section exercise

What commands will print out random whole numbers in the following ranges?

1. From 1 to 40 inclusive.

2. From 0 to 39 inclusive.

3. From 11 to 20 inclusive.

4. From 0 to 1279 inclusive.

5. From 640 to 1279 inclusive.

6. From 512 to 1023 inclusive.

7. Amend the two dice program of section 10.3 so that it works out the average score for 20 throws.

Suggested answers are in chapter 43.

10.8 End of section summary

1. If `n` is a whole number greater than 1, `RND( n)` returns a random whole number from 1 to `n` inclusive.

2. `RND( a#,b#)` returns a random whole number from `a#` to `b#` inclusive.

3. `RND( 1)` returns a random decimal value between 0 and 1.

4. `TIME` is a system variable measured in hundredths of a second. The value of `TIME` is continually being updated by the computer. `TIME` can be assigned a value. When the computer is switched on `TIME` is assigned the value 0.

5. `NULL` is the statement to do nothing.

11 Graphics

In this section we shall learn how to use some of the graphics facilities available in Acornsoft COMAL.

11.1 Graphics modes

Only modes 0, 1, 2, 4 and 5 can have graphics displays.

11.2 The coordinates

In every graphics mode the screen is considered to be like a graph with a scale numbered across the screen and a scale numbered up the screen.

In the same way as with graphs, each point marked has a pair of 'coordinates'. When we switch on the computer the 'origin' of coordinates is at the bottom left of the display. The first coordinate is the position across the screen and the second coordinate is the position up the screen.

When we switch on, the screen coordinates number from

0 to 1279 across

and 0 to 1023 up.

The bottom left hand corner of the screen is then 0,0 and the top right hand corner is 1279,1023.

11.3 DRAW

Change mode to 5 by entering

```
mode := 5
```

and enter the direct command (with a space after `draw`):

```
draw 1000,800
```

Starting from the bottom left of the screen a 'line' has been drawn to the point which has coordinates 1000 across and 800 up. The graphics marker starts at the bottom left of the screen after a change of mode.

Enter

```
draw 1200,400
```

The line has been drawn from the last point of the previous line to the point 1200 across and 400 up.

`DRAW x,y` is the command to draw a line to the point with coordinates `x` units across the screen and `y` units up the screen. Each line is drawn from the last position visited.

Change to mode 6.

Enter `new` and then

```
 10 left := 0
 20 right := 1279
 30 bottom := 0
 40 top := 1023
 50 *key 0 mode := 6 |m list |m
 60 mode := 5
 70 draw right,bottom
 80 draw right,top
 90 draw left,top
100 draw left,bottom
```

Run the program.

Press the f0 function key to change mode to 6 and list.

Change 30 to

```
 30 bottom := 200
```

and run again.

11.4 MOVE

We don't get the neat rectangle, because the first line starts in the wrong place. We need to start our drawing from the point with coordinates 0,200. We need to `MOVE` our graphics marker to that point before starting to `DRAW`.

`MOVE x,y` is the command to transfer the graphics marker (cursor) to the point `x,y` without drawing a line.

Enter

```
 65 move left,bottom
```

Run the program again then press the f0 key.

11.5 Graphics colours

We can easily use the colours that are available in each mode. The graphics colours are changed quite independently of the colours in which we print text.

GCOL

The command `GCOL` is used to control the choice of graphics colours. Two numbers then follow the command. For the moment we shall use nought, then a comma, then the colour number we choose.

In mode 5, red is colour 1, so enter

```
  67 GCOL 0,1
```

and run the program again.

Similarly you can change to yellow with

```
  67 GCOL 0,2
```

`GCOL 0,n` will set the graphics colour to the number `n` where `n` is any number in the list of colours available for the mode being used.

The more experienced BBC Microcomputer user can refer to the section on `GCOL` in Part III for the meaning of `GCOL` statements other than `GCOL 0,n`.

Random example

Change 30 to

```
  30 bottom := 0
```

and add

```
 110 move 640,512 // centre screen
 120 for line := 1 to 20
 130 x := rnd(1279)
 140 y := rnd(1023)
 150 number := rnd(3)
 160 gcol 0,number
 170 draw x,y
 180 next line
```

Run the program.

11.6 Clearing graphics

Now enter the direct command

```
clg
```

`CLG` is the ‘clear graphics area’ command.

11.7 Background graphics colours

If the number used after `GCOL 0`, is ‘128 + a colour number’ then it is the graphics background colour that is changed. This only shows up if we use a `CLG` command.

Enter

```
   62 gcol 0,128 + 3
```

and

```
   64 clg
```

change 150 to

```
  150 number := rnd(0,2)
```

then run the program.

The black lines are now apparent.

The command `CLG` clears the graphics display area and leaves it in the current graphics background colour.

Text background colour is set by a `COLOUR` command. `CLS` clears the text area using text background colour.

Run the program again and then enter `CLS` to see the screen cleared in current text background colour black.

11.8 Graphics in eight colours

In mode 2 the other colours are readily available. Press f0 and then make the following changes:

```
   60 MODE := 2
   62 GCOL 0,128 + 7 // background white
  150   number := RND( 0,6)
```

producing

```
   10 left := 0
   20 right := 1279
```

```
  30 bottom := 0
  40 top := 1023
  50 *key 0 mode := 6 |m list |m
  60 MODE := 2
  62 GCOL 0,128 + 7 // background white
  65 MOVE left,bottom
  67 GCOL 0,2
  70 DRAW right,bottom
  80 DRAW right,top
  90 DRAW left,top
 100 DRAW left,bottom
 110 MOVE 640,512 // centre screen
 120 FOR line = 1 TO 20 DO
 130   x := RND( 1279)
 140   y := RND( 1023)
 150   number := RND( 0,6)
 160   GCOL 0,number
 170   DRAW x,y
 180 NEXT line
```

Now run the program.

11.9 Plotting points

The BBC Microcomputer has very powerful facilities for plotting points, lines and triangles. (Details are given in Part III.) If we only intend to mark a point then we need only one of the `PLOT` statements. Press function key f0 and make the following changes:

```
 120 for dot := 1 to 500
 170 plot 69,x,y
 180 next dot
```

Then run it.

`PLOT 69,x,y` marks the point with coordinates `x,y` in the current graphics foreground colour.

11.10 POINT(

It is possible to reverse the process and find out what colour is being used at any position on the screen. Taking care not to use a space before the bracket, enter

```
print point(640,512)
```

The number given is the colour number at 640,512.

Enter

```
print point(2000,2000)
```

The position is off the screen. The value -1 indicates that there is no colour displayed.

`POINT( x,y)` returns the colour number at the point `x,y` on the screen. If the point is off the screen then -1 is returned.

11.11 Filling triangles

Another powerful `PLOT` statement enables us to fill in triangles with colour. Press the f0 key and make the following changes:

```
  120 for triangle := 1 to 20
  170 plot 85,x,y
  180 next triangle
```

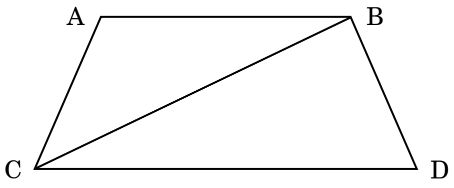
Then run the program.

`PLOT 85,x,y` draws a line to the point `x,y` in the current graphics colour and fills in a triangle made by the point `x,y` and the last two points visited.

Other shapes can be filled in using triangles. If the sequence of points is chosen carefully then only four points need be visited to fill a quadrilateral.

Press f0, enter `new` and then the following program.

```
   10 mode := 5
   20 gcol 0,1 // red
   30 move 400,700 // visit point A
   40 move 800,700 // visit point B
   50 plot 85,200,300 // fill triangle ABC
   60 plot 85,1000,300 // fill triangle BCD
```



You will need to watch closely to see the individual triangles being drawn.

11.12 Windows

It is possible to define rectangular areas of the screen which may have text and rectangular areas that may contain graphics. These areas (or ‘windows’) may overlap. The text and graphics windows are both set to the complete screen when there is a change of mode.

VDU commands

There is a set of visual display unit (VDU) commands, which affect the output from the computer to the screen or a printer. They are listed in Part III. For the moment we will learn how to use the ‘window’ commands only.

Text windows

The command to define a text window is `VDU 28`, it is followed by the `TAB` position of the bottom left corner of the window then the `TAB` position of the top right of the window. The `TAB` positions are the across and down values separated by commas.

Example: `VDU 28,2,24,17,16`

will define a text window with the bottom left corner at position 2 across and 24 down. The top right of the window will be at position 17 across and 16 down.

If you define a window that will not fit on the screen then the text area is taken to be the whole screen.

Press f0, enter `new` and the following program.

```
 10 mode := 5
 20 print tab( 2,5);"Colour choices"
 30 print tab( 2,7);"1. Red"
 40 print tab( 2,9);"2. Yellow"
 50 print tab( 2,11);"3. White"
 60 print tab( 2,14);
 70 input "Choice 1-3 > ":choice
 80 vdu 28,2,24,17,16
 90 colour 128 + choice
100 cls
110 colour 0
120 print "Text window"
130 end
```

Run the program.

Did you notice that `CLS` only cleared the window area?

The print positions are defined from the top left of the text window.

Keep pressing the RETURN key until you drive the `Text Area` message out of the text window.

Enter the direct command

```
print tab( 3,2);"Here"
```

Printing will only take place in the text window. If the `TAB( )` position cannot occur in the window then the instruction is ignored and the printing occurs without regard to the `TAB( )` part of the command.

Enter

```
print tab( 20,20);"Can't"
```

The position does not occur in the window so the `TAB( )` part of the command has no effect and printing occurs on the next line.

`VDU 28,` is the command to set up a text window. It must be followed by four values separated by commas. The first two values give the position of the bottom left corner of the window. The last two values give the position of the top right corner of the window.

When a text window is defined, a `CLS` command clears only the area defined by the window.

Graphics windows

The command to define a graphics window is `VDU 24,`. It is followed by four numbers each of which has a semi-colon after it. The first two numbers are the coordinates of the bottom left of the graphics area and the last two numbers are the coordinates of the top right of the graphics area.

Example:

```
VDU 24,400;200;900;800;
```

This command defines a graphics window with the bottom left hand corner at the point 400 across and 200 up. The top right corner is at 900 across and 800 up.

A comma follows the `VDU 24`, but all the other numbers must be followed by semi-colons. The reason for this is explained under `VDU` in Part III.

If you define a graphics window which will not fit on to the screen, then the graphics window will be set to the whole screen.

Press f0, enter `new` and the following program.

```
 10 red := 1
 20 yellow := 2
 30 white := 3
 40 mode := 5
 50 vdu 24,400;200;900;800;
 60 gcol 0,128 + red
 70 clg // set background to red
 80 gcol 0,yellow
 90 for y := 0 to 1000 step 40
100 move 0,y // left side of screen
110 draw 1000,500
120 next y
130 end
```

Run the program.

Did you notice that `CLG` only cleared the graphics area?

The lines have start and finish points which are outside the graphics window. The lines only show up where they cross the window.

`VDU 24,` is the command to set up a graphics window. It is followed by four values each of which should be followed by a semi-colon. The first two values are the coordinates of the bottom left of the graphics area. The last two values are the coordinates of the top right of the graphics area.

When a graphics window is defined, a `CLG` command clears only the area defined by the window.

11.13 End of section exercise

Make a note of your answers and check them against the suggested solutions in chapter 43. There may be some differences in the way you have written the programs or in the names for the variables that you have used.

1. Does mode 6 have graphics facilities?

2. If you are using a graphics mode, what commands would draw a line from the top left corner to the bottom right corner of the screen?

Use mode 5 for the following questions.

3. Write a program to draw a set of horizontal lines across the screen at intervals of 100 units.

4. Write a program to draw a set of vertical lines down the screen at intervals of 100 units.

5. Write a program, making use of the `PLOT 69,` command to plot a set of points at intervals of 80 units across the middle of the screen.

6. Write a program, making use of the `PLOT 85,` command, to draw a red square of side 200 in the middle of the screen on a yellow background.

7. What command would define a text window which takes up the top half of the screen?

8. What command would define a graphics window taking up the bottom half of the screen?

11.14 End of section summary

1. Modes 0, 1, 2, 4 and 5 can have graphics displays.

2. When we switch on the screen coordinates are from 0 to 1279 across and from 0 to 1023 up.

3. `DRAW x,y` is the command to draw a line to the point with coordinates `x,y` from the last point visited.

4. `MOVE x,y` is the command to move to the point with coordinates `x,y`.

5. `GCOL 0,n` is the command to set the graphics colour to the colour given by number `n` for the mode in use.

6. `GCOL 0, 128 + n` will set the graphics background colour to the colour given by the number `n`.

7. `CLG` is the command to clear the graphics area with the current graphics background colour.

8. `PLOT 69,x,y` is the command to plot the point with coordinates `x,y` using the current graphics colour.

9. `PLOT 85,x,y` is the command to draw a line to the point `x,y` in the current graphics colour and to fill a triangle formed by the point `x,y` and the last two points visited.

10. `POINT( x,y)` is a function which returns the number of the colour at the point `x,y`. If the point is off the screen then `-1` is returned. There must not be a space between `POINT` and the open bracket.

11. `VDU 28,` is the command used to set up a text window. It is followed by the `TAB( )` positions of the bottom left and top right of the window with commas placed between the values.

12. The `CLS` command clears the area defined by a text window using the current text background colour.

13. `VDU 24,` is the command to set up a graphics window. It is followed by the coordinates of the bottom left and top right of the window. Each coordinate should be followed by a semi-colon.

12 STOP and CONT

In this section we shall use the statement `STOP` *and discover when a program may be continued.*

12.1 STOPping a program

A program may be brought to a temporary halt with the `STOP` statement or by using the ESCAPE key.

`STOP` is the statement that causes a program to stop and report the line on which it stopped.

Enter `new` and then the following:

```
10 for j := 1 to 2
20 print j
30 stop
40 next j
50 print "The End"
```

When the program is run the value 1 is given then the message `STOP at line 30` is displayed.

12.2 CONT

Enter

```
cont
```

The value 2 is printed and the program stops again.

Enter

```
cont
```

and the program finishes.

`CONT` is the command to attempt to continue a program which has been stopped.

CONT after ESCAPE

In certain circumstances a program can be continued after use of the ESCAPE key.

Change 30 to

```
   30 INPUT "Enter a word > ":word$
```

Don't enter a word when you run the program. Instead press ESCAPE.

If ESCAPE is pressed whilst a program line is being run, then an `Escape at line ...` message is displayed.

Look at the last value printed and notice what happens when you enter `CONT` and `INPUT` a word.

The program should have continued from the line indicated by the ESCAPE message. If the line was not fully executed when ESCAPE was pressed then that line will be repeated.

Complete the program or press ESCAPE again.

Changing variables

Run the program and ESCAPE again. Now enter the direct commands `x := 45` and `j := 0.5` then enter

```
cont
```

New and existing variables may be assigned during a break in the program.

Changing the program

Run the program and ESCAPE again. Then enter the program line

```
   45 print "This is"
```

and enter

```
cont
```

A program will not continue if it is changed following a `STOP` or `ESCAPE`.

Errors in direct mode

Run the program and ESCAPE again. Now make an error by entering

```
prunt
```

Now try to continue the program by entering

```
cont
```

Run the program and ESCAPE again. Whilst in this direct mode press ESCAPE a second time.

Now enter

```
cont
```

A program may not continue if ESCAPE is pressed or an error is made while in the direct mode.

12.3 End of section summary

A program may continue if `CONT` is entered either:

1. after a `STOP`, or

2. after ESCAPE is pressed whilst a program is running

provided that:

- the program has not been altered;
- ESCAPE has not been pressed again;
- no error has occurred whilst in direct mode.

If the break occurs before a program line has been completed, then that line will be repeated when `CONT` is entered (if the program can be continued).

13 PROCedures

13.1 Creating a program

When we have an idea for a program we need to think in two ways:

– What do we want to do?
– How can we do it?

We do this by dividing the program into its separate components, for example: title, instructions, displays, inputs and responses.

Then we decide how we can make the computer do what we want.

Our thinking is affected by the language which we are using, ie by the ideas which we can express. It is no use having a good idea if we cannot find words to express it. The language should be part of the solution and not part of the problem.

COMAL has a way of helping to break down a program into separate stages: procedures can be used.

A procedure is a set of program statements that define a particular task. The procedure is then called into action by using the name of the procedure.

For example, a program might contain the lines:

```
 110 assign_variables
 120 select_background_colour
 130 select_text_colour
```

Each of these lines indicates that a procedure is to be executed. The names of the procedures usually explain what they are intended to do. The names must obey the same rules as for a variable.

Here is a very simple example of what might be contained in the procedure `select_background_colour`:

```
1000 PROC select_background_colour
1010   CLS
1020   PRINT TAB( 10,5);"Background Colour"
1030   PRINT TAB( 13,10);"0. Black"
1040   PRINT TAB( 13,12);"1. Red"
1050   PRINT TAB( 13,14);"2. Yellow"
1060   PRINT TAB( 13,16);"3. White"
```

```
 1070   PRINT TAB( 10,20);
 1080   INPUT "Your Choice 0-3 > ":choice
 1090   COLOUR 128 + choice
 1100 END PROC select_background_colour
```

The procedure begins with the keyword `PROC` and then the name of the procedure. The procedure ends with the keywords `END PROC` and the name of the procedure. There must be spaces between the `END`, the `PROC` and the procedure name.

When a procedure is called, the flow of the program is diverted into that procedure and back again. Indeed, the same procedure may be called into use several times in the program.

The procedure has only one entry point and only one exit point. The flow of the program is in at the beginning and out at the end.

When a program is listed the commands inside the procedure are indented by two places.

A procedure name need not be entered after `END PROC`. It will be inserted automatically. Leaving out the procedure name after `END PROC` is particularly useful when you have long names for the procedures.

The program lines within the `PROC` definition are only carried out when the procedure is called by name.

13.2 Example program

If we want to create a program that will test our reaction time we might consider the following to be the stages in the program.

1. Select the mode.
2. Display instructions.
3. Pause for a random length of time.
4. Display a symbol and set the `TIME` to 0.
5. Measure the response time.
6. Display the result.

When creating a program it is a very good idea to be clear in your mind what it is that the program must do.

By breaking the program down into small stages it is possible to cope with each stage quite easily by writing it as a procedure.

This forms the basis of what is called ‘top down analysis’. Each of the stages may be broken down into smaller easier steps until we can see how to carry out the simple steps.

Below is a program example of a reaction timer which demonstrates the use of procedures. It makes frequent use of the comment symbol `//` to break up the program into sections and to add extra comments.

When a program listing is given it should include enough comments to enable you (and anyone else) to see the structure of the program and what is happening. Whilst this does take up computer memory, we have plenty and it is better to write good programs which are clear, rather than incomprehensible programs which might use a little less memory. A well structured program with clear comments is, in fact, easier to write and usually very much faster to correct if it does have an error (or bug) in it.

The program makes use of a `*fx` command. These commands control many of the internal features or effects of the BBC Microcomputer. They are features of the machine and not COMAL. See the *BBC Microcomputer System User Guide* or the selection of `*fx` commands given in Part III.

Enter `auto` and then the following program.

```
 100 // REACTION TESTER
 110 //
 120 MODE := 6
 130 //
 140 // Main program structure
 150 //
 160 display_instructions
 170 pause
 180 show_symbol
 190 measure_response_time
 200 display_result
 210 //
 220 // That's all!
 230 // Now for the procedures.
 240 //
1000 PROC display_instructions
1010   PRINT TAB( 3,5);"Press SPACE when you see a *"
1020 END PROC display_instructions
1030 //
1100 PROC pause
1110   delay := RND( 1500,3500)
1120   FOR pause := 1 TO delay DO
1130     NULL
1140   NEXT pause
1150 END PROC pause
1160 //
```

```
1200 PROC show_symbol
1210   PRINT TAB( 18,12);"*"
1220   TIME := 0
1230 END PROC show_symbol
1240 //
1300 PROC measure_response_time
1310   // next line removes any stored key presses
1320   *fx 21,0
1330   response$ := GET$
1340   wait := TIME
1350 END PROC measure_response_time
1360 //
1400 PROC display_result
1410   PRINT TAB( 5,21);"You reacted in ";
1420   PRINT ;wait / 100;" seconds"
1430 END PROC display_result
1440 //
1500 END
```

Run the program but *don't* enter `new` just yet.

Enter `LIST ,999`

Notice that by using line numbers from 1000 upwards, we can easily separate the main program structure from the procedures.

Note: some of the points in the following sections are designed to be used with this program.

13.3 Changing MODE

`MODE` should not be assigned on a program line which is inside a procedure (unless you are also using a second processor). A `Not allowed` error will be produced when that program line is encountered.

13.4 CLEAR

If you press ESCAPE and interrupt the program or if a `STOP` is encountered inside a procedure, you will not be allowed to change mode until you have entered `CLEAR`. This is so that it is possible to use `CONT` to continue the program, if required. The command `CLEAR` removes all of the user variables from the computer's memory except the resident integer variables `a#` to `z#`.

13.5 EDIT

Enter

```
edit
```

`EDIT` is the command to list a program without the indentation. It may be used with line numbers to restrict the range displayed, eg

`EDIT 1100,1150`	will show lines 1100 to 1150 inclusive.
`EDIT 1340`	will show just line 1340.
`EDIT 1000,`	will show lines 1000 onwards.

Enter

```
EDIT ,210
```

which should show lines up to 210. Notice the `EXEC`s.

13.6 EXEC

Technically the statement `EXEC procedure_name` should be used to call a procedure into action. All the `EXEC`s in a program are shown when `EDIT` is used.

An `EXEC` is automatically inserted as the line is entered. However, when a program is listed, that `EXEC` is not displayed. If you enter `EXEC` in a program line, then that `EXEC` will always appear in the listing.

13.7 Listing longer programs

If your program has more lines than the screen can display then when you list it in full you can only read the last lines. You can list the first few lines with a command such as `LIST ,240` or you can press ESCAPE part way through the listing.

An alternative method is to make a listing pause as it is being displayed. There are two ways of achieving this.

1. Locate the CTRL (Control) key on the left of the keyboard, just above SHIFT. In a moment you will need to use two fingers of the left hand to press both the CTRL and SHIFT keys at the same time. When you list a program and depress these keys, the listing stops until they are released again. The listing may be interrupted and held whenever you want. Try it.

2. Hold the CTRL key down with the left hand and then press the N key and release both keys. When you list the program you will find that only one ‘page’ of the program will be displayed. To see the next page you must press the SHIFT key once. Try it now. You may use the ESCAPE key if you want to stop

the listing at a particular page. To turn off this ‘paged mode’ listing you can hold the CTRL key down and press the letter O key then release both keys. Turn off the paged listing.

13.8 Using procedures from direct mode

Provided that a program containing the procedure has just been `LIST`ed, `EDIT`ed, `DEBUG`ged or `RUN` then any procedure from that program may be called whilst in the direct mode.

If you still have the reaction timer program in the computer then enter

```
list
cls
```

and then

```
show_symbol
```

to execute the procedure.

13.9 End of section exercise

Suggested answers are given in chapter 43.

1. What amendments would you make to the reaction timer program of section 13.2 to make the `*` appear in a random position along any row from 6 to 20?

2. What further changes would you make so that in mode 1 the `*` appears in red and the instructions in yellow?

3. Arrange the following procedures into a sequence that would be suitable for a test question program.

```
display_response
mark_answer
assign_variables
ask_question
display_instructions
input_answer
```

4. What command will list a program with no indentation and with the `EXEC` commands showing?

5. What variables are not removed when the command `CLEAR` is entered?

6. If a program is interrupted in the middle of a procedure what command would you use before trying to change mode?

7. If you use CTRL N to turn on the page mode for listing, and then enter `list`, how do you get the following page to list?

8. How is paged listing turned off?

9. How can you interrupt and hold a listing?

What is wrong with the following program lines?

10. `200 proctest`

11. `200 PROC EDIT`

12. `300 endproctest`

13.10 End of section summary

1. A procedure is called into action by using its name or `EXEC` followed by the procedure name.

2. When a program is `EDIT`ed an `EXEC` is always displayed in front of the procedure calls and no indentation is shown.

3. A procedure is defined between lines with `PROC` and `END PROC` followed by the procedure name.

4. The procedure name need not be given after `END PROC`. It will be inserted automatically.

5. If a program is interrupted in the middle of a procedure then `CLEAR` must be entered before a new `MODE` is assigned, unless a second processor is being used.

6. `MODE` may not be assigned within a procedure unless a second processor is being used.

7. CTRL N and CTRL O can be used to turn paging on or off when listing a program.

8. CTRL SHIFT will cause a listing to pause.

9. A procedure may be called in the direct mode provided that a program containing the procedure has just been `LIST`ed, `EDIT`ed, `DEBUG`ged or `RUN`.

14 Saving, loading and running programs

If you have typed in a long program such as the reaction timer given in chapter 13, then you will probably want to save it for future use.

14.1 Cataloguing

The `*CAT` is the command to give a CATalogue of the programs that are on your tape or disc. It can be used if you want to work your way through a tape seeing what programs are saved on it. A good index on your cassette cover is also helpful and it saves time if you can use the tape counter numbers.

14.2 Filenames

A 'filename' is used for any program or set of data stored on cassette or disc. The filename is the title by which you save or load the program. On a disc system the filename can be up to seven letters whereas on cassette the filename can be up to ten letters. It should not include spaces or punctuation marks.

14.3 Saving a program on cassette

If you have your cassette recorder connected up, with the seven pin plug in the socket at the back of the computer and have a cassette in place then decide on the title of your program (we have used the name `Title`).

Enter

```
*tape
```

Leaving a space after the word `save` enter:

```
save Title
```

When saving, no quotation marks are needed around the title, though they may be used.

When the prompt appears on the screen, press the PLAY and RECORD buttons and then press the RETURN key.

When the computer has finished its display of the number of blocks of program, it gives a set of symbols that represent the program length.

You may now press STOP on the cassette recorder and rewind. You can test if the program has saved properly. If you have a tone control, set it as high as possible.

Enter

```
*cat
```

Press only the PLAY button.

After a while the computer should show you the title and the block numbers as it checks them. If there are no error messages on the screen, then it has saved properly and you may press ESCAPE. If you get a `data?/rewind` message then you should press ESCAPE, rewind your tape and try playing it again with a different volume setting (usually higher, but not always).

14.4 Loading from cassette

With the cassette in the recorder at the place you believe is correct:

Enter

```
*tape
```

and

```
*cat
```

then press the PLAY button.

You may use the FAST FORWARD or REWIND buttons to move over unwanted programs.

When you see the title of the program that you are looking for, press ESCAPE and rewind the tape just enough to get back to the beginning of that program.

Making sure you leave a space after the word `load` enter

```
load title
```

and press the PLAY button.

A loading message will be displayed when the program is located on the tape.

The title does not need quotations around it. A program title is considered to be the same whether it is typed in capitals or small letters or a mixture.

14.5 Saving on disc

(If your disc has not been formatted then refer to your *Disc Filing System User Guide* to find how to format a disc.)

Enter

```
*disc
```

Place the floppy disc fully inside the drive unit, but don't close the door yet.

Enter

```
*cat
```

and whilst the disc is being spun inside the unit, close the door.

The catalogue of programs saved on the disc should appear on the screen.

Enter

```
save Title
```

where `Title` is the program filename.

A further `*cat` will show that the program name is in the catalogue of programs on the disc.

No warning

If you choose a filename that already exists on your catalogue of programs then you will lose the original program when you save the new one. No warning message is given that the filename already exists, unless the file is locked. Refer to the *Disc Filing System User Guide* under `*ACCESS` for details of locking and unlocking files.

14.6 Loading from disc

Enter

```
*disc
```

Insert the disc as explained in the section above, and then

```
load Title
```

where `Title` is the name of the program.

The program can then be run.

14.7 Deleting a file from disc

The command `delete Title` will remove the program called `Title` from the catalogue of programs, unless that program is locked.

14.8 Combined loading and running

The command

```
run Title
```

will cause a program called `Title` to be loaded and run.

It will work with disc or cassette. It is the equivalent of the BBC BASIC command `CHAIN "Title"`.

15 Decisions

In this section we shall learn about the keyword `IF`.

15.1 Do, repeat and decide

At the lowest level, there are three basic building blocks in constructing a computer program. Firstly, there are simple instructions such as `PRINT` or `INPUT` or assign. Secondly there is the ability to repeat a set of statements, as in the `FOR... NEXT...` loop. The third essential feature is the ability to decide whether or not a set of statements is to be carried out.

All these features may then be used to form the procedures which comprise the basic structure of the program.

15.2 Relations

A decision must be based on a 'condition' which may be true or false. When we write a program we do not know what the decision will be, but our program must cope with the various possibilities. The 'condition' is a relation between two values. We may compare in many ways. The following symbols are used.

Symbol	**Meaning**
=	The item on the left of the symbol is equal to the item on the right of the symbol. Notice that this symbol represents equality, and, therefore, does not have a `:` in front.
<	The item on the left of the symbol is less than the item on the right.
>	The item on the left of the symbol is greater than the item on the right.
<=	The item on the left of the symbol is less than or equal to the item on the right.
>=	The item on the left of the symbol is greater than or equal to the item on the right.
<>	The item on the left of the symbol is not equal to the item on the right.
IN	The string on the left of the symbol is contained in the string on the right.

15.3 Using conditions

Enter the direct command

```
if 2 < 3 then print "I believe that"
```

Use the arrow keys and COPY key to help you enter

```
if 2 > 3 then print "I believe that"
```

Nothing should have happened this time because the condition `2 > 3` is false and therefore the computer did not carry out the `print` command.

A single line `IF... THEN...` command can be used, when there is only one action to carry out.

Now use the copy facility again with each of the symbols

`>=` `<=` and `<>`

to enter similar `IF... THEN...` commands.

15.4 Comparing strings

All symbols are known by the computer as numbers. This means that we can compare letters in exactly the same way, by using the relations <, > and = to test whether one symbol occurs before another.

The computer compares the ASCII codes of the letters, which are in alphabetic order. Lower case letters have higher ASCII codes than capital letters.

Enter

```
if "AAA" < "AAB" then print "True"
```

and

```
if "AAA" < "AAa" then print "True"
```

and

```
if "TOM" < "TIM" then print "True"
```

Using IF... THEN... in a program

Now enter the following program.

```
   10 mode := 6
   20 print ''"Intelligence Test"
   30 print ''"In which city is the Tower of London?"''
   40 input "Your answer is > ":reply$
   50 print ''"You have ";
   60 if reply$ <> "LONDON" then print "not ";
   70 print "passed the test."'
```

Now make sure the CAPS LOCK is on and run the program a few times with right and wrong answers.

Upper and lower case

Then turn the CAPS LOCK off and run again with `london` as the answer.

The string `"london"` is not the same as `"LONDON"` and will be considered wrong.

Omitting THEN

Now change line 60 to leave out the `THEN` producing

```
   60 IF reply$ <> "LONDON" PRINT "NOT";
```

Now list line 60 by entering `list 60`

Notice that `THEN` has been inserted into the line.

`THEN` may be left out when entering an `IF` statement. It will be inserted automatically.

15.5 The long form of IF

A different form of the `IF` statement must be used when we require several results to follow. We are not allowed to have several separate commands on one line so the commands that are to be carried out cannot appear on the program line after the `THEN`. They are given separate lines and an `END IF` marker is used to show that the list is complete. The `END IF` occurs on a line of its own. See lines 1310 to 1340 on the next page.

Enter `new` and then the following program.

```
  100 // Five question test
  110 //
  120 correct := 0
  130 MODE := 6
  140 FOR question := 1 TO 5 DO
  150   make_up_question
  160   ask_question
  170   take_in_answer
  180   mark_and_respond
  190 NEXT question
  200 give_score
  210 //
 1000 PROC make_up_question
 1010   first := RND( 2,9) // random number 2 to 9
 1020   second := RND( 2,9)
 1030   answer := first * second
 1040 END PROC make_up_question
 1050 //
 1100 PROC ask_question
 1110   CLS
 1120   PRINT TAB( 5,5);"What is ";
 1130   PRINT ;first;" times ";second;" ?"
 1140 END PROC ask_question
 1150 //
 1200 PROC take_in_answer
 1210   PRINT TAB( 5,8);
 1220   INPUT "Your answer > ":reply
 1230 END PROC take_in_answer
 1240 //
 1300 PROC mark_and_respond
 1310   IF reply = answer THEN
 1320     PRINT 'TAB( 5);"Correct"
 1330     correct :+1
 1340   END IF
 1350   IF reply <> answer THEN PRINT 'TAB( 5);"Sorry,
it should be ";answer
 1360   PRINT 'TAB( 5);"Press Space Bar"
 1370   pause$ := GET$
 1380 END PROC mark_and_respond
 1390 //
 1400 PROC give_score
 1410   CLS
 1420   PRINT TAB( 5,5);"You got ";
 1430   PRINT ;correct;" right out of 5."
 1440 END PROC give_score
 1500 END
```

List the program and notice that the commands on lines 1320 and 1330 have been indented by two further places.

Run the program and then list it again.

15.6 ELSE

In line 1350 we are using the opposite test to the one used in line 1310. The commands are alternatives. We can make that clearer using the keyword `ELSE`.

Make the following changes and additions:

```
1340 else
1350 print 'tab(5);"Sorry, it should be ";answer
1355 end if
```

List the program. Notice that the `ELSE` occurs in line with the `IF` and the `END IF` and that the lines between `ELSE` and `END IF` are further indented by two places.

Run the program to prove that it still works.

15.7 End of section exercise

What is wrong with the `IF` commands in the following:

1.

```
190 IF reply = answer THEN
200   PRINT "Correct"
210   correct :+1
220   endif
```

2.

```
180 IF reply < answer THEN PRINT "Too low"
190 wrong :+ 1
200 END IF
```

3.

```
220 ELSE PRINT "Wrong"
```

4.

```
170 IF reply$ = answer$ THEN PRINT "correct"
180 ELSE
190 PRINT "wrong"
200 END IF
```

5.

```
150 IF reply > 9 THEN
160   PRINT "Too high"
170 END IF
180 ELSE
190 PRINT "OK"
```

6.

```
  140 IF this = that THEN PRINT "Correct"
  150 END IF
```

In the questions that follow, the answer programs should be short and don't need procedures.

7. Write a program to input two strings and print them out in alphabetical order.

8. Extend the program in question 7 so that `before` or `after` is printed between the words as appropriate, eg

```
CHICKEN before EGG
NIGHT after DAY
```

Suggested answers are contained in chapter 43.

15.8 End of section summary

1. The following relations may be used in a test of a particular statement.

=	equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal to
IN	one string contained in another

2. When a single instruction is to be carried out if a particular condition is true then a single line statement may be used. It will take the form

```
IF... THEN...
```

`THEN` will be inserted automatically, if left out.

3. When several instructions are to be carried out if a particular condition is true then a multi-line `IF` structure must be used. It will take the form

```
IF... THEN
  Command 1
  Command 2
  etc
END IF
```

`THEN` will be inserted, if left out.

The `END IF`, which must have a space between `END` and `IF`, marks the end of the statements that are to be carried out if the condition is met.

All the statements inside the structure will be indented by two places.

Multi-statement lines are *not* allowed.

4. When an alternative set of instructions is to be carried out if the condition is not met then an `ELSE` statement may be used. `ELSE` must have a line to itself. A single line `IF... THEN...` cannot precede an `ELSE`. The `ELSE` statement will appear in line with the `IF` and `END IF`.

```
IF... THEN
  ...
ELSE
  ...
END IF
```

16 A conditional loop

In this chapter we shall learn how to use the `REPEAT UNTIL...` *loop and encounter* `TRUE` *and* `FALSE`.

In chapter 6 we met the `FOR... NEXT...` loop which carried out a set of statements a precise number of times.

Sometimes, when we are writing a program, we are not sure exactly how many times we may need to carry out a set of statements. For example, we may not know how many times we will need to take in an answer before it is correct.

In this situation a 'conditional loop' is needed. The loop will be carried out over and over again `UNTIL` the condition is satisfied. As an example let us print out random numbers from 1 to 6 until we get a 6. We don't know how many we shall print, other than that we shall print at least one.

16.1 REPEAT

`REPEAT` is the keyword to start a conditional loop, with a test at the end of the loop. The test is given after the keyword `UNTIL`. `REPEAT` must be on a line of its own.

Enter `new` and then

```
10 mode := 6
20 repeat
30 x := rnd(6) // note :=
40 print x
50 until x = 6 // no colon
```

In line 50 the test is whether x equals 6. The x is not being assigned the value 6.

List the program.

Statements between the `REPEAT` and the `UNTIL` are indented.

Run the program a few times.

Notice that the number of loops varies quite a lot. When the program is run, we don't know in advance how many numbers will be printed out.

At least once

A `REPEAT UNTIL...` loop is carried out at least once.

The condition that ends the loop is not encountered until we have passed through the loop the first time. This is not the same as with the `FOR... NEXT...` loop, which will not be carried out at all, if the `FOR` command sets an impossible loop.

16.2 Conditional endings

Now add lines

```
 5 total := 0
```

and

```
35 total :+ x // increase by x
```

Change 40 to

```
40 PRINT x,total
```

and 50 to

```
50 UNTIL total > 50
```

which should list as

```
 5 total := 0
10 MODE := 6
20 REPEAT
30   x := RND( 6) // note :=
35   total :+ x // increase by x
40   PRINT x,total
50 UNTIL total > 50
```

Run the program a few times.

We can even make `TIME` the condition to end the loop. Add

```
15 TIME := 0
```

and change 50 to

```
50 UNTIL TIME > 50
```

List and then run the program a few times.

Enter `new` and then the following program.

```
   10 mode := 5
   20 repeat
   30 number := rnd(3)
   40 colour number
   50 across := rnd(0,19)
   60 down := rnd(0,29)
   70 print tab(across,down);"*"
   80 until across + down
```

Think about the condition that will end the program.

Then run it a few times.

Any condition that is not `FALSE` will end a `REPEAT UNTIL...` loop.

16.3 TRUE and FALSE

The `REPEAT UNTIL...` loop is continued until the final condition is met. In other words, whilst the final condition is `FALSE` the loop will continue.

We can now look at the concept which the computer has of `TRUE` and `FALSE`.

Numerical values of TRUE and FALSE

The computer considers `TRUE` and `FALSE` conditions to be represented by numbers. We can make the computer tell us what these numbers are.

Change mode to 6 then enter

```
print false
```

and

```
print true
```

`TRUE` and `FALSE` are system functions that return the values −1 for `TRUE` and 0 for `FALSE`. They will list in capitals.

16.4 Numbers that end loops

Enter `new` and then

```
   10 enough := false
   20 mode := 6
   30 repeat
   40 print "Hear ";
   50 until enough
```

Run the program.

Press the ESCAPE key when you want that to stop.

The loop kept on running because the condition at the end was `FALSE`.

Change line 10 to

```
    10 enough := true
```

and run again.

This time the loop is ended, first time through, because the end condition is `TRUE` (ie not `FALSE`).

Now change line 50 to

```
    50 until 0
```

Delete line 10 and run again.

You will need the ESCAPE key again.

List the program.

The computer will accept integers as representations of the conditions `FALSE` and `TRUE`.

Change line 50 to

```
    50 until -1
```

and run again.

The loop is ended because −1 represents the condition `TRUE`.

Change line 50 to

```
    50 until 42
```

and run again.

The loop ends just the same as if `TRUE` had been encountered.

The computer will consider any integer which is not zero (ie not `FALSE`) to mean that the loop is to end.

Truncation

Change line 50 to

```
    50 until 0.5
```

and run again.

The ESCAPE key will be needed this time.

If `UNTIL` is followed by a real value (ie a number which may have a decimal part) then the value is truncated (ie has its decimal part removed) and the integer remaining is examined to see if it represents a true or false condition.

-0.5 would truncate to -0 (or 0) representing `FALSE`
+1.5 would truncate to 1 representing not `FALSE` ie true
-1.5 would truncate to -1 representing `TRUE`

16.5 Using GET$ and IN

As an example let's try taking in a letter from the keyboard until one of the set Y, y, N, n is pressed.

We can use `GET$` to take in the key which is pressed and use the `IN` function to test whether the letter is in the set.

Enter `new` and the following program

```
    10 mode := 6
    20 print ''"Press Y or N"
    30 repeat
    40 reply$ := get$
    50 until reply$ in "YyNn" <> 0
    60 print "You pressed ";reply$
```

List and then examine line 50.

Not FALSE

If the key Y or N is pressed, `reply$ in "YyNn"` will produce 1, 2, 3 or 4, depending on whether we have the CAPS LOCK set on or off.

1, 2, 3 and 4 are all not zero and so would bring a `REPEAT UNTIL...` loop to an end.

If any other key is pressed, `reply$ in "YyNn"` will produce 0 and so the loop will continue.

Run the program.

Try pressing the number and letter keys other than Y and N.

Although the key press is detected, `GET$` does not cause the symbol to be printed.

Now try Y.

Repeat the program with y, N and n.

16.6 End of section exercise

1. Print out a random number from 1 to 26 until 26 is printed.

2. Amend the program in question 1 to print out a random number from 65 to 90 until 90 is produced.

3. Amend the program in question 2 to print random letters of the alphabet until a letter `Z` is printed.

4. Create a program to print random letters of the alphabet in random positions on the screen until a letter `Z` is printed.

5. Amend the stars program of section 16.2 to work in seven colours in mode 2.

6. Amend question 5 to print a random letter of the alphabet instead of a star. Then amend further to print a random character whose ASCII code is from 32 to 126.

7. Write a program to ask a question which requires a Yes/No answer, accepts a press of the Y or N key and then prints `Yes` in full or `No` in full.

8. Write a program to test multiplication of two random whole numbers (each from 2 to 9), allowing as many questions as possible to be answered in 30 seconds and then giving a score.

Suggested solutions are given in chapter 43.

16.7 End of section summary

1. `REPEAT`, which must be on a line on its own, is the keyword to start carrying out a set of commands until a condition is found to be not `FALSE` (ie true).

2. The statements which are to be repeated are placed on the following lines. They are indented in a listing.

3. The condition to end the loop is placed after the keyword `UNTIL`.

4. A `REPEAT UNTIL...` loop will carry out the statements at least once.

5. `TRUE` and `FALSE` are system functions returning the values −1 and 0 respectively.

6. Whilst the condition after `UNTIL` is `FALSE` the loop will continue.

7. In a test of the condition after `UNTIL`, any value which, after truncation to an integer, is non-zero will be taken as not `FALSE`, and the loop will end. Note that the truncation means that, for example, 0.1 and −0.1 will both be taken as 0 giving `FALSE` conditions, but −1.1 will truncate to −1 which represents a `TRUE` condition.

17 Storing information in a program

In this section we shall learn how to use the `READ` *statement, the* `DATA` *statement and* `EOD`.

In an earlier section we learnt how to take in values from the keyboard using the `INPUT` and `GET` statements. They were particularly useful where we wanted to control some part of the program, eg the colour to be used. However, the statement `INPUT` can be very inefficient: every time we run the program we have to enter the information. There is an alternative way in which we may give information to the computer using data statements.

17.1 DATA

Data can be thought of as 'a collection of values that may be referred to'. The items of data can be stored on a program line.

`DATA` is a keyword that may be placed at the beginning of a program line. It indicates that the items which follow are data available for use in the program, eg `1000 DATA 2,3,4,5,6,7,8,9,10,Jack,Queen,King,Ace`

The values stored on a `DATA` line must be separated by commas, but strings need not normally have quotation marks.

17.2 READ

The `READ` statement is used whenever we require to take in a value from the list of `DATA` and assign it to a variable, eg `READ number` or `READ name$`.

Enter `new` and then the following program:

```
10 mode := 6
20 repeat
30 read number
40 print number
50 until number < 0
60 data 5,4,3,2,-999
```

Run the program.

Change 60 to

```
    60 DATA 5,4,3,2,1
```

and run the program again.

What is the meaning of `EOD` in the error message?

17.3 EOD

When there is no more data, we don't normally want the computer to produce an error message and decline to continue.

In COMAL we have a marker to indicate when we have come to the End Of Data (`EOD`).

`EOD` is a system function that returns one of two values, `TRUE`, −1 or `FALSE`, 0. Whilst there is more `DATA` to be `READ`, `EOD` is set to `FALSE`. When all the data has been `READ`, `EOD` is set to `TRUE`.

Change line 50 to

```
    50 UNTIL eod
```

and run the program again.

This time the program comes to an end without producing an error message.

Now enter

```
     5 data 10,9,8,7,6
```

and list and run the program.

Reading data in sequence

The `DATA` is read into the computer in the order in which it occurs in the program. A 'pointer' is used to keep track of the next item of `DATA` which can be read. When the pointer has no more data to indicate, `EOD` is set to `TRUE`.

17.4 RESTORE

It is sometimes useful to be able to refer to the same set of data whenever it is needed. We need to be able to set the pointer back to the beginning of the list.

The keyword `RESTORE` can be used to reset the pointer to the beginning of the `DATA` in the program.

Change lines 20 and 50 to

```
20 for item := 1 to 7
50 next item
```

and add

```
70 restore
80 read number
90 print number
```

A listing should produce

```
 5 DATA 10,9,8,7,6
10 MODE := 6
20 FOR item := 1 TO 7 DO
30   READ number
40   PRINT number
50 NEXT item
60 DATA 5,4,3,2,1
70 RESTORE
80 READ number
90 PRINT number
```

Run the program.

Once the first seven items have been read and printed `RESTORE` sets the data pointer back to the first item of data. It is that item which is read next and printed.

Labels

We can store several sets of quite different `DATA` in a program. If we need to refer to a particular set then we must be able to reset the pointer to the beginning of that set of `DATA`. We can do this by using 'labels' that mark the beginning of a set of `DATA`.

A 'label' may be placed anywhere in a program on a line of its own. The label is indicated by placing a colon `:` immediately after a name. The label name must obey the same rules as for a variable.

RESTORE with a label

Enter

```
55 second_set:
```

Change 70 to

```
   70 RESTORE second_set
```

RESTORE can be used with a label to indicate the place from which DATA is to be read next.

Run the program.

List the program and check through to see how RESTORE to a label has worked.

17.5 READing strings

String data can be READ just as easily as numeric data. Quotations need not normally be placed around strings, though if you require to have spaces at the ends of a string or to include punctuation in a string then quotation marks should be used around the string.

17.6 READing several items at once

Commands such as

```
READ x,y,z
```

or

```
READ a$,b$,c$
```

or

```
READ name$,code,tel_number
```

enable several variables to be assigned in one line.

Enter new.

Here is a demonstration program that makes use of quotations in the data statements.

```
   10 MODE := 6
   20 REPEAT
   30   READ name$,address1$,address2$,town$,code$
   40   CLS
   50   PRINT ''name$'address1$'address2$'town$'code$
   60   PRINT TAB( 5,20);"Press Space Bar"
   70   space$ := GET$
   80 UNTIL EOD
   90 DATA "Mr J.R. Smith","123, Parkside"
```

```
  100 DATA "Newton St. Thomas",SOMETOWN,SO11 1CE
  110 DATA "Sir Edwin Drood, M.P.",The Manor
  120 DATA Littleton,Nr ROCHESTER,RO1 5TE
```

Add your own data lines and run the program.

17.7 Quotation marks and spaces inside DATA items

If a quotation mark occurs as the first non-space character in a data item then it is taken as indicating the start of the string. The end of the string is indicated by the next single quotation mark. Neither quotation mark is taken as part of the string.

In order to get a quotation mark as an item in string data which is, itself, contained in quotation marks use two consecutive quotation marks, which will then be interpreted as one.

If the first non-space character of the data item is not a quotation mark then the string will be `READ` exactly as it appears in the `DATA` list, including any quotation marks, leading spaces (except on the first item in a data list) or trailing spaces (except on the last item in a data list).

For example:

```
 2200 DATA The "Boss","""Equality"""
 2210 DATA "2, Par's Avenue",NOWHERE
 2220 DATA NO1 4US
```

used in the above program would produce

```
The "Boss"
"Equality"
2, Par's Avenue
NOWHERE
NO1 4US
```

This is a most useful feature. Without it we would be limited in our use of data items or we should have to go through some unpleasant programming to enter quotations into our strings.

17.8 Demonstration program

In this example we read numeric and string data. Enter new and then:

```
  100 MODE := 6
  110 //
  120 REPEAT
  130   READ x,y,good$,bad$
```

```
140   PRINT'"What is ";x;" times ";y'
150   INPUT "Your answer is > ":answer
160   IF answer = x * y THEN
170     PRINT ',good$'
180   ELSE
190     PRINT ',bad$'
200   END IF
210 UNTIL EOD
220 //
230 DATA 7,9,Good,Sorry
240 DATA 12,8,Splendid,Incorrect
250 DATA 23,76,Excellent,Not quite
260 DATA 2,2,Well done,Oh dear !
```

Now, test the program using right and wrong answers.

17.9 Data that can be numeric or string

If figures are given as a DATA item then they can be READ as either string or numeric (unless quotes are placed around the figure). Enter new and then

```
10 mode := 6
20 print '"x$","x$ IN ""12345"""
30 repeat
40 read x$
50 print 'x$,x$ in "12345"
60 until eod
70 data 3,4,5,1,2,6
```

Run the program. The data is taken to be string data if the READ command is to read a string.

Change 20 to

```
20 PRINT '"Numeric x"
```

40 to

```
40 READ x
```

and 50 to

```
50 PRINT x
```

and run the program again. This time the data is taken to be numeric because the READ statement indicates that numeric DATA is expected.

17.10 End of section exercise

Suggested answers are in chapter 43.

What is wrong with the following parts of programs?

1.
```
100 here
120 DATA 1,2,3,4,5,6
130 READ x
140 PRINT x
150 RESTORE here
```

2.
```
100 DATA 9,10,Jack,Queen,King,Ace
110 FOR card = 1 TO 3
120   READ number
130   PRINT number
140 NEXT card
```

3.
```
100 DATA 2 3 4 5 6 7 8 9
110 REPEAT
120   READ number
130   PRINT number
140 UNTIL EOD
```

4.
```
100 suit:
110 DATA Clubs,Diamonds,Hearts,Spades
120 FOR choice := 1 TO RND( 4)
130   READ suit$
140   RESTORE suit
150 NEXT choice
160 PRINT suit$
```

5. Write a program to `READ` in a multiple choice type question, four alternative answers and the number of the correct answer. Display the question and suggested answers, then take in a response and mark the choice as right or wrong.

6. Amend the program for question 5 so that it keeps accepting responses until the correct answer is chosen.

7. Amend the program for question 6 so that it will continue taking in questions and answers until it runs out of data.

17.11 End of section summary

1. String and numeric `DATA` may be stored on a program line which begins with the keyword `DATA`.

2. The values may be assigned to a variable by the keyword `READ`.

3. The values stored on a `DATA` line must be separated by commas, but strings do not normally need quotes.

4. A double quotation mark (`""`) may be used to indicate the position of a single quotation mark when a string has quotes around it.

5. A pointer to the next unread `DATA` item is kept by the computer.

6. The pointer can be reset to the beginning of the `DATA` by the command `RESTORE`.

7. A label may be placed on its own line in a program. The label is indicated by a colon following immediately after any name which meets the requirements of a variable.

8. The pointer can be set to a particular place by the command `RESTORE` followed by the name of a label. Example: `RESTORE here`, where `here:` is a label in the program.

9. `EOD` is a system function which returns either `-1`, `TRUE` or 0, `FALSE` depending on whether the End Of Data has been reached. `EOD` can be used as a condition to end a `REPEAT UNTIL...` loop.

18 More functions, operators and variables

In this chapter we shall encounter the system functions `POS`*,* `VPOS`*,* `COUNT`*,* `ABS` *and* `SGN`*. We shall also use the operators* `DIV` *and* `MOD` *and control printing with* `WIDTH`*.*

18.1 Screen position (POS and VPOS)

The position of the cursor on the screen can be read using the same numbers that are used for `TAB( )`. The numbers across start at 0 and then depending on the mode being used may go up to 19 or 39 or 79. The numbers down start at 0 and then may be up to 24 or 31 depending on the mode.

`POS` is a function which returns the number of the current cursor position across the screen. It is the position at which printing will continue.

`VPOS` is a function which returns the number of the current cursor position down the screen. It is the position at which printing will continue.

The computer can tell us the position of the cursor using `POS` for position across the screen and `VPOS` for the vertical position down the screen.

Enter (in the direct mode)

```
print pos
```

and

```
print vpos
```

`POS` is zero because when the RETURN key is pressed the `POS`ition at which printing will continue is at the beginning of the next line.

You cannot move the cursor by trying to assign values to `POS` and `VPOS`. They are functions that return values. Any attempt to assign values will produce a `Not allowed` error message. Try it if you like.

18.2 Examples of POS and VPOS

Enter the following program to show the effect of `POS`

```
10 mode := 6
20 repeat
30 print "*";
40 until pos = 19
```

and run it.

Now add:

```
15 print tab(20);
```

Run and list.

After line 15 `POS` returns a value greater than 19, so printing continues with an overflow into the next row causing `POS` to return the value 0. When `POS` returns the value 19 the printing stops.

Delete line 15 and change line 40 to

```
40 until vpos = 19
```

List and run.

Now enter

```
15 print tab(0,20);
```

List and run.

You will need to press the ESCAPE key to stop the printing. After line 15 `VPOS` returns the value 20 and then increases with each new line until it reaches 24 (at the bottom of the screen). The cursor does not go to the top of the screen. The top screen line is lost, all the other lines move up and a new line becomes available at the bottom of the screen.

Delete line 15.

18.3 Carriage return and line feed

In the days when computer programs were typed at a mechanical keyboard the term ‘carriage return’ came into use for the instruction that returned the carriage to the beginning of a line. A ‘line feed’ would cause the paper to move up by one line. Both a carriage return and a line feed would be needed if a new line of entry was required.

Nowadays, when we are entering data on an electronic keyboard, connected to a screen display, the RETURN key has the effect of a carriage return and a line feed. When a PRINT command finishes without a comma or semi-colon then a carriage return is used to make the next print position move on to a fresh line. A carriage return is not considered to have occurred if the printing overflows from one line into the next.

The terms carriage return and line feed are still in use. They also apply to the actions of a printer when the print head moves to the beginning of a line and when the paper is moved up.

18.4 COUNT

The computer keeps a record of how many characters it has printed since the last 'carriage return'.

COUNT is the function which returns the number of characters printed since a carriage return.

When printing 'overflows' into the next line COUNT does not reset to zero, but continues to count on.

Change line 40 to

```
    40 UNTIL COUNT = 100
```

List and run.

Change line 40 to

```
    40 UNTIL COUNT = 255
```

and run again.

This is the maximum value that COUNT can return.

18.5 WIDTH

Add the line

```
    15 WIDTH := 15
```

and run the program.

WIDTH is a system variable that may be used to control the number of characters printed before the text cursor is made to move on to the next line with a carriage return and line feed (COUNT is then reset to zero).

In line 15 change WIDTH to 8 and run the program.

Press ESCAPE and then list.

Entertaining, but not very helpful in a listing.

Enter the direct command

```
WIDTH := 0
```

and list again.

This is the default (or normal) value for `WIDTH`.

18.6 ABS

`ABS` is a function which returns the 'absolute' value of a number, ie its size disregarding sign, eg `ABS(3)` and `ABS(-3)` both return `3`.

Enter `new` and

```
    10 mode := 6
    20 for j := -11 to 11
    30 print j,abs(j)
    40 next j
```

List and run the program.

We can use `ABS` to control the `TAB` position.

Change 30 to

```
    30 PRINT TAB( ABS(j));"Acornsoft COMAL"
```

List and run again.

If you want to move the shape towards the centre of the screen then change line 30 to

```
    30 PRINT TAB( 5 + ABS(j));"Acornsoft COMAL"
```

18.7 SGN

`SGN` is a function which returns `1`, `0`, or `-1` to indicate a positive, zero or negative argument (value).

Change line 30 to

```
    30 PRINT j,SGN(j)
```

List and run.

18.8 DIV

`DIV` is a binary operator that has an effect similar to division, except that it expects to work with integers and it produces an integer.

The whole number which is to be divided, goes in front of `DIV` and the whole number by which we are dividing, follows `DIV`.

`6 DIV 3` gives `2`, but `7 DIV 3` also gives `2` because `DIV` returns only the whole number result.

If we apply `DIV` to a decimal value then, in Acornsoft COMAL, the decimal fraction part is ignored, for example: `6.2 DIV 3` returns `2`.

If we use a non-integral value for `DIV` to divide by, then that has its fractional part ignored, for example: `6 DIV 2.9` returns `3`.

`DIV` returns the whole number of times that one whole number may be divided by another whole number.

Change 30 to

```
   30 PRINT j,j DIV 3 // Notice spaces
```

The spaces are needed so that the keyword `DIV` is not thought to be part of a variable `jdiv3`.

List and run.

Notice how the negative numbers are treated.

18.9 MOD

`MOD` is a binary operator that returns the integral remainder when one integer is divided by another. If `MOD` operates on non-integral values then, in Acornsoft COMAL, it removes the fractional parts and returns the whole number remainder of the division, for example:

```
24 MOD 6 returns 0          24 MOD 5   returns 4
 6 MOD 6 returns 0          24.5 MOD 5 returns 4
 5 MOD 6 returns 5          24 MOD 5.2 returns 4
```

`MOD` returns the whole number remainder of dividing one whole number by another.

Change 30 to

```
   30 PRINT j, j DIV 3, j MOD 3
```

List and run.

Care with MOD

The effect of `MOD` is not quite the same as in 'modulo' arithmetic. In arithmetic modulo 3, the values 0, 1, 2 are the only numbers that can result from a calculation. −1 would be represented by 2, −2 by 1 and −3 by 0. −7 modulo 3 would produce 2.

`MOD` in Acornsoft COMAL returns the remainder when the division has taken place a whole number of times. `-7 MOD 3` produces `-1`.

18.10 Precedence of MOD and DIV

`MOD` and `DIV` have a higher precedence than `+` and `-` but the same as `*` or `/`.

Enter

```
PRINT 2*5 MOD 3
```

which produces the value given by `(2*5) MOD 3` ie `1`.

Enter

```
PRINT 2 + 5 DIV 3
```

which produces the value given by `2 + (5 DIV 3)` ie `3`.

Take care with `MOD` and `DIV`. Brackets may be needed. See section 40.6 for a full precedence table.

18.11 Reversing the process

For any particular set of whole numbers, `MOD` and `DIV` are related. For example:

```
59 = 8 * (59 DIV 8) + 59 MOD 8
77 = 6 * (77 DIV 6) + 77 MOD 6
a# = b# * (a# DIV b#) + a# MOD b#
```

18.12 End of section exercise

What would you expect each of the following commands to produce?

1. `PRINT ABS(-5)`
2. `PRINT 15 DIV 4`
3. `PRINT SGN(15)`
4. `PRINT 15 MOD 3`
5. `PRINT -SGN(-15)`
6. `PRINT ABS(-9 DIV 4)`
7. `PRINT 15.5 MOD 4`
8. `PRINT 16.8 DIV 2.6`
9. `PRINT 4 * 10 DIV 4 + 10 MOD 4`

Using the functions `ABS`, `SGN`, `DIV`, `MOD` and `INT` what programs would produce the following sets of numbers?

10.	5	11.	1	12.	3
	4		1		3
	3		0		3
	2		-1		2
	1		-1		2
	0		-1		2
	1		-1		1
	2				1
	3				1
13.	4	14.	2	15.	2
	3		1		2
	2		0		1
	1		2		1
	0		1		0
	4		0		0
	3		-1		-1
	2		-2		-1
	1		0		-2
	0		-1		-2

Suggested answers are given in chapter 43.

18.13 End of section summary

1. `POS` and `VPOS` are functions that return the positions of the cursor across and down the screen. They cannot be set as a way of controlling the cursor.

2. `COUNT` is a function that returns the number of characters printed since a carriage return.

3. `WIDTH` is a system variable that may be used to control the number of characters that will be printed before a carriage return and line feed are given.

4. `ABS` is a function that returns absolute value, ie the value without sign.

5. `SGN` is a function returning 1, 0 or -1 to indicate a positive, zero or negative argument.

6. `DIV` is an operator which returns the whole number of times that one whole number may be divided by another.

7. `MOD` is an operator that returns the whole number remainder when one whole number is divided by another.

8. `MOD` and `DIV` have a higher precedence than `+` or `-` and the same precedence as `*` or `/`.

19 Arrays

In this section we shall learn how to use an array to store a set of similar items and how to dimension the array.

19.1 Introduction

Imagine a squad of servicemen being given drill instruction. The instructor sees a fault in the way an arm is positioned and snaps out a command.

'Straighten your arm: rear rank, number three!'

The whole squad is being told, but the instruction is directly addressed to one member of the squad who is identified by the rank and by position along that rank.

We can use a similar method to handle sets of information in the computer. We can use one variable to refer to the 'squad' of values and then identify individual items by using numbers that indicate the row and position in that row.

The computer squad is called an 'array' and the numbers that identify individual items are called 'subscripts'. For example, if we want to keep a record of pieces on a chess board then `piece$(5,4)` could represent the name of the piece which is in the fifth row and on the fourth place. The 5 is the first subscript and the 4 is the second subscript.

An array is a collection of values that are referred to by the same identifying name, but each individual value is located by subscripts.

19.2 Dimensions

The array need not have two 'subscripts', it could have just one, or in Acornsoft COMAL it could have as many as eight. Each subscript represents a 'dimension' of the array.

`total(9)` could be assigned as, say, the total marks scored by candidate number `9`. `total( )` is a one-dimensional array.

`item$(3,2,12)` could refer to the name of the component which is stored in tray `12` of level `2` of the store cabinet `3`. `item$` is a three dimensional array.

19.3 Storing arrays

The computer memory is certainly not eight dimensional. How can it cope with the storage of array elements?

If we go back to the ‘squad’ idea then we can see what happens. Imagine that the instructor wants to take his squad into a classroom for a lesson. He cannot march them into the room as a squad, because only one person at a time can go through the doorway. However, if the instructor turns the squad to face the side, he can tell the front rank to march off into the classroom followed by the centre rank and finally the rear rank.

```
                      ***********
                      ***********
                      ***********
```

becomes

```
                      ***********
                      ***********
           ***********
```

which becomes

```
                      ***********
           ***********
***********
```

and finally

```
*********************************
```

A continuous list of values can be stored in the computer quite easily, but how does the computer then find the fifth item in the second row?

If the computer knows how many items are in each row then it can go past that number of items and then start counting the second row until it gets to the fifth item and similarly for other rows.

To store and identify items in an array the computer must be told the number of dimensions and the number of elements in each dimension.

19.4 DIMensioning arrays

The process of dimensioning is used to declare the number of dimensions (or subscripts) required and the range of values required for each subscript.

DIM price(20) will set aside space for the array of items referred to as price(0) through to price(20), ie 21 elements in total.

Enter new and the following program.

```
   10 mode := 6
   20 dim price(20)
   30 //
   40 for number := 0 to 20
   50 print price(number),
   60 next number
```

Run the program.

When a numeric array is dimensioned all the elements are set to zero. When a string array is dimensioned, all the elements are null strings, ie ""

Now add

```
   45 price(number) := rnd(99)
```

and list.

This program will place a random number from 1 to 99 into each of the elements of the array. The computer then obtains the value from the element and prints it out.

Run the program.

We can find an individual element of the array by using a direct command.

Enter

```
print price(20)
```

Setting the bounds of an array

The ‘bounds’ (or outer limits) of the array were set to 0 and 20 by our dimension statement: DIM price(20)

Sometimes we don’t really want the lower bound to be zero or even 1. Suppose we want to keep a record of the scores when we throw two dice.

We can use a dimension statement to restrict the bounds, eg DIM score(2:12). There are even occasions when it may be useful to set negative bounds.

The lower and upper bounds of a subscript can be set when the array is dimensioned. For example:

`DIM diagonal#(-7:7)` will create a one dimensional integer array from `diagonal#(-7)` to `diagonal#(7)`.

`DIM piece$(1:8,1:8)` will create a two dimensional string array from `piece$(1,1)` to `piece$(8,8)`.

When a string array is dimensioned, the room reserved for each element is the same as for an ordinary string, ie 40 places.

We may reserve more or less space if we wish by using the `OF` command, for example: `DIM initial$(1:100) OF 5`. This command would reserve five places for each of the elements `initial$(1)` through to `initial$(100)`. This would represent a saving of 100 times 35 memory places, ie 3500 in total – a most significant saving.

An existing array cannot be redimensioned, nor can the length of the string array elements be changed once they are set.

19.5 Example program

We are going to develop a program to display the results of simulating the shaking of two dice 60 times.

The process of dimensioning arrays and preparing the computer is usually called initialisation.

The stages in the program are initialisation, display of headings, the display itself and then finish.

Enter `new` and the following lines

```
100 // Frequency Chart
110 //
120 mode := 6
130 //
140 initialise
150 headings
160 display
170 finish
180 //
```

Lines 140 to 170 call the procedures which form the program.

If we can now write each procedure then the program is complete.

We can place the procedures on predetermined lines, usually beginning on a round number.

Enter

```
1000 proc initialise

1080 end proc
1090 //
1100 proc headings

1180 end proc
1190 //
1200 proc display

1280 end proc
1290 //
1300 proc finish

1380 end proc
1390 //
1400 end
```

Run the program.

If nothing appears to happen then the program is structurally sound. If error messages appear then list and check the lines again.

We can generate the 60 sets of results of throwing two dice. Enter

```
1210 for shake := 1 to 60
1220 die_1 := rnd(6)
1230 die_2 := rnd(6)
1240 sum := die_1 + die_2
1250 print die_1,die_2,sum
1260 next shake
```

List the program and then run it.

If we want to add up the number of times that a 2 occurs, the number of times a 3 occurs, and so on… then an array with a subscript from 2 to 12 would help. This should be done in the `initialise` procedure.

Enter

```
1010 dim score(2:12)
```

Each time the sum is 2, we want the element `score(2)` to go up by 1 and similarly for the other elements.

Enter

```
 1245 score(sum) :+ 1
```

Run and list the program.

The array, `score(2)` through to `score(12)`, has totalled up the number of twos, threes and so on up to twelves. We can find out how many times that seven occurred, by entering

```
print score(7)
```

It would be more interesting to see the scores build up as they are 'shaken'.

Add

```
 1110 print tab(0,20);"Score.2..3..4..5.
.6..7..8..9..10.11.12"'''"Total"
```

Change 1250 to:

```
 1250 print tab(3*sum,22);score(sum)
```

Line 1250 prints `score(sum)` when it changes.

List and run the program.

Improving the display

We can make the display look more interesting by printing a star above the score, one for each time it occurs. However `TAB` works from the top of the screen. This means tabbing across to the score and coming down by a smaller amount as `score(sum)` goes up. The first star occurs in row 19 and the second in row 18 etc.

Add

```
 1255 print tab(3*sum,20 - score(sum));"*"
```

Run the program.

To provide a title, clear the screen, then enter

```
 1120 print tab(0,2);"Frequency chart"
 1130 print "for 60 throws"
 1140 print "of two dice."
```

and run the program.

19.6 Cursor on and off

We can make use of one of the `VDU` commands to improve the presentation of our program.

The flashings of the cursor as it moves around the screen can be distracting. There is a VDU command to turn the cursor off and a similar one to turn it back on again.

Clear the screen and then, taking care with spaces, commas and semi-colons enter

```
 1020 vdu 23,1,0;0;0;0; // cursor off
```

and

```
 1310 vdu 23,1,1;0;0;0; // cursor on
 1320 print tab(0,5); // park cursor
```

The full listing should now be:

```
  100 // Frequency Chart
  110 //
  120 MODE := 6
  130 //
  140 initialise
  150 headings
  160 display
  170 finish
  180 //
 1000 PROC initialise
 1010   DIM score(2:12)
 1020   VDU 23,1,0;0;0;0; // cursor off
 1080 END PROC initialise
 1090 //
 1100 PROC headings
 1110   PRINT TAB( 0,20);"Score.2..3..4..5..6..7..8..9
..10.11.12"''"Total"
 1120   PRINT TAB( 0,2);"Frequency chart"
 1130   PRINT "for 60 throws"
 1140   PRINT "of two dice."
 1180 END PROC headings
 1190 //
 1200 PROC display
 1210   FOR shake := 1 TO 60 DO
 1220     die_1 := RND( 6)
 1230     die_2 := RND( 6)
 1240     sum := die_1 + die_2
 1245     score(sum) :+ 1
 1250     PRINT TAB( 3*sum,22);score(sum)
 1255     PRINT TAB( 3*sum,20 - score(sum));"*"
```

```
1260   NEXT shake
1280 END PROC display
1290 //
1300 PROC finish
1310   VDU 23,1,1;0;0;0; // cursor on
1320   PRINT TAB( 0,5); // park cursor
1380 END PROC finish
1390 //
1400 END
```

19.7 End of section exercise

Make a note of your answers and check them in chapter 43.

What commands would dimension the following arrays?

1. A one dimensional string array, `name$`, with bounds 0 to 50.

2. A one dimensional integer array, `points#`, with bounds 10 to 80.

3. A two dimensional array, `grid`, with bounds 1 to 3 for each subscript.

4. A two dimensional array, `ref_number`, with bounds 1 to 3 for the first subscript and 1 to 50 for the second.

5. A three dimensional string array, `item$`, with bounds 0 to 5 for each subscript.

6. Repeat question 5 but reserving only five places for each element.

7. Write a program to `READ` from `DATA` into the arrays `name$` and `cost#` the names of five items and the cost of those items in pence.

8. Add to the program in question 7 so that after the `DATA` has been `READ`, the item names and costs are shown and an `INPUT` is invited of the number of each item required. A final total cost is calculated and shown.

19.8 End of section summary

1. An array is a set of variables that are referred to by the same identifying name, but are individually located by subscripts, for example: `surname$(89)` or `sum(4,2)` or `place#(2,3,4)`.

2. An array must be dimensioned before it is used, eg `DIM surname$(100)` or `DIM sum(10,10)`.

3. The bounds of a dimension in the array may be chosen and can include negative values. If no lower bound is given then it is set to 0, for example: `DIM total(2:12)` allows the use of `total(2)` up to `total(12)`.

4. Up to eight dimensions may be used in an array. Example:

```
DIM coins#(1:2,1:2,1:2,1:2,1:2,1:2,1:2,1:2)
```

5. When a numeric array is dimensioned, all its elements are set to zero. When a string array is dimensioned all its elements are set to null strings.

6. 40 spaces are reserved for each element of a string array unless the keyword `OF` is used to set the length. Example: `DIM postcode$(100) OF 9` will reserve nine spaces for each of the 101 elements of the array `postcode$(0)` to `postcode$(100)`.

7. An existing array cannot be redimensioned. Nor can the length of the string array elements be changed once they are set.

20 More on the keyboard, loops and functions

In this chapter we shall learn about timed tests for key presses. We shall READ *data into an array using loops and also encounter* VAL *and* STR$.

In earlier sections we used GET or GET$ to wait until a key was pressed. We also used TIME to measure a delay. They can be combined in one statement.

20.1 INKEY

The internal ‘clock’ works in hundredths of a second so a delay of 200 represents a pause of two seconds.

---

x := INKEY(n) is the instruction to await a key press for n hundredths of a second. If a key is pressed then x is assigned its ASCII code: otherwise x is assigned −1.

Enter new and

```
   10 MODE := 6
   20 FOR go := 1 TO 20
   30   PRINT TAB( 6,8);"Press a key"
   40   answer := INKEY (200)
   50   IF answer = -1 THEN
   60     PRINT TAB( 25,8);"Too slow"
   70   ELSE
   80     PRINT TAB( 25,8);"Code ";answer
   90   END IF
  100   x := INKEY (100) // to see comment
  110   CLS
  120 NEXT go
  130 PRINT ''"Program complete"
```

Run the program testing quick and slow responses.

Press ESCAPE to get out of the program and then list.

20.2 INKEY$

`x$ := INKEY$(n)` is the instruction to await a key press for `n` hundredths of a second. If a key is pressed then `x$` is assigned the symbol of that key: otherwise a null string is assigned to `x$`.

Change the following lines so that we can use `INKEY$`. The changes are underlined.

```
  40 answer$ := INKEY$ (200)
  50 IF answer$ = "" THEN
  80   PRINT TAB( 25,8);answer$
 100   x$ := INKEY$ (100) // to see comment
```

Run again to check the effect of `INKEY$`.

20.3 Keys without ASCII codes

On the Acorn computers there are some keys that do not normally return ASCII codes, for example, the arrow keys or the function keys. How can we test whether one of these keys is pressed?

Each key has a ‘keyboard scan number’ which we can use to see if a particular key is pressed. The number is unique to the Acorn keyboard.

20.4 INKEY minus

`INKEY(-n)` is a function that tests whether the key with ‘keyboard scan number’ `n` is being pressed. There is no time delay used. If the key is being pressed at the time when the test is made then the value `TRUE` (`-1`) is given, otherwise `FALSE` (`0`) is produced.

We can use the keyboard scan number to test for normal as well as special keys.

Enter `new` and then

```
  10 mode := 6
  20 print ''"Press the RETURN key"
  30 delay := inkey(100) // pause for title
  40 repeat
  50 test := inkey(-74)
  60 print test
  70 until time<0 // loop for ever
```

Press the RETURN key several times as the program is running.

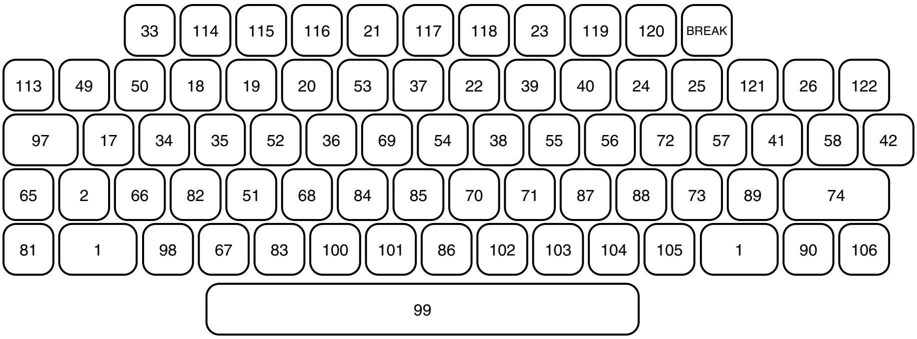
ESCAPE, list, delete lines 20 and 30 and change line 50 to

```
    50 test := inkey(-99)
```

Now run the program and see if you can find which key on the keyboard has a scan number of 99.

20.5 Keyboard scan codes for the BBC Microcomputer

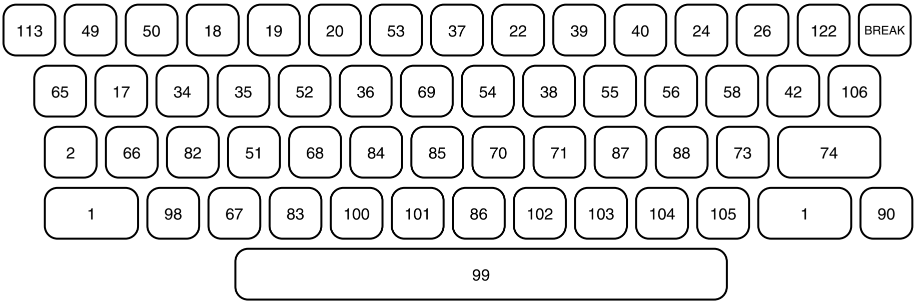
The keyboard scan codes are as shown on the following diagram:



The scan codes for 1 and ! are the same. To distinguish between them we should need to test for the SHIFT or SHIFT LOCK.

20.6 Acorn Electron keyboard scan codes

The keyboard scan codes for the Acorn Electron are essentially the same as for the BBC Microcomputer, but there are small differences due to the different keyboard layout.



The ‘function’ key being used is found by testing for both the FUNC key and the appropriate number key being pressed at the same time.

20.7 Example program

Enter this `inkey` minus program.

```
 100 // Movement
 110 //
 120 MODE := 4
 130 initialise
 140 REPEAT
 150   test_keys
 160   test_off_screen
 170   show_star
 180 UNTIL FALSE
 190 //
1000 PROC initialise
1010   across := 19 // Start Mid Screen
1020   down := 15
1030   VDU 23,1,0;0;0;0; // Cursor off
1040   left := -26 // Inkey values
1050   right := -122
1060   up := -58
1070   back := -42 // down already used
1080 END PROC initialise
1090 //
1100 PROC test_keys
1110   IF INKEY (left) THEN across:-1 // take one off
1120   IF INKEY (right) THEN across:+1 // add one on
1130   IF INKEY (up) THEN down:-1 // TAB down less
1140   IF INKEY (back) THEN down:+1 // TAB down more
1180 END PROC test_keys
1190 //
1200 PROC test_off_screen
1210   // if off sides then swap
1220   IF across < 0 THEN across := 39
1230   IF across >39 THEN across := 0
1240   // if off top/bottom then swap
1250   IF down <0 THEN down := 29
1260   IF down >29 THEN down := 0
1280 END PROC test_off_screen
1290 //
1300 PROC show_star
1310   PRINT TAB( across,down);"*"
1320   FOR delay := 1 TO 28 DO
1330     NULL // slow to our speed
```

```
1340    NEXT delay
1350    // Now wipe off
1360    PRINT TAB( across,down);" "
1380 END PROC show_star
1390 //
1400 END // Change mode to get cursor
```

Now run the program using arrow keys to move the star. Try pairs of keys to see if you can get diagonal movement.

20.8 Use of procedures

The program above could have been written without using procedures. It does, however, show how easy it is to analyse an idea into its simple stages and then to use procedures to program each stage.

Once you have thought out the correct main program you can write each procedure separately. In fact several people could share the writing of a program by taking a procedure each and then putting them together.

After a while you may find it useful to keep a library of useful procedures. You can then add (merge) them into your programs when needed.

It has been claimed that writing programs this way takes half the time of writing a non-structured program and about a tenth of the time to remove errors.

Programs written using procedures tend to look fairly long. That is not a particular drawback if it is easy to understand the logic of the program and it is possible to see how each procedure works.

Shorter programs which are difficult to understand will be extremely difficult to correct if they have errors and will be almost impossible to adjust for other purposes.

20.9 Nested loops

Enter `new` and the following program.

```
10 MODE := 6
20 FOR j:=1 TO 15 DO
30   FOR k:= 1 TO 5 DO
40     PRINT TAB( j,k);"*"
50     delay:=INKEY (10) // to slow down
60   NEXT k
70 NEXT j
```

Run the program.

Inner loop completed first

List the program.

Notice that the stars are printed as a series of columns.

The inner loop is carried out fully and will be carried out every time the outer loop increases by one.

Change line 40 to

```
40 print tab(k,j);"*"
```

and run.

This time the rows are completed first.

Mixed up loops

`FOR` and `NEXT k` are 'nested' inside the `FOR` and `NEXT j` (one loop is inside the other). If you mix up the loop variables, then you will receive a set of error messages when you try to run the program. Try swapping the `j` and `k` by entering

```
60 NEXT j
70 NEXT k
```

Run the program.

The program can be corrected by entering

```
60 NEXT
70 NEXT
```

The variables will be inserted correctly. List to see that the structure is now correct.

Enter `new` and the following array reading program.

```
 10 DIM array(3,5)
 20 MODE := 6
 30 FOR j:= 1 TO 5 DO
 40   FOR k := 1 TO 3 DO
 50   READ array(k,j)
 60   PRINT TAB( k,j);array(k,j)
 70   delay:= INKEY (10) // slow down
 80   NEXT k
 90 NEXT j
100 DATA 1,1,1,2,2,2,3,3,3,4,4,4,5,5,5
```

Run the program and then list it.

Now change line 60 to

```
   60 PRINT TAB( j,k);array(k,j)
```

and run the program again.

The shape has been reversed by the `TAB` instruction.

20.10 Matching subscripts to variables

Change 10 to

```
   10 DIM array(5,3)
```

and try to run again.

List the program.

The error arises because the second subscript of the array is only allowed to go up to 3 and we are trying to use subscript values of 4 and 5. If we want to read data into an array then we must be careful to match the 'shape' of the array to the loop shapes.

20.11 VAL

The point has been made that we cannot carry out a calculation with a string. However, we do have a function which can evaluate a suitable string.

`VAL` is a function which returns the value that a string would have produced, if it had been a mathematical expression.

Enter the direct command

```
this$ := "2 * x + 3"
```

`this$` is a string, but it has the appearance of a mathematical calculation. If we knew the value of `x` then we could evaluate `this$`.

Enter

```
x := 4
```

and then

```
print val(this$)
```

`VAL` takes the string called `this$` and tries to work out its value as if it had been a normal calculation.

Enter

```
that$ := "len nonsense"
```

and

```
print val(that$)
```

If a string does not make sense as a statement of a calculation, its VALue cannot be found.

A string can contain figures, eg `reply$ := "123"`

`VAL` may be used to obtain the numeric equivalent, 123.

Enter `new` and then

```
   10 MODE := 6
   20 FOR question = 1 TO 6
   30   INPUT "Enter some figure > ":reply$
   40   PRINT 'reply$,VAL(reply$)'
   50 NEXT question
```

Run the program.

Notice that `reply$` is printed to the left of its zone because it is a string, whereas `val(reply$)` is printed to the right of its zone because it is a number.

Run the program using a range of figures including some with decimal places.

20.12 STR$

There is a process for converting numeric elements to strings.

`STR$` is a function which takes a numeric element as the argument and returns a string with the same appearance.

Change 30 to

```
   30 INPUT "Enter a number > ":reply
```

and 40 to

```
   40 PRINT 'reply,STR$ (reply)'
```

List the program.

The numeric variable `reply` will be printed to the right of its zone, but its string value `STR$(reply)` will be printed alongside at the left of the next zone.

Run the program using integral and real values with decimal places.

20.13 End of section exercise

Make a note of your answers and check them in chapter 43.

1. What commands will have the following effects?

a) Waiting for a key to be pressed for up to one second and to assign the symbol of the key to `a$`.
b) Waiting for a key to be pressed for up to two seconds and to assign the ASCII code of the key to `b`.
c) Testing to see if the key C is being pressed and printing `-1` if it is.

2. What is wrong with the following program?

```
10 FOR j := 1 TO 5 DO
20   FOR k := -5 TO 1 DO
30     PRINT j*j - k*k
40   NEXT j
50 NEXT k
```

3. If `x := 2` and `d$ := "3 * 4 + x"`, what is `VAL(d$)`?

4. If `y := 5.678` what will be the result of the command
`print len(str$(y))`

20.14 End of section summary

1. `INKEY$(n)` is a function that waits for a duration of up to `n` hundredths of a second for a key to be pressed. If a key is pressed then it immediately returns the symbol. If no key is pressed, then a null string is returned.

2. `INKEY(n)` is a function that waits for a duration of up to `n` hundredths of a second for a key to be pressed. If a key is pressed then its ASCII code is returned immediately. If no key is pressed, then `-1` is returned.

3. `INKEY(-n)` is a function that tests if the key with 'keyboard scan number' `n` is being pressed. The values `-1` for `TRUE` or `0` for `FALSE` are returned.

4. `VAL` is a function which interprets its string argument as if it were an expression entered from the keyboard.

5. `STR$` is the function that returns the string expression which is the equivalent of its numeric argument.

21 Logical operators

In this chapter we shall extend the scope of the conditions we may use in `IF` *structures and* `REPEAT` *loops. We shall use the logical operators* `OR`, `AND`, `EOR` *and* `NOT`.

`FALSE` (meaning `0`) will usually appear in capital letters however `TRUE` (meaning `-1`) and true (any integer except 0) will be distinguished by using capitals or lower case respectively.

21.1 OR

In the section on the `REPEAT UNTIL...` loop we placed a condition at the end of the loop. If the condition is met then the loop ends. We sometimes wish to end a loop when either one `OR` other of two conditions is met.

The logical operator `OR` may be used to provide a true or `FALSE` result after examining two conditions. If one `OR` other of the conditions is true then the result is true.

Enter `new` and

```
 10 goes := 0
 20 actual := RND( 9)
 30 MODE := 6
 40 PRINT TAB( 0,5);"Guess my number, 1 to 9"
 50 //
 60 REPEAT
 70   INPUT "Your guess ":guess
 80   goes :+ 1
 90 UNTIL guess = actual OR goes = 3
100 //
110 IF guess = actual THEN
120   PRINT ''"Correct"
130 ELSE
140   PRINT ''"It was ";actual
150 END IF
```

Run the program a few times.

The logical operator `OR` can also be used with an `IF` test where one `OR` another condition (or both) must be met.

21.2 AND

We may want to bring a loop to an end only when one condition AND another are both met. Similarly an IF test could be based on both one condition AND another being true.

The logical operator AND is used to test two conditions. If both conditions are TRUE then TRUE is returned. If either of the conditions is FALSE then FALSE is returned.

Change the following lines to

```
   90 UNTIL guess = actual OR goes = 6
  110 IF guess = actual AND goes < 3 THEN PRINT
"Excellent"
  120 IF guess = actual AND goes > 2 AND goes < 5 THEN
PRINT "Good"
  130 IF guess = actual AND goes > 4 THEN PRINT "Slow"
  140 IF guess <> actual THEN PRINT "Hard luck"
```

and delete line 150.

21.3 EOR

The logical operator EOR (Exclusive OR) can be used to test whether one or other (but not both) of two conditions is true.

In the following program we shall print a star if the row is less than 12 OR if the column is less than 19, but not if both conditions apply.

Enter new and

```
   10 MODE := 6
   20 FOR down := 0 TO 23 DO
   30   FOR across := 0 TO 39 DO
   40     IF down < 12 EOR across < 19
THEN PRINT TAB( across,down);"*"
   50   NEXT across
   60 NEXT down
```

Run the program.

21.4 Truth tables

We can show the results of the operators OR, AND, EOR in tables where T stands for TRUE and F for FALSE.

For example: if statement 1 is T(RUE) but statement 2 is F(ALSE)

then T OR F will have the result T(RUE)
T AND F will have the result F(ALSE)
T EOR F will have the result T(RUE)

		Statement 2	
	OR	T	F
Statement	T	T	T
1	F	T	F

		Statement 2	
	AND	T	F
Statement	T	T	F
1	F	F	F

		Statement 2	
	EOR	T	F
Statement	T	F	T
1	F	T	F

21.5 Logical operators

OR, AND, EOR are all ‘logical operators’. They are implemented in Acornsoft COMAL as operations carried out on binary digits (zeros and ones). This means that it is possible to make statements such as

```
PRINT 1 OR 3
PRINT 2 AND 1
PRINT 2 EOR 3
```

Each binary digit of one number is compared with the corresponding binary digit of the other number according to the rules of `OR`, `AND` and `EOR`. 1 or 0 is then used in the appropriate place of the returned value.

As an example using just two binary digits we can see how the operator `OR` works.

`1 OR 3` will return the value 3. Comparisons are made for each of the columns and if there is a 1 in either one `OR` other of the places then a 1 is used in that column of the result.

	Number (base 10)		Binary digits Column values 2	1
	1		0	1
		`OR`		
	3		1	1
Result	3		1	1

The binary number 11 (decimal 3) is returned by the operation `1 OR 3`.

If the value returned by the `OR` operator is 0 then it represents a `FALSE` condition. Any other (integer) value represents a true condition.

The logical operator `AND` will give a 1 in the result if there is a 1 in the corresponding places of both the arguments.

	Number (base 10)		Binary digits Column values 2	1
	2		1	0
		`AND`		
	1		0	1
Result	0		0	0

The binary number 00 (decimal 0) is returned by the operation `2 AND 1`.

The operator `EOR` gives a 1 in the result place only if there is a 1 in one or other of the corresponding places of the argument but not in both.

	Number (base 10)		Binary digits Column values 2	1
	2		1	0
		EOR		
	3		1	1
Result	1		0	1

The binary number 01 (decimal 1) is returned by the operation `2 EOR 3`.

21.6 NOT

The keyword `NOT` reverses the values of `TRUE` and `FALSE`.

Enter

```
PRINT NOT TRUE
```

and

```
PRINT NOT FALSE
```

`NOT` is an operator that reverses `TRUE` and `FALSE` values.
`NOT FALSE` ie `NOT 0` produces `-1` or `TRUE`.
`NOT TRUE` ie `NOT -1` produces `0` or `FALSE`.

21.7 What is true?

Care must be taken with values other than −1.

`NOT` has the effect in the computer of taking the number away from −1 although strictly it inverts each binary digit turning ones into zeros and vice versa.

`NOT` is one of the highest priority operators. It will be carried out before any of the operators `OR`, `AND`, `EOR`.

Enter the direct command

```
PRINT NOT 5
```

The computer considers any value which is not zero to represent a true condition. For example, neither 5 nor −6 is `FALSE`, so they both represent a condition that will be taken to mean true.

Enter `new` and the following program.

```
10 MODE := 6
20 PRINT TAB( 5,5);"Press any number key"
30 REPEAT
40   x$ := GET$
50 UNTIL x$ IN "1234567890"
60 PRINT x$'"Finished"
```

This program waits for a number key to be pressed.

Run it to prove that it works correctly.

When a figure key is pressed then the function `x$ IN "1234567890"` will produce a number from 1 to 10, ie the position at which the figure occurs in the string. Since the value isn't zero (not `FALSE`) it is taken to mean the condition is true and the loop ends.

Change line 50 to

```
50 UNTIL NOT (x$ IN "1234567890")
```

The brackets are needed so that the `IN` function is carried out before the `NOT`. `NOT` cannot act on a string. Now run the program and press the 5 key.

`x$ IN "12345657890"` and `NOT (x$ IN "1234567890")` both produce a value that isn't zero. Therefore, they both represent conditions that end the loop.

If we want to exclude a set of symbols then an exact condition should be tested. Change 50 to

```
50 UNTIL x$ IN "1234567890" = 0
```

Handle `NOT` with care!

21.8 Truncation to integers

All numbers used in logical tests are truncated before they are tested, for example:

`NOT 0.1` is the same as `NOT 0`, ie `-1` or `TRUE`.

21.9 Precedence of operators

When a 'logical' condition has to be interpreted there must be an order of precedence between the operators `NOT`, `AND`, `OR`, `EOR`. This is the logical equivalent of the precedence used with `*` and `+` in the arithmetic.

`NOT` has the highest precedence.

`AND` has a higher precedence than both `OR` and `EOR`.

`OR` and `EOR` have the lowest precedence.

If `OR` and `EOR` are encountered in the same expression then they are applied in the order in which they occur.

Example:

```
2<3   EOR   3<4 AND 5<4
```

is treated as

```
2<3   EOR ( 3<4 AND 5<4 )
```

that is

```
TRUE  EOR       FALSE       
```
returning `TRUE`

whereas

```
( 2<3 EOR 3<4 ) AND 5<4
```

that is

```
     FALSE      AND FALSE
```
returns `FALSE`

A full precedence table is given in section 40.6.

21.10 End of section exercise

Do the following statements return `TRUE` or `FALSE`?

1. `2 < 3 OR 3 < 4`
2. `2 < 3 EOR 3 < 4`
3. `2 < 3 AND 3 < 4`
4. `3 > 2 OR 3 > 4`
5. `3 > 2 EOR 3 > 4`
6. `3 > 2 AND 3 > 4`
7. `NOT (2 < 3)`
8. `NOT (2 = 3)`

What numbers are represented by

9. `NOT 0`

10. `NOT -1`

11. `NOT 1`

The answers are given in chapter 43, but they can also be checked by instructing the computer to `PRINT` the statements given in the questions.

21.11 End of section summary

1. Any non-integral value is truncated to an integer before it is used by a logical operator.

2. `FALSE` has the value `0` and `TRUE` has the value `-1`, though any integral value that is not zero will be considered to represent a true condition. Any condition that isn't `FALSE` is considered true. The reverse cannot be assumed.

3. `OR` can be used to return true or `FALSE` as the result of two conditions. If one `OR` other of the conditions is true then the result is true. This includes the case when both conditions are true. Only if both conditions are `FALSE` will the result be `FALSE`.

4. `EOR` (Exclusive `OR`) can be used to return true or `FALSE` as the result of two conditions. It is a type of `OR` test but will exclude the case when both conditions are `TRUE`. If both conditions are `TRUE` the result is `FALSE`. If both conditions are `FALSE` then the result is `FALSE`.

5. `AND` can be used to return true or `FALSE` as the result of two conditions. If both conditions are `TRUE` then the result is `TRUE`.

6. `NOT` will reverse `TRUE` and `FALSE`. Any integral value other than `-1` (representing a condition which is taken as true) will produce a non-zero value (also representing a true condition). `NOT` is one of the highest priority operators.

7. More precisely, `OR`, `AND`, `EOR`, `NOT` all perform bitwise logical operations on the arguments.

Part II – for the more experienced programmer

22 Summary of Part I

This chapter contains a brief summary of the keywords introduced in Part I of the manual. There is then a general discussion of programming in COMAL, followed by a summary of other points raised in Part I. Chapter 23 onwards contains the material not covered in Part I.

The keywords given have been introduced in Part I, but the brief outline of each word is an indication of its function. Fuller explanations are provided in the sections indicated and syntax diagrams are given in the keyword summary in Part III.

Keyword	Brief outline	Section
`ABS (j)`	Absolute value of `j`	18.6
`AND`	Bitwise logical AND operator normally used to test if both conditions are met	21.2
`AUTO`	Provide automatic line numbering	5.11
`CHR$ (j)`	Character with ASCII code `j`	7.6
`CLEAR`	Clears memory of variables except system/resident integer variables	4.6
`CLG`	CLear the Graphics display area	11.6
`CLS`	CLear the Screen text display area	7.1
`COLOUR 3`	Sets text colour to number 3	8.3
`COUNT`	Number of symbols printed since the last carriage return	18.4
`DEBUG`	Test for structural errors in the program and report any errors found	7.3
`DEL 30,90`	Delete lines 30 to 90 inclusive	5.10
`DELETE it`	Delete a file named `it`	14.7
`DIM set(4)`	Dimension an array with elements `set(0)` to `set(4)`	19.4
`DIV`	The integral quotient	18.8
`DRAW 30,400`	Draws a line to point 30,400	11.3
`EDIT`	Lists program with ‘invisible’ `EXEC`s and without indentation	13.5
`ELSE`	Alternative commands following `IF`	15.6
`END IF`	Indicates end of `IF` structure	15.5

Keyword	Brief outline	Section
`END PROC x`	Marks the end of the commands that are contained in procedure `x`	13.1
`EOD`	End of `DATA` indicator	17.3
`EOR`	Bitwise logical EOR operator normally used to test if only one condition is met	21.3
`FALSE`	0	16.3
`FOR j:=`	Indicates the beginning of a loop with loop variable `j`	6.1
`FREE`	Amount of available memory	3.12
`*FX`	Command to control machine effects	13.2
`GET`	ASCII code of next input symbol	9.5
`GET$`	String of next input symbol	9.6
`GCOL 0,2`	Sets graphics colour to number 2	11.5
`IF`	Start of decision	15.3
`IN`	Position of one string in another	4.3
`INKEY (n)`	Wait up to `n` hundredths of a second for input. Returns ASCII code	20.1
`INKEY$ (n)`	Wait up to `n` hundredths of a second for input. Returns string symbol	20.2
`INKEY (-99)`	Test if key with code 99 is being pressed	20.4
`INPUT a$`	Take in string and assign it to `a$`	9.1
`INT (j)`	Integral part of `j`	3.11
`LEN (a$)`	Number of characters in `a$`	4.2
`LIST`	Lists the program with indentation	5.4
`LOAD`	Load a program from current filing system	14.4, 14.6
`MOD`	Integral remainder after division	18.9
`MODE := 6`	Select mode 6	8.1
`MOVE 10,200`	Move graphics cursor to the point 10,200	11.4
`NEW`	Prepare for a new program	5.1
`NEXT j`	Marks the end of a `FOR` loop	6.1
`NOT j`	Inverts the binary digits of `j` Reverses `TRUE` and `FALSE`	21.6
`OF 5`	Indicates the length of a string or the elements of a string array	4.5
`OLD`	Attempt to recover last program	5.8

Keyword	Brief outline	Section
`OR`	Bitwise logical OR operator normally testing if one or both conditions are met	21.1
`ORD ("A")`	ASCII code for A	7.5
`PI`	Constant with value 3.141592653	3.13
`PLOT 69,x,y`	Plot the point with coordinates `x`, `y`	11.9
`POINT( x,y)`	Number of colour at position `x`, `y`	11.10
`POS`	Horizontal cursor position	18.1
`PRINT a$,b`	Prints values of `a$` and `b`	2.4
`PROC x`	Marks the beginning of the commands that form procedure `x`	13.1
`READ a$`	Assign next `DATA` item to `a$`	17.2
`RENUMBER`	Renumbers program in tens from 10	5.5
`REPEAT`	Start of a conditional loop	16.1
`RESTORE`	Resets pointer to start of `DATA`	17.4
`RESTORE aid`	Resets pointer after label `aid:`	17.4
`RND( a,b)`	Random whole number from `a` to `b`	10.1
`RUN`	Tests program structure and runs	5.3
`SAVE it`	Save program with title `it`	14.3, 14.5
`SGN (j)`	Sign of `j` (indicated by 1, 0, −1)	18.7
`STEP`	Size of increment in `FOR...` loop	6.4
`STR$ (5)`	The string equivalent for number 5	20.12
`TAB( a)`	Sets text cursor `a` places across screen	7.2
`TAB( a,d)`	Sets text cursor `a` across, `d` down	7.4
`THEN`	Marker for consequence of `IF`	15.3
`TIME`	Time since reset in 1/100 sec	10.5
`TRUE`	`-1`	16.3
`UNTIL`	Marker for end of `REPEAT` loop	16.1
`VAL`	Numeric value of evaluated string	20.11
`VDU`	Visual display unit code	11.12
`VPOS`	Vertical cursor position	18.1
`WIDTH`	Characters printed before carriage return and new line	18.5

22.1 The programming environment

Acornsoft COMAL has many features that are designed to help the programmer.

1. Each program line is checked for syntax as it is entered. If an error is found then the place at which it has been detected is indicated by an arrowhead.

2. A pre-run check tests whether there are any errors in the structure of the program. If any are found, then they are reported and the program does not run.

3. Multi-statement lines are not allowed. This makes programs slightly longer, but they are much clearer to read, understand and edit.

4. Keywords will list in upper case and user variables in lower case. This makes the actions of a program stand out.

5. Statements which are inside loops or other structures are indented when a program is listed. The overall structure of the program is then clearly visible.

6. Many of the formal details of syntax are entered automatically.

7. Spaces are stripped off the front and end of a line on entry and spaces after keywords are cut down to one.

8. Procedures may be called by their name without use of an `EXEC`. The procedure is defined between statements `PROC` and `END PROC` each of which is followed by the procedure name.

22.2 Points introduced in Part I

Keywords

After a keyword a space must usually be typed, unless a non-alphanumeric symbol such as `"` or `(` follows. The keywords `RND(` `TAB(` and `POINT(` include `(` as part of the keyword, and so no spaces should be used before the open bracket, for example: `TAB(5)` and not `TAB (5)`.

Statements

A null `FOR... NEXT...` loop in which the end is before the start is ignored. The loop will be bypassed.

The loop variable need not be stated in a `NEXT` statement. The variable will be deduced and inserted automatically.

`NULL` is the statement to do nothing.

A continuous sequence of program lines can be deleted using the `DEL` command. For example, `DEL 30,90` will remove lines 30 to 90 inclusive. By using `DEL` with a line number (or the line number and pressing the RETURN key) a single line can be removed.

The comment symbol `//` causes everything following it on that line to be ignored. It is the equivalent of `REM` in BASIC.

When several statements are to be carried out following an `IF` condition then a multi-line structure must be used. The structure is closed by `END IF` which must have a space between the words. The format is

```
IF... THEN
  Command 1
  Command 2
  etc
END IF
```

In both forms of `IF` the keyword `THEN` may be omitted on entry. It will be inserted automatically.

`ELSE`, which must occur on a line of its own, indicates the start of the alternative statements to be carried out should the `IF` condition fail. `ELSE` cannot be used with a single line `IF` statement.

Constants

`TRUE` has the value `-1` and `FALSE` has the value 0. In a test of a condition, any non-zero value is considered to represent true. Note, however, that real values are truncated to integers before being tested in conditions.

`PI` returns the value 3.141592653.

Variables

The proper symbol for assignment is `:=` though if the `:` is omitted it will be inserted automatically and shown in listings, eg `10 number = 2` becomes `10 number := 2`.

Any user variable appearing on the right of an assignment must have been introduced previously.

Apart from using keywords, any word or collection of letters, figures and the underline symbol may be used as a variable provided that it starts with a letter. A variable may contain a keyword anywhere as part of its name. The following are valid variable names:

```
k9  r2d2  me_and_u  printer_on  total  cost
```

An integer variable is indicated by `#`, eg `integer#`.

When a string is first assigned, 40 places of memory are reserved for it, unless a `DIM` statement allocates more or less memory.

`OF` is the keyword used when indicating the amount of memory to reserve for a string, for example:

```
DIM my$ OF 60
```

The maximum number of places reserved for a string may not be altered once set.

An attempt to assign more characters to a string than have been reserved, will produce an error message.

A quotation mark may be placed inside a string by using a double quotation mark `""` to indicate its position.

`MODE` is a system variable which can be assigned a value from 0 to 7 (although on the Acorn Electron assigning the value 7 will give mode 6). It may be used in the same way as any other variable. For example, the mode currently in use can be found from `PRINT MODE` and the mode may be changed by a statement such as `MODE := 6`.

`EOD`, which stands for End Of Data, is set to `TRUE` when all data has been read, but otherwise is set to `FALSE`.

System variables may only be assigned directly and not, for example, by using `INPUT`.

Labels

A label may be placed anywhere in a program on a line of its own. It is indicated by placing a colon immediately after any name which meets the requirements of a numeric variable name, for example: `here:` or `prices:`. The colon is not used when the label is indicated, for example: `RESTORE here`.

Arrays

Real, integer and string arrays are all allowed and may have up to eight dimensions. The array must be dimensioned before being used, for example:

```
DIM this$(3,4,5)
```

The bounds of a subscript may be set in the dimension statement and can include negative values.

Example: `DIM this$(1:3,-4:-2,4:5)`

If no lower bound is given then it is assumed to be 0.

Example: `DIM whole#(7)` gives `whole#(0)` to `whole#(7)`.

When a string array is dimensioned, 40 places are reserved for each element of the array, unless more or less are reserved by using `OF`. For example, `DIM a$(3,4) OF 5` would create a string array from `a$(0,0)` to `a$(3,4)` with five places reserved for each element.

Filenames

Quotations are not needed around filenames when loading, saving, deleting or running. The filename is considered to be the same irrespective of case. For example: `MYPROG` is the same filename as `myprog` or `MyPrOg`.

Data

Quotations are not needed around strings in data statements except when the string contains punctuation or is required to start or finish the `DATA` statement with spaces.

The data pointer can be reset to the first item of data by the command `RESTORE`.

`RESTORE` can be used with a label to restore the data pointer to the first element of data that follows the label, eg `RESTORE prices`.

Functions and operators

`IN` is a binary operator which returns the first position at which one string occurs inside another.

`ORD` is the function that returns the ASCII code of the first character of a string. It is the equivalent of `ASC( )` in BASIC, eg `PRINT ORD("A")` or `PRINT ORD(a$)`.

`RND( n)` where `n` is a whole number greater than 1 will produce a random whole number from 1 to `n`.

`RND( a,b)` where `a` and `b` are integers will produce a random whole number from `a` to `b` inclusive.

`RND( 1)` returns a random decimal value between 0 and 1.

`VAL` returns the value that the string argument would have produced if it had been a numerical expression.

`:+` is the command to increment the variable on the left of the sign by the amount on the right. For example, `price_of_fish :+ change` has the same effect as the longer statement `price_of_fish = price_of_fish + change`. It is particularly useful when long variable names are included.

`:+` may also be used with strings, eg if `a$ := "BRAIN"` then `a$ :+ "STORM"` makes `a$` become `"BRAINSTORM"`.

`:-` is the instruction to decrement the variable on the left of the sign by the amount on the right. For example, `total :- discount` has the same effect as the longer statement `total := total - discount`.

`:-` may not be used with strings.

System variables may be reassigned using `:+` or `:-`.

Care is needed with regard to the effect of `:+` and `:-`. See the section on assignment operators in Part III.

Syntax

Print separators (`,` or `'` or `;`) are required between all items in a `PRINT` list.

Example: `PRINT TAB(9);"here"`

or: `PRINT this$,that#'other`

If a prompt message is used in an `INPUT` statement then a colon is required between the message and the variable. No question mark is given if a prompt message is used.

23 Output to printers and PRINT USING

The results of a program or its listing can be output to the screen and, if required, to one of two printer connectors (known as ports).

Note that the Acorn Electron needs a Plus 1 expansion unit to interface to printers. Reference should be made to the *Acorn Electron Plus 1 User Guide* for details of using printers.

Characters may be sent to printers in two ways: serially, where the eight bits forming the one-byte character are sent one after another down a single wire, or in parallel, where the eight bits are sent simultaneously down eight wires.

The RS423 port, which is at the rear of the BBC Microcomputer, is used for serial output. The parallel (Centronics) printer port is situated underneath the BBC Microcomputer.

23.1 SELECT OUTPUT

The command `SELECT OUTPUT S` will cause any listing or printing to appear on the screen and to be passed through the serial output.

The command `SELECT OUTPUT P` will cause any listing or printing to appear on the screen and to be passed through the parallel output.

The command `SELECT OUTPUT D` will cause output to the display (screen) only.

Any symbols may follow `SELECT OUTPUT`. It is the initial symbol that determines the output. Any symbol other than `S` or `P` will direct output to the screen only.

23.2 Printing formats

There are two ways in which the format of numeric output may be controlled. The most versatile assigns a value to the system variable `ZONE` and the simplest to use demonstrates the format required as part of a `PRINT USING` command.

23.3 ZONE

`ZONE` is usually assigned a hexadecimal (base 16) value built up in three parts. For example `ZONE := &020309` sets the printing to take place according to format 02 (which is with a fixed number of decimal places) 03 indicates three decimal places and 09 indicates a zone width of nine characters. The `&` indicates a base 16 number.

23.4 ~

We can find the current value of `ZONE` by instructing the computer to print its value. First look for the symbol ˜ which is the shifted `^` key (next to the = sign). The ˜ sign (often pronounced 'twiddle') is placed in front of a numeric variable to convert the output from a decimal number to a string containing the hexadecimal representation. When using mode 7 ˜ is represented on the screen by the division symbol, ÷.

Enter

```
PRINT ~ZONE
```

23.5 PRINT USING

The alternative way of indicating the printing format is to create a string example of the format required and instruct the computer to print to that format.

The format shown in the string is the one in which the number will be printed.

Example:

```
PRINT USING "###.##":number
```

The value of number will be rounded to two decimal places before it is printed with three places before the decimal point and two after it.

Strings may be printed as part of a `PRINT USING` command.

Example:

```
PRINT USING "#####.##":"Price is ";price
```

which will print the value of price with five places before the decimal point and two after it.

The string must contain exactly one decimal point if the format is to be correctly defined. The `#` symbol is the standard symbol for use inside the string, though any other symbols would do.

Example:

```
PRINT USING "1234.123":length
```

would have four places before the decimal point and three after it. The length of the string and the position of the decimal point are the only significant features.

If a number is too large to fit into the format then it will be printed with the extra places before the decimal point, but with the required number of decimal places.

Allow for the minus sign

If it is anticipated that a value will be negative then an extra place should be allowed for the - sign. For example -99.99 is the largest negative number that could be positioned correctly with `PRINT USING "###.##"`. `PRINT USING` is particularly useful for printing columns of figures correctly ranged.

Example:

```
PRINT USING "#####.###":0.3'34.5'345.6'-345.6789
```

produces

```
    0.300
   34.500
  345.600
 -345.679
```

24 WHILE

`WHILE` is used as the introductory statement for a loop.

24.1 The possibly not loop

The loops that we have met so far are:

– `FOR... NEXT...` which is carried out a precise number of times;

– `REPEAT UNTIL...` which is carried out at least once, but an unspecified number of times dependent on the condition given after `UNTIL`.

The `WHILE` loop is used where we want to carry out a set of statements an undetermined number of times, and possibly not at all. A test is made before the statements are encountered.

The format of the `WHILE` loop is

```
WHILE ... DO
  Command1
  Command2
  etc
END WHILE
```

Following the `WHILE` a condition is given. If the condition is satisfied (ie it has a true result) then the statements inside the loop are carried out. The loop continues until the condition becomes false.

The keyword `DO` may be omitted on entry. It will be inserted automatically.

The statements inside the loop will be indented by two places.

The `END WHILE` must contain a space.

24.2 Examples

```
10 WHILE NOT EOD
20   READ text$
30   PRINT text$
40 END WHILE
50 PRINT "That's all folks."
60 DATA What's up Doc ?
```

```
210 INPUT "Enter a number of pence ":amount
220 INPUT "Enter cost of an apple in pence ":cost
230 number := 0
240 WHILE amount >= cost DO
250   amount :- cost
260   number :+ 1
270 END WHILE
280 PRINT "You have enough for ";
290 PRINT ;number;" apples"
```

```
810 space_bar = -99
820 PRINT "Press Space Bar to Continue"
830 WHILE NOT INKEY (space_bar) DO
840   NULL // pause until Space Bar pressed
850 END WHILE
```

```
900 DIM a(1:1000)
910 shift := -1
920 FOR j := 1 TO 1000
930   PRINT j,a(j)
940   WHILE INKEY (shift) DO
950     NULL // pause if shift pressed
960   END WHILE
970 NEXT j
```

25 CASE

`CASE` is the introductory statement for a decision. It is best used when a whole series of alternatives has to be considered and use of the `IF` command would be untidy. The different `CASE`s are tested and then acted on.

25.1 OF

`CASE` must be followed by a variable or an expression which can be evaluated and should be followed by the keyword `OF`. The value is then compared with a set of alternatives. The format of the command is shown by this example. If `OF` is omitted on entry then it will be inserted automatically.

```
 10 MODE := 6
 20 PRINT''"I am thinking of an integer 1-99"
 30 PRINT''"Try to guess it"''
 40 actual := RND( 99)
 50 REPEAT
 60   INPUT "Your guess > ":guess
 70   PRINT TAB( 18);
 80   difference := ABS(actual-guess)
 90   CASE difference OF // OF need not be typed
100   WHEN 0 // a single value
110     PRINT "Exactly"
120   WHEN 1,2 // a list of alternative values
130     PRINT "White Hot"
140   WHEN < 5 // a condition
150     PRINT "Red Hot"
160   WHEN < 10
170     PRINT "Warm"
180   WHEN < 20
190     PRINT "Cool"
200   WHEN < 40
210     PRINT "Freezing"
220   OTHERWISE
230     PRINT "Get thermal underwear"
240   END CASE
250 UNTIL difference = 0
```

The `CASE`s `OF` the value of `difference` are considered.

25.2 The CASE format

There must be a space between `END` and `CASE`.

The statements inside `WHEN` and `OTHERWISE` are indented by two places.

Following the `WHEN` there may be:

- A set of precise values, separated by commas.
  Example: `WHEN 0,1,2` or `WHEN "A","B","C"`
- A single expression. Example: `WHEN 2*x-3`
- A condition. Example: `WHEN IN "YyNn"`

or

- A combination of precise values and conditions.

  Example:

  ```
  WHEN 5, < -5
  ```
  or
  ```
  WHEN >5, <-5, 99
  ```

Only one CASE succeeds

The cases are considered in sequence. `WHEN` one case is satisfied, the statements following it are carried out and the program then jumps to the end of the `CASE` structure. Only one set of statements is ever carried out.

`OTHERWISE` is a general trap for the action when none of the conditions is satisfied. There need not be an `OTHERWISE` in a `CASE` structure, but if no action is required then this can be made clear by use of the `NULL` statement.

26 Short forms

Where only one instruction has to be carried out there are single line forms of the `FOR... NEXT...` and the `WHILE...` loops.

No indenting occurs with any short form.

26.1 Short form FOR

In the single line `FOR` statement the action is placed after the `TO` or `STEP` values and the command `DO`. Example:

```
FOR j := 1 TO 100 DO array(j) := RND( 999)
```

The single line version of the `FOR` loop may be given as a direct command and the `DO` need not normally be included.

```
FOR j := 1 TO 100 array(j) := j
```

In a program statement the `DO` can also normally be left out on entry, its position will be deduced and it will be inserted automatically.

```
  100 FOR k := 20 TO 15 STEP -1 PRINT k
```

will list as

```
  100 FOR k := 20 TO 15 STEP -1 DO PRINT k
```

The only exceptions are with the use of `?` and `!` where there could be possible confusion. Example:

```
FOR i = 1 TO n DO ? node(i) = 0
```

The `DO` is necessary because `n ? node(i) = 0` is a valid expression.

26.2 Short form WHILE

In the single line `WHILE` statement the action is placed after the condition and the keyword `DO`.

```
  100 WHILE INKEY(-99) = 0 DO PRINT"."; // Space Bar
```

The single line version of the `WHILE...` loop may be given as a direct command and the `DO` need not be included because it will be inserted automatically.

```
WHILE INKEY(-99) = 0 PRINT ".";
```

If there is an error in the command then the position of the `DO` can be seen when the error message is reported.

In a program statement the `DO` can also be left out on entry, its position will be deduced and it will be inserted automatically.

```
  100 WHILE n <= 0 INPUT "Enter a positive number":n
```

will list as

```
  100 WHILE n <= 0 DO INPUT "Enter a positive number":n
```

26.3 Exceptions

Short form `FOR`, `IF` or `WHILE` statements cannot be combined so that one is the result of the other. For example the following command is `Not allowed`.

```
IF reply = 5 THEN FOR j := 1 TO 10 DO a(j) := j
                      ^
```

27 Multiple assignments and dimensions

Whilst several instructions cannot be placed on the same line, it is possible to have more than one assignment or dimension on a program line.

27.1 Assignments

The assignments must be separated by semi-colons, for example:

```
100 red := 1; yellow := 2; white := 3 // assign colours
110 r$ := "Red"; a$ := "Amber"; g$ := "Green"
120 temp := this; this := that; that := temp // swap
```

A mixture of variable types may be assigned on the same line, for example:

```
130 a# := 1; b$ := "two"; c := 3; MODE := 4
```

and multi-assignments may be the result of a short form command, for example:

```
140 FOR j := 1 TO 20 DO a(j) := j; b(j) := j*j
```

27.2 Dimensions

When placing several dimension statements on a line they must be separated by commas, but `DIM` need only be given once, for example:

```
100 DIM this$ OF 5, that$ OF 10, other$ OF 200
```

```
100 DIM array(5,4), name$(10) OF 25, integer#(3,3)
```

28 ELIF

ELIF is the statement which has the same effect as ELSE combined with IF. It can only be used inside an IF structure and is designed to avoid further nested IF conditions inside the IF loop.

28.1 Format for ELIF

The format for using ELIF inside an IF structure is

```
IF ......... THEN
   ..........
   ..........
ELIF ....... THEN
   ..........
   ..........
ELIF ....... THEN
   ..........
   ..........
ELSE
   ..........
END IF
```

There may be more than one ELIF inside an IF structure.

The general trap, using ELSE, may be omitted if required, though for clarity a NULL statement may be used to indicate the action of the program.

The first test satisfied

Only one of the sets of statements is carried out. As soon as a test is satisfied, the statements following it are carried out. The program then passes on to the line that follows the END IF.

28.2 Demonstration program

```
   10 INPUT "Enter the number of a year > ":year
   20 //
   30 IF year MOD 4 <> 0 THEN
   40   //year doesn't divide by 4
   50   leap := FALSE
   60 ELIF year MOD 100 <> 0 THEN
   70   //year/4, not century
```

```
 80   leap := TRUE
 90 ELIF year MOD 400 <> 0 THEN
100   //year/100 not by 400
110   leap := FALSE
120 ELSE
130   leap := TRUE
140 END IF
150 //
160 PRINT year;" is";
170 IF NOT leap THEN PRINT " not";
180 PRINT " a leap year"
```

Each ELIF condition is only reached if all previous conditions are not met.

29 Substrings

A substring is any part of a string from a null string up to the complete string itself.

In Acornsoft COMAL operations may be performed using a substring, including the replacement of that substring using an assignment.

29.1 Selection of a substring

A substring is selected by indicating the number of the place where we wish to start and the number of the place where we wish to end the substring.

`a$(start:finish)` specifies the substring of `a$` beginning at the place given by the value of `start` and continuing up to and including the place given by `finish`.

Enter

```
town$ := "EDINBURGH"
```

and

```
PRINT town$(5:8)
```

The substring starts at the fifth letter from the left, which is `B` and continues up to and including the eighth letter from the left.

29.2 Substring specifiers

The numbers used in the bracket are called ‘substring specifiers’ because they specify or define the substring.

Those readers familiar with `LEFT$`, `RIGHT$` and `MID$`, from BASIC may be surprised at this method of defining substrings. However, a whole range of powerful commands is defined using this simple notation.

Left substrings

Enter

```
PRINT town$(1:5)
```

The first five letters of `town$` are printed.

The command can be simplified by leaving out the 1 if a leftmost substring is being specified.

Enter

```
PRINT town$(:5)
```

Enter

```
FOR j := 1 TO 9 DO PRINT town$(:j)
```

Right substrings

Enter

```
PRINT town$(6:9)
```

Whilst this has produced the rightmost set of four letters, the 'substring specifiers' are not so easy to find. There are two ways of easing the problem.

Enter

```
PRINT town$(6:)
```

If the right substring specifier is omitted then it is assumed that the substring continues to the end of the string.

Negative substring specifiers

Enter

```
PRINT town$(-4:)
```

A negative substring is taken to mean a substring measured from the right of the string.

Enter

```
FOR j := 1 TO 9 DO PRINT town$(-j:)
```

The use of substring specifiers is not restricted to variables.

Enter

```
PRINT "ABCDEFGHIJKL"(3:-3)
```

producing a string starting at the third element on the left and going up to the third element from the right.

29.3 Precedence

Substring specifiers have a higher precedence than string addition. For example:

```
PRINT "DUN" + "DEER"(1:3)
```

will produce `DUNDEE`.

Brackets have the highest precedence so

```
PRINT ("LIME" + "RICKMANSWORTH")(:8)
```

will produce `LIMERICK`.

```
PRINT ("      " + a$)(-6:)
```

will print `a$` on the right of a six-element string with spaces padding out any gaps on the left.

An example for the more advanced programmer is

```
PRINT (label$ + "       ")(:7);("000" + ~x)(-4:)
```

which might be used to tabulate labels and a hexadecimal address.

29.4 Single element substrings

It is possible to specify a particular single symbol with a command such as `PRINT town$(5:5)`. However, there is an alternative.

Leave a space between the string symbol and the bracket as you enter

```
PRINT town$ (5)
```

The space is necessary because without it `town$(5)` would have the appearance of an array element.

`town$ (5)` is the single symbol which is the fifth from the left in `town$`.

`town$ (-5)` is the single symbol which is the fifth from the right in `town$`.

Enter the direct command

```
FOR j := 1 TO 9 DO PRINT "CAMBRIDGE" (j)
```

and then enter

```
FOR j := 1 TO 9 DO PRINT TAB(j);"CAMBRIDGE" (j)
```

Enter the following program.

```
   10 INPUT "Enter a possible palindrome > ":phrase$
   20 letter := 1 ; middle := LEN (phrase$) / 2
   30 WHILE letter <= middle AND phrase$ (letter) =
phrase$ (-letter) DO letter :+ 1
   40 PRINT 'phrase$''"is ";
   50 IF letter <= middle THEN PRINT "not ";
   60 PRINT "a palindrome"
```

Run the program and try entering

```
MADAM
MADam
maDam
ABLE WAS I ERE I SAW ELBA
MADAM I'M ADAM
```

29.5 Elements of a string array

Substring specifiers can be used with an array element. For example:

```
array$(2,3) = "abcdefghijkl"
PRINT array$(2,3)(4:5)
```

will produce `de`.

When a single symbol of an array element is required then a space is not needed between the bracketed subscripts and the substring specifier bracket. For example:

```
array$(2,3) := "abcdefghijkl"
PRINT array$(2,3)(3)
```
will produce `c`

29.6 Replacing substrings

The substring notation can be used in an assignment of a new value to part of an existing string.

`a$(start:finish) := replace$` uses `replace$` to replace the symbols of `a$` from the position given by the value of `start` to the place given by the value of `finish`, truncating `replace$`, if necessary.

Enter

```
town$ := "AMSTERDAM"
```

and

```
town$(:3) := "ROT"
```

and

```
PRINT town$
```

The first three symbols of `town$` have been replaced.

Enter

```
town$(4:7) := "HERH"
```

and

```
PRINT town$
```

The symbols from 4 to 7 have been replaced.

Short measure

Now enter

```
town$ := "PORTSMOUTH"
```

and

```
PRINT LEN (town$)
```

Now enter

```
town$(:5) := "PLY"
```

and

```
PRINT town$
```

and

```
PRINT LEN (town$)
```

In this case five symbols are replaced with only three. The resulting string is closed up to leave no gap and the length of `town$` is accordingly decreased to eight characters.

Excess

Now enter

```
town$(4:) := "NLIMON"
```

and

```
PRINT town$
```

and

```
PRINT LEN (town$)
```

The length of a string may not be increased by using a substring assignment. The leftmost part of the inserted string is used until the available spaces are filled.

Summary

In general, a substring assignment may not increase the length of a string, but a string may decrease when there is a deficiency in the assignment.

Substring assignment is extremely versatile and may be used with string expressions.

Example:

```
a$(:m) := (b$(x,y)(n:p) + c$(z)(-q:))(:n)
```

is a valid assignment.

30 FUNCtions

A function has many similarities with a procedure. It represents a diversion of the flow of the program to a set of statements elsewhere. The difference is that the purpose of a function is to provide one value and to return that value to the original part of the program. The function may be string or numeric depending on what type of value you wish to return.

The function is called by giving its name. It is defined elsewhere in a program by the statement `FUNC` followed by the name. The statements that perform the function then follow and an `END FUNC` statement completes the function. If the name is omitted on entry it will be inserted automatically. The statement `RETURN` is used to indicate the value to be 'returned'. Alternative conditions may require the use of more than one `RETURN` statement.

30.1 Example programs

```
 10 // Function example
 20 MODE := 6
 30 DIM a(1:100)
 40 FOR go := 1 TO 10 DO
 50   PRINT "Filling the array."
 60   FOR j := 1 TO 100 DO a(j) := RND( 1000)
 70   PRINT "Greatest was ";highest
 80 NEXT go
 90 //
100 FUNC highest
110   max := 0
120   FOR j := 1 TO 100 DO
130   IF a(j) > max THEN max := a(j)
140   NEXT j
150   RETURN max
160 END FUNC highest
```

Run the program.

List the program and examine line 70.

The function `highest` is called and a value is returned for printing.

You will have to provide the missing procedures if you want to run this program.

```
 10 REPEAT
 20   make_up_question
 30   ask_question
 40   take_in_answer
 50 UNTIL more$ = "N" // until TRUE
 60 //
100 FUNC more$
110   PRINT '"Another question ? Y/N > ";
120   REPEAT
130     reply$ := GET$
140   UNTIL reply$ IN "yYnN"
150   //
160   IF reply$ IN "yY" THEN
170     RETURN "Y" // loop again
180   ELSE
190     RETURN "N" // end the loop
200   END IF
210 END FUNC more$
```

`FUNC more$` returns either `Y` or `N` (which brings the `REPEAT` loop to an end).

30.2 Use of FUNCtions

A similar program could have used a procedure instead of a function. A variable `more$` could have been assigned inside the procedure. This example of the use of a function is slightly artificial. Functions are most useful when they have particular values (the arguments) to work on. The arguments are passed to the function as parameters. The use of parameters is dealt with in the next chapter.

Program lines within a `FUNC` definition are ignored if they are encountered whilst following through the normal flow of the program.

A major advantage of a function is that the same process can be used in different parts of a program without duplicating statements.

31 Parameters

Procedures and functions may have parameters allocated to them. A parameter is a value that the procedure or function uses to produce its result. If we take as an example the calculation of an average of two numbers, then the result depends on which particular numbers are chosen.

However, we know what to do with any two numbers which we are given. Similarly, the function can define what to do with two variables. These variables are the parameters. They are shown in brackets after the function name and are separated by commas.

Enter the following program

```
100 //
110 FUNC average(x,y) // formal parameters
120   sum := x + y
130   RETURN sum / 2
140 END FUNC average
```

The `x` and `y` are called the 'formal parameters'.

31.1 Calling a FUNCtion with parameters

When we want to use the function, we must provide two values for it to work on. We can use numbers immediately.

Enter

```
 10 PRINT average(5,6)
```

List and run the program.

The function is called in line 10, `x` is assigned the value 5 and `y` is assigned the value 6. The average of `x` and `y` is calculated and `RETURN`ed so as to be available for printing.

The 5 and 6 are called the 'actual parameters'.

Formal parameters are variables

The formal parameters used in the `FUNC` statement must be variables. `FUNC average(x,1)` called by `average(5,6)` could not give the value of 6 to a number 1. The line input syntax check would find this error.

Actual parameters may be variables

Change and add lines 10 and 20 to produce:

```
  10 a := 6; b := 5
  20 PRINT average(a,b) // actual parameters
 100 //
 110 FUNC average(x,y)
 120   sum := x + y
 130   RETURN sum / 2
 140 END FUNC average
```

and run the program.

The function is called in line 20, x is assigned the value of a, y is assigned the value of b. The average of x and y (and therefore a and b) is calculated and RETURNed for printing.

31.2 Local variables

Enter the direct command

```
PRINT x,y
```

The formal parameters x and y are 'local' variables. When the function is called, the variables x and y are created. When the function RETURNs a value, those local variables are no longer available.

31.3 Global variables

The resident integer variables a# to z# and the system variables (eg MODE) are called global variables because they are available for use or assignment both inside and outside a procedure or function. They may not be used as formal parameters. For example: PROC test(a#) would produce a syntax error at line entry. Other integer variables such as aa# and zz# are acceptable as formal parameters.

31.4 Main block variables

Any variable (other than a global variable), which is created in an open situation is stored (with its value) in a 'main block'. These main block variables are available automatically inside and outside any open procedure or function.

Add

```
  30 PRINT "sum is ";sum
```

List and run.

The variable sum, which was created inside the function, is available outside the function.

Change line 30 to:

```
  30 PRINT c
```

and add:

```
 115 c := a
```

List and run the program.

The variable a, which was assigned outside the function is available inside the function. sum, a and c are examples of main block variables.

31.5 CLOSED PROCedures and FUNCtions

Change 110 to

```
 110 FUNC average(x,y) CLOSED
```

producing

```
  10 a := 6; b := 5
  20 PRINT average(a,b) // actual parameters
  30 PRINT c
 100 //
 110 FUNC average(x,y) CLOSED
 115   c := a
 120   sum := x + y
 130   RETURN sum / 2
 140 END FUNC average
```

and run the program.

a cannot be found at line 115 because it does not occur in the ‘local variable block’.

When a procedure is CLOSED, only the local variables and global variables (a# to z# and the system variables) are automatically available inside the procedure.

31.6 IMPORT

Main block variables can be used in a CLOSED function or procedure following an IMPORT statement.

Enter

```
 112 IMPORT a
```

List and run the program.

The value of c is not available at line 30. When the program was run, c was stored in the local variable block because it was created in a CLOSED environment. Any variables which are first assigned inside a CLOSED function or procedure are not available outside it.

Change 30 to:

```
 30 PRINT c#
```

and 115 to:

```
115 c# := a
```

List and run the program.

c# is a global variable so it is available outside the closed procedure.

The only main block variables that may be accessed from inside a CLOSED function or procedure are those that are passed to it as parameters or IMPORTed.

Change 30 to

```
 30 print a
```

and 115 to

```
115 a :+ 1
```

producing

```
 10 a := 6; b := 5
 20 PRINT average(a,b)
 30 PRINT a
100 //
110 FUNC average(x,y) CLOSED
112   IMPORT a
115   a :+ 1
120   sum := x + y
130   RETURN sum / 2
140 END FUNC average
```

Run the program.

The average of 5.5 is printed out, followed by the value 7 which is the new value of a. The main block value of an imported variable is affected by anything that happens inside a CLOSED function.

Several variables or arrays can be brought in with a command such as:

```
IMPORT a,b$,c#,d(,)
```

where `d(,)` indicates a two dimensional array.

31.7 Using the same actual and formal parameter

Enter `new` and the following program.

```
   10 x := 5
   20 print test(x)
   30 print x
  100 //
  110 func test(x)
  120 x :+ 10
  130 return x
  140 end func test
```

Great care must be taken if the same variable is used as the formal parameter in a function and also as the actual parameter outside the function. Although the same symbol is used, it refers to distinct variables.

Run the program.

This result needs some explanation.

The initial main block value of `x` is 5.

The value, 5, is given to the local parameter of the function, which happens to have the same name as one of the main block variables. The value is stored in the local variable block and is subsequently increased to 15 and `RETURN`ed. The function has now been completed and so the original main block variables are now made available.

When `x` is required in line 30, it is the original `x` and not the value used in the function.

If you wish to avoid any confusion as to the value of a variable, then use different names for the actual and formal parameters.

31.8 REFerenced parameters

Should you wish to use the same variable inside a procedure (or function) as in the main part of a program then there is a way to do it, provided that you realise that the value outside the procedure or function may be changed by the statements inside.

When the formal parameter is listed in brackets after the `PROC` or `FUNC` name, the keyword `REF` must be placed in front of the parameter. The variable is passed by `REF`erence.

Enter `new` and the following

```
  10 a := 3; b := 4
  20 print a,b
  30 swap(a,b)
  40 print a,b
 100 //
 110 proc swap(ref x,ref y) closed
 120 temp := x; x := y; y := temp
 130 end proc swap
```

Run the program.

When the procedure is called in line 30, `x` is linked to the value of `a` and `y` is linked to the value of `b`.

Line 120 carries out an exchange of values.

At the end of the procedure, `a` and `b` have the values to which `x` and `y` were linked. `x`, `y` and `temp` are no longer available and the program continues.

It is important to note that, if a formal parameter is `REF`erenced, the actual parameter must be a variable. For example: `50 swap(3,4)` could not be used as a call to the procedure `PROC swap(REF x, REF y)`. The 3 and 4 are not variables and do not have a position in the main block of variables so there is no location to which the `x` and `y` could be referred. Such an error would be discovered as the program was running.

The correct method is as shown above where the values 3 and 4 are assigned to variables `a` and `b` then the procedure is called with `swap(a,b)`.

31.9 String parameters

String variables and string expressions may be passed to procedures and functions in the same way as numeric variables, though with two added points.

When a string variable is declared as a formal parameter, it is assumed that a string dimension of 40 is required unless the length is declared by using `OF`.

Example:

```
PROC string(name$ OF 10,number)
```

The only variables that may be used for the length of a string parameter are resident integer variables and the system variables. The reason is that they are global variables and will be available inside the closed dimensioning process.

Any other variable would not be found when the program was run.

Enter `new` and try this string parameter example.

```
100 REPEAT
110   INPUT "A word of up to 10 letters > ":word$
120 UNTIL LEN(word$) < 11
130 //
140 REPEAT
150   INPUT "Number of copies > ":n
160 UNTIL n>0
170 //
180 string(word$,n)
190 //
200 PROC string(name$ OF 10,number)
210   FOR go := 1 TO number
220     PRINT name$;
230   NEXT go
240   PRINT
250 END PROC string
```

If string variables are passed by `REF`erence then the length must *not* be dimensioned in the `PROC` or `FUNC` declaration, for example: `PROC reverse(REF a$,REF b$)`. Any use of a `REF`erenced string variable will apply to the value held in the main block of variables where its length has been established so that `OF` is not needed (or allowed). Any attempt to use `OF` in this situation will be rejected at line entry.

31.10 Passing arrays by REFerence

A single element of an array may be used as a parameter in the normal way. However, if it is required to use the whole array inside the procedure or function then the array must be passed by `REF`erence.

Links must be established between the procedure or function and the original array. An indication is needed of the number of dimensions that the array possesses. This is done by showing the number of commas used in dimensioning the original array, for example:

`REF line()`	would refer to a 1 dimensional array
`REF table(,)`	would refer to a 2 dimensional array
`REF block(,,)`	would refer to a 3 dimensional array

The same method is used to indicate the type of array in an `IMPORT` statement.

The following procedure could be used to carry out a scaling of an array.

```
2000 PROC scale(REF matrix(,),factor,dimension)
2010   FOR j := 1 TO dimension DO
2020     FOR k := 1 TO dimension DO
2030       matrix(j,k) := matrix(j,k) * factor
2040     NEXT k
2050   NEXT j
2060 END PROC scale
```

The procedure could be called with a statement such as

```
 150 scale(mat(,),3,6)
```

which would scale the two dimensional 6 by 6 array called `mat` by a factor of 3.

32 File handling

A file is a collection of information which is usually kept together for some common purpose. In an office, the files may be in the form of wallets, boxes, microfilms or computer storage devices.

Computer storage has many advantages including: speed of access, ease of sorting and compactness.

COMAL has statements which make storage of information quite an easy process.

There are two sorts of file that we can use.

32.1 Sequential files

One type of file is like a wallet of papers. Every time another piece of paper is added to the file, it is added to the back of the wallet. When you want to find something in the wallet, you have to go through all the papers until you find the one you want. This type of file is called ‘sequential’. The information is added to the file in sequence and has to be taken out in sequence.

32.2 Random (or direct) access files

The other type of file requires more organisation. It is as if each set of information is placed on identical pieces of paper. Every piece of paper is numbered, so that if we know the number of the item then we can find it immediately, without having to sort through. The position of each item is fixed by its number.

This type of file is called ‘random access’ or ‘direct access’ because the records can be found directly without having to read through every preceding record. The word random is slightly misleading because the records are not kept at random, but may be accessed in a random order.

A ‘sequential’ file can be kept on tape or on disc. A ‘random access’ file can only be kept on disc. Each has its advantages. Sequential files are simpler, and are economical in the space they take. Direct (random) files are often faster to use and are suitable for continuous processing.

32.3 Filenames

Every file has a name, which on disc can be up to seven letters and on cassette can be up to ten letters. The symbols should not include a space. On disc a filename should not include punctuation. See the *Disc Filing System User Guide* for details. The filename need not have quotation marks around it, unless it also happens to be a keyword.

32.4 File numbers

When a file is being used it has a number (from 0 to 5) associated with it. We use the number to refer to the file rather than its name. There may be up to five files in use at the same time.

32.5 Closing a file

The fact that we start the section on files by indicating how to close a file should warn you that it is vital always to close a file after using it. If you do not, then unwanted information may be placed in the wrong file and there is a danger that you will do irreversible damage to existing disc programs.

The command `CLOSE` will close all files that are in use. Individual files can be closed with a command such as `CLOSE FILE 1`.

32.6 Opening a sequential file

The file may be opened to store information in sequence from the beginning (`WRITE`), to add on to the end of the file (`APPEND`) or to obtain information from the file (`READ`).

The open sequential file statement has three parts.

1. The number of the file (0 to 5).
2. The filename.
3. The purpose for which the file is opened: `READ` or `APPEND` or `WRITE`.

A command such as:

```
OPEN FILE 1, Tapes, WRITE
```

will prepare to write information into a file with the name `Tapes` starting at the beginning of the file. If we are using a disc then a check is made as to whether such a file already exists. If not, then a new filename will be created on the disc directory and a new file with that name will be opened on the disc. Note that any existing file of the same name will be overwritten unless it is locked, in which case a message to that effect will be generated. Spaces are required after the keywords `OPEN` and `FILE`.

A command such as:

```
OPEN FILE 2, Discs, READ
```

will prepare to read information from the file with the name `Discs` starting at the beginning of the file.

A command such as:

```
OPEN FILE 3, Books, APPEND
```

can only be used with a disc system to add information to the file `Books` continuing from the last item entered in the file.

32.7 Placing information into the file

When you want to place information into the file the command `PRINT FILE` may be used. It is followed by the number used in the `OPEN FILE` command (not the filename), a colon and then the item, for example:

```
PRINT FILE 4:name$
```

If you want to place several items into the file at the same time then the items are separated by commas, for example:

```
PRINT FILE 4:name$,address$,town$,code$
```

Variables, constants and expressions can be used in the list of items to be stored, for example:

```
PRINT FILE 4:name$,2*x,7,"Hi"
```

`WRITE FILE` can be used instead of `PRINT FILE` but only when storing the values of variables and not constants and expressions, for example:

```
WRITE FILE 5:a$,b,c#
```

Enter this program which creates the file `Randcol` which contains 50 sets of three numbers:

```
10 OPEN FILE 1, Randcol, WRITE
20 FOR set := 1 TO 50 DO
30   c := RND( 7)
40   x := RND( 1279)
50   y := RND( 1023)
60   PRINT c,x,y // screen display only
70   PRINT FILE 1:c,x,y // into file
80 NEXT set
90 CLOSE FILE 1 // MUST close file
```

and save it under the name `COL_IN` ready for later use.

If you are going to use this program with cassette, then prepare a blank section of cassette tape in the recorder. When line 10 is reached a prompt to press `RECORD then RETURN` will be given in just the same way as with saving a program. There will be pauses between the blocks as they are saved. A further prompt will appear when the program is complete.

Run the program.

32.8 Retrieving information from a file

Whenever we want to retrieve information from the file, it must be opened (and closed) in the same way as for storing, except that we use the keyword `READ` at the end of the `OPEN FILE` statement. The keywords `INPUT FILE` are used to retrieve information from the file. They must be followed by the number of the file, a colon and then the name of the variable to which the information will be assigned, for example:

```
INPUT FILE 1:name$
```

If several pieces of information are to be taken in at one time, then commas must separate the variables to which the information will be assigned, for example:

```
INPUT FILE 1:name$,address$,town$,postcode$
```

`READ FILE` has exactly the same effect as `INPUT FILE`.

Here is the matching program that will retrieve the information saved on file and perform a multicolour drawing sequence.

```
10 MODE := 2 // or 5 if BBC Model A
20 MOVE 640,512 // Middle of screen
30 OPEN FILE 1, Randcol, READ
40 FOR set := 1 TO 50 DO
50   READ FILE 1:c,x,y
60   GCOL 0,c // Graphics colour
70   DRAW x,y
80 NEXT set
90 CLOSE FILE 1 // MUST close file
```

If you are using a cassette recorder, rewind the tape until you have wound back to the beginning of the file `Randcol`.

Run the program (and press PLAY if you are using tape).

32.9 EOF

The function `EOF` (End Of File) is available. The function must be followed by the number of a file. If there is no more information available in the file, then `EOF` returns `TRUE`. If the end of the file has not been reached, then `EOF` returns `FALSE`.

Change lines 40 and 80 to

```
40 WHILE NOT EOF 1
80 END WHILE
```

(Rewind the cassette, if appropriate) and run again. Save this program under the name `COL_OUT`.

32.10 APPENDing to a sequential file

On disc it is possible to add further information on to the end of an existing file using the keyword `APPEND` provided that there is space available at the appropriate point on the disc.

`APPEND` cannot be used for cassette sequential files. The tape cannot be rewound under computer control for the additional information to be inserted on top of the end of file marker. If you need to extend a cassette file then dimension enough space in the computer, load the existing data from the tape, add the extra items, then save the whole file again.

If you can foresee the requirement to extend a sequential file, then it would be wise to create the file with extra sets of dummy information. The first items in the file could be the number of actual records and the maximum possible number that may be stored in the file. The number of actual records can be changed as more true information replaces the spurious. This approach is effective, but spoils the idea of appending. As an example of appending enter the following program, either directly or by loading the program `COL_IN` and changing line 10 to use `APPEND`.

```
10 OPEN FILE 1, Randcol, APPEND
20 FOR set := 1 TO 50 DO
30   c := RND( 7)
40   x := RND( 1279)
50   y := RND( 1023)
60   PRINT c,x,y
70   PRINT FILE 1:c,x,y
80 NEXT set
90 CLOSE FILE 1
```

Run the program.

(If you receive a `Can't extend` message then there is insufficient additional space available at this point of the disc. Enter `CLOSE` to close the file.)

Load the program `COL_OUT` and run it to test the effect of appending. If `COL_IN` has just generated a `Can't extend` error then it is very likely that `Randcol` will end with an incomplete set of data, so line 70 of `COL_OUT` will generate an `EOF` error when that point is reached. Again, `CLOSE` should be entered to close the file.

32.11 Random access files

When creating a random access file, more organisation is needed. Each set of information (or 'record') may contain several sections (or 'fields'), for example:

reference number, surname, initials, amount due

All the records in a random access file need to be the same length so that a particular numbered record can be found directly. The length of record needed depends on what type of fields it is to contain. There are three such field types: integer, real, and string.

32.12 Record length required

An integer requires five bytes.

Any other real number requires six bytes.

Each string requires two bytes more than the length of the string (one byte to indicate that it is a string and one byte to give the length of the string).

As an example we can work out the length required for a record consisting of:

– a reference number, which may be up to 8 figures;
– a surname, which may be up to 20 characters;
– a set of initials, which may be up to 5 characters;
– an amount due, which may be up to 10,000.00.

The reference number is an integer needing 5 bytes; the surname is a string requiring 20 + 2 bytes; the initials are a string requiring 5 + 2 bytes; the amount due is a real number requiring 6 bytes.

The total record length would be 5 + 22 + 7 + 6 = 40.

When we open the random access file we must declare the record length to be 40.

32.13 Opening random access files

The keywords `OPEN FILE` must be followed by a file number (from 0 to 5), a comma, then the filename, a comma, the word `RANDOM` and the number or logical expression for the record length. For example:

```
OPEN FILE 1, Invoice, RANDOM 40
```

Read or write

Once a random access file is opened, it may be used to read or write, unless the keywords `READ ONLY` (with exactly one space) are added to the end of the open file command in order to allow access to locked files. For example:

```
OPEN FILE 2, Payment, RANDOM 40 READ ONLY
```

No comma is needed (or accepted) between the record length and the `READ ONLY` command.

32.14 Placing data into a random access file

The keywords `PRINT FILE` and the file number are given. A comma, the record number and a colon then follow. Commas separate the items to be placed into the record. For example:

```
PRINT FILE 1,record#:refer#,name$,initial$,amount
```

`PRINT FILE` allows use of variables and expressions for the record number and the items. `WRITE FILE` may be used if the values of variables, only, are being placed in the file rather than expressions.

In this example we shall write some data into a random access file for use later. Enter

```
   10 DATA 12345678,Jones,ABC,111.11
   20 DATA 13579135,Smith,DEF,222.22
   30 DATA 24680246,Brown,GHI,333.33
   40 DATA 56789012,Evans,ACV,123.45
   50 //
   60 OPEN FILE 1,Bills,RANDOM 40
   70 record# := 0
   80 REPEAT
   90   READ refer#,name$,initial$,amount
  100   record# :+ 1
  110   PRINT FILE 1,record#:refer#,name$,initial$,amount
  120   PRINT name$;" ";initial$;TAB( 20);refer#,amount
  130 UNTIL EOD
  140 CLOSE FILE 1 // MUST close file
```

Run the program.

32.15 Retrieving data from a random access file

The keywords `INPUT FILE` may be used to retrieve data from a file.

`INPUT FILE` is followed by the file number, a comma, the record number and a colon. The items to be taken in from the record are then listed, separated by commas, for example:

```
INPUT FILE 2,record#:refer#,name$,initial$,amount
```

The command `READ FILE` can be used instead of `INPUT FILE`.

Enter the following program:

```
  100 MODE := 6
  110 PRINT TAB( 0,2);
  120 INPUT "Which record ? " :record#
  130 //
  140 OPEN FILE 2,Bills,RANDOM 40 READ ONLY
  150 INPUT FILE 2,record#:refer#,name$,initial$,amount
  160 CLS
  170 CLOSE FILE 2
  180 //
  190 PRINT 'name$;" ";initial$
  200 PRINT "Reference number ";refer#
  210 PRINT "Amount owing >>> ";amount
```

Run the program a few times trying different records.

The records are accessed directly. We do not have to read our way through to the appropriate information.

32.16 On-line processing

It is very easy to carry out continuous processing of data. Whilst a file is open, data can be read, processed and written back (provided that the file wasn't opened for `READ ONLY`). As an example, we can change the amount due from one of the people in our random access file.

Change 140 to

```
  140 OPEN FILE 2,"Bills",RANDOM 40
```

(removing the `READ ONLY` restriction).

Delete line 170 and add the following

```
220 PRINT '"How much extra > ";
230 INPUT extra
240 owing := amount + extra
250 PRINT '"New balance >";owing'
260 PRINT FILE 2,record#:refer#,name$,initial$,owing
270 CLOSE FILE 2
```

Run the program a few times seeing how the same account is being updated.

In practice, when carrying out such a procedure, careful records would need to be kept of the original amount and the transactions that caused the change.

A file of transactions could be built up and a new file created showing the updated amounts. This could be done with either a random access file or a sequential file.

If the transactions are placed in the same sequence as the records then continuous processing can be carried out with a sequential file. The original file can be read through and records either copied directly into the new file or amended and then written into the file.

32.17 The EXTent of a file

`EXT` is a function that will return the size of an opened file. The number of the file must be given after `EXT`, eg `PRINT EXT 2`.

An error message is given if the file is not open.

32.18 Extra records

A random access file may only be extended if there is sufficient space available on the disc at the end of the file. If you foresee the requirement to add extra records at some stage, then it is wise to create the whole file at the outset. This can be achieved by printing some values into the last record needed. Space is then reserved for all the preceding records.

Example:

```
10 OPEN FILE 1,BILLS,RANDOM 40
20 PRINT FILE 1,100:999999,"ZZZZZ","ZZ",0
30 CLOSE FILE 1
```

will create a file called `BILLS` and ensure that space is reserved for records 0 up to 100. Record 100 will contain the dummy information shown in line 20 above.

33 Other functions and facilities

33.1 Standard mathematical functions

Acornsoft COMAL also provides the following standard mathematical functions:

`ACS(number)` returns the arc(inverse)-cosine in radians.

`ASN(number)` returns the arc(inverse)-sine in radians.

`ATN(number)` returns the arc(inverse)-tangent in radians.

`COS(angle)` returns the cosine of an angle given in radians.

`DEG(radian)` returns the degree equivalent.

`EXP(number)` returns the exponential power.

`LN(positive_number)` returns the logarithm to base e.

`LOG(positive_number)` returns the logarithm to base 10.

`RAD(degree)` returns the radian equivalent.

`SIN(angle)` returns the sine of an angle given in radians.

`SQR(positive_number)` returns the square root.

`TAN(angle)` returns the tangent of an angle given in radians.

33.2 Sound

The BBC Microcomputer has extensive facilities to generate sounds using the keywords `SOUND` and `ENVELOPE`. (The Acorn Electron has similar, but less extensive facilities and reference should be made to the *Acorn Electron User Guide* for details.) For simple effects the `SOUND` statement can be used on its own, but greater control is given by using the `ENVELOPE` statement. At its simplest `SOUND` is followed by four arguments, for example:

```
SOUND c,a,p,d
```

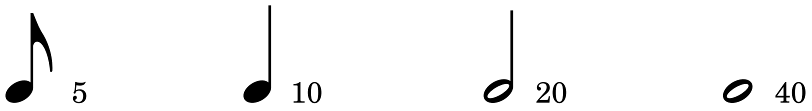
c	is the channel number	0 to 3
a	is the amplitude or loudness	0 to −15
p	is the pitch	0 to 255
d	is the duration	1 to 255

The channel number determines which of the four ‘voices’ is to be used. Channel 0 produces only ‘noise’ whereas channels 1, 2 and 3 produce purer notes.

The amplitude can be varied from 0 (off) to −15 (loud).

The pitch selects notes in intervals of a quarter of a semi-tone. Middle C is produced when the pitch is set to 53, and the other notes are generated with the values given below, spanning a full five octaves.

The duration determines the length of the note and is given in twentieths of a second. Those used to reading music will find that ‘Moderato 𝅗𝅥 = 60’ will be about right with the following settings for the duration.



	Octave number						
Note	**1**	**2**	**3**	**4**	**5**	**6**	**7**
B	1	49	97	145	193	241	
A #	0	45	93	141	189	237	
A		41	89	137	185	233	
G #		37	85	133	181	229	
G		33	81	129	177	225	
F #		29	77	125	173	221	
F		25	73	121	169	217	
E		21	69	117	165	213	
D #		17	65	113	161	209	
D		13	61	109	157	205	253
C #		9	57	105	153	201	249
C		5	53	101	149	197	245

As a simple example, enter the following program lines which play a well-known film theme:

```
100 SOUND 1,-15,97,10
110 SOUND 1,-15,105,10
120 SOUND 1,-15,89,10
130 SOUND 1,-15,41,10
140 SOUND 1,-15,69,20
```

The `SOUND` facility may be extended in two main areas. Firstly, an ‘envelope’ can be selected to vary the amplitude and pitch of the note while it is playing;

secondly it is possible to ensure that notes are synchronised so that chords start together.

These methods of controlling the sound are very powerful, but also somewhat complex. A detailed description is given in Part III under `SOUND` and `ENVELOPE`.

An example of the more complex sounds possible using the `ENVELOPE` statement is

```
ENVELOPE 1,1,-26,-36,-45,255,255,255,127,0,0,-127,126,0
SOUND 1,1,255,255
```

33.3 Analogue input

The BBC Microcomputer Model B (or an expanded Model A) has an analogue to digital (A-D) converter facility. The Plus 1 expansion unit also provides a similar facility for the Acorn Electron, but reference should be made to the *Acorn Electron Plus 1 User Guide* for details.

There are four analogue to digital converters in the Model B BBC Microcomputer. Each converter takes an analogue input voltage and outputs a digital representation of that voltage. The analogue voltage might be controlled by, for example, the position of a joystick, or it might be the output from some sensor which the software is using.

The range of acceptable input voltages is from 0V to 1.8V. The corresponding range of digital output is 0 to 65520, that is a 16-bit binary number. At present the hardware in the computer can only return the value to an accuracy of 12 bits, but the facility exists for expansion to 16-bit if and when the hardware becomes available.

The numbers returned by the A-D converter, therefore, increase in steps of 16 in the range 0 to 65520, but a simple division by 16 will give numbers in the range 0 to 4095 if preferred.

The exact method by which the analogue input channels are controlled is described in Part III under `ADVAL`.

34 Low level aspects of COMAL

In this section various aspects of low level programming in the COMAL environment are discussed.

34.1 System functions and variables

Acornsoft COMAL supports the following keywords to provide information to the user regarding the memory being used by the language: `PAGE`, `SIZE` and `FREE`.

`PAGE` specifies the first location that COMAL uses for storing the text of programs. As its name suggests, it must be set to the start of a page of memory in the machine (ie it must be an exact multiple of 256, &xx00). Its default value depends on the precise configuration of the BBC Microcomputer but a few sample values are:

Tape filing system	&E00
Disc filing system	&1900
Disc and Econet	&1B00
6502 Second Processor	&800

`PAGE` is a system variable – it can be both read and assigned. The main use of it is to protect machine code, as described in the section below on using machine code from COMAL.

`SIZE` is a read only function that returns the size of the program currently in memory. The program actually occupies the locations `PAGE` to `PAGE + SIZE - 1` inclusive, which is a total of `SIZE` bytes. Note that the start- and end-of-program bytes are always present in memory so that, after a `NEW`, `SIZE` is always 2.

`FREE` is a system function that returns the number of bytes of free memory available at the instant the keyword is interpreted. Note that the free memory is defined as the memory between the top of the global variable space and the bottom of the software stack (see memory map, below) and so this varies throughout a calculation as intermediate results are stored on the stack. This means that `PRINT FREE + SIZE` will give a different answer to `PRINT SIZE + FREE`.

34.2 Using machine code from COMAL

The user can call machine code routines from COMAL by means of the USR function. This takes a single argument, the address of the machine code to be called, and returns a value taken from the 6502's Accumulator, X register, Y register and Processor status register in the order AXYP lo-byte to hi-byte. On entry to the machine code, the accumulator, X and Y registers are set to the LSBytes of a#, x# and y#, and the carry flag is set to the LSBit of c#. No assembler is included in COMAL, but machine object code can be called from it. The code can either be placed in ROM or one of the unused OS buffers (eg RS423 buffers on a disc system), or it can be placed in user RAM and protected. In the latter case it should be loaded into the bottom of memory and PAGE set above the last required location. For example, if 4K of machine code is required in a disc-based machine which initially has PAGE set at &1900 then a two-part program can be run:

```
 10 // Part One
 20 //
 30 PAGE :+ 4096
 40 RUN "part2"

 10 // Part Two
 20 //
 30 *load code 1900
 40
 50   rest of program
 ..
130 a:=USR &1900
```

34.3 Memory organisation and map

The overall memory map for Acornsoft COMAL is as follows:

Locations			**Contents**
&0000	to	&0084	Zero page COMAL usage
&0085		&008F	Spare
&0090		&00FF	Operating system usage
&0400	to	&046B	ZONE and a#-z#
&046C		&047F	Temporary floating point workspace
&0480		&04B5	Variable catalogue
&04B6		&04E9	File characteristics table
&04EA		&04EB	Pointer to first PROC or FUNC
&04EC		&04FF	Spare
&0500		&05FF	Loop stack
&0600		&06FF	String accumulator (&680-&69F doubles as file control block, &6A0-&6AF as OSWORD block)
&0700		&077F	Direct mode line buffer
&0780		&07FF	Spare
PAGE			Start of COMAL program
PRGTOP			End of COMAL program
LOMEM		See	Start of global variable block
VARTOP		below	End of global variable block
		for	(Free memory)
AESTKP		details	Bottom of software stack
LOCALL		of	Bottom of local variable block
LOCALH		locations	Top of local variable block
HIMEM-1			Top of software stack Top of user language RAM
HIMEM	to	&7FFF	Video memory (unless in Second Processor)
&8000		&BFFF	COMAL interpreter
&C000		&FFFF	Operating system (&F800-&FFFF in Second Processor)

Useful locations in zero page are:

&00,01	HIMEM	
&02,03	LOMEM	
&04,05	AESTKP	
&06,07	PAGE	See memory map
&08,09	VARTOP	for explanation
&0C,0D	PRGTOP	
&0E,0F	LOCALL	
&10,11	LOCALH	
&12	CLOSED	Flag – &FF if inside `CLOSED PROC` or `FUNC`
&13,14	TXTPTR	Two byte pointer to current text
&15	TXTOFF	One byte offset to TXTPTR
&3F	ERFLAG	Most recent error number (Though reset very frequently)
&68	PDBUGD	Non-zero if program structure correct. Corrupt with care only.
&6F	LSTKP	Loop stack pointer

34.4 Program structure

Each line of program is stored in a tokenised form in memory (see section 40.4 for list of tokens). The line format, though similar, is not compatible with BBC BASIC. It is:

&0D	HI	LO	LL	IN	Text of line	&0D (next line)

HI and LO are the hi- and lo- bytes of the line number.
LL is the length of the line (including header).
IN is the indentation of the line in structures (2 spaces per structure).

The end of program is marked by HI being set to &FF. Tokens for keywords are represented by bytes in the range &80 to &FE. Substring specifiers are preceded by a 01 byte. Anyone writing 'toolkit' or similar utilities for the language should bear in mind that, since syntax is checked at line entry, the run-time interpreter has been written assuming that the line syntax is correct. No trapping of incorrect syntax will occur at run time, and introducing any into the program text is likely to cause fatal crashes. Any modifications to the program text should also reset PDBUGD (see memory map) to indicate to the interpreter that the structure needs to be re-checked before running or listing.

Implementers of utilities should note that subroutines exist inside the interpreter to perform the following functions.

– List a single line of program.
– Decrunch a single byte.
– Relink the program and update size pointers.
– Search the program for a given character.
– Crunch a single line of program text.
– Check the syntax of a crunched program line.

34.5 Some simple examples

Two facilities the user may require are put-byte-to-file and get-byte-from-file. These may be performed on files opened using a COMAL `OPEN` statement as follows:

```
1000 FUNC file_get(fnumber)
1010   y#:=&04B6?(8*fnumber)
1020   a#:=USR &FFD7
1030   IF a# AND &1000000 THEN
1040     PRINT "EOF"
1050     STOP
1060   END IF
1070   RETURN 255 AND a#
1080 END FUNC file_get

2000 PROC file_put(fnumber,byte)
2010   y#:=&04B6?(8*fnumber)
2020   a#:=byte
2030   byte:=USR &FFD4
2040 END PROC file_put
```

Any other functions that the user implements in machine code should be implemented in a similar style, ie if they are used more than once then the individual users should call a procedure or function that sets up and performs the `USR` call.

35 Merging COMAL programs

There are two possible methods for merging COMAL programs.

If you consider that full error reporting would be helpful then the first method should be used. It is effectively the same as the BBC BASIC routine given in the *BBC Microcomputer System User Guide*.

The second method is much simpler and generally the one to use. It works in COMAL because ~ provides a string value, which can be combined with the load command and then passed to `OSCLI`, which is the Operating System Command Line Interpreter. This will interpret the command implied by the string expression.

Method 1

Step	Command
1	`load part2`
2	`renumber 20000,1`
3	`*spool temp`
4	`list`
5	`*spool`
6	`load part1`
7	`renumber 1,1`
8	`*exec temp`
9	`renumber`

Method 2

Step	Command
1	`load part1`
2	`oscli("LOAD part2 "+ ~(PAGE+SIZE-2))`
3	`old`
4	`renumber`

Note the space between `part2` and the quotation mark.

36 Converting programs to COMAL

Although BBC BASIC appears to be very similar to COMAL, there is little, if anything, to be gained from simply entering slightly modified BASIC code as COMAL. However, returning to the original algorithm for the program and then recoding in COMAL, making full use of its wide range of structures, will result in a program that is much easier to read and maintain.

Programs originally written in Pascal are more amenable to conversion to COMAL, since the structured nature of the two languages has much in common. However, although the structure required for the program will be very similar, the code itself is somewhat different. In this case, returning to the structure diagram (or flow chart) rather than right back to the original algorithm is likely to be sufficient.

It is possible using the VIEW word processor to convert programs from one language format to another by using the operating system commands `*SPOOL` and `*EXEC`. This enables a listing of a program in one language to be stored as a sequence of ASCII codes rather than in the usual tokenised form, and then reloaded in another language as if it were keyboard input.

The method is described in outline below, but reference should be made to the *BBC Microcomputer System User Guide* or the *Disc Filing System User Guide* for further details of the `*SPOOL` and `*EXEC` commands, and to the VIEW manuals.

Example: to produce a COMAL program `comtest` corresponding to the BASIC program `bastest` the following steps are necessary:

```
*BASIC
LOAD "bastest"
*SPOOL temp
LIST
*SPOOL
```

Enter

`*WORD` to change to VIEW

and

`READ temp` to load the file.

At this point the file `temp` may be edited to split multi-statement lines and to produce the necessary COMAL structures by, for example, placing `ELSE` on its own line and completing `IF` structures with an `END IF`. The powerful `REPLACE` facility can be used to make changes using the Y or N keys.

Example:

`REPLACE = :=`	for assignment but not in a condition
`REPLACE % #`	for integer variables
`REPLACE :REM //`	for comments

Then

`SAVE temp`	to save VIEW file
`*COMAL`	to change language
`*EXEC temp`	to load the file into memory as a program

When the program runs satisfactorily in COMAL you may save it by entering

```
SAVE comtest
```

37 Error messages

In this section all the error messages that can be generated by COMAL are listed alphabetically, along with their error numbers (which may be of use to low level programmers), and indications as to their cause.

All error messages generated while a program is being run or debugged will be followed by a listing of the line where the error was detected.

Bad DIM **36**

This message indicates that an attempt has been made to `DIM`ension an array or string with bounds that are not allowed.

Bad GOTO **29**

This is generated if an attempt is made to jump out of or into a structure where this is not permitted.

Bad program **21**

If the section of memory containing the program has been corrupted and an attempt is made to perform some operation on that program (eg `OLD`, `RUN`, `SAVE` etc) then this error will be generated.

Bad type **23**

This error is generated if an attempt is made to perform an operation which requires one variable type with another (in `CASE` structures, when reading data from files or with `RETURN`).

Bad value **32**

This message indicates that an operation has been attempted for which the numeric argument given was not in the valid range (eg `SQR(-1)`).

Can’t CONT **18**

`CONT` will only work if program execution has been halted by a `STOP` command or by the ESCAPE key being pressed, and if the program has not since been corrupted. An attempt to `CONT` otherwise will give this error.

EOD **33**

When attempting to `READ` from `DATA` statements, this error will be generated if there is no more data available in the program.

EOF **41**

This error will be generated if an attempt is made to `READ` or `INPUT` items from a file after the end of the file has been reached.

Escape **17**

This indicates that the ESCAPE key has been pressed.

File open **38**

If an attempt is made to open a file which is already open then this error will be generated.

Name mismatch **25**

This error is generated by the prerun structure checker if the name or variable at the end of a structure does not agree with that at the beginning. For example, if the name following a `PROC` statement is not the same as that after the corresponding `END PROC`.

No (keyword) **28**

This error is generated by the prerun structure checker if an end of structure statement is found without a corresponding start of structure.

No RETURN **39**

If when a function is executed, the `END FUNC` is reached before a `RETURN` is encountered then this error is generated.

No room **20**

This message indicates that there is insufficient memory available for the operation attempted.

Not allowed **26**

This error will be generated when the interpreter encounters an instruction which, although it may be syntactically correct, is not allowed at that point.

Not found **31**

This error will be generated either if an attempt is made to access a file which is not present on the current filing system, or if an operation is attempted on a variable which has not been previously assigned or on a procedure or function which has not been defined.

Not open **35**

If an attempt is made to access a file which is not open then this error is generated.

Parm block error **40**

This message indicates that there is some error in the parameters used to call a procedure or function.

Record overflow **34**

This error will be generated if an attempt is made to `WRITE` or `PRINT` more information to a random access file than may be accommodated in the record length assigned.

STOP **19**

If a `STOP` command is encountered by the program interpreter then this message is generated, and execution halts with a report of the line number.

String too long **30**

This error will be generated if an attempt is made to assign a string to a variable which is longer than the allocated maximum length for that string (which defaults to 40 characters). If the string has not been dimensioned previously then a dimension of 40 is assumed and the string is assigned the null string. A subsequent dimension statement will cause a `Variable exists` error.

Syntax error **22**

If the line input syntax checker detects an error in the syntax of a line then this message is generated. A type mismatch error is reported as a `Syntax error`.

Too complex **27**

This error will be generated if the level of 'nesting' either of parentheses or structures is too great for the interpreter to cope with.

Unclosed at **24**

The prerun structure checker generates this error if it does not find an end of structure statement where one is required. It is followed by either the line itself or `END`, indicating where the omission was detected.

Variable exists **37**

This error will be generated if an attempt is made to redimension an array or string.

Part III – reference section

38 Guide to the reference section

This reference section is designed for anyone who needs to check the details of the use of some keyword or operator.

38.1 Format of summaries

The keyword and operator summary which follows contains descriptions of all the operators used in the language, followed by descriptions of all the COMAL keywords, in alphabetical order.

Each keyword is described under the following headings:

KEYWORD **Minimum form**

The keyword is given and in some cases it is followed by a few words which explain the derivation of the word. At the right of the page is the minimum form of the keyword which will be accepted by the computer.

Description

The purpose and method of use of the keyword is described in normal computer terminology. For certain of the mathematical functions this description will have no significance unless users are familiar with the concepts involved.

Reference

A reference is given to the section in the earlier parts of the manual where the keyword is first introduced, and sometimes to other keywords or chapters if they are likely to be useful.

Syntax

A syntax diagram is given. This is the clearest way of displaying the many alternative and syntactically correct constructions.

The diagram is interpreted from left to right, starting with the keyword or the line from the left hand side (indicating that some other element must precede the construction). The end of the construct is indicated either by an end of line (eol), a carriage return (cr) or a line to the right indicating that some further construction is required.

Any path which takes you towards the end of the diagram represents an allowable syntax.

Arrowed paths indicate where the flow may take alternative routes or involve repetition.

An arrowed path with a star indicates that if the element which is bypassed is omitted, then it will be inserted automatically.

The syntax diagrams for the expressions, elements, lists, constants, variables and other terms used in the individual diagrams are given at the beginning of the keyword and operator summary.

Examples

One or more short examples are given to indicate possible uses of the keyword. Both direct mode and program line examples are given where appropriate, and reference is sometimes made to other keywords where further relevant examples may be found. Note that for a number of operators brackets have been used in the examples where they are not specifically required by the syntax, but may be an aid to comprehension.

Associated keywords

A list is given of other keywords used in the same structures or with related effects.

OPERATORS

Operators are described in essentially the same way as keywords, but in some cases are grouped together under a single entry. See the first page of chapter 39 for the order and grouping of the operators.

Note that the term 'binary operator' is used to describe many operators. It is the term used for describing an operator which takes two arguments, one before it and the other after (eg `1 + 2`, or `a AND b`).

The precedence with which operators act is given in section 40.6.

38.2 Reference tables

The reference tables in chapter 40 summarise the information which the user may need for certain operations. The tables supplied are as follows:

1. ASCII codes, giving the codes for all the available characters and operations.

2. VDU codes, giving a summary of the VDU codes available and the arguments which they require.

3. *FX/OSBYTE calls, a summary of some of the more useful Operating System calls available with *FX or OSBYTE.

4. A list of all the keywords and operators along with their minimum forms and tokens (where appropriate).

5. A list of the possible error messages along with their error numbers.

6. A precedence list showing the order in which operators will act in the absence of brackets, as well as the valid ranges for arithmetic operations and their accuracy.

39 Keyword and operator summary

This chapter contains the following sections:

– Subdefined elements for syntax diagrams
– Assignment operators `:=  :+  :-`
– Arithmetic operators `+  -  *  /  ^`
– Relational operators `=  <  >  IN`
– Substring specifiers `(:)`
– Hexadecimal operators `&  ~`
– Indirection operators `?  !`
– Comment delimiter `//`

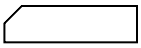
followed by a complete alphabetical keyword summary.

Subdefined elements for syntax diagrams

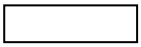
This section contains the definitions of all the subdefined elements used in the syntax diagrams in terms of basic elements that may be entered from the keyboard.

The subdefined elements are listed in alphabetical order for ease of reference.

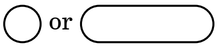
The notation used in the syntax diagrams is as follows:



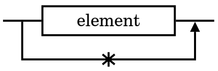
indicates that the element is a keyword which must be followed by a non-alphanumeric character.



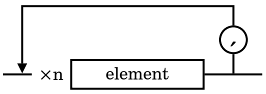
indicates that the item is a subdefined element.



indicates that the item is a basic element.

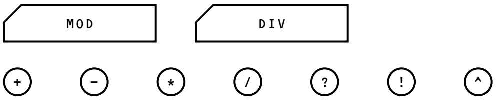


indicates that if the element is omitted on entry then it will be inserted automatically.

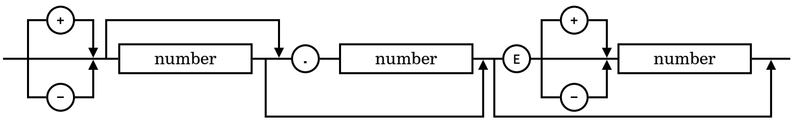


indicates that n elements should be entered, separated by commas.

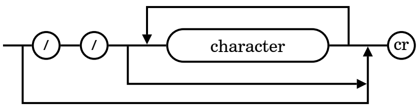
binary operator



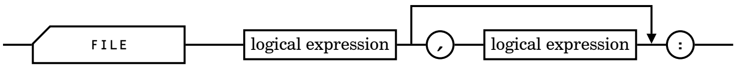
constant



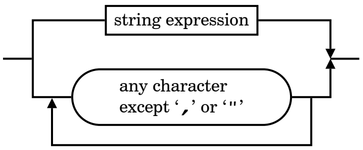
end of line (eol)



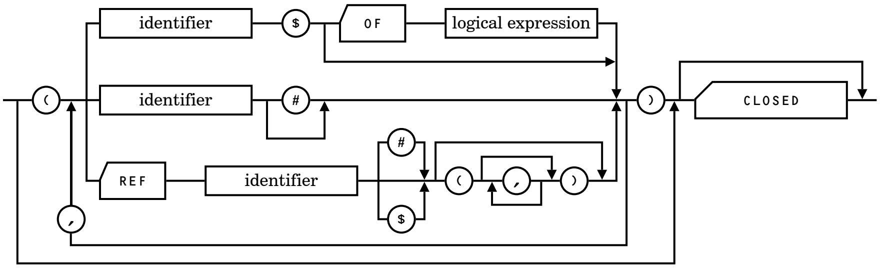
file designator



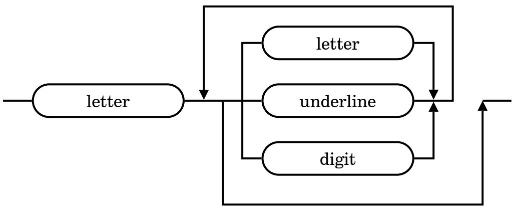
filename



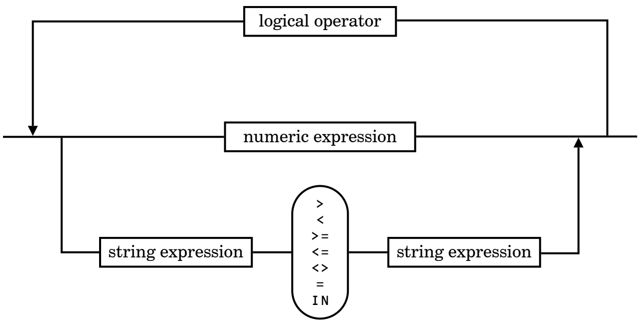
head appendix



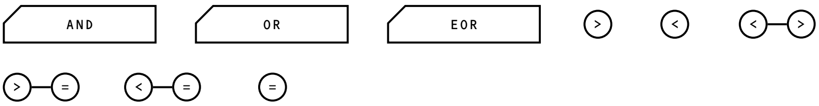
identifier



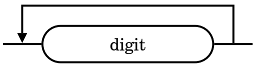
logical expression



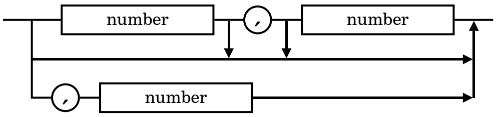
logical operator



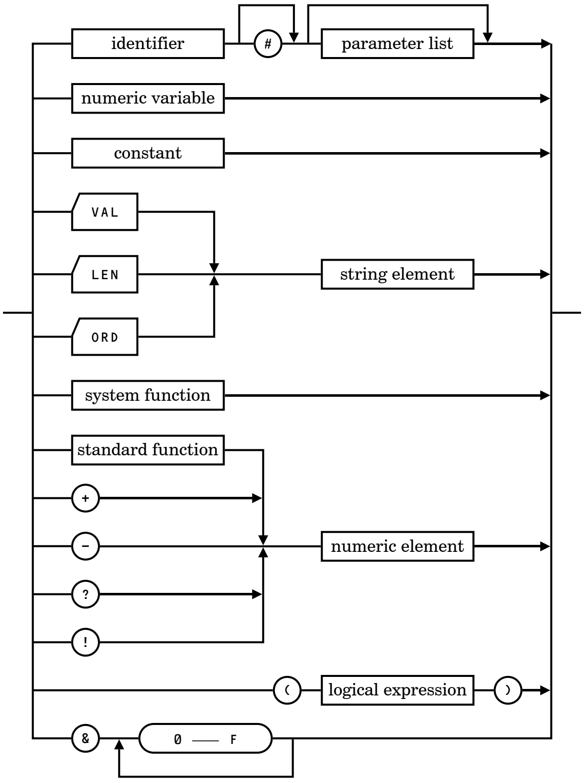
number



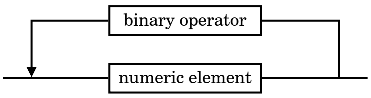
number list



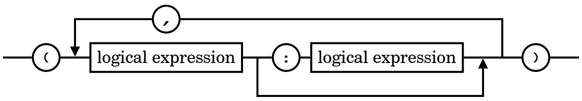
numeric element



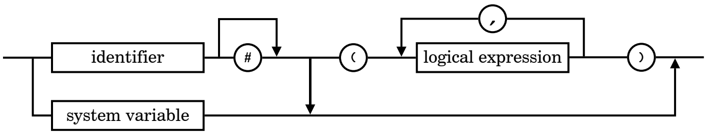
numeric expression



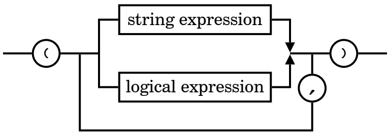
numeric list



numeric variable



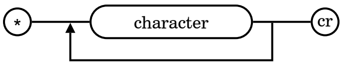
parameter list

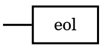


simple statement

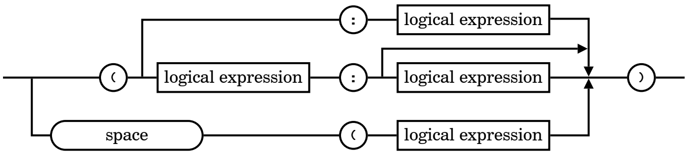
A syntactically correct version of any one of the following keywords or elements will be a valid simple statement.

CLEAR	CLG	CLOSE	CLS
COLOUR	DELETE	DRAW	ENVELOPE
EXEC	GCOL	GOTO	INPUT
MOVE	NEW	NULL	OPEN
OSCLI	PLOT	PRINT	READ
RESTORE	RETURN	RUN	SELECT OUTPUT
SOUND	VDU	WRITE	

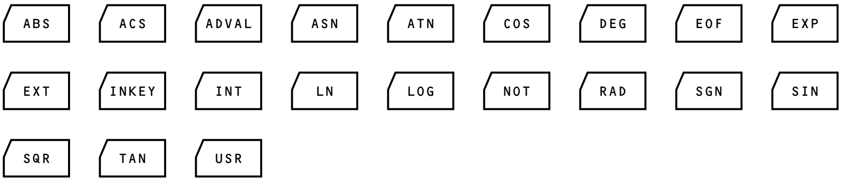




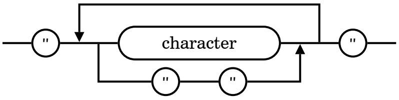
specifier



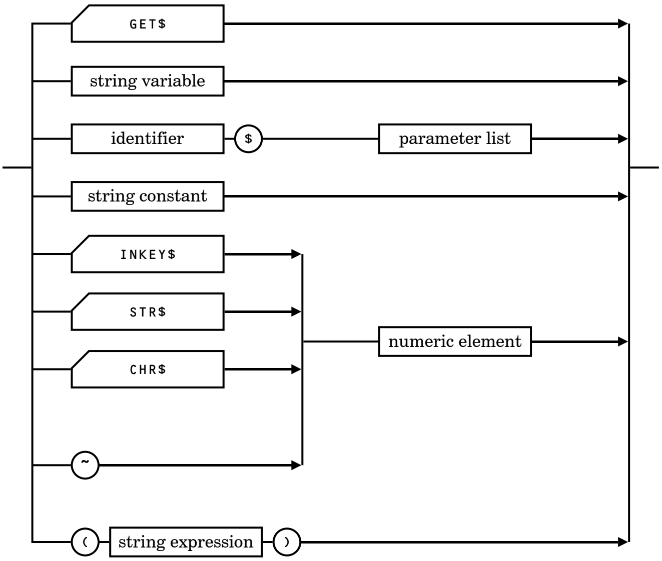
standard function



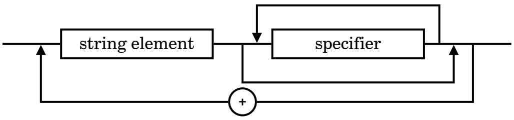
string constant



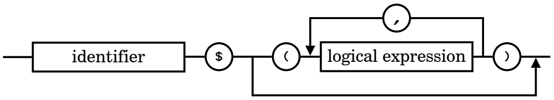
string element



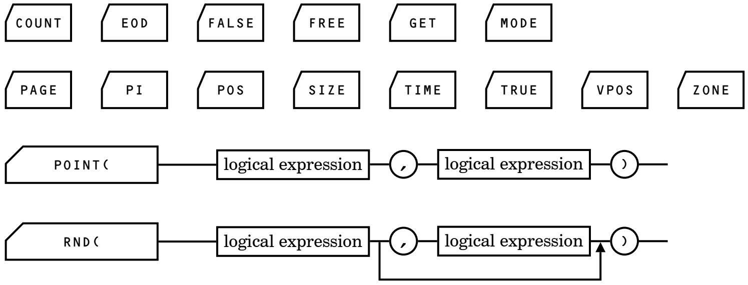
string expression



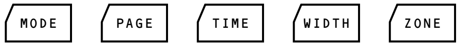
string variable



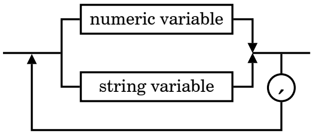
system function



system variable



variable list



`:=` `:+` `:-` Assignment operators

Description

These operators are used for assigning values to numeric and system variables, the contents of specific memory locations, and (`:=` `:+` only) for assigning strings to string variables.

`:=` is used to assign the expression following it to the variable preceding it. The `:` is optional on entry and will be inserted automatically if omitted.

`:+` is used with numeric variables to add the value of the expression following it to the existing value of the variable, and assigning the result back to the variable. With string variables it is used to add the string expression following it to the end of the current content of the variable. It cannot be used with a substring on the left of the assignment symbol.

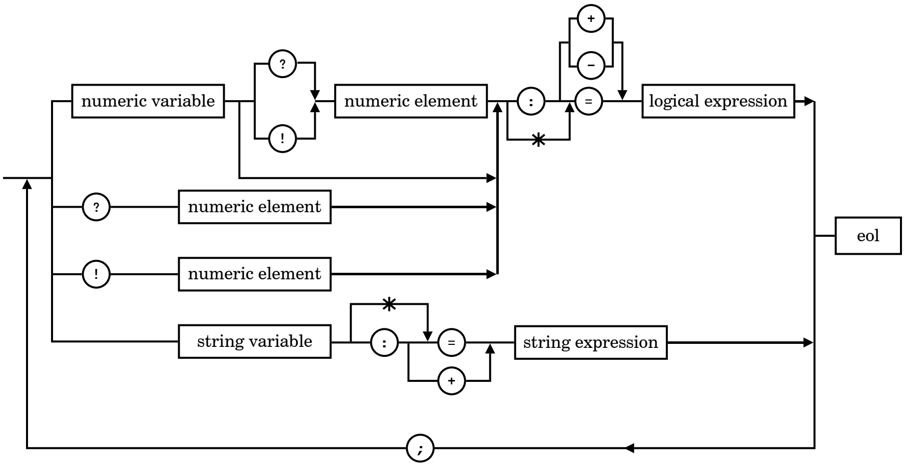
`:-` is used only with numeric variables. It takes the value of the expression following it and subtracts it from the current value of the variable. The result is assigned back to the variable.

Note that, for example:
`a :+ b AND c` is equivalent to `a := a + (b AND c)` not
`a := (a + b) AND c`

Reference

Sections 3.2, 6.7, 6.8 and the precedence table (40.6).

Syntax



Examples (see also sections 6.7, 6.8)

```
a := number1 * number2
b# := a / d
c$ := message$
MODE :=6

d(i#) :+ i#
e$(2) :+ "end of string"
f :+ g ^ 2

g :- 17.5 * const1
h(6) :- d(4) * 6
?x :- 1
```

+ − * / ^ Arithmetic operators

Description

These are the standard arithmetic operators.

`+` is either a binary operator which returns the result of adding the two operands, or a unary operator indicating that the following argument is not to have its sign changed (ie having no effect).

`-` is either a binary operator which returns the result of subtracting the second operand from the first, or a unary operator indicating that the following argument is to have its sign changed (ie multiplied by `-1`).

`*` is a binary operator which returns the result of multiplying the two operands together.

`/` is a binary operator which returns the result of dividing the first operand by the second.

`^` is a binary operator which returns the result of raising the first operand to the power of the second. Note that, for example, 2^3^4 is (2^3)^4.

Known problem

In the first release of Acornsoft COMAL a slash (`/`) immediately followed by a point (`.`) will be interpreted as the minimum form of the comment delimiter. When dividing by constants between 0 and 1 it is, therefore, necessary either to put a space or a zero before the decimal point.

Reference

Sections 2.4, 2.7 and the precedence table (40.6).

Syntax

See numeric element, constant, binary operator.

Examples

```
twenty_three := 3 + 4 * 5

minus_seventeen := 30.4 - 11.85 * 2 ^ 2

twenty_five := 5 * 20 / 4

plus_one := -1 ^ 4
```

= < > IN Relational operators

Description

These are all binary operators used in specifying a logical relationship which will return a result which is either true or FALSE. They may be used either singly or in pairs as follows.

Note that 'true' means not FALSE (ie non-zero), whereas TRUE has the value -1.

= will return TRUE if the two operands evaluate either to equal numbers or identical strings, and FALSE otherwise (but beware of rounding errors).

< compares the values of numerical expressions or the ASCII codes of string characters. With numerical expressions < returns TRUE if the first expression evaluates to less than the second, and FALSE otherwise. With strings the two operands are evaluated and compared letter by letter until the ASCII codes of the two characters are different. If the code for the character from the first string is lower than that from the second then < returns TRUE, otherwise it returns FALSE. If the end of one string is reached before any differences are found then the shorter string is taken to have the lower ASCII code.

<= compares the two operands and returns TRUE if the first is less than or equal to the second, and FALSE otherwise. The result obtained will be the opposite to that returned by >.

> compares the two operands in the same way as <, but returns TRUE if the first expression is greater than the second, and FALSE otherwise. It will return the opposite result to <=.

>= compares the two operands and returns TRUE if the first is greater than or equal to the second. The result obtained will be opposite to that returned by <.

<> returns TRUE if the two operands are not equal numbers (or identical strings), and FALSE otherwise. The result will be opposite to that returned by =.

IN is used as a relational operator to compare two string operands. If the first string appears in the second string then IN returns its position (which is considered a true result), and if it does not then it returns FALSE. See the entry in the keyword summary for further details of IN.

Reference

Sections 15.2, 15.3, 15.4 and the precedence table (40.6).

Syntax

See logical operator, logical expression, `WHEN`.

Examples (see also chapter 15, `IN` and all examples of conditional structures)

```
100 IF x < 0 THEN PRINT "Negative"

150 WHEN >= 5, <= 10

 95 UNTIL repeat$ <> "Yes"
```

(:) Substring specifiers

Description

It is possible to perform operations on portions of a string or string expression by using substring specifiers.

A substring is specified by giving the string identifier (which may be an element of a string array or a string expression) followed by an expression of the form `(start:finish)` where `start` and `finish` are the positions of the first and last characters of the substring in the string. Either specifier may be omitted and they will default to the beginning and end of the string respectively.

With positive arguments the position is counted from the start of the string (the first character being at position 1).

With negative arguments the position is counted from the end of the string back towards the beginning (the last character being at position `-`1).

If the first position is after the second then the substring is returned as a null string.

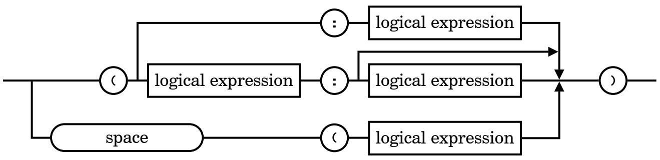
Assignments to substrings may either reduce the length of the substring or leave it unchanged. It is not possible to extend a substring. If this is attempted the string being assigned to the substring is truncated from the right until it is the same length.

Note that single character substrings (eg `a$(x:x)`) may be specified with just a single value in the brackets (eg `a$ (x)`), but a space is usually necessary before the brackets to avoid confusion with string arrays.

Reference

Chapter 29.

Syntax



Examples (see also chapter 29)

```
PRINT a$(3:6)

  200 b$ :+ a$(-6:-3)

  315 c$ := ("  " + d$)(-6:)

  400 UNTIL a$ (1) = "Y"

  500 a$(1:3) := "HI" // reduce length of substring

  600 d$ := a$(x:y)(2:3)
```

&   ˜   Hexadecimal operators

Description

These operators are employed when numbers are used in their hexadecimal form.

& is used to immediately precede a numeric constant to indicate that it is expressed in hexadecimal notation. The number is then used in exactly the same way as any other.

˜ is used to precede a numeric element to indicate that the number should be expressed in hexadecimal form as a string.

Note that both these operators have the highest level of precedence.

Reference

Precedence table (40.6).

Syntax

See numeric element, string expression.

Examples (see also chapter 34)

```
PAGE := &1A00

PRINT ˜FREE

ffffffff$ := ˜(-1)

zero$ := ˜0.1
```

? ! Indirection operators

Description

Indirection operators are used to read and write directly at specific memory locations.

? (pronounced query) operates on a single byte. ?x means 'the contents of memory location x'. It may be used to read the contents of that location (PRINT ?x) or to set them (?x := y).

! (pronounced pling) operates on four successive bytes, that is the one indicated by the argument and the three successive bytes. Four bytes is the amount of memory used to store an integer variable, so pling may be used to manipulate them. The least significant byte of the integer variable is stored in the byte indicated by the argument, and the other bytes are stored in order of significance up to the most significant byte in the highest number location.

Additionally both query and pling may be used as binary operators (with variables or constants as the operands). For example, x?8 is taken as the contents of (memory location x+8)'.

Reference

Precedence table (40.6).

Syntax

See numeric element, binary operator, assignment.

Examples

```
PRINT ~?&3000

   35 !y := i#

  100 FOR i#:= 0 TO 16 DO PRINT ~(x+i#), ~(x?i#)
```

//   Remark

Description

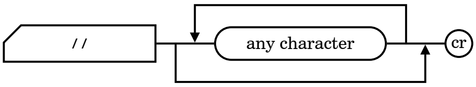
`//` is used to preface a comment in the program. All characters between the `//` and the end of the line are ignored by the interpreter.

A remark may be placed on a separate program line or at the end of any valid program line except after: `DELETE LOAD RUN SAVE SELECT OUTPUT` or Operating System `*` commands.

Reference

Section 5.12.

Syntax



Example

```
  100 x_coord:=640; y_coord:=512 // centre of screen
```

Associated keywords

None.

ABS Absolute value AB.

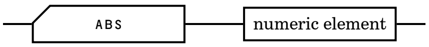
Description

This function returns the absolute value of its argument. For example, the absolute value of −3.67 is 3.67, while the absolute value of 1.72 is 1.72.

Reference

Section 18.6.

Syntax



Examples

```
PRINT ABS (speed# - 30)

   10 size_of_offset := ABS (reading - reference_value)

  120 four := ABS (-2) + 2
```

Associated keyword

SGN

ACS Arc-cosine AC.

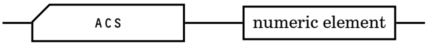
Description

ACS returns the principal value of the angle whose cosine is the argument. The value returned is in radians in the range 0 to PI. DEG may be used to convert this value to degrees.

Reference

Section 33.1.

Syntax



Examples

```
PRINT ACS (0.5)

   35 angle_in_degrees := DEG (ACS (0.3))
```

Associated keywords

ASN ATN SIN COS TAN RAD DEG

ADVAL Analogue to digital converter value AD.

Description

Note that on the Acorn Electron this function is not the same as the BBC Microcomputer version described here. Reference should be made to the *Acorn Electron User Guide* for details of `ADVAL` which performs exactly as in BASIC.

`ADVAL` performs three different functions depending on whether the argument following it is positive, zero, or negative.

With arguments in the range 1 to 4 `ADVAL` returns the most recently converted value of the analogue to digital channel with that number. The number returned has 12 bit resolution, but is scaled to 16 bit (ie it has a range of 0 to 65520 in steps of 16).

`ADVAL (0)` can be used to indicate which 'Fire' buttons are pressed on the games paddles, and also to indicate which was the last A-D channel to complete conversion. The information can be determined as follows:

`(ADVAL (0)) AND 3` will return 0 if no button is pressed, 1 if the left side fire button is pressed, 2 if the right side fire button is pressed or 3 if both fire buttons are pressed.

`ADVAL (0) DIV 256` will return the number of the last A-D channel to complete conversion, 0 indicating that no channel has yet done so.

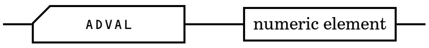
With negative arguments `ADVAL` is used to determine how full any of the internal buffers are.

`ADVAL (-1)` returns the number of characters in the keyboard buffer.
`ADVAL (-2)` returns the number of characters in the RS423 input buffer.
`ADVAL (-3)` returns the number of free spaces in the RS423 output buffer.
`ADVAL (-4)` returns the number of free spaces in the printer output buffer.
`ADVAL (-5)` returns the number of free spaces in the `SOUND` channel 0 buffer.
`ADVAL (-6)` returns the number of free spaces in the `SOUND` channel 1 buffer.
`ADVAL (-7)` returns the number of free spaces in the `SOUND` channel 2 buffer.
`ADVAL (-8)` returns the number of free spaces in the `SOUND` channel 3 buffer.
`ADVAL (-9)` returns the number of free spaces in the speech buffer.

Reference

Section 33.3 and the *BBC Microcomputer System User Guide*.

Syntax



Examples (see also the *BBC Microcomputer System User Guide*)

```
PRINT ADVAL 2

  100 *FX 2 1
  105 // line 100 takes input from RS423
  110 REPEAT
  120   message$ :+ get$
  130 UNTIL ADVAL (-2) = 0
  140 *FX 2
  145 // line 140 returns to keyboard input
```

Associated keyword

SOUND

AND

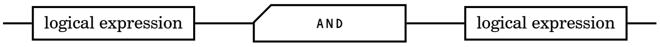
Description

AND is a logical operator which performs a bitwise AND on the two operands, ie it looks at each bit of the binary representations of the two operands in turn and, for each of them, sets the corresponding result bit to 1 if both are set or to 0 otherwise (giving a 32-bit number).

Reference

Section 21.2.

Syntax



Examples

```
210 IF a<>FALSE AND b<>FALSE THEN
220   PRINT "Both a and b"
230 ELSE
240   PRINT "At least one is false"
250 END IF
```

Associated keywords

OR NOT EOR

APPEND

Description

`APPEND` is a qualifying keyword used when `OPEN`ing a file which already exists for sequential writing from the current end of that file.

Reference

Section 32.10 and `OPEN`.

Syntax

See `OPEN`.

Examples (see also `OPEN`)

```
OPEN FILE 1,name$,APPEND
```

Associated keyword

```
OPEN
```

ASN Arc-sine AS.

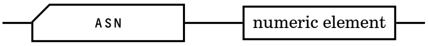
Description

ASN returns the principal value of the angle whose sine is the argument. The value returned is in radians in the range -PI/2 to +PI/2. DEG may be used to convert this value to degrees.

Reference

Section 33.1.

Syntax



Examples

```
PRINT ASN (-0.7)

   45 phase_shift := ASN (value_when_reference_zero)
```

Associated keywords

ACS ATN SIN COS TAN RAD DEG

ATN Arc-tangent AT.

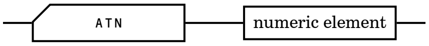
Description

ATN returns the principal value of the angle whose tangent is the argument. The value is in radians in the range -PI/2 to +PI/2. DEG may be used to convert this value to degrees.

Reference

Section 33.1.

Syntax



Examples

```
PRINT ATN (-1)

   25 angle := ATN slope
```

Associated keywords

ACS ASN SIN COS TAN RAD DEG

AUTO Automatic line numbering A.

Description

This command facilitates the input of programs from the keyboard by automatically generating the number of the next line each time RETURN is pressed. `AUTO` cannot be used as part of a program line.

Two arguments may be used, the first specifying the starting line number and the second the increment to be used. Both arguments default to 10 if omitted.

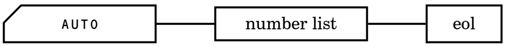
If a syntax error is found on a line that is entered then `AUTO` reprompts with the same line number.

`AUTO` mode may be left either by pressing ESCAPE or by generating a line number greater than 32767.

Reference

Section 5.11.

Syntax



Examples

```
AUTO 100,20
```

```
AUTO 100
```

```
AUTO ,20
```

Associated keyword

`RENUMBER`

CASE

Description

This is the declaration statement for the multiple choice `CASE` structure.

The argument after the `CASE` statement is evaluated and used to select one of the blocks of statements on subsequent lines which are declared using `WHEN` or `OTHERWISE`. To find which block is to be executed the argument is tested with each of the expressions in the `WHEN` clauses in turn until a true relation is found. The block of statements between that `WHEN` clause and the next structure element (`WHEN`, `OTHERWISE` or `END CASE`) is executed, after which the program continues from the line following the `END CASE` statement.

If no match for the argument is found in the `WHEN` clauses then the block of statements after the `OTHERWISE` clause is executed, if present. If no `OTHERWISE` statement is found then no case has been satisfied and execution proceeds from after the `END CASE` statement.

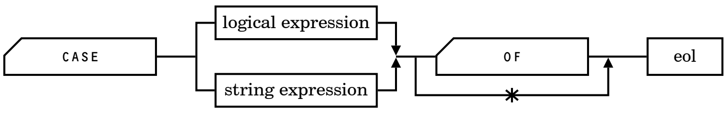
Note that only one of the blocks of statements is ever executed. This is the block nearest to the top of the structure that satisfies the `CASE` selector.

An `OF` statement will be inserted at the end of the `CASE` line since this is required by the language definition. It is always optional on entry.

Reference

Chapter 25.

Syntax



Examples (see also chapter 25)

```
   10 CASE MODE OF
   20 WHEN 0,3
   30   max_col := 79
   40 WHEN 2,5
   50   max_col := 19
   60 OTHERWISE
   70   max_col := 39
   80 END CASE
```

Associated keywords

`WHEN  OTHERWISE  END CASE`

CHR$ Character string CH.

Description

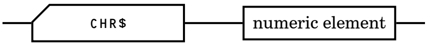
`CHR$` is a string function requiring a single numeric argument. It returns a single character string which is the ASCII character corresponding to the least significant byte of the numeric argument.

`CHR$` is needed to construct strings containing characters which are not available from the keyboard (for example, `CHR$(10)`, a line feed). It should be noted that it is often easier to use `VDU` for outputting control codes to the screen or other output stream.

Reference

Section 7.6 and table of ASCII codes (40.1).

Syntax



Examples

```
message$ := CHR$ (7) + text$ // CHR$ (7) = "sound bell"
   10 hi$ := CHR$ (34) + hi$ + CHR$ (34) // add quotes
```

Associated keywords

`ORD VDU VAL STR$`

CLEAR

CLE.

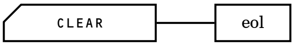
Description

`CLEAR` deletes all variables from the memory except for the system integer variables `a#` to `z#` and the system functions and variables, eg `TIME`, `MODE` etc.

Reference

Section 4.6.

Syntax



Examples

```
CLEAR
```

```
  100 CLEAR
```

Associated keyword

`NEW`

CLG Clear the graphics screen

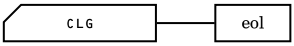
Description

`CLG` clears the current graphics area and leaves it in the current graphics background colour. The graphics cursor is moved to the graphics origin (0,0) (which is at the bottom left of the graphics screen unless it has been reset using `VDU 29`).

Reference

Section 11.6 and `GCOL`.

Syntax



Examples

```
CLG
```

```
  125 CLG
```

Associated keywords

`CLS GCOL`

CLOSE

Description

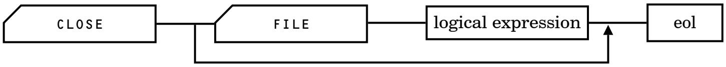
CLOSE is used to close files. It may be followed by FILE and a single argument giving the number of the file to be closed.

If used without any qualifiers CLOSE closes all open files on the current filing system, including, for example, *SPOOL and *EXEC files, as well as COMAL files.

Reference

Section 32.5.

Syntax



Examples (see also chapter 32)

```
CLOSE FILE 3
```

```
CLOSE
```

```
  115 CLOSE FILE file_number
```

Associated keywords

FILE OPEN

CLOSED

Description

`CLOSED` is used to declare that all variables used in a `PROC`edure or `FUNC`tion are local unless `IMPORT`ed or passed by `REF`erence. It appears at the end of the declaration line.

Reference

Section 31.5.

Syntax

See `PROC` and head appendix.

Examples

```
   10 PROC zap(a$) CLOSED
```

Associated keywords

`PROC FUNC IMPORT REF`

CLS Clear the text screen CLS

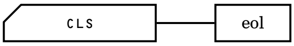
Description

`CLS` clears the current text area and leaves it in the current text background colour. In addition, the text cursor is moved to the ‘home’ position at the top left of the text area.

Reference

Section 7.1 and `COLOUR`.

Syntax



Examples

```
CLS
```

```
  135 CLS
```

Associated keywords

`CLG COLOUR`

COLOUR

COL.

Description

`COLOUR` sets the logical colours which are to be used for the background and foreground of the text area in `MODE`s 0-6.

The default actual colours for each of the logical colours in the different modes are as follows.

MODE 0, 3, 4 or 6

Foreground	Background	Colour
0	128	black
1	129	white

MODE 1 or 5

Foreground	Background	Colour
0	128	black (normal background)
1	129	red
2	130	yellow
3	131	white (normal foreground)

MODE 2

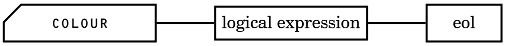
Foreground	Background	Colour
0	128	black (normal background)
1	129	red
2	130	green
3	131	yellow
4	132	blue
5	133	magenta
6	134	cyan
7	135	white (normal foreground)
8	136	flashing black-white
9	137	flashing red-cyan
10	138	flashing green-magenta
11	139	flashing yellow-blue
12	140	flashing blue-yellow
13	141	flashing magenta-green
14	142	flashing cyan-red
15	143	flashing white-black

Note that the actual colours referred to by the logical colours may be changed by using `VDU 19`. The actual colour numbers are the same as the default logical colours in `MODE 2`.

Reference

Section 8.3 and chapter 11.

Syntax



Examples

```
COLOUR 3

  110 COLOUR 5
```

Associated keywords

`GCOL  VDU 19`

CONT Continue CO.

Description

`CONT` resumes execution of a program after it has been halted by either a `STOP` command, a run-time error, or ESCAPE being pressed.

Execution recommences with the line on which the error occurred or, in the case of a `STOP` command, from the following line.

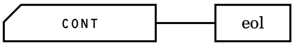
If the program is altered, or an error is generated while in direct mode, then continued execution may not be possible, in which case a `Can't CONT` error is generated.

Note that it is not possible to `CONT`inue if a procedure or function has been called while in direct mode.

Reference

Section 12.2.

Syntax



Example

```
CONT
```

Associated key word

`STOP`

COS Cosine

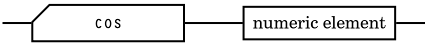
Description

`COS` returns the cosine of the argument, which is taken to be in radian measure. `RAD` may be used to convert degrees to radians.

Reference

Section 33.1.

Syntax



Examples

```
PRINT COS (RAD (30))

   15 adj := hyp * COS (angle)
```

Associated keywords

`ACS ASN ATN SIN TAN DEG RAD`

COUNT

COU.

Description

`COUNT` is a system function which returns the number of characters sent using a `PRINT` command since the last carriage return. Note that `VDU` commands do not alter the value of `COUNT`.

Reference

Section 18.4.

Syntax

Examples

```
PRINT COUNT

  115 no_of_characters := COUNT
```

Associated keywords

```
POS  TAB  SELECT OUTPUT
```

DATA

DA.

Description

`DATA` is used at the beginning of a program line to indicate that the information which follows is to be assigned using a `READ` statement. Data items are separated by commas.

Leading or trailing spaces on numeric data items are ignored.

String data items may or may not be enclosed in quotes, but they are only necessary if the string itself includes commas. However, it is advisable to put the string in quotes if the string itself contains quotation marks, in which case double quotes (`""`) should be used (see example). If quotes are to be used round data items then there should be no spaces between the preceding comma and the opening quotation mark.

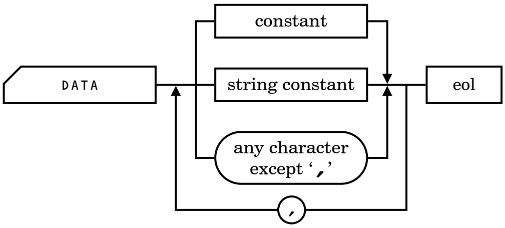
If quotes are not used round string items then all the characters between the preceding and following commas are taken to be part of the string, including leading and trailing spaces.

Note that it is only possible to read 'hexadecimal' data items such as `&FFE0` as strings.

Reference

Chapter 17.

Syntax



Examples

```
265 DATA 1,4,9,16,25,36,49

270 DATA """one""","quote "" mark",Vogon,petunias
```

Associated keywords

`READ RESTORE`

DEBUG

Description

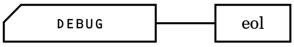
DEBUG reports structural errors in a program, such as unclosed structures, crossed loops and non-existent labels. It is performed automatically before programs are RUN.

Note that DEBUG clears all user variables.

Reference

Section 7.3.

Syntax



Example

```
DEBUG
```

Associated keywords

LIST RUN

DEG Radians to degrees

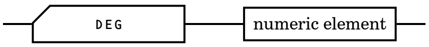
Description

DEG converts angles from radian measure to degrees. 1 radian is (180/PI degrees (ie about 57.3 degrees).

Reference

Section 33.1.

Syntax



Examples

```
PRINT DEG 1

  100 ninety := DEG (PI/2)
```

Associated keyword

RAD

DEL Delete program lines

Description

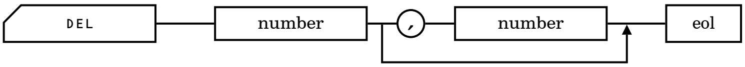
`DEL` deletes the program lines specified by one or two arguments. As a safety measure, two parameters are necessary to delete more than one line. No action results if the second parameter is lower than the first, but otherwise all the program lines between the two arguments are deleted (inclusively).

`DEL` followed by a single parameter will delete one program line. Typing the line number followed by RETURN will have the same result.

Reference

Section 5.10.

Syntax



Examples

```
DEL 10,30
```

```
DEL 25
```

Associated keywords

`LIST OLD NEW`

DELETE Delete file DEL.

Description

This command deletes the specified file on the current filing system.

It will not work on cassette files, but will not generate an error if this is attempted. Trying to delete a file which does not exist will also not give an error message.

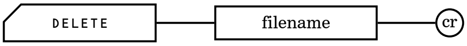
Note that attempts to use keywords in specifying filenames without using quotation marks will not produce the normal results since they are not recognised as keywords (eg `DELETE STR$2` will delete the file `str$2`).

Note also that if a string variable name is used in specifying the filename then it must be in lower case.

Reference

Section 14.7.

Syntax



Examples

```
DELETE myprog

   10 DELETE fname$
```

Associated keywords

`SAVE LOAD RUN`

DIM Dimension DI.

Description

Both numeric and string arrays must be declared before the first assignment to them is made. Additionally, the maximum length of strings may be declared using `OF` if the default value of 40 characters is not suitable. `DIM` performs these operations.

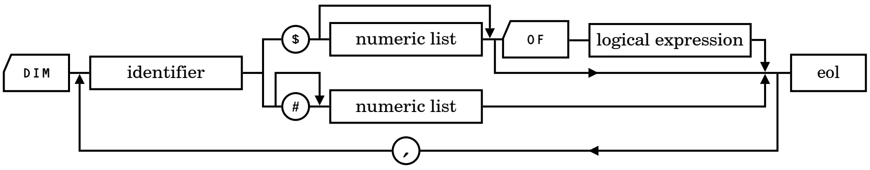
For arrays, upper and lower bounds may be placed on each dimension. The lower bound defaults to 0, but the upper bound must always be specified. If the lower bound is declared to be greater than the upper bound then a run-time `Bad DIM` error results.

The maximum number of dimensions allowed in any array is eight. The maximum length permitted for strings is 253 characters.

Reference

Sections 4.5, 19.2.

Syntax



Examples

```
DIM name$ OF 12

   25 DIM image(-3:4,3:8,-3:-2)
   35 DIM output$(1:5) OF 20
   45 DIM a$ OF 5, b$ OF 10
   55 DIM c$(a,b#) OF c
   65 DIM d#(6,p)
```

Associated keyword

`OF`

DIV Integer division

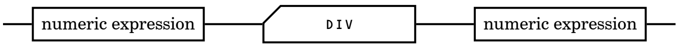
Description

DIV is a binary operator which truncates the two operands to integers before dividing the first by the second. The result is also truncated to an integer.

Reference

Section 18.8.

Syntax



Examples

```
PRINT 17 DIV 3

   55 eleven# := 44.1 DIV 4.9
```

Associated keyword

MOD

DO

Description

`DO` is used in the `FOR` and `WHILE` structures, it is optional on entry except in a few cases, such as:

```
FOR loc = &7C00 TO &8000 DO ?loc = 0
```

because `&8000 ? loc=0` is a valid logical expression.

Reference

`FOR` and `WHILE`.

Syntax

See `FOR` and `WHILE`.

Examples (see also `FOR` and `WHILE`)

```
WHILE GET <> 32 DO example_screen
   55 FOR i := 1 TO 10 DO PRINT i
   75 WHILE waiting DO NULL
```

Associated keywords

```
FOR WHILE
```

DRAW

Description

This facility enables lines to be drawn on the screen in any of `MODE`s 0, 1, 2, 4 or 5. The line is drawn in the current graphics foreground colour from the current position of the graphics cursor to the point specified by the x- and y-coordinates following the `DRAW` statement.

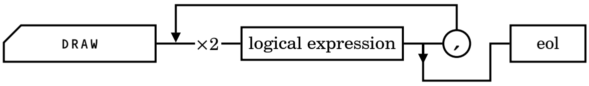
The x- and y-coordinates are relative to the current graphics origin (0,0). The default origin is at the bottom left of the graphics area, but this may be changed by using `VDU 29`. The screen is 1280 units wide (x-axis) and 1024 units high (y-axis) irrespective of which graphics mode is in use.

`DRAW x,y` is identical in effect to `PLOT 5,x,y`.

Reference

Section 11.3 and `PLOT`.

Syntax



Examples (see also `PLOT`)

```
DRAW 350,250
  635 DRAW x,y
```

Associated keywords

`MODE PLOT MOVE GCOL CLG VDU`

EDIT List program without indentation E.

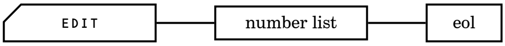
Description

EDIT lists the specified range of program lines without any indentation of structures. The parameters default to the start and end of the program. The entire program is listed if no parameters are given. Unlike LIST, EDIT displays EXEC before all procedure calls where none are already present.

Reference

Section 13.5.

Syntax



Examples

```
EDIT
```

```
EDIT 100,200
```

```
EDIT ,150
```

```
EDIT 300,
```

Associated keyword

LIST

ELIF Else if EL.

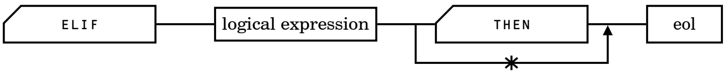
Description

`ELIF` is a statement that may be used in the long form of the `IF` structure. It declares another condition to be tested if the condition after the initial `IF` or any previous `ELIF` statement fails.

Reference

Chapter 28 and `IF`.

Syntax



Examples (see also `IF`)

```
20 IF boss_free
30   route_call(boss_phone)
40 ELIF secretary_free
50   route_call(sec_phone)
60 ELSE
70   return_engaged_tone
80 END IF
```

Associated keywords

`IF   THEN   ELSE   END IF`

ELSE

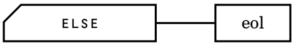
Description

This statement is used in the long form of the `IF` structure. The block of program lines following it is executed if none of the `IF` or `ELIF` conditions earlier in the structure is satisfied.

Reference

Section 15.6 and `IF`.

Syntax



Examples (see also `IF`)

```
30 IF hours < 12 THEN
40   greet("Good Morning!")
50 ELSE
60   greet("Good Afternoon!")
70 END IF
```

Associated keywords

`IF` `THEN` `ELIF` `END IF`

END

EN.

Description

END indicates that the end of the program has been reached. It is optional and may be used as many times as required in a program, although it is not good practice to have more than one.

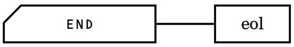
In Acornsoft COMAL the end of structure statements (END IF, END PROC etc) are each stored as two separate keywords and, therefore, require a space after END.

Note that END is not allowed inside structures, where STOP must be used to halt execution.

Reference

Section 5.13.

Syntax



Example

```
1000 END
```

Associated keyword

STOP

END CASE

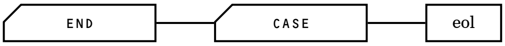
Description

This indicates the end of a `CASE` structure. Note that the space between `END` and `CASE` is obligatory.

Reference

Chapter 25 and `CASE`.

Syntax



Examples (see also `CASE`)

```
825 CASE number OF
830 WHEN <0
835   PRINT "Negative"
840 WHEN 0
850   PRINT "Zero"
860 OTHERWISE
870   PRINT "Positive"
880 END CASE
```

Associated keywords

`CASE WHEN OTHERWISE`

END FUNC

Description

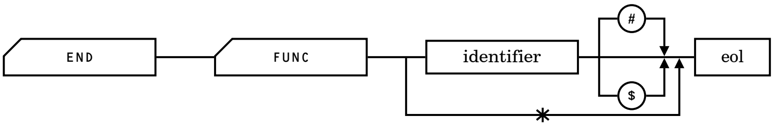
This indicates the end of a FUNCtion declaration. Note that the space between END and FUNC is obligatory.

The function name is optional on entry, and will be inserted automatically if omitted.

Reference

Chapter 30 and FUNC.

Syntax



Examples (see also FUNC)

```
355 FUNC fib(n)
360   IF n < 3 THEN
365     RETURN 1
370   ELSE
375     RETURN fib(n-1)+fib(n-2)
380   END IF
385 END FUNC fib
```

Associated keywords

FUNC CLOSED REF IMPORT RETURN

END IF

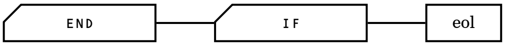
Description

This indicates the end of a long form `IF` structure. Note that the space between `END` and `IF` is obligatory.

Reference

Section 15.5.

Syntax



Examples (see also `IF`)

```
635 IF temp > max_temp THEN
640   PRINT "*OVERHEATING*"
650   turn_off_heater
660 END IF
```

Associated keywords

`IF THEN ELSE ELIF`

END PROC

Description

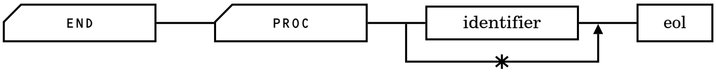
This indicates the end of a PROCedure declaration. Note that the space between END and PROC is obligatory.

The procedure name is optional on entry, and will be inserted automatically if omitted.

Reference

Section 13.1 and PROC.

Syntax



Examples (see also PROC)

```
425 PROC laim(message$)
430   enable_voice_synthesiser
440   say(message$)
450   disable_voice_synthesiser
460 END PROC laim
```

Associated keywords

PROC CLOSED REF IMPORT

END WHILE

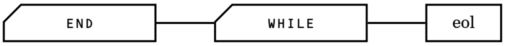
Description

This indicates the end of a long form `WHILE` structure. Note that the space between `END` and `WHILE` is obligatory.

Reference

Chapter 24 and `WHILE`.

Syntax



Examples (see also `WHILE`)

```
30 WHILE results < excellent DO
40   suggest_more_work
50   reset_test
60 END WHILE
```

Associated keywords

`WHILE DO`

ENVELOPE ENV.

Description

Note that on the Acorn Electron this function is not the same as the BBC Microcomputer version described below. Reference should be made to the *Acorn Electron User Guide* for details of `ENVELOPE` which performs exactly as in BASIC.

`ENVELOPE` is used with `SOUND` to provide greater control over the sounds produced. It is a very powerful, but somewhat complex command, and reference should be made to the *BBC Microcomputer System User Guide* for a more detailed description than the parameter list given below.

`ENVELOPE` is followed by 14 parameters as follows:

```
ENVELOPE N,T,PI1,PI2,PI3,PN1,PN2,PN3,AA,AD,AS,AR,ALA,ALD
```

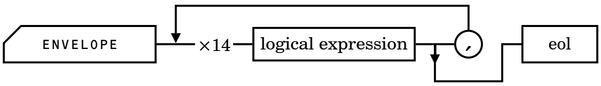
Parameter	**Range**	**Function**
N	1 to 4	Envelope number
T bits 0-6	0 to 127	Length of each step in hundredths of a second
bit 7	0 or 1	0=auto-repeat pitch envelope 1=no auto-repeat
PI1	−128 to 127	Change of pitch per step in section 1
PI2	−128 to 127	Change of pitch per step in section 2
PI3	−128 to 127	Change of pitch per step in section 3
PN1	0 to 255	Number of steps in section 1
PN2	0 to 255	Number of steps in section 2
PN3	0 to 255	Number of steps in section 3
AA	−127 to 127	Change of amplitude per step during attack phase
AD	−127 to 127	Change of amplitude per step during decay phase
AS	−127 to 0	Change of amplitude per step during sustain phase
AR	−127 to 0	Change of amplitude per step during release phase
ALA	0 to 126	Target of level at end of attack phase
ALD	0 to 126	Target of level at end of decay phase

Note that if the RS423 and cassette buffers are not being used then envelope numbers in the range 1 to 16 are allowed.

Reference

Section 33.2 and the *BBC Microcomputer System User Guide*.

Syntax



Examples (see also the *BBC Microcomputer System User Guide*)

```
ENVELOPE 2,10,5,0,-10,5,5,15,50,0,-10,-15,125,125

  100 ENVELOPE 4,133,0,0,0,35,0,0,125,-5,0,-10,100,60
```

Associated keywords

```
SOUND ADVAL
```

EOD End of data

Description

EOD is a system function which indicates whether the end of the available data statements has been reached. While there is still data in the program that can be read it returns FALSE (0). After reading the last available data item it returns TRUE (−1).

Reference

Section 17.3.

Syntax

Examples

```
10 a:=0
20 WHILE NOT EOD
30   READ array(a)
40   a:+1
50 END WHILE
```

Associated keywords

DATA READ

EOF End of file EO.

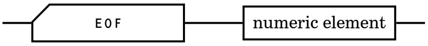
Description

EOF is a function which shows whether the end of a file has been reached. It is followed by a single argument indicating the number of the file which is to be checked. EOF returns TRUE (−1) if no more data can be successfully read from the file, and FALSE (0) otherwise. If no file is open with the specified number then a Not open error is generated.

Reference

Section 32.9.

Syntax



Examples

```
 80 OPEN FILE 1,"Test",READ
 90 i:=0
100 REPEAT
110   READ FILE 1:a(i)
120   i:+1
130 UNTIL EOF 1
140 CLOSE FILE 1
```

Associated keywords

OPEN FILE READ RANDOM READ ONLY CLOSE

EOR Exclusive-OR operator EOR

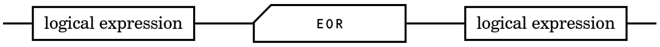
Description

EOR is a logical operator which performs a bitwise exclusive-OR on the two operands, ie it looks at each bit of the binary representations of the two operands in turn and, for each of them, sets the corresponding result bit to 1 if one or the other is set, but to 0 if they both have the same value (giving a 32-bit number).

Reference

Section 21.3.

Syntax



Examples

```
310 IF a <> FALSE EOR b <> FALSE THEN
320   PRINT "Either a or b, but not both"
330 ELSE
340   PRINT "Either both a and b, or neither"
350 END IF
```

Associated keywords

AND OR NOT

EXEC Execute procedure EX.

Description

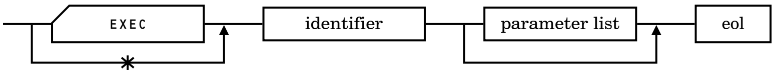
`EXEC` is used to call procedures. It is always optional on entry, and if it is omitted then an ‘invisible’ `EXEC` is inserted. `EDIT` shows invisible `EXEC`s, but `LIST` does not.

Procedures may be called from direct mode, but only if the program has been `LIST`ed, `EDIT`ed, `DEBUG`ged or `RUN`.

Reference

Section 13.6.

Syntax



Example

```
EXEC myproc
```

Associated keyword

`PROC`

EXP Exponent

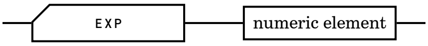
Description

EXP is a mathematical function which returns e (2.718281828) raised to the power of the argument.

Reference

Section 33.1.

Syntax



Examples

```
PRINT EXP (3.5)

  175 y := EXP (x)
```

Associated keywords

LN LOG

EXT Extent

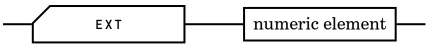
Description

`EXT` is a function which returns the length in bytes of the file with the file number specified by the argument. If there is no file open with that number then a `Not open` error is generated.

Reference

Section 32.17.

Syntax



Examples

```
PRINT EXT 2

   35 size_of_file := EXT 1
```

Associated keywords

`OPEN FILE CLOSE`

FALSE

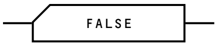
Description

`FALSE` is a system function which returns the value 0.

Reference

Section 16.3.

Syntax



Example

```
   65 UNTIL FALSE // infinite loop
```

Associated keyword

`TRUE`

FILE

Description

FILE is used as a qualifying keyword to indicate that the expression which follows is the file number to which the qualified keyword is to be applied. When a file is OPENed the file is assigned to a number given by the expression following the FILE statement. All operations on that file until it is next CLOSEd refer to it by this number. See the file handling commands (OPEN, CLOSE, WRITE, READ) for further details of the use of FILE.

Reference

Chapter 32.

Syntax

See OPEN and CLOSE.

Examples

Refer to the keyword which is being qualified.

Associated keywords

```
OPEN CLOSE WRITE READ PRINT INPUT
```

FOR

Description

FOR is used to initialise an unconditional loop structure.

There are two forms: the short form (single line), and the long form (multi-line). In both cases FOR is followed by a loop variable assignment which has lower and upper bounds separated by a TO statement. Additionally there may then be a STEP statement to indicate the loop variable increment. The default value is 1.

In the short form this is followed by the simple statement to be repeated.

If more than one command is to be executed in the loop then each must be entered on the lines following the FOR statement, and, finally, a NEXT statement entered to indicate the end of the loop.

In both cases a DO statement is automatically inserted if not present at entry.

Note that if the values taken by a loop variable are always going to be whole numbers then it will make execution much faster if an integer variable is used.

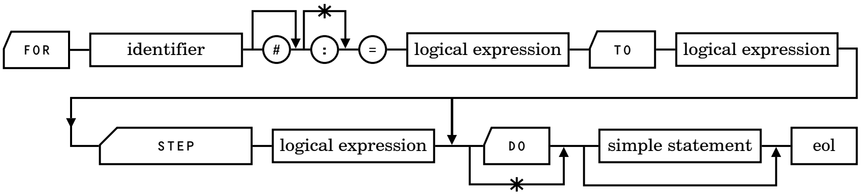
Note also that if the start and end values of the loop variable and the increment are incompatible (eg `FOR i := 1 TO 10 STEP -1`) then the loop is not performed at all and execution moves to the line after the end of the FOR structure.

It is advisable, if possible, to use integral step sizes and divide down the loop variable if required, since inaccuracies may result with fractional step sizes.

Reference

Chapter 6.

Syntax



Examples

Short form:

```
FOR i# := 1 TO 10 DO PRINT i#
```

```
   10 FOR i# := 5 TO 11 STEP 2 DO READ a(i#)
```

Long form:

```
   10 FOR j := 2 TO 10 DO
   20   PRINT j
   30   myproc(j2)
   40 NEXT j
```

Associated keywords

```
TO STEP DO NEXT
```

FREE Free memory FR.

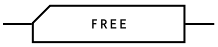
Description

`FREE` is a system function which returns the number of bytes of free memory available.

Reference

Section 34.1 and memory map in section 34.3.

Syntax



Examples

```
PRINT FREE
```

```
  100 FUNC legal_change(new_mode)
  110   RESTORE mode_size_list
  120   FOR m := 0 TO new_mode DO READ new_mode_size
  130   RESTORE mode_size_list
  140   FOR m := 0 TO MODE DO READ old_mode_size
  150   IF (new_mode_size - old_mode_size) >= FREE THEN
  160     RETURN FALSE
  170   ELSE
  180     RETURN TRUE
  190   END IF
  200 END FUNC legal_change
  210 mode_size_list:
  220 DATA 20480,20480,20480,16384,10240,10240,8192,1024
```

Associated keywords

SIZE PAGE

FUNC Function FU.

Description

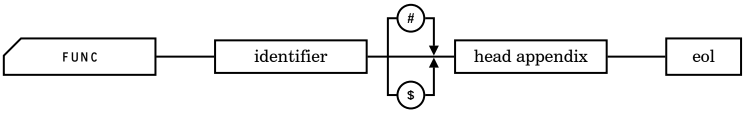
FUNC is the first word used on a function declaration line. It is followed by the name which identifies the function, the formal parameter list (if any) and, possibly, CLOSED to denote a closed (local) function. The body of the function is between this line and the next END FUNC statement, and must contain a RETURN statement to indicate the value to be returned by the function.

FUNC can only appear at the start of a program line and not inside any other structures.

Reference

Chapter 30.

Syntax



Examples (see also chapter 30)

```
235 FUNC factorial(number#)
240   IF number# > 1 THEN
245     RETURN number# * factorial(number#-1)
250   ELIF number# >= 0
255     RETURN 1
260   ELSE
265     RETURN 0
270   END IF
275 END FUNC factorial
```

Associated keywords

CLOSED REF IMPORT END FUNC RETURN PROC

GCOL Graphics colour G.

Description

`GCOL` sets the logical colours to be used for the background and foreground of the graphics area in all subsequent operations.

Two arguments follow the `GCOL` command. The first specifies the method of displaying the colour on the screen, as follows:

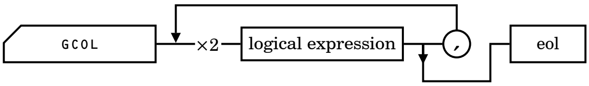
0	plot the specified colour
1	`OR` the specified colour with that already there
2	`AND` the specified colour with that already there
3	`EOR` the specified colour with that already there
4	invert the colour already there

The second argument specifies the logical colour to be used. Numbers less than 128 define the foreground colour, while those greater than 127 define the background colour. Note that although the second argument in `GCOL 4,x` has no effect on the result a number must be entered.

Reference

Section 11.5 and `COLOUR`.

Syntax



Examples (see also chapter 11)

```
GCOL 1,131

   95 GCOL 0,4
```

Associated keywords

`COLOUR CLG MODE PLOT DRAW MOVE VDU`

GET

Description

GET is used to make execution wait until a key is pressed. It returns the ASCII code of the key pressed, but the character typed is not printed on the screen. See section 40.1 for details of ASCII codes.

Note that GET reads the buffer corresponding to the current input stream. Thus, the key may be pressed prior to the GET being executed provided that some other command does not remove it from the buffer.

If the input stream is changed (*FX 2 command or *EXEC) then GET may be used to take characters from sources other than the keyboard.

Reference

Section 9.5.

Syntax

Examples

```
PRINT CHR$ GET

   35 WHILE GET <> 32 DO NULL
```

Associated keywords

GET$ INKEY INKEY$ INPUT

GET$

GE.

Description

`GET$` makes execution wait until a key is pressed. It returns a string containing the key pressed, but the character is not displayed on the screen.

Note that `GET$` reads the buffer corresponding to the current input stream. Thus, the key may be pressed prior to the `GET$` being executed provided that some other command does not remove it from the buffer.

If the input stream is changed (`*FX 2` command or `*EXEC`) then `GET$` may be used to take characters from sources other than the keyboard.

Reference

Section 9.6.

Syntax

Examples

```
q1$ := GET$
```

```
   25 q2$ := GET$
   30 PRINT q2$
```

Associated keywords

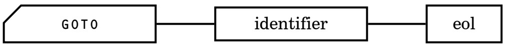
`GET INKEY INKEY$ INPUT`

GOTO

Description

GOTO causes program execution to jump unconditionally to a named label (*not* a line number). See RESTORE for details of label definition. Note that prolific use of GOTOs tends to destroy the structure of a program and is not recommended.

Syntax



Examples

```
10 i := 0
20 again: //label
30 i:+1
40 PRINT "This is appalling style"
50 GOTO again
```

Associated keywords

None.

IF

Description

The `IF` structure is used to make the execution of some program lines dependent on whether some logical expression is true. It can be used in two forms, the short form and the long form.

The short form is a single line structure allowing only a simple statement (or a multiple assignment) to be executed if the condition is true.

The long form allows the block of lines following the `IF` statement to be dependent on it, and also allows the `ELIF` and `ELSE` commands to be used. These introduce other blocks of statements to be performed if the original condition in the `IF` statement is not met. The long form `IF` is terminated by an `END IF` statement.

Note that `THEN` is optional on entry, except in a few cases, such as:

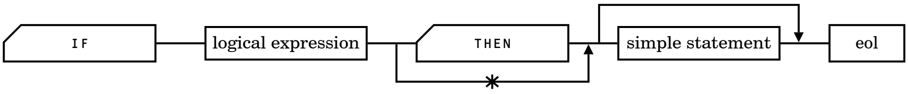
```
IF x = 10 THEN ? y = x
```

where `10 ? y = x` would be a valid expression.

Reference

Sections 15.3, 15.5.

Syntax



Examples (see also chapter 15)

```
IF x < 0 THEN PRINT "Negative"

   30 IF x = 1 THEN
   40   MOVE x,y
   50 ELSE
   60   DRAW x,y
   70 END IF
```

Associated keywords

`END IF  ELSE  ELIF  THEN`

IMPORT

Description

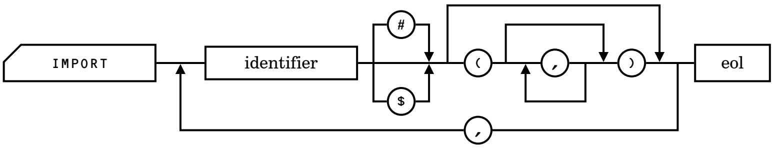
`IMPORT` is used to declare which global variables are accessible from inside a closed procedure or function. This is equivalent to passing these variables by reference to the procedure or function, in that assignments to them inside the structure will change their values globally.

Note that `IMPORT` may be used anywhere within a structure, provided that a variable with the same name as that to be `IMPORT`ed does not already exist within the structure.

Reference

Section 31.6.

Syntax



Examples (see also `PROC` and chapter 31)

```
35 IMPORT a,b#,name$,array(,,)
```

Associated keywords

`PROC FUNC CLOSED REF`

IN

Description

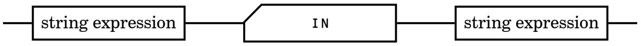
`IN` is a binary operator which returns the position of the first string in the second.

If no match is found then `IN` returns 0. If the first argument is a null string then it returns 1.

Reference

Section 4.3.

Syntax



Examples

```
PRINT "345" IN "12345"
```

```
  145 DIM a$ OF 10
  150 REPEAT
  155   PRINT "Answer Yes or No >>"
  160   a$ := GET$
  165 UNTIL a$ IN "YNyn"
```

Associated keywords

None.

INKEY

Description

INKEY has two forms depending on whether the argument following it is positive or negative.

With a positive argument INKEY is used to halt program execution for up to a specified time whilst waiting for a key to be pressed. If a key is pressed within the specified time then INKEY immediately returns the ASCII code of the key pressed (see section 40.1 for ASCII codes). Otherwise it returns -1 and program execution continues. The time for which INKEY waits is specified by the argument, in hundredths of a second (in the range 0 to 32767). INKEY reads the keyboard buffer so the key may be pressed at any time before the statement occurs, so long as it is not removed from the buffer by some other command in the mean time.

With a negative argument INKEY looks at the keyboard (not the buffer) to see whether a particular key is pressed at the instant the command is invoked. The argument determines which key is to be tested according to the table at the end of this keyword. Note that the codes for the Acorn Electron are different in a few cases, so reference should be made to the *Acorn Electron User Guide* where appropriate.

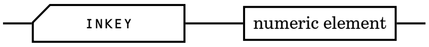
Known problem

In the first release of Acornsoft COMAL for the BBC Microcomputer (but not for the Acorn Electron), INKEY needs arguments in the range -255 to 255 to give the expected result. In order to achieve the correct results with values outside this range, for example, to perform INKEY &PQRS (where P, Q, R and S are any hexadecimal digits), INKEY &PQ0000RS should be used. See chapter 42 for an example function to perform the required manipulation.

Reference

Sections 20.1, 20.4.

Syntax



Examples

```
PRINT INKEY (200)

   75 x := INKEY (100)
```

Associated keywords

```
INKEY$ GET GET$
```

Key	**Number**	**Key**	**Number**
f0	−33	1	−49
f1	−114	2	−50
f2	−115	3	−18
f3	−116	4	−19
f4	−21	5	−20
f5	−117	6	−53
f6	−118	7	−37
f7	−23	8	−22
f8	−119	9	−39
f9	−120	0	−40
A	−66	-	−24
B	−101	^	−25
C	−83	\	−121
D	−51	@	−72
E	−35	[	−57
F	−68	_	−41
G	−84	;	−88
H	−85	:	−73
I	−38	]	−89
J	−70	,	−103
K	−71	.	−104
L	−87	/	−105
M	−102	ESCAPE	−113
N	−86	TAB	−97
O	−55	CAPS LOCK	−65
P	−56	CTRL	−2
Q	−17	SHIFT LOCK	−81
R	−52	SHIFT	−1
S	−82	SPACE BAR	−99
T	−36	DELETE	−90
U	−54	COPY	−106
V	−100	RETURN	−74
W	−34	↑	−58
X	−67	↓	−42
Y	−69	←	−26
Z	−98	→	−122

INKEY$

Description

`INKEY$` returns a string containing the key pressed within the time specified by the argument. It scans the keyboard buffer rather than the keyboard itself.

If no key is pressed within the specified time limit then a null string is returned. The argument specifies the time limit in hundredths of a second (in the range 0 to 32767).

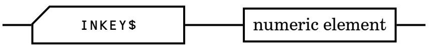
Known problem

In the first release of Acornsoft COMAL for the BBC Microcomputer (but not for the Acorn Electron), `INKEY$` needs arguments in the range 0 to 255 to give the expected result. In order to achieve the correct results with values outside this range, for example, to perform `INKEY$ &PQRS` (where `P`, `Q`, `R` and `S` are any hexadecimal digits), `INKEY$ &PQ0000RS` should be used. See chapter 42 for an example function to perform the required manipulation.

Reference

Section 20.2.

Syntax



Examples

```
PRINT INKEY$ (100)

   10 key$ := INKEY$ (150)
```

Associated keywords

`INKEY GET GET$`

INPUT

Description

This statement takes data from the current input stream and assigns it to a variable. If there is no message then a question mark prompt is generated. A prompt message may immediately follow the `INPUT` command, in which case no additional prompt is supplied.

If insufficient data is supplied then a `??` prompt is generated for more. If string data is supplied when numeric is required then `Bad value` is printed and `??` prompts for re-entry.

If the string supplied is longer than the maximum length possible for the string variable to which it is to be assigned, then a `String too long` error is reported and execution stops. If any other error occurs then input starts again, including the prompt, and from the first item in the list.

`INPUT FILE` may be used to input data from files and is identical to `READ FILE`.

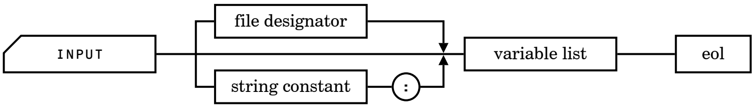
Known problem

In the first release of Acornsoft COMAL any string beginning with a valid numeric constant (including `.`) will be accepted when a numeric input is required and taken as having the value of that constant (as in BBC BASIC). No error is generated.

Reference

Section 9.1.

Syntax



Examples (see also `READ FILE`)

```
235 INPUT "Please enter your choice >> ":choice$

125 INPUT a,b#,c$
```

Associated keywords

`GET GET$`

INT Integer part

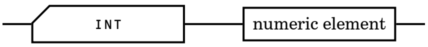
Description

INT returns the largest whole number equal to or less than the argument. For example, INT (2.5) is 2, but INT (-2.5) is -3.

Reference

Section 3.11.

Syntax



Examples

```
PRINT INT (-15.36)

  100 integer# := INT real_number

  200 minus_nine := INT -8.7
```

Associated keywords

None.

LEN Length of string LE.

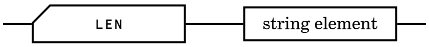
Description

LEN returns the number of characters in the string argument which follows it.

Reference

Section 4.2.

Syntax



Examples

```
PRINT LEN (x$)
```

```
 1100 no_of_characters := LEN (test$)
```

Associated keywords

None.

LIST .

Description

This command lists the program from the line given by the first argument to that given by the second. The two arguments default to the beginning and end of the program. If only one argument is given then just that line is listed.

Line numbers are ranged right in a five character field and are followed by a space. Two extra spaces are inserted after the start of each structure and removed before its last line. Keywords are given in upper case and variables in lower case, although both may be entered in either case.

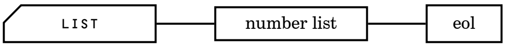
CTRL N and CTRL O will switch 'paged mode' listing on and off respectively, with SHIFT to move to the next page.

Pressing SHIFT and CTRL together will cause scrolling to stop until one of them is released.

Reference

Section 5.4.

Syntax



Examples

```
LIST

LIST 100,200

LIST ,50

LIST 300,

LIST 75
```

Associated keywords

```
EDIT NEW OLD
```

LN Natural logarithm

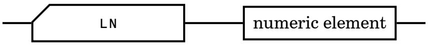
Description

LN returns the logarithm to the base e (the Natural or Napierian logarithm) of the argument.

Reference

Section 33.1.

Syntax



Examples

```
PRINT LN (0.5)

  325 half_life := LN (2) / decay_constant

  135 delta_g := -gas_const*abs_temp*LN(equil_const)
```

Associated keywords

EXP LOG

LOAD

LO.

Description

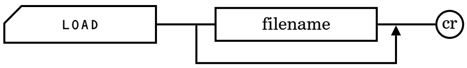
`LOAD` loads a program from the current filing system. Quotes round the filename are optional (except where a keyword is used as a filename). Any string expression may be used as the argument. The maximum length of the filename is dependent on the filing system in use.

Note that if a string variable is used in specifying the filename then it must be in lower case.

Reference

Sections 14.4 and 14.6.

Syntax



Examples

```
LOAD "myprog"
```

```
LOAD myprog
```

```
a$ := "my"
LOAD a$ + "prog"
```

Associated keywords

`SAVE RUN`

LOG Common logarithm

Description

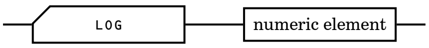
`LOG` returns the common logarithm (ie to the base 10) of the argument. Anti-logarithms are calculated using:

```
a_log := 10^logarithm
```

Reference

Section 33.1.

Syntax



Examples

```
PRINT LOG (2)
```

```
  465 ph := - LOG (h_plus_conc)
```

Associated keywords

```
LN EXP
```

MOD Modulus MOD

Description

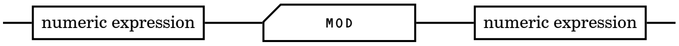
`MOD` is a binary operator which returns the signed remainder obtained by dividing the first argument by the second. Both arguments are first truncated to integers, as is the result.

Note that, for example, `-14 MOD 5` is −4.

Reference

Section 18.9.

Syntax



Examples

```
PRINT (TIME/100) MOD 60

  365 remainder# := x MOD y
```

Associated keyword

`DIV`

MODE

MOD.

Description

`MODE` is a system variable containing the current screen mode which may be read as well as assigned. Note, however, that `MODE` may only be altered by using an assignment statement and not, for example, by using `INPUT`.

It is not possible to change mode inside a procedure or function unless using a 6502 Second Processor. Changing modes clears the screen. Attempting to change to a mode for which there is insufficient memory available will result in a `No room` error.

The `MODE`s available are as follows.

Mode	Graphics	Colours	Text	Memory used
0	640 × 256	2 colour display	80 × 32 text	20K
1	320 × 256	4 colour display	40 × 32 text	20K
2	160 × 256	16 colour display	20 × 32 text	20K
3		2 colour text only	80 × 25 text	16K
4	320 × 256	2 colour display	40 × 32 text	10K
5	160 × 256	4 colour display	20 × 32 text	10K
6		2 colour text only	40 × 25 text	8K
7		Teletext display	40 × 25 text	1K

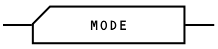
On Model A computers with only 16K of memory `MODE`s 4, 5, 6 and 7 may be selected.

On Acorn Electrons `MODE` 7 is not available and attempting to enter this mode will give `MODE` 6 instead.

Reference

Section 8.1.

Syntax



Examples

```
MODE := 6
```

```
PRINT MODE
```

```
  115 MODE :+ 2
```

Associated keywords

```
CLS CLG FREE
```

MOVE

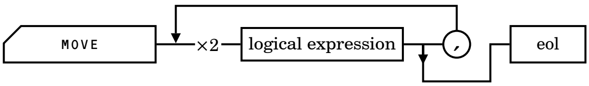
Description

`MOVE` moves the graphics cursor to the position specified by the two arguments. These will be the x- and y-coordinates relative to the current graphics origin. `MOVE` is identical in effect to `PLOT 4`.

Reference

Section 11.4.

Syntax



Examples (see also `PLOT 4`)

```
MOVE 200,100
```

```
  375 MOVE next_x, next_y
```

Associated keywords

```
DRAW PLOT VDU CLG
```

NEW

Description

NEW clears the program and variables (except the system integer variables a# to z#) from the memory and resets the relevant system functions (SIZE and FREE).

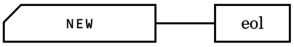
The program (but not the variables) may be retrieved by typing OLD provided that it has not been corrupted.

Note that NEW will not close any files which are open on the current filing system. This is to enable the user to employ *EXEC files which contain the keywords NEW or *COMAL. Correct use of OPEN and CLOSE statements will ensure that no problems result from this feature.

Reference

Section 5.1.

Syntax



Example

```
NEW
```

Associated keyword

OLD

NEXT

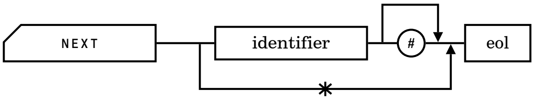
Description

NEXT declares the end of a FOR loop. The loop variable is optional on entry and is automatically inserted if omitted. NEXT is not used in short form loops.

Reference

Section 6.1 and FOR.

Syntax



Examples (see also FOR)

```
    75 NEXT i
```

Associated keywords

FOR STEP DO

NOT

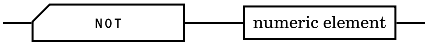
Description

NOT performs a logical bitwise inversion on its argument. It is often used to reverse the result of some test in a conditional structure, but may also be used for numerical calculations.

Reference

Section 21.6.

Syntax



Examples

```
minus_six := NOT 5

  790 IF NOT (q1 = 17 AND q2 = -3) THEN PRINT "Again"
```

Associated keywords

AND OR EOR

NULL

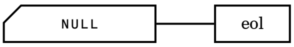
Description

`NULL` is an operation which has no effect, but is useful in, for example, delay loops or elsewhere to indicate that no action results.

Reference

Section 10.6.

Syntax



Example

```
495 WHILE INKEY (-1) DO NULL
```

Associated keywords

None.

OF

Description

`OF` is a qualifying keyword used in `DIM` statements and formal parameter lists to declare the amount of memory to be reserved for strings if the default of 40 characters is not suitable. The maximum length of string allowed is 253 characters.

`OF` is also used for syntactic completeness in `CASE` statements, but may always be omitted on entry.

Reference

Section 4.5, `DIM` and `CASE`.

Syntax

See `DIM` and `CASE`.

Examples (see also `DIM` and `CASE`)

```
  615 DIM a$(10,5) OF b
```

Associated keyword

`DIM`

OLD

O.

Description

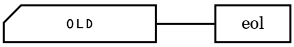
`OLD` recovers a program which has been deleted using `NEW` or by pressing the BREAK key. It will only work if no program lines have been entered and no new variables created since the program was deleted.

A `Bad program` error will result if `OLD` fails and `NEW` should be typed to clean up the memory. If the first line of the program is numbered greater than 255 then `OLD` will result in this line only being given the wrong number.

Reference

Section 5.8.

Syntax



Example

```
OLD
```

Associated keyword

`NEW`

OPEN

Description

OPEN opens a file with the specified name on the current filing system. The file is later referred to by the number following FILE which may be in the range 0 to 5.

After the filename is the file type specifier, which is one of: READ, WRITE, APPEND or RANDOM.

READ indicates that a previously created sequential file is to be opened for reading.

WRITE indicates that a sequential file is to be created for writing. Care should be taken since any unlocked file of the same name will be overwritten.

APPEND opens a previously created sequential file to write more data on to the end.

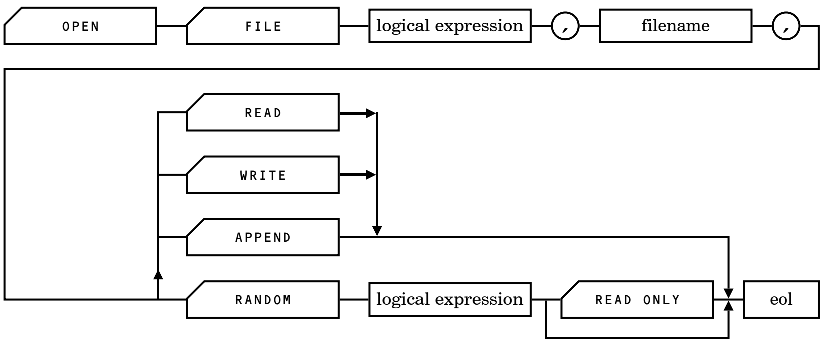
RANDOM opens a random access file, and must be followed by the record length used in the file (which must be in the range 0 to 65535). This may then be followed by READ ONLY which allows a locked file to be read.

Note that the format used for storing data in files is identical to that used by BBC BASIC. Sequential files are, therefore, directly compatible.

Reference

Sections 32.6, 32.13.

Syntax



Examples (see also chapter 32)

```
OPEN FILE 3, "myfile", APPEND
```

```
OPEN FILE 1, "another", RANDOM 26 READ ONLY
```

```
  100 OPEN FILE f#, fname$, RANDOM rec_len
```

Associated keywords

```
FILE  CLOSE  READ  WRITE  APPEND  RANDOM  READ ONLY
```

OR

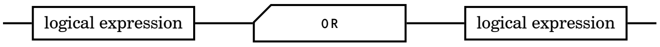
Description

`OR` is a logical operator which performs a bitwise OR on the two operands, ie it looks at each bit of the binary representations of the two operands in turn and, for each of them, sets the corresponding result bit to 1 if either or both of the two bits are set and to 0 otherwise (giving a 32-bit number).

Reference

Section 21.1.

Syntax



Examples

```
PRINT 5 OR 7
```

```
  710 IF a OR b THEN
  720   PRINT "Either a or b or both"
  730 ELSE
  740   PRINT "Neither a nor b"
  750 END IF
```

Associated keywords

`AND EOR NOT`

ORD Return ASCII value ORD

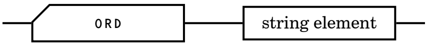
Description

`ORD` returns the ASCII code of the first character of the argument, which must be a string expression. If the argument is a null string then `-1` will be returned. See section 40.1 for a table of ASCII codes.

Reference

Section 7.5.

Syntax



Examples

```
number_65 := ORD ("A")
```

```
PRINT ORD ("g")
```

```
  195 code# := ORD (trial$)
```

Associated keyword

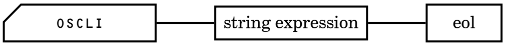
`CHR$`

OSCLI OS command line interpreter OS.

Description

`OSCLI` is a function which evaluates its string argument and passes it to the Operating System command line interpreter. The argument should evaluate to one of the OS `*` commands (without the `*`).

Syntax



Examples (see also chapter 35)

```
OSCLI "SAVE " + fname$ + " " + ~start + " " + ~end
```

```
 1805 OSCLI command$
```

Associated keywords

None.

OTHERWISE OT.

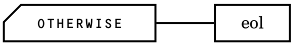
Description

OTHERWISE is used in the CASE structure to declare a block of program lines to be executed if all the previous conditional tests in the structure fail.

Reference

Chapter 25 and CASE.

Syntax



Examples (see also CASE)

```
  765 OTHERWISE
```

Associated keywords

CASE OF WHEN END CASE

PAGE

PA.

Description

PAGE is a system variable containing the address of the start of the program in memory. It may be set by the user, and its default value will depend on the particular machine configuration (disc, Tube etc). The least significant byte of PAGE is automatically set to zero so PAGE can only be adjusted by multiples of 100 bytes hex (256 decimal). NEW should be performed after adjustments to PAGE to avoid Bad program errors and to give the correct value of SIZE.

Note that PAGE may only be changed using an assignment statement and not, for example, using INPUT.

A memory map is given in section 34.3.

Reference

Section 34.1.

Syntax

Examples

```
PRINT PAGE

  395 PAGE :+ &200
```

Associated keywords

FREE SIZE

PI

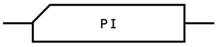
Description

`PI` is a system function which returns the value 3.141592653. It is, amongst other things, the ratio of the circumference of a circle to its diameter.

Reference

Section 3.13.

Syntax



Examples

```
PRINT SIN (PI/3)

  365 circumference := PI * diameter
```

Associated keywords

None.

PLOT

PL.

Description

`PLOT` is used to put points, lines or triangles on the graphics screen. It requires three arguments, the first determining the method of plotting to the point which has the second and third arguments as its x- and y-coordinates relative either to the current graphics origin or to the present position, depending on the first argument.

The effects corresponding to the values of the first argument are as follows:

0	move relative to last point
1	draw line relative in current graphics foreground colour
2	draw line relative in the logical inverse colour
3	draw line relative in current graphics background colour
4	move to absolute position
5	draw line absolute in current graphics foreground colour
6	draw line absolute in the logical inverse colour
7	draw line absolute in current graphics background colour

Higher values have similar effects to 0-7 as follows:

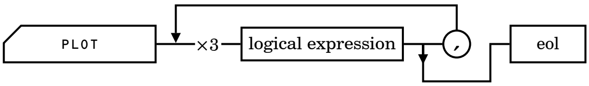
8-15	as 0-7 but with the last point in the line omitted in 'inverting actions', eg when using `GCOL 4`
16-23	as 0-7 but with a dotted line
24-31	as 0-7 but with a dotted line and without the last point on the line
32-63	reserved for the Graphics Extension ROM
64-71	as 0-7 but only a single point is plotted
72-79	as 0-7 but plot points left and right until non-background colour
80-87	as 0-7 but plotting and filling the triangle formed by the point given and the last two points visited
88-95	as 0-7 but plot points to the right until background colour
96-255	reserved for future expansion

`PLOT 4` is identical in effect to `MOVE`, and `PLOT 5` to `DRAW`.

Reference

Section 11.9.

Syntax



Examples (see also chapter 11)

```
PLOT 2,640,512

  715 PLOT 85,0,0
```

Associated keywords

`MODE MOVE DRAW CLG GCOL POINT VDU`

POINT(PO.

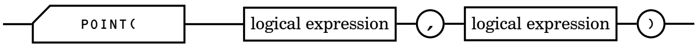
Description

POINT returns the logical colour present at the point whose x- and y-coordinates are specified by the two arguments. Brackets are obligatory round the coordinates and there must not be a space before the open bracket. If the point specified is off the screen then POINT returns -1.

Reference

Section 11.10.

Syntax



Examples

```
PRINT POINT(640,512)
```

```
  915 IF POINT(x_pos,y_pos) = 3 THEN PRINT "Zapped!"
```

Associated keywords

PLOT DRAW MOVE GCOL

POS Position

Description

POS returns the number of the screen column containing the text cursor. The left hand column is numbered 0 and the number of columns on the screen depends on the MODE selected.

Reference

Section 18.1.

Syntax

Examples

```
   65 horiz_distance := column_distance * POS
```

Associated keywords

COUNT TAB VPOS

PRINT

P.

Description

`PRINT` is used to send characters to the current output device(s). The items to be sent follow the `PRINT` statement in the 'print list'.

Items in the print list surrounded by quotes will be printed exactly as they appear in the list. Items without quotes are evaluated before printing. The format in which the items are printed depends on the separators used in the list. The screen is divided into 'zones' which are initially ten columns wide (use `ZONE` to adjust this).

Using a comma (`,`) as the separator will result in the next item being printed in the next zone after that in which the print marker currently resides unless it is already at the start of that zone. If it is a string it will be ranged left in the zone, and if it is a number it will be ranged right.

Using a semi-colon (`;`) as the separator will result in the next item being printed immediately after the previous item.

Using an apostrophe (`'`) will result in the next item being printed on a new line.

Note that to obtain a quotation mark when a string constant is `PRINT`ed it is necessary to put two quotation marks in the string (see example) or use a `CHR$ 34`.

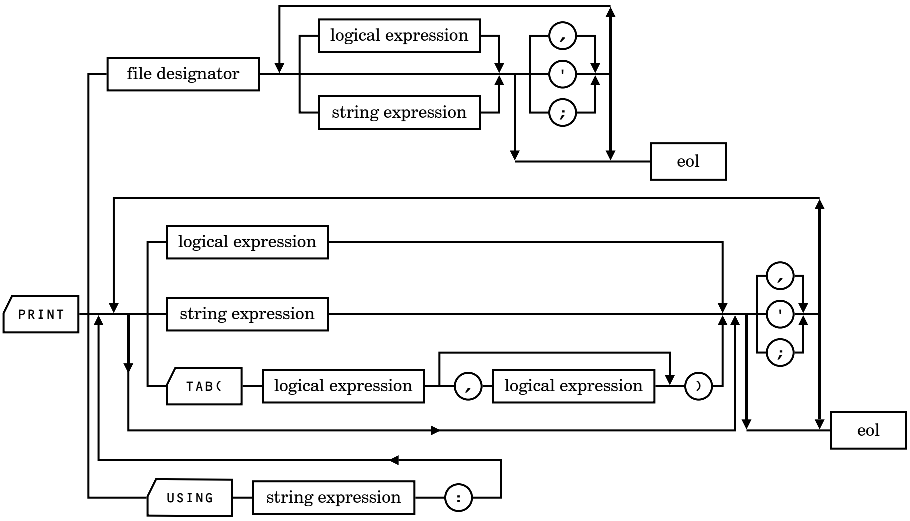
Numeric formatting is possible with `PRINT USING` where the string following `USING` is of the form `"##.####"` . The length of this string specifies the field width and the position of the decimal point indicates the accuracy to which the following numbers are to be printed. `ZONE` is a more versatile formatting command.

`PRINT FILE` is similar in operation to `WRITE FILE`, except that `PRINT FILE` may be followed by expressions rather than just a variable list. Any separators may be used between the items in the list.

Reference

Section 2.4 and `WRITE FILE`.

Syntax



Examples

```
PRINT a,b#;c$'(" "+a$)(-6:);

PRINT "Quotation "" mark"

PRINT x$(number)(3:8)

   65 PRINT ;message$(:35)'message$(36:70)

   85 PRINT FILE file_no,rec_no:i#;i#^2,"string"'c$ (4)
```

Associated keywords

TAB POS WIDTH INPUT VDU ZONE VPOS

PROC Procedure declaration PRO.

Description

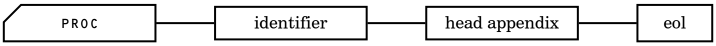
PROC is the first word used on a procedure declaration line. It is followed by the name which identifies the procedure, the formal parameter list (if any), and, possibly CLOSED to denote a closed (or local) procedure. The body of the procedure is between this line and the END PROC line.

PROC can only appear at the start of a program line, and not inside any other structures.

Reference

Chapters 13 and 31.

Syntax



Examples (see also chapters 13 and 31)

```
1010 PROC pause(centiseconds) CLOSED
1020   finish_time := TIME + centiseconds
1030   WHILE TIME < finish_time DO NULL
1040 END PROC pause
```

Associated keywords

CLOSED   REF   IMPORT   END PROC   EXEC   FUNC

RAD Radian RA.

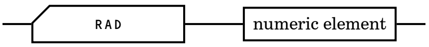
Description

RAD converts angles from degree to radian measure. 1 radian is (180/PI) degrees (ie about 57.3 degrees).

Reference

Section 33.1.

Syntax



Examples

```
PRINT RAD 30
```

```
  625 quarter_of_pi := RAD (45)
```

Associated keyword

DEG

RANDOM

RAN.

Description

`RANDOM` is a qualifying keyword used when `OPEN`ing a file for random access reading or writing. It is followed by an expression which must evaluate to the length of record used in the file. There may then be the qualifier `READ ONLY` which allows a file which is locked to be read.

The number of bytes used for file storage is as follows:

Floating point (real) numbers	6 bytes
Integers	5 bytes
Strings	(Number of characters + 2) bytes

Reference

Section 32.2 and `OPEN`.

Syntax

See `OPEN`.

Examples (see also chapter 32)

```
OPEN FILE 1, "myfile", RANDOM 30

  100 OPEN FILE a#, name$, RANDOM rec_len
```

Associated keyword

`OPEN`

READ

Description

READ may be used either to read DATA items from program lines, or to read information from files. It is also used to qualify OPEN when opening a sequential file for reading.

When used with DATA items READ assigns the data to the variables which follow it. If the data item is a not a valid number when the variable to which it is to be read is numeric then a Bad value error results. See DATA for more details of data format.

When used with files READ is followed by FILE and a numeric argument to specify the file from which the information is to be read. If the file to be read is a random access file then there may follow a numeric argument to specify the number of the record to be read. If the record number is not specified then READ reads the next item of data from the file (the first record if the file has just been OPENed). At the end of a READ statement the internal file pointer is moved to the start of the next record. This means that when using random access files, all the items in a single READ list must be contained in a single record, and that it is not possible to selectively read fields from the middle or end of a record except by reading the earlier fields into dummy variables.

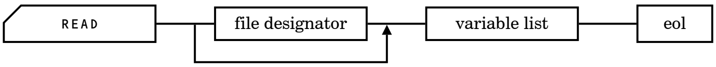
After the file designator there follows the list of variables to which the information is to be assigned, which are separated by commas. If the data to be read is not of the same type as the variable then a Bad type error is generated.

Note that attempting to read items from a sequential file which has been opened for writing or appending will generate a Not allowed error.

Reference

Section 17.2, DATA and OPEN.

Syntax



Examples (see also `OPEN` and chapter 17)

```
READ FILE 1:a$

  100 READ a,b#,c$ // from DATA statements

  165 READ FILE file_no,rec_no : name$,address$,age#

  235 READ FILE 4 : a,b,c
```

Associated keywords

```
WRITE  INPUT  FILE  OPEN  CLOSE  READ ONLY  RANDOM
```

READ ONLY

Description

READ ONLY is a qualifying statement used when OPENing a random access file so that writing to that file is not possible. It also allows a locked file to be accessed.

Note that READ ONLY will not be accepted with more than one space between the words.

Reference

OPEN and section 32.13.

Syntax

See OPEN.

Examples (see also chapter 32 and RANDOM)

```
OPEN FILE 2, "test_file", RANDOM 20 READ ONLY
```

Associated keyword

OPEN

REF

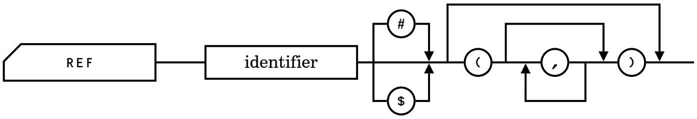
Description

REF is used in a PROCedure or FUNCtion declaration statement to indicate a parameter which is to be passed by reference rather than by value. Note that arrays must always be passed by reference.

Reference

Section 31.8.

Syntax



Examples (see also chapter 31)

```
  135 PROC invert(REF mat(,))

  275 FUNC reverse$(REF length#)
```

Associated keywords

PROC CLOSED IMPORT

RENUMBER

REN.

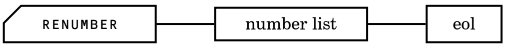
Description

This renumbers a program according to the two arguments following it. The first specifies the first line number to be used, and the second the increment for the numbers. Either parameter defaults to 10.

Reference

Section 5.5.

Syntax



Examples

```
RENUMBER

RENUMBER 100

RENUMBER 1000,30

RENUMBER ,20
```

Associated keywords

`LIST EDIT`

REPEAT

Description

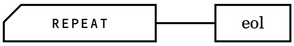
REPEAT is used to declare the start of a REPEAT UNTIL... loop structure. The lines between the REPEAT and UNTIL statements are executed repeatedly until the logical expression in the UNTIL statement becomes true. Program execution then continues from the line after the UNTIL statement.

Note that the loop will always execute at least once. (Use WHILE if this is not suitable.)

Reference

Section 16.1.

Syntax



Examples (see also chapter 16)

```
35 REPEAT
40   display_menu
45   take_user_choice
50 UNTIL valid_choice
```

Associated keyword

UNTIL

RESTORE

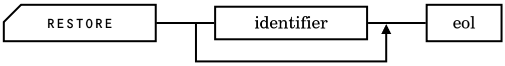
Description

RESTORE is used to reset the data pointer to some item of data in the program. If it is followed by a label then the pointer is reset to the first data item after that label. Otherwise the pointer is reset to the first data item in the program. Labels are defined by typing them on their own on a program line and terminating them with a colon.

Reference

Section 17.4.

Syntax



Examples

```
RESTORE

    35 RESTORE

    45 RESTORE second_set
```

Associated keywords

DATA EOD

RETURN

RET.

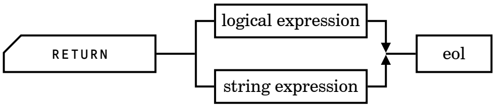
Description

RETURN is used in FUNCtion declarations to indicate the value that is to be returned by the function. All functions must return a value which is obtained by evaluating the expression following the RETURN statement. Immediately after the RETURN statement, program execution returns to the statement which called the function.

Reference

Chapter 30 and FUNC.

Syntax



Examples (see also FUNC and chapter 30)

```
1690 FUNC sinh(x)
1700   RETURN (EXP (x) - EXP (-x))/2
1710 END FUNC sinh(x)
```

Associated keyword

FUNC

RND(Random number RN.

Description

RND is used to generate psuedo-random numbers.

It functions as follows:

RND(-x) returns the value -x and resets the random number generator to a number based on x.

RND(0) repeats the last number given by RND(1).

RND(1) generates a random number between 0 and 1-2^(-31) (effectively 0 and 1).

RND(x) generates a random integer in the (inclusive) range 1 to x.

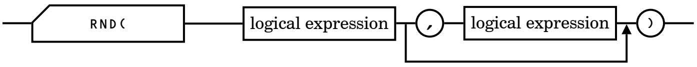
RND(x,y) generates a random integer in the range x to y (inclusive). If y is less than x then a Bad value error is generated at run-time. Note that x and y are truncated to integers if fractional values are given.

Note that no space is allowed before the open bracket.

Reference

Chapter 10.

Syntax



Examples

```
PRINT RND( 1,10)
  355 chance := RND( 1)
  765 score := RND( a,b)
```

Associated keywords

None.

RUN

RU.

Description

`RUN` is used to run a program. If no filename is supplied then the program run is that currently in the machine memory. If a filename is supplied then that program is first loaded from the current filing system.

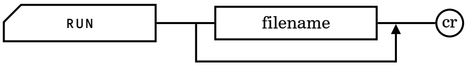
`RUN` clears all variables except the system integer variables.

Note that if a string variable is used in specifying the filename then it must be in lower case.

Reference

Sections 5.3 and 14.8.

Syntax



Examples

```
RUN
```

```
RUN myprog
```

```
RUN "myprog"
```

```
a$ = "myprog"
```

```
RUN a$
```

Associated keywords

`LOAD NEW OLD LIST EDIT`

SAVE

Description

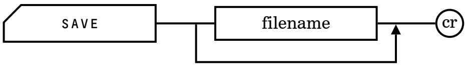
`SAVE` is used to store the program in the machine memory on the current filing system. The filename need not be in quotes. A string expression which evaluates to the filename may be used (except one which uses substring specifiers).

Note that if string variables are used in specifying the filename then they must be in lower case.

Reference

Sections 14.3 and 14.5.

Syntax



Examples

```
SAVE myprog

SAVE "myprog"

SAVE :2.$.name$
```

Associated keywords

```
LOAD RUN
```

SELECT OUTPUT SE.

Description

This is used to select which of the three possible output streams are to be used. `SELECT OUTPUT` is followed by a string, the effect of which is as follows.

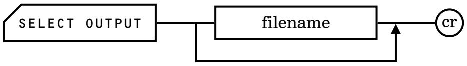
First letter	**Effect**
`D` or `d`	output to Display screen only
`P` or `p`	output to Parallel printer and screen
`S` or `s`	output to RS423 Serial port and screen

Any other initial letter will direct output to the screen only. The recommended strings for referring to the output streams are `DS`, `PP` and `SP` respectively.

Reference

Section 23.1.

Syntax



Examples

```
SELECT OUTPUT "S"

   15 SELECT OUTPUT output$(choice)
```

Associated keywords

None.

SGN Sign of argument SG.

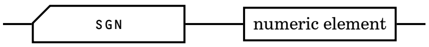
Description

`SGN` is a function which returns `-1`, 0 or 1 when its argument is less than zero, zero or greater than zero respectively.

Reference

Section 18.7.

Syntax



Examples

```
PRINT SGN (change)
```

```
   95 trend := SGN (difference)
```

Associated keyword

`ABS`

SIN Sine SI.

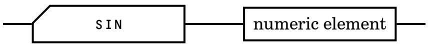
Description

SIN returns the sine of the argument which is taken to be in radian measure. Use RAD to convert degrees to radians.

Reference

Section 33.1.

Syntax



Examples

```
PRINT SIN (RAD (45))

   85 opp := SIN (angle) * hyp
```

Associated keywords

ACS ASN ATN COS TAN DEG RAD

SIZE

Description

SIZE is a system function which returns the number of bytes of memory currently occupied by a program. With no program it will return the value 2 since the beginning and end of program markers are still present.

See also the memory map in section 34.3.

Reference

Section 34.1.

Syntax

Examples

```
PRINT SIZE

   35 IF SIZE > large THEN MODE := 6
```

Associated keywords

PAGE FREE

SOUND

Description

Note that on the Acorn Electron this function is not the same as the BBC Microcomputer version described below. Reference should be made to the *Acorn Electron User Guide* for details of `SOUND` which performs exactly as in BASIC.

`SOUND` is used to instruct the sound generator to activate one of its synthesis channels to give an audio output through the internal loudspeaker. It is a very powerful, but also somewhat complex command, and reference should be made to the *BBC Microcomputer System User Guide* for a more detailed description than that given below.

There are four sound channels which are added together to give the audio output. Channels 1 to 3 can each generate a square wave of programmable frequency, while channel 0 can produce psuedo-random noise or a pulse waveform.

`SOUND` may be used by itself to produce simple sounds, but `ENVELOPE` allows the user to have greater control over the quality of the sounds.

`SOUND` is followed by four parameters, the first of which is a four-digit hexadecimal number, thus:

```
SOUND &HSFC,A,P,D
```

	Range	Function
H	0 or 1	Continuation
S	0 to 3	Synchronisation
F	0 or 1	Flush
C	0 to 3	Channel number
A	−15 to 4	Amplitude or envelope number
P	0 to 255	Pitch
D	1 to 255	Duration

The `H` parameter is set to 1 to indicate a dummy note (ie the amplitude and pitch parameters will have no effect). This feature may be used to allow the previous note to complete in full its release phase before the next note starts.

The `S` parameter is used to control synchronisation of sound on the different channels. If it is set to 0 the sound will be played as soon as it reaches the head of the queue for that channel. If it is set to 1, 2 or 3 the sound is not played until that many other channels have a sound waiting for synchronisation.

The `F` parameter is normally set to 0 meaning that the sound will be put in the queue and will only be played when those preceding it have been completed. However, if it is set to 1 then the queue will be flushed and the note played immediately.

The `C` parameter sets the channel number to be used.

The `A` parameter controls the amplitude of the sound. This may be done either by selecting an envelope to be used using the envelope number (1 to 4 or 1 to 16 if the RS423 and cassette buffers are unused), or by setting the amplitude directly with a number in the range −15 (loudest) to 0.

The `P` parameter determines the pitch of the note. Values in the range 0 to 255 may be used.

The `D` parameter sets the duration of the sound in twentieths of a second. If the amplitude has been given explicitly by a negative or zero value of the `A` parameter then this is the total length of the note, otherwise it will be the length of the attack, decay and sustain periods only (not the release period).

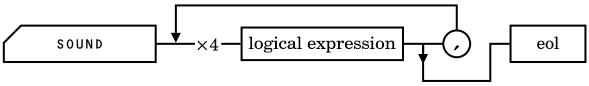
When used with channel 0 (the 'noise' channel) the `P` parameter has a different effect from that described for channels 1 to 3. The results of using the various values for `P` are as follows:

0	High frequency periodic noise
1	Medium frequency periodic noise
2	Low frequency periodic noise
3	Periodic noise of frequency determined by the pitch setting of channel 1
4	High frequency 'white' noise
5	Medium frequency 'white' noise
6	Low frequency 'white' noise
7	Noise of frequency determined (continuously) by the pitch setting of channel 1

Reference

Section 33.2 and the *BBC Microcomputer System User Guide*.

Syntax



Examples (see also section 33.2 and the *BBC Microcomputer System User Guide*)

```
SOUND 1,-5,53,128

  110 SOUND &0102,2,103,100 // waits for synchronisation

  230 SOUND channel,envelope,pitch,duration
```

Associated keywords

```
ENVELOPE ADVAL
```

SQR Square root SQ.

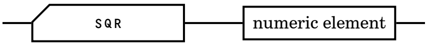
Description

`SQR` returns the square root of the argument. A negative argument results in a `Bad value` error.

Reference

Section 33.1.

Syntax



Examples

```
PRINT SQR (5)
```

```
  175 x1 := (-b+SQR (b*b-4*a*c))/(2*a)
  185 x2 := (-b-SQR (b*b-4*a*c))/(2*a)
```

Associated keywords

None.

STEP

Description

STEP is used in the FOR structure to indicate the increment for the loop variable, if omitted it defaults to 1. The increment may be positive or negative.

If the start and end values of the loop variable and the increment are incompatible (eg FOR i := 1 TO 10 STEP -1 DO) then the loop is not performed at all and execution moves to the line after the end of the FOR structure.

Reference

Section 6.4 and FOR.

Syntax

See FOR.

Examples (see also FOR)

```
  725 FOR i := 15 TO 1 STEP -2 DO PRINT i
```

Associated keywords

FOR TO DO NEXT

STOP

ST.

Description

STOP interrupts program execution and displays the message:

```
STOP at line xxx
```

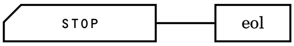
Execution may be restarted by using CONT provided that the program has not been altered or an error generated while in the direct mode.

Note that STOP may not be used in short form loops.

Reference

Chapter 12.

Syntax



Examples

```
 3010 STOP
 2500 IF x < 0 THEN STOP
```

Associated keywords

CONT END

STR$ String STR.

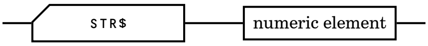
Description

`STR$` returns a string containing the numeric argument as it would have been printed. The form of the string created is affected by the field width and format imposed by `ZONE`.

Reference

Section 20.12 and `ZONE`.

Syntax



Examples

```
PRINT STR$ (a+b#)

   75 number$ := STR$ (number)
```

Associated keywords

```
VAL ZONE
```

TAB(Tabulation TAB.

Description

`TAB` is used to format `PRINT` output. There are two forms of the statement.

With one argument `TAB( x)` outputs (`x - COUNT`) spaces provided that this is greater than zero. If it is negative then `256 + x MOD 256` spaces are printed.

With two arguments, ie `TAB( x,y)`, the cursor is moved to the position whose x- and y-coordinates are the arguments. The coordinates are relative to the origin which is at the top left of the text area.

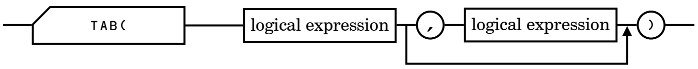
Note that brackets are necessary round the arguments, as is a comma to separate them if there are two.

Note also that `TAB( x,y)` will not work with a printer since it uses `VDU 31`. (`TAB( x)` will work with printers.)

Reference

Sections 7.2 and 7.4.

Syntax



Examples

```
PRINT TAB( 15,20); "Hi There!"
```

```
  375 PRINT TAB( 15); "Enter choice >> ";
```

Associated keywords

`PRINT INPUT POS VPOS COUNT`

TAN Tangent TA.

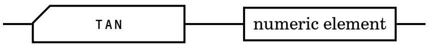
Description

TAN returns the tangent of the argument which is taken to be in radian measure. Use RAD to convert degrees to radians.

Reference

Section 33.1.

Syntax



Examples

```
PRINT TAN (RAD (210))

   75 opp := adj * TAN (angle)
```

Associated keywords

ACS ASN ATN SIN COS DEG RAD

THEN

Description

`THEN` is a keyword used in the long and short forms of the `IF` structure. It is optional on entry, except in a few cases such as:

```
IF x = &6000 THEN ? x = 0
```

where `x = &6000 ? x = 0` would be a valid expression.

Reference

Section 15.3 and `IF`.

Syntax

See `IF`.

Examples (see also `IF`)

```
  205 IF x < 0 THEN PRINT "Negative"
```

Associated keyword

`IF`

TIME

Description

TIME is a system variable which is used to set or read the internal elapsed time clock. The clock returns the elapsed time in hundredths of a second. Pressing BREAK or SHIFT BREAK does not reset the clock (but CTRL BREAK does).

Reference

Section 10.5.

Syntax

Examples

```
PRINT TIME

  505 time_now := TIME

  705 TIME := ((hrs * 60 + mins) * 60 +secs) * 100
```

Associated keywords

None.

TO

Description

`TO` is used in the `FOR` structure to separate the lower and upper bounds of the loop variable.

Reference

Section 6.1 and `FOR`.

Syntax

See `FOR`.

Examples (see also `FOR`)

```
    75 FOR i := 1 TO 10 DO PRINT i
```

Associated keyword

`FOR`

TRUE

Description

`TRUE` is a system function which returns the value `-1` (ie &FFFFFFFF).

Reference

Section 16.3.

Syntax

Example

```
   95 IF user$ = answer$ THEN answer(i#) := TRUE
```

Associated keyword

`FALSE`

UNTIL

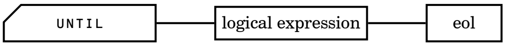
Description

UNTIL is used to declare the terminating condition in a REPEAT structure.

Reference

Chapter 16 and REPEAT.

Syntax



Examples (see also REPEAT and chapter 16)

```
    10 UNTIL result
```

Associated key word

REPEAT

USING

Description

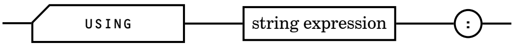
`USING` is used with `PRINT` statements to provide (fixed format) numeric formatting. It is followed by a string of the form `"###.##"` where the length of the string determines the field width, and the number of characters after the decimal point indicates the accuracy to which numbers are to be printed. Any characters may be used in the string provided that there is one decimal point, but to be compatible with other systems '`#`' should be used.

Note that `ZONE` may be used for this and for other formatting.

Reference

Section 23.5, `ZONE` and `PRINT`.

Syntax



Examples (see also `PRINT`)

```
   35 PRINT USING "##.###": a,b
```

Associated keywords

`PRINT ZONE`

USR User US.

Description

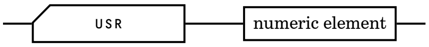
USR is a standard function which is used to call the machine code subroutine at the address specified by the argument.

USR enters the subroutine with the Accumulator, X and Y registers set to the least significant bytes of a#, x# and y# respectively, and the carry flag set to the least significant bit of c#. It returns with a value which is a four byte integer set up from the A, X, Y and P registers (in the order least significant to most significant).

Reference

Section 34.2.

Syntax



Examples (see also chapter 34)

```
PRINT USR &3000

  175 byte := USR osbget
```

Associated keywords

None.

VAL Evaluate string VA.

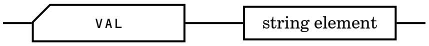
Description

VAL takes the string which follows it and interprets it as if it were an expression entered at the keyboard. Hence it can access variables, call functions, etc. It is often used to pass a function into a program from a user input.

Reference

Section 20.11.

Syntax



Examples

```
PRINT VAL (a$)

  100 x_input := VAL input1$

  200 answer := VAL (function$ + " " + STR$( x))
```

Associated keyword

STR$

VDU

Description

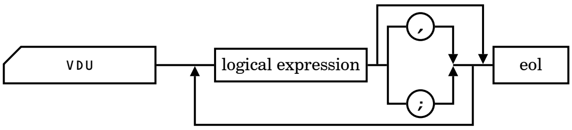
Note that on the Acorn Electron this function is not the same as the BBC Microcomputer version described below. Reference should be made to the *Acorn Electron User Guide* for details of `VDU` which performs exactly as in BASIC.

The `VDU` statement is equivalent to `PRINT CHR$`, but is generally used to generate the ASCII control codes. These codes are interpreted by the VDU driver and are summarised on the next page and in section 40.2. Those which require no extra arguments are generally obvious in their effect and are not discussed further here. Those which require more parameters are described in detail after the table. If the arguments are separated by commas then single bytes are sent. If any argument is followed by a semi-colon then that argument is sent as two bytes, with the least significant byte being sent first. Reference should be made to the *BBC Microcomputer System User Guide* for further details of the operation of `VDU` codes.

Reference

Section 11.12 and the *BBC Microcomputer System User Guide*.

Syntax



Associated keywords

`CHR$ PRINT`

VDU code summary

Code	CTRL	Bytes needed	Operation
0	@	0	no effect
1	A	1	send next character to printer only
2	B	0	enable printer
3	C	0	disable printer
4	D	0	write text at text cursor
5	E	0	write text at graphics cursor
6	F	0	enable VDU drivers
7	G	0	sound bell
8	H	0	backspace cursor one character
9	I	0	forwardspace cursor one character
10	J	0	move cursor down one line
11	K	0	move cursor up one line
12	L	0	clear text area (`CLS`)
13	M	0	move cursor to start of current line
14	N	0	page mode on
15	O	0	page mode off
16	P	0	clear graphics area (`CLG`)
17	Q	1	define text colour (`COLOUR`)
18	R	2	define graphics colour (`GCOL`)
19	S	5	define logical colour
20	T	0	restore default logical colours
21	U	0	disable VDU drivers or delete line
22	V	1	select screen mode (`MODE:=`)
23	W	9	reprogram display character
24	X	8	define graphics window
25	Y	5	`PLOT`
26	Z	0	restore default windows
27	[	0	no effect
28	\	4	define text window
29	]	4	define graphics origin
30	^	0	home text cursor to top left of area
31	_	2	move text cursor to `x`,`y` (`TAB(x,y)`)
127	DELETE	0	backspace and delete

Detailed descriptions

1 This code sends the character immediately following it to the printer only and not to the screen. The printer must already have been enabled using `VDU 2`. This facility enables control characters to be sent to a printer if it requires them to access certain features. It also allows characters to be sent to the printer which would otherwise be ignored because the `*FX 6` facility (see section 40.3) had been used.

Example: `VDU 1,14`

17 This code functions exactly like the COMAL statement `COLOUR`. It is followed by a single argument specifying the colour. See the entry under `COLOUR` for further details.

Example: `VDU 17,3`

18 This code functions exactly like the COMAL statement `GCOL`. It is followed by two arguments specifying the method by which the colour is to be plotted and the colour itself. See the entry under `GCOL` for further details.

Example: `VDU 18,2,129`

19 This code enables the user to change the actual colour referred to by a logical colour. The first argument specifies the logical colour and the second the actual colour which is to be assigned to it. There then need to be three arguments of 0 which are reserved for future expansion. The actual colour numbers are given in the entry for `COLOUR`.

For example, in a four colour mode:
`VDU 19,2,6,0,0,0` or `VDU 19,2,6;0;`
will reset colour 2 from its default of yellow to cyan.

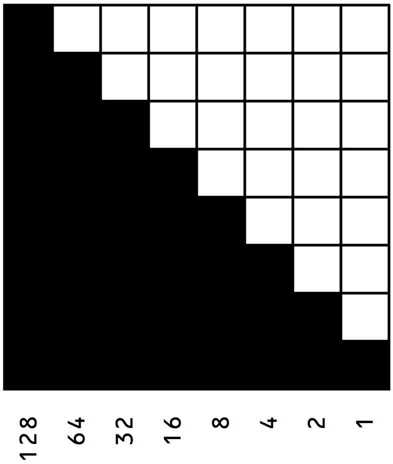
22 This code is equivalent to the COMAL statement `MODE:=` and is followed by a single argument specifying the new mode. Note that, unlike `MODE`, `VDU 22` does not check that the memory will not be corrupted before changing the video memory allocation. It may, however, be used inside procedures, but should only be used to change to a mode requiring the same amount or less memory.

Example: `VDU 22,3`

23 This code is used to redefine the characters corresponding to various ASCII codes. Codes 224 to 255 are initially available for this purpose. Other codes may also be redefined, but the procedure is more complicated and reference should be made to the *BBC Microcomputer System User Guide* for details. `VDU 23` requires nine further arguments. The first specifies the character to be redefined and the other eight give the contents of each row of dots, as shown in the following example.

To generate a character as shown in the following grid and assign it to ASCII code 245 the `VDU` command is:

```
VDU 23,245,128,192,224,240,248,252,254,255
```



```
                = 128
       64 + 128 = 192
  32 + 64 + 128 = 224
            ... = 240
            ... = 248
            ... = 252
            ... = 254
            ... = 255
```

Note that it is not possible to redefine characters used in `MODE 7`.

It is also possible to use `VDU 23` to alter the contents of the 6845 CRTC circuit. See the *BBC Microcomputer System User Guide* for further details.

24 This code is used to define a graphics window, ie to change the area of the screen in which graphics operations are possible. Four arguments follow the `VDU 24` and they must each be followed by a semi-colon. The arguments specify the x- and y-coordinates of the bottom left corner of the window, followed by the x- and y-coordinates of the top right corner.

Example: `VDU 24,200;250;1080;774;`

25 This code is identical to the COMAL command `PLOT`. The `VDU 25` is followed by three arguments. The first determines the method of plotting (see `PLOT` for details), the second gives the x-coordinate, and the third the y-coordinate. The coordinates must be followed by semi-colons since they must be two byte numbers.

Example: `VDU 25,5,640;512;` draws a line in the current foreground colour from the current cursor position to the centre of the screen.

28 This code is used to define a text window, restricting the area in which text operations are possible. The four arguments following the `VDU 28` give the x- and y-coordinates of the bottom left corner of the window, followed by the x- and y-coordinates of the top right corner. The arguments specify the column or row as appropriate, and the maximum values will depend on the screen mode in use. The origin (0,0) is at the top left of the screen.

Example: `VDU 28,10,20,30,10`

29 This code is used to move the graphics origin to the position specified by the two arguments following the `VDU 29`. The first gives the new x-coordinate, and the second the new y-coordinate. Both arguments should be followed by semi-colons.

Example: `VDU 29,640;512;` moves the origin to the centre of the screen.

31 This code is used to position the text cursor on the screen at the position given by the arguments following it. The first gives the x-coordinate and the second the y-coordinate. The maximum values allowable depend on the screen mode in use. Note that the coordinates specified are relative to the top left of the current text window. `VDU 31` is identical in effect to the COMAL command `TAB( x,y)`.

Example: `VDU 31,20,16`

VPOS Vertical position of cursor VP.

Description

VPOS returns the vertical position of the text cursor. The top of the screen returns VPOS = 0.

Reference

Section 18.1.

Syntax

Examples

```
PRINT TAB( POS,VPOS + 5);

  175 position := VPOS
```

Associated keyword

POS

WHEN

Description

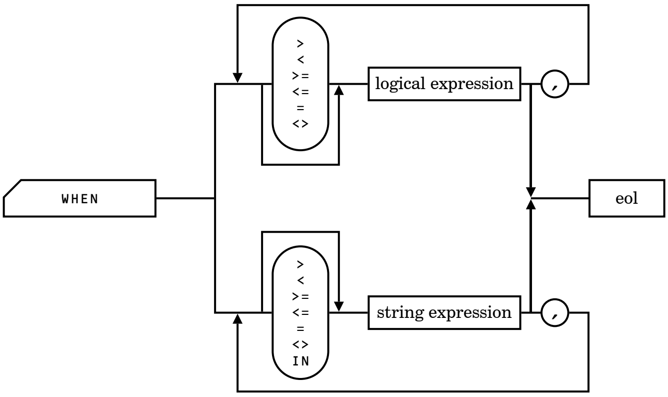
WHEN is used to declare a block of statements in a CASE structure which are to be executed when the variable in the CASE statement satisfies the condition after the WHEN statement.

The condition after the WHEN statement may use any of the following operators: `> < >= <= = <>` or no operator at all (indicating equality). Additionally if a string is being tested then the IN operator may be used. More than one condition may be used after the WHEN statement (see example), in which case the block of program lines will be executed if any of the conditions is true.

Reference

Chapter 25 and CASE.

Syntax



Examples (see also CASE and chapter 25)

```
375 WHEN <-5 , > 5

915 WHEN IN "YyNn"
```

Associated keywords

CASE OF END CASE OTHERWISE

WHILE

Description

WHILE is used to declare a conditional structure which is executed while the condition is true. There are two forms: the short form on a single program line, or in direct mode, which consists of a simple statement to be repeated WHILE the condition is true, and the long form consisting of a block of program lines between a WHILE statement and an END WHILE command.

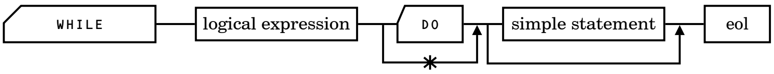
In both cases, since the condition is at the beginning of the structure, the loop need not be executed at all, unlike REPEAT loops which are always executed at least once.

Note that DO is optional on entry (except in a very few cases such as: WHILE x DO ? y = 0) and is inserted automatically if omitted.

Reference

Chapter 24.

Syntax



Examples (see also chapter 24)

Short form:

```
   70 WHILE GET$ <> " " DO PRINT "Press Space Bar"
```

Long form:

```
   95 WHILE NOT finished DO
  100   update_screen
  105   display_menu
  110 END WHILE
```

Associated keywords

DO END WHILE REPEAT

WIDTH

Description

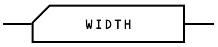
WIDTH is a system variable which contains the number of characters that may be printed on a line either on the screen or the printer before a carriage return and line feed is automatically generated. If WIDTH is set to 0 (the default) then no carriage return/line feeds are printed automatically.

Note that it is only possible to change WIDTH with an assignment statement (not, for example, from an INPUT).

Reference

Section 18.5.

Syntax



Examples

```
PRINT WIDTH

   65 WIDTH :- 10
```

Associated keywords

COUNT ZONE

WRITE

WR.

Description

WRITE is used to write data items to a file. It is also used as a qualifier when OPENing a sequential file for writing.

When used to write to a file WRITE is followed by FILE and an expression evaluating to the number of the file to which the data is to be written. If the file is random access there may then be an expression evaluating to the number of the record to be used. If it is omitted writing commences from the record after the last one used (the first record if the file has just been OPENed). For both sequential and random access files there then follows the list of variables to be written to the file. The variable names must be separated by commas.

With random access files a separate WRITE command is necessary for each new record. If the variables following the command will not fit in the record then as many as possible are written and then a Record overflow error is generated.

Note that PRINT FILE is identical to WRITE FILE except that PRINT FILE may have expressions rather than just variables as its arguments and may use any separators in the list.

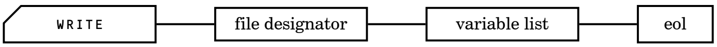
See RANDOM for details of the number of bytes required for storing data in files.

When using WRITE as a qualifier in OPENing a sequential file care should be taken since any existing unlocked file of the same name will be overwritten. APPEND should be used when OPENing an existing file to add information to the end.

Reference

Sections 32.7 and 32.14.

Syntax



Examples (see also OPEN)

```
WRITE FILE 2: a,b$,c#

   35 WRITE FILE file_no,rec_no: d(i#),i#,e$(i#)
```

Associated keywords

```
OPEN RANDOM READ PRINT FILE APPEND CLOSE
```

ZONE

Description

ZONE is a system variable used for setting the format, precision, and field width for the PRINT statement. It is essentially the same as the @% system variable in BBC BASIC, although B4 is not implemented, and reference should be made to the *BBC Microcomputer System User Guide* for a fuller discussion than that below. Note that STR$ always uses ZONE.

The argument following ZONE is normally expressed as a hexadecimal number, and so is preceded by the & operator. The argument consists of three bytes, each of which is specified by two hex digits. The most significant byte (ie the leftmost) is called B3, and the others are B2 and B1, the least significant byte.

B3 selects the basic format thus:

00 General (G) format. Integers are printed as integers. Numbers between 0.1 and 1 are printed 0.1 etc. Numbers less than 0.1 are printed in exponent form.

01 Exponent (E) format. Numbers are always printed in standard form, ie 1700 is printed as 1.7E3 etc.

02 Fixed (F) format. Numbers are printed with a fixed number of decimal places. If the number will not fit into the selected field width it reverts to G format. Decimal points will be aligned vertically making this format ideal for tables etc.

Numbers greater than 02 are taken as 00.

B2 sets the total number of digits printed in the selected format. If this number is too large or too small for that format then it is taken to be 9. The number to be printed is rounded to fit the B2 field.

In E format B2 specifies the total number of digits to be printed in the mantissa (ie not including those after the E). It may be in the range 1 to 9.

In F format B2 specifies the number of digits to follow the decimal point. It may be in the range 1 to 9.

In G format B2 gives the maximum number of digits that may be printed before returning to E format. It may be in the range 1 to 9.

B1 sets the overall print field width and may have any value in the range &00 to &FF (0 to 255 decimal).

The default value of `ZONE` is &00090A.

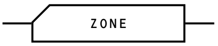
Leading zeros may, of course, be omitted.

Note that `ZONE` may only be changed using an assignment statement (not, for example, from an `INPUT` statement).

Reference

Section 23.3 and the *BBC Microcomputer System User Guide* (`@%`)

Syntax



Examples (see also section 23.3 and the *BBC Microcomputer System User Guide*)

```
ZONE := &02040A // F format, 4 dec places, 10 field

  100 ZONE := &010309
```

Associated keywords

```
PRINT USING  WIDTH
```

40 Reference tables

This chapter contains the following reference tables:

1. ASCII codes.
2. VDU code summary.
3. *FX call summary.
4. Keywords, operators and tokens.
5. Error messages and numbers.
6. Precedence of operators and valid numeric ranges.

40.1 ASCII codes

ASCII code table

Action/character	Code (Decimal)	Action/character	Code
Nothing	0	#	35
Next to printer	1	$	36
Start printer	2	%	37
Stop printer	3	&	38
Separate cursors	4	'	39
Join cursors	5	(	40
Enable VDU	6	)	41
Beep	7	*	42
Back	8	+	43
Forward	9	,	44
Down	10	-	45
Up	11	.	46
Clear text area	12	/	47
Carriage return	13	0	48
Paged mode on	14	1	49
Paged mode off	15	2	50
Clear graphics area	16	3	51
Define text colour	17	4	52
Define graphic colour	18	5	53
Define logical colour	19	6	54
Default logical colours	20	7	55
Erase line or Disable VDU	21	8	56
Select Mode	22	9	57
Reprogram characters	23	:	58
Define graphics area	24	;	59
Plot	25	<	60
Default screen areas	26	=	61
Nothing	27	>	62
Define text area	28	?	63
Define graphic origin	29	@	64
Move text cursor to 0,0	30	A	65
Move text cursor to X,Y	31	B	66
Space	32	C	67
!	33	D	68
"	34	E	69

Action/character	Code	Action/character	Code
F	70	c	99
G	71	d	100
H	72	e	101
I	73	f	102
J	74	g	103
K	75	h	104
L	76	i	105
M	77	j	106
N	78	k	107
O	79	l	108
P	80	m	109
Q	81	n	110
R	82	o	111
S	83	p	112
T	84	q	113
U	85	r	114
V	86	s	115
W	87	t	116
X	88	u	117
Y	89	v	118
Z	90	w	119
[	91	x	120
\	92	y	121
]	93	z	122
^	94	{	123
_	95	|	124
£	96	}	125
a	97	~	126
b	98	Backspace and delete	127

40.2 VDU code summary

Code	CTRL	Bytes needed	Operation
0	@	0	no effect
1	A	1	send next character to printer only
2	B	0	enable printer
3	C	0	disable printer
4	D	0	write text at text cursor
5	E	0	write text at graphics cursor
6	F	0	enable VDU drivers
7	G	0	sound bell
8	H	0	backspace cursor one character
9	I	0	forwardspace cursor one character
10	J	0	move cursor down one line
11	K	0	move cursor up one line
12	L	0	clear text area (`CLS`)
13	M	0	move cursor to start of current line
14	N	0	page mode on
15	O	0	page mode off
16	P	0	clear graphics area (`CLG`)
17	Q	1	define text colour (`COLOUR`)
18	R	2	define graphics colour (`GCOL`)
19	S	5	define logical colour
20	T	0	restore default logical colours
21	U	0	disable VDU drivers or delete line
22	V	1	select screen mode (`MODE:=`)
23	W	9	reprogram display character
24	X	8	define graphics window
25	Y	5	`PLOT`
26	Z	0	restore default windows
27	[	0	no effect
28	\	4	define text window
29	]	4	define graphics origin
30	^	0	home text cursor to top left of area
31	_	2	move text cursor to x,y (`TAB(x,y)`)
127	DELETE	0	backspace and delete

40.3 *FX call summary

Below are given selected `*FX` calls of use to the COMAL programmer. For other calls see the *BBC Microcomputer System User Guide*.

`*fx 0`	Print version number of the operating system		
`*fx 2`	Enable/Disable input device		
`,0`	enable keyboard input and disable RS423		
`,1`	enable input from RS423		
`,2`	enable keyboard input and enable RS423		
`*fx 3`	Select output device(s)		
`,0`	Printer	Screen	
`,1`	Printer	Screen	RS423
`,2`	Printer		
`,3`	Printer		RS423
`,4`		Screen	
`,5`		Screen	RS423
`,6`			
`,7`			RS423
`*fx 4`	Reset/Disable COPY and arrow keys		
`,0`	reset system for normal cursor editing		
`,1`	disable editing, keys generate codes 135-139		
`,2`	keys become user definable keys, 11-15		
`*fx 5`	Select type of printer output		
`,0`	lose output without ‘hang-up’		
`,1`	parallel output		
`,2`	serial (RS423) output		
`,3`	select user defined printer driver		
`*fx 6`	Set code to be ignored by printer driver		
`,10`	ignore line feed		
`*fx 7`	Set RS423 baud receive rate		
`,1`	75		
`,2`	150		
`,3`	300		
`,4`	1200		
`,5`	2400		
`,6`	4800		
`,7`	9600		
`,8`	19200		

`*fx 8`	Set RS423 baud transmit rate
`,1`	75
`,2`	150
`,3`	300
`,4`	1200
`,5`	2400
`,6`	4800
`,7`	9600
`,8`	19200
`*fx 9`	Set flash period of first colour
`,p`	where `p` is number of fiftieths of a second
`*fx 10`	Set flash period of second colour
`,q`	where `q` is number of fiftieths of a second
`*fx 11`	Set auto repeat delay
`,r`	where `r` is number of hundredths of a second
`*fx 12`	Set period before auto repeat starts
`,s`	where `s` is number of hundredths of a second
`*fx 15`	Flush internal buffers
`,0`	flush all internal buffers
`,1`	flush currently selected input buffer
`*fx 16`	Disable/Enable analogue to digital channels
`,0`	disable all ADC channels
`,1`	enable channel 1
`,2`	enable channels 1 and 2
`,3`	enable channels 1, 2 and 3
`,4`	enable channels 1, 2, 3 and 4
`*fx 17`	Select channel and initiate A/D conversion
`,n`	where `n` is number of the channel
`*fx 18`	Clear user defined keys
`*fx 19`	Wait for screen refresh before next display

`*fx 21`	Select and flush buffer
`,0`	flush keyboard buffer
`,1`	flush RS423 (serial) input buffer
`,2`	flush RS423 (serial) output buffer
`,3`	flush the printer output buffer
`,4`	flush sound channel 0 (white noise)
`,5`	flush sound channel 1
`,6`	flush sound channel 2
`,7`	flush sound channel 3
`,8`	flush speech synthesis buffer
`*fx 200`	Enable/Disable ESCAPE key
`,0`	enable ESCAPE
`,1`	disable ESCAPE
`*fx 210`	Enable/Disable sound
`,0`	enable sound
`,1`	disable sound
`*fx 229`	ESCAPE key action
`,0`	normal action
`,n`	(n>0) treat as ASCII code

40.4 Keywords, operators and tokens

Keyword etc	Minimum form	Token		
:=	=	150		=
:+	:+	150	followed by	+
:-	:-	150		-
+	+	—		
-	-	—		
*	*	—		
/	/	—		
^	^	—		
=	=	—		
<	<	—		
>	>	—		
:	:	—		
&	&	—		
~	~	—		
?	?	—		
!	!	—		
//	//	206		
ABS	AB.	179		
ACS	AC.	165		
ADVAL	AD.	180		
AND	AN.	133		
APPEND	AP.	139		
ASN	AS.	166		
ATN	AT.	167		
AUTO	A.	245		
CASE	CA.	239		
CHR$	CH.	186		
CLEAR	CLE.	196		
CLG	CLG	198		
CLOSE	CLOSE	220		
CLOSED	C.	253		
CLS	CLS	199		
COLOUR	COL.	227		
CONT	CO.	249		
COS	COS	168		
COUNT	COU.	154		
DATA	DA.	207		

Keyword etc	Minimum form	Token
DEBUG	D.	250
DEG	DEG	169
DEL	DEL	205
DELETE	DEL.	211
DIM	DI.	214
DIV	DIV	134
DO	DO	140
DRAW	DR.	221
EDIT	E.	247
ELIF	EL.	236
ELSE	ELS.	234
END	EN.	229
END CASE	END CA.	229 + 239
END FUNC	END FU.	229 + 237
END IF	END I.	229 + 241
END PROC	END PRO.	229 + 238
END WHILE	END W.	229 + 242
ENVELOPE	ENV.	226
EOD	EOD	155
EOF	EO.	181
EOR	EOR	135
EXEC	EX.	203
EXP	EXP	170
EXT	EXT	182
FALSE	FA.	151
FILE	FI.	141
FOR	F.	243
FREE	FR.	159
FUNC	FU.	237
GCOL	G.	222
GET	GET	156
GET$	GE.	161
GOTO	GO.	204
IF	I.	241
IMPORT	IM.	232
IN	IN	138
INKEY	INKEY	183
INKEY$	INK.	187
INPUT	IN.	217

Keyword etc	Minimum form	Token
INT	INT	177
LEN	LE.	162
LIST	.	248
LN	LN	171
LOAD	LO.	210
LOG	LOG	172
MOD	MOD	136
MODE	MOD.	191
MOVE	M.	223
NEW	NEW	200
NEXT	N.	230
NOT	NO.	184
NULL	NU.	197
OF	OF	142
OLD	O.	251
OPEN	OP.	216
OR	OR	137
ORD	OR.	163
OSCLI	OS.	215
OTHERWISE	OT.	233
PAGE	PA.	192
PI	PI	152
PLOT	PL.	224
POINT(	PO.	190
POS	POS	157
PRINT	P.	244
PROC	PRO.	238
RAD	RA.	173
RANDOM	RAN.	143
READ	READ	219
READ ONLY	R.	252
REF	REF	144
RENUMBER	REN.	246
REPEAT	REP.	240
RESTORE	RES.	202
RETURN	RET.	231
RND(	RN.	189
RUN	RU.	208
SAVE	SA.	209

Keyword etc	Minimum form	Token
SELECT OUTPUT	SE.	212
SGN	SG.	178
SIN	SI.	174
SIZE	SIZ.	158
SOUND	S.	225
SQR	SQ.	175
STEP	STE.	145
STOP	ST.	201
STR$	STR.	188
TAB(	TAB.	146
TAN	TA.	176
THEN	TH.	147
TIME	T.	193
TO	TO	148
TRUE	TR.	153
UNTIL	U.	228
USING	USI.	149
USR	US.	185
VAL	VA.	164
VDU	V.	213
VPOS	VP.	160
WHEN	WHE.	235
WHILE	W.	242
WIDTH	WI.	194
WRITE	WR.	218
ZONE	Z.	195

40.5 Error messages and numbers

Message	Number
`Bad DIM`	36
`Bad GOTO`	29
`Bad program`	21
`Bad type`	23
`Bad value`	32
`Can't CONT`	18
`EOD`	33
`EOF`	41
`Escape`	17
`File open`	38
`Name mismatch`	25
`No ...`	28
`No RETURN`	39
`No room`	20
`Not allowed`	26
`Not found`	31
`Not open`	35
`Parm block error`	40
`Record overflow`	34
`STOP`	19
`String too long`	30
`Syntax error`	22
`Too complex`	27
`Unclosed at ...`	24
`Variable exists`	37

40.6 Precedence of operators and valid numeric ranges

The operators available in COMAL are divided into eight groups, as follows, with group 8 having the highest precedence and group 1 the lowest.

Operators in the same group are applied in the order in which they are encountered.

Group 1: `OR EOR`

Group 2: `AND`

Group 3: `= < > <= >= <> IN`

Group 4: `+ -` (binary)

Group 5: `* / DIV MOD`

Group 6: `^` substring specifiers

Group 7: `? !`

Group 8: unary functions `NOT ( ) & ~`

The term ‘unary function’ covers all the operators and keywords which take a single argument and return a value. These are:

`& ~` (unary) `- ABS ACS ADVAL ASN ATN CHR$ COS DEG EOF INKEY INKEY$ INT LEN LN LOG ORD RAD SGN SIN SQR STR$ TAN USR VAL`

Valid ranges for arithmetic operations

Floating point numbers are stored with a single byte exponent, giving permissible values in the range −(2^128) to 2^128. The smallest non-zero value is 2^(−128).

Integer variables are stored in the range −&80000000 to +&7FFFFFFF.

Accuracy

All arithmetic values are stored with 32-bit accuracy.

Part IV

41 History of COMAL

COMAL was initially developed in Denmark during 1973/4 by Benedict Loefstedt (University of Aarhus) and Borge Christensen (State Teachers' College, Tonder). It was originally conceived as an extension to BASIC, mainly for schools' use, and implemented recent developments in programming languages.

Additional facilities were added over the next few years and in 1979 representatives of manufacturers, schools and universities met to discuss standardisation of the language. This group developed the standard for what became known as COMAL-80, which contained a common kernel and recommended extensions to the language.

The Acornsoft implementation of COMAL exceeds the COMAL-80 Standard as published in May 1982 and in addition has many extensions to take full advantage of the facilities of the BBC Microcomputer or Acorn Electron.

42 Supplementary programs

```
 920 // ****************************
 930 // * This function overcomes  *
 940 // * the known INKEY problem  *
 950 // * on the BBC Microcomputer.*
 960 // * For example              *
 970 // * code := new_inkey(300)   *
 980 // ****************************
 990 //
1000 FUNC new_inkey(n)
1010   a:=n MOD 256
1020   b:=n DIV 256
1030   RETURN INKEY (a+&01000000*b)
1040 END FUNC new_inkey
1050 //
```

```
 920 // ****************************
 930 // * This function overcomes  *
 940 // * the known INKEY$ problem *
 950 // * on the BBC Microcomputer.*
 960 // * For example              *
 970 // * key$ := new_inkey$(300)  *
 980 // ****************************
 990 //
1000 FUNC new_inkey$(n)
1010   a:=n MOD 256
1020   b:=n DIV 256
1030   RETURN INKEY$ (a+&01000000*b)
1040 END FUNC new_inkey$
1050 //
```

```
  10 // Attractive line drawings
  20 REPEAT
  30   MODE :=0
  40   REPEAT
  50     turn:=RND( 40,320)
  60     scale:=RND( 975,995)/1000
  70   UNTIL turn<>180
  80   PRINT "TURN ";turn;" degrees"
  90   PRINT "SCALE ";scale
 100   VDU 19,1,RND( 6),0,0;// COLOUR
 110   VDU 29,640;512; // MOVE ORIGIN
 120   x:=0;y:=-512 // START POINT
 130   MOVE x,y
 140   a:= scale*COS (RAD (turn))
 150   b:=-scale*SIN (RAD (turn))
 160   REPEAT
 170     x1:=x;y1:=y
 180     x:=x1*a+y1*b;y:=y1*a-x1*b
 190     DRAW x,y // TURN AND SCALE
 200   UNTIL ABS (x)+ABS (y) < 80
 210   delay:=INKEY (100)
 220 UNTIL FALSE
```

```
  10 // Adapted from a program to generate anagrams by
  20 // Andy Tonks
  30 //
  40 MODE := 0 // or MODE := 6
  50 //
  60 PRINT TAB( 0,5);
  70 INPUT "Enter anagram base >> ":word$
  80 INPUT "Lowest length of anagram   >> ":low#
  90 INPUT "Highest length of anagram  >> ":high#
 100 //
 110 found#:=0
 120 anag(word$,"") // Start of recursive procedure.
 130 PRINT found#;" selections were shown"'
 140 //
1000 PROC anag(word$,part$) CLOSED
1010   IMPORT low#,high#,found#
1020   FOR letter#:=1 TO LEN word$ DO
1030     result$:= part$ + word$ (letter#) // one letter
1040     IF LEN result$ >= low# AND LEN result$ <= high#
THEN
1050       PRINT result$
1060       found#:+1
1070     END IF
1080     IF LEN word$ > 1 AND LEN result$ <= high# THEN
```

```
 1090       anag(word$(:letter#-1) + word$(letter#+1:),
result$)
 1100     END IF // Short form IF could be used
 1110   NEXT letter#
 1120 END PROC anag
 1130 //
 1140 END

   10 // Program to calculate coins to
   20 // be given as change for a given
   30 // price and amount offered.
   40 //
   50 MODE :=6
   60 initialise
   70 get_price
   80 get_amount_offered
   90 give_change
  100 //
  110 END
  120 //
 1000 PROC initialise
 1010   amount#:=0 // amount offered
 1020   change#:=0 // change in pence
 1030   name$:=""  // name of note/coin
 1040   unit:=0    // note/coin value
 1050   number#:=0 // notes/coins needed
 1060   value#:=0  // value returned
 1070   return_key:=13 // ASCII code
 1080 END PROC initialise
 1090 //
 1200 PROC get_price
 1210   PRINT ''"Price of item   > ";
 1220   cost# := value  // a function
 1230 END PROC get_price
 1240 //
 1300 PROC get_amount_offered
 1310   PRINT ''"Amount offered  > ";
 1320   WHILE value < cost# DO // reject
 1330     PRINT "NOT ENOUGH"'
 1340     PRINT "Try amount again  > ";
 1350   END WHILE
 1360   amount# := value#  // when enough
```

```
1370 END PROC get_amount_offered
1380 //
1500 PROC give_change
1510   change# := amount# - cost# // pence
1520   PRINT ''"Change ";
1530   PRINT USING ".##":;change#/100'
1540   choose_coins
1550 END PROC give_change
1560 //
1600 PROC choose_coins
1610   WHILE change# > 0 DO
1620     READ name$,unit#
1630     number# := change# DIV unit#
1640     change# := change# MOD unit#
1650     IF number# > 0 THEN PRINT number#;" ";name$
1660   END WHILE
1670   DATA Fifty Pound,5000,Twenty Pound,2000
1680   DATA Ten Pound,1000,Five Pound,500
1690   DATA One Pound,100,Fifty Pence,50
1700   DATA Twenty Pence,20,Ten Pence,10
1710   DATA Five Pence,5,Two Pence,2
1720   DATA One Penny,1
1730 END PROC choose_coins
1740 //
1800 FUNC value
1810   value$ := ""
1820   REPEAT
1830     key$ := GET$
1840     IF key$ IN "1234567890." THEN
1850       PRINT key$;
1860       value$ :+ key$
1870     END IF
1880   UNTIL key$ = CHR$ (return_key)
1890   value# := 100 * VAL (value$) + 0.5
1900   RETURN value#
1910 END FUNC value
```

43 Suggested answers to exercises

This chapter contains suggested answers to the exercises set at the end of the chapters in Part I. Where variable names are used in these answers they may, of course, be replaced by any other suitable variable of the same type. In many cases there will be a number of alternative ways of producing the required results which may be just as good as that given, although you should ensure that you understand the suggested method.

Section 2.10

1. `PRINT 6 + 7`
2. `PRINT 1 + 2 + 3 + 4 + 5`
3. `PRINT 4 * 5 * 9 / 6`
4. `PRINT 7 * 7 - 6 * 6`
5. `PRINT 24 / (5+3)`
6. `PRINT 12345679 * 7 * 9`
7. `PRINT 1 / 0.11`
8. `PRINT 3.142 * 3.5 ^ 2 / 3`
9. `PRINT 5.40 * 120`
10. `PRINT 450 / 120`

Section 3.14

1. `2u` begins with a number
   `tax paid` has a space in the middle
   `print` and `free` are both keywords
2. 9E9 ie 9 times 10^9
3. 8.76E-3 ie 8.76 divided by 10^3
4. `print out` and `PRINT OUT` will both print 3
5. 3 ie the same as the value of `lower`
6. 4.5 the integer variable makes no difference.

7. 3 the integer below 3.5
8. 3 the integer below 3.141592653

Section 4.11

1. `PRINT a$,b$,c$`
2. `PRINT a$;" ";b$;c$` or `PRINT a$ + " " + b$ + c$`
3. `PRINT a$'b$'c$`
4. `PRINT b$;" ";b$;" ";b$` or `PRINT b$+" "+b$+" "+b$`
5. `1` `Y` is the first element in `YyNn`
6. `0` `Y` is not in `Nn`
7. `3` `Y` is the third element in `RGYBMCW`
8. `1` the first place at which `Y` occurs
9. `1` the length of `reply$` is 1
10. `DIM ton$ OF 100`
11. A dimension statement is not needed. 40 places will be reserved when `test$` is first assigned. No error would arise from entering

    `DIM test$ OF 40`
12. `DIM ten$ OF 10`

Section 5.15

1. `AUTO 200,5`
2. Press the ESCAPE key.
3. `RENUMBER 100` or `RENUMBER 100,10`
4. `NEW`
5. `OLD`
6.
```
10 a$ := "SUPER"
20 b$ := "CALI"
30 c$ := "FRAGI"
40 d$ := "LISTIC"
50 PRINT a$,b$,c$,d$
```

then either

```
60 PRINT a$,b$,c$;d$
70 PRINT a$,b$;c$;d$
80 PRINT a$;b$;c$;d$
```

or

```
60 PRINT a$,b$,c$+d$
70 PRINT a$,b$+c$+d$
80 PRINT a$+b$+c$+d$
```

followed by either

```
90 PRINT a$'b$'c$'d$
```

or

```
 90 PRINT a$
100 PRINT b$
110 PRINT c$
120 PRINT d$
```

7.

```
10 x := 111
```

followed by either

```
20 PRINT x,x*x,x*x*x
```

or

```
20 PRINT x,x^2,x^3
```

Section 6.10

1.

```
10 FOR number := 21 TO 30 DO
20   PRINT number
30 NEXT number
40 END
```

2.

```
10 FOR number := 10 TO 100 STEP 10 DO
20   PRINT number
30 NEXT number
40 END
```

3.

```
10 FOR number := 1 TO 19 STEP 2 DO
20   PRINT number
30 NEXT number
40 END
```

4.
```
10 FOR number := 2 TO 20 STEP 2 DO
20   PRINT number
30 NEXT number
40 END
```

5.
```
10 FOR number := 99 TO 0 STEP -11 DO
20   PRINT number
30 NEXT number
40 END
```

6.
```
10 FOR number := 2 TO 6 STEP 0.5 DO
20   PRINT number
30 NEXT number
40 END
```

7.
```
10 star$ := "*"
20 PRINT star$
30 pling$ := "!"
40 FOR go := 1 TO 7 DO
50   star$ :+ pling$
60   PRINT star$
70 NEXT go
80 END
```

8.
```
10 silly$ := "ABAD"
20 PRINT silly$
30 FOR go := 1 TO 3 DO
40   silly$ :+ silly$
50   PRINT silly$
60 NEXT go
70 END
```

Section 7.7

1. `PRINT TAB( 4,2);"here"`
2. `PRINT TAB( 8,14);"now"`
3. `PRINT ORD ("G")`
4. `PRINT CHR$ (70)`

In the following, the loop variable need not be `row`. The particular variable name is a matter of choice. `row#` could be used to point out the use of an integer. It is possible to use the loop variable as part of a `TAB( , )` command as shown in the alternative answers to questions 5 and 8.

5.
```
10 CLS
20 FOR row := 1 TO 9 DO
30   PRINT TAB( 4);"here"
40 NEXT row
```

or

```
10 CLS
20 FOR row := 0 TO 8 DO
30   PRINT TAB( 4,row);"here"
40 NEXT row
```

6.
```
10 CLS
20 FOR row := 1 TO 9 DO
30   PRINT TAB( 4);row
40 NEXT row
```

7.
```
10 CLS
20 FOR row := 1 TO 9 DO
30   PRINT TAB( row);row
40 NEXT row
```

8.
```
10 CLS
20 PRINT TAB( 0,10);
30 FOR row := 11 TO 19 DO
40   PRINT ;CHR$ (70 + row)
50 NEXT row
```

or delete 20 and use

```
40 PRINT TAB( 0,row-1);CHR$ (70+row)
```

Section 8.6

1.
```
COLOUR 1 or red := 1
            COLOUR red
```

2.
```
100 MODE := 5
110 FOR code := 65 TO 90 DO
120   PRINT CHR$ (code);
130 NEXT code
140 END
```

3.
```
COLOUR 3 or white := 3
            COLOUR white
```

4.
```
100 red := 1
110 white := 3
120 MODE := 5
```

```
    130 COLOUR red
    140 FOR code := 65 TO 90 DO
    150   PRINT CHR$ (code);
    160 NEXT code
    170 COLOUR white
    180 END
```

5.
```
    115 yellow := 2
    123 COLOUR 128 + yellow
    126 CLS
```

6.
```
COLOUR 2 or green := 2
            COLOUR green
```

7.
```
COLOUR 134 or cyan := 6
CLS           COLOUR 128 + cyan
              CLS
```

8.
```
    100 green := 2
    110 cyan := 6
    120 white := 7
    130 MODE := 2
    140 COLOUR 128 + cyan
    150 CLS
    160 COLOUR green
    170 FOR number := 1 TO 9 DO
    180   PRINT number
    190 NEXT number
    200 COLOUR white
    210 END
```

9. `*key 2 RENUMBER`

10. `*key 2 RENUMBER |m`

11. `*key 2 RENUMBER |m MODE := 6 |m LIST |m`

Section 9.7

1.
```
     10 INPUT "Enter a number please > ":first
     20 INPUT "and another please    > ":second
     30 total := first + second
     40 product := first * second
     50 PRINT ''"Their total is   ";total
     60 PRINT ''"Their product is ";product
```

2.
```
 10 INPUT "Length  > ":length
 20 INPUT "Breadth > ":breadth
 30 INPUT "Height  > ":height
 40 volume := length * breadth * height
 50 PRINT ''"The volume is ";volume
```

3. Add

```
 60 base := length * breadth
 70 front := length * height
 80 side := breadth * height
 90 area := 2*(base + front + side)
100 PRINT ''"The surface area is ";area
```

4.
```
 10 INPUT "Principal > ":p
 20 INPUT "Rate      > ":r
 30 INPUT "Time      > ":t
 40 i := p * r * t / 100
 50 PRINT ''"The Simple Interest is ";i
```

5. The following are the changes and additions.

```
 23 PRINT TAB( 5,1);
 26 INPUT "How many numbers are there ? ":n
 30 PRINT TAB( 8,3);"Average of ";n;" numbers"
 40 FOR row := 1 TO n DO
 90 PRINT 'TAB( 8); The average is";
100 PRINT ;total / n
```

Section 10.7

1. `PRINT RND( 40)`

2. `PRINT RND( 0,39)`

3. `PRINT RND( 11,20)`

4. `PRINT RND( 0,1279)`

5. `PRINT RND( 640,1279)`

6. `PRINT RND( 512,1023)`

7. Add the following lines

```
  5 total := 0
 65 total :+ score
 90 PRINT '"The average was "; total/20
```

Section 11.13

1. No. Only modes 0, 1, 2, 4 and 5 have graphics.

2.
```
MOVE 0,1023
DRAW 1279,0
```

3.
```
 10 MODE := 5
 20 FOR height := 0 TO 1000 STEP 100 DO
 30   MOVE 0,height
 40   DRAW 1279,height
 50 NEXT height
```

4.
```
 10 MODE := 5
 20 FOR over := 0 TO 1200 STEP 100 DO
 30   MOVE over,1023
 40   DRAW over,0
 50 NEXT over
```

5.
```
 10 MODE := 5
 20 FOR across := 40 TO 1240 STEP 80 DO
 30   PLOT 69,across,511
 40 NEXT across
```

6.
```
 10 red := 1        // Assigning
 20 yellow := 2     // values to
 30 left := 539     // variables
 40 right := 739    // makes the
 50 top := 611      // program
 60 bottom := 411   // MUCH EASIER
 70 fill := 85      // to read.
 80 MODE := 5
 90 //
100 GCOL 0,128 + yellow
110 CLG // to fill background in yellow
120 GCOL 0,red // set graphics colour
130 MOVE left,top
140 MOVE right,top
150 PLOT fill,left,bottom  // Uses last 3
160 PLOT fill,right,bottom // points.
170 END
```

7.
```
VDU 28,0,15,19,0
```

Bottom left corner at position 0 across, 15 down
Top right corner at position 19 across, 0 down

8. `VDU 24,0;0;1279;511;`

   Bottom left corner at coordinates 0,0
   Top right corner at coordinates 1279,511

Section 13.9

1.
```
1203 across := RND( 0,39)
1206 down := RND( 6,20)
1210 PRINT TAB( across,down);"*"
```

2.
```
 120 MODE := 1
 122 red := 1
 124 yellow := 2
 126 white := 3

1005 COLOUR yellow

1208 COLOUR red

1215 COLOUR yellow

1425 COLOUR white
```

3.
```
assign_variables
display_instructions
ask_question
input_answer
mark_answer
display_response
```

4. `EDIT`

5. `a#` to `z#` the resident integer variables (and the system variables).

6. `CLEAR`

7. With a single press of the SHIFT key.

8. By holding down the CTRL key and pressing the letter O key.

9. By holding down the CTRL and the SHIFT key at the same time.

10. No space between `PROC` and `TEST`.

11. `EDIT` is a keyword and so cannot be used for a procedure name.

12. No spaces between `END PROC` and `TEST`.

Section 15.7

1. `endif` should have a space inside. `END IF` will list in capitals.
2. This is not suitable for a single line `IF` command. There are two actions to carry out. The `print "Too low"` should be on a separate line.
3. `ELSE` must have a line to itself. The `PRINT "Wrong"` should be on a separate line.
4. When an `ELSE` condition is used the `IF` command cannot be a single line statement. The `PRINT "correct"` should be on a separate line.
5. Line 170 should occur after line 190. The `END IF` must come after the `ELSE`.
6. A single line `IF` command doesn't have an `END IF`
7.
```
10 PRINT TAB( 0,5);
20 INPUT "Enter a word please > ":first$
30 INPUT "and another please  > ":second$
40 IF first$ < second$ THEN
50   PRINT first$,second$
60 ELSE
70   PRINT second$,first$
80 END IF
```
8. Change lines 50 and 70 to
```
50 PRINT first$;" before ";second$
70 PRINT first$;" after ";second$
```

Section 16.6

1.
```
10 REPEAT
20   number := RND( 26)
30   PRINT number
40 UNTIL number = 26
```
2. Change 20 to `20   number := RND( 65,90)`
   and 40 to `40 UNTIL number = 90`
3. Change 30 to `30 PRINT CHR$(number)`
4.
```
10 MODE := 5
20 REPEAT
30   across := RND( 0,19)
40   down := RND( 0,29)
50   number := RND( 65,90)
60   PRINT TAB( across,down);CHR$ (number)
70 UNTIL number = 90
```

5. Change the following lines

```
   10 MODE := 2
```

and

```
   30 number := RND( 7)
```

6. Add `65 symbol := RND( 65,90)`
   Change `70 PRINT TAB( across,down);CHR$ (symbol)`
   Then change 65 to `65 symbol := RND( 32,126)`

7.

```
   10 MODE := 6
   20 PRINT ''"Did you write this program ? ";
   30 REPEAT
   40   reply$ := GET$
   50 UNTIL (reply$ IN "YyNn") <> 0
   60 IF (reply$ IN "Yy") <> 0 THEN
   70   PRINT "Yes"
   80 ELSE
   90   PRINT "No"
  100 END IF
```

8.

```
   10 TIME := 0
   20 correct := 0
   30 REPEAT
   40   CLS // or MODE := 6
   50   first := RND( 2,9)
   60   second := RND( 2,9)
   70   PRINT ''first;" times ";second;" is ";
   80   INPUT answer
   90   IF answer = first * second THEN correct :+1
  100 UNTIL TIME > 3000
  110 PRINT ''"You got ";correct;" right."
```

Section 17.10

1. Line 100 should be `100 here:`
   A label must have a colon immediately after it.

2. Line 120 should be `120 READ number$`
   Line 130 should be `130 PRINT number$`
   The `DATA` contains strings.

3. There should be commas between the `DATA`

4. `RESTORE` is inside the `FOR... NEXT...` loop.
   The only value that will be `READ` is `Clubs`
   Swap 140 and 150 to get a random suit choice.

5.
```
 10 MODE := 6
 20 //
 30 READ question$
 40 PRINT ''question$
 50 //
 60 FOR offer := 1 to 4
 70   READ answer$
 80   PRINT ';offer;". ";answer$
 90 NEXT offer
100 //
110 READ correct$ // note use of string
120 //
130 PRINT '"Which number is correct ";
140 REPEAT
150   choice$ := GET$ // note string
160 UNTIL (choice$ IN "1234") <> 0
170 PRINT choice$ // to display it
180 //
190 IF choice$ = correct$ THEN
200   PRINT 'choice$;" is Correct"
210 ELSE
220   PRINT '"Sorry ";choice$;" is Wrong"
230 END IF
240 //
250 DATA What does a car run on ?
260 DATA Tyres,The Road,Petrol,Hedgehogs,3
```

6. Add lines

```
125 REPEAT
235 UNTIL choice$ = correct$
```

7. Add lines

```
 25 REPEAT
237 UNTIL EOD
```

Section 18.12

1. 5
2. 3
3. 1
4. 0

5. 1

6. 2

7. 3

8. 8

9. 12 – `*` is evaluated before `DIV`

These programs are not the only right answers. If you have a program that successfully prints out the values then it may well be just as good an answer.

10.
```
10 FOR number := 5 TO -3 STEP -1
20   PRINT ABS (number)
30 NEXT number
```

11.
```
10 FOR number := 2 TO -4 STEP -1
20   PRINT SGN (number)
30 NEXT number
```

12.
```
10 FOR number := 11 TO 3 STEP -1
20   PRINT number DIV 3
30 NEXT number
```

13.
```
10 FOR number := 9 TO 0 STEP -1
20   PRINT number MOD 5
30 NEXT number
```

14.
```
10 FOR number := 5 TO -4 STEP -1
20   PRINT number MOD 3
30 NEXT number
```

15.
```
10 FOR number := 2.5 TO -2 STEP -0.5
20   PRINT INT (number)
30 NEXT number
```

Section 19.7

1. `DIM name$(50)`

2. `DIM points#(10:80)`

3. `DIM grid(1:3,1:3)`

4. `DIM ref_number(1:3,1:50)`

5. `DIM item$(5,5,5)`

6. `DIM item$(5,5,5) OF 5`

7.
```
 10 // Name and Cost
 20 //
 30 MODE := 6
 40 //
 50 initialise
 60 read_data
 70 //
 80 PROC initialise
 90   DIM name$(1:5)
100   DIM cost#(1:5)
110 END PROC initialise
120 //
130 PROC read_data
140   FOR item := 1 TO 5 DO
150     READ name$(item)
160     READ cost#(item)
170   NEXT item
180   PRINT TAB( 11,10);"All values read."
190 END PROC read_data
200 //
400 DATA Fish,87,Cheese,152,Peas,30,Beans,22,Bread,45
410 END
```

8. Add the following lines

```
 65 take_order
105 total := 0
210 PROC take_order
220   CLS
230   FOR item := 1 TO 5 DO
240   PRINT 'name$(item)
250   PRINT "Cost ";cost#(item);" pence"'
260   REPEAT // no negative numbers
270     INPUT "How many would you like > ":number#
280   UNTIL number# >= 0 // number# is an integer
290   total :+ number# * cost#(item)
300   NEXT item
310   PRINT '"Total cost ";total;" pence."
320 END PROC take_order
330 //
```

Section 20.13

1. a) `a$ := INKEY$ (100)`

   b) `b := INKEY (200)`

   c) `PRINT INKEY (-83)`

2. The loops are not properly nested. The `j` and `k` after the `NEXT` commands should be reversed.

3. 14

4. 5

Section 21.10

1. `TRUE OR TRUE` will produce `TRUE` (`-1`)

2. `TRUE EOR TRUE` will produce `FALSE` (`0`)

3. `TRUE AND TRUE` will produce `TRUE` (`-1`)

4. `TRUE OR FALSE` will produce `TRUE` (`-1`)

5. `TRUE EOR FALSE` will produce `TRUE` (`-1`)

6. `TRUE AND FALSE` will produce `FALSE` (`0`)

7. `NOT TRUE` will be `FALSE` (`0`)

8. `NOT FALSE` will be `TRUE` (`-1`)

9. -1

10. 0

11. -2

Index

COMAL

on the BBC Microcomputer and Acorn Electron

About this book

This manual serves both as a beginners' introduction to computing in COMAL and as a reference manual for the more experienced user.

COMAL is recognised in many European countries as one of the best languages in education. It has the facilities for writing well-structured programs and includes excellent interactive debugging responses. Its ability to handle procedures, functions, strings and arrays, and its other advanced features place COMAL in the forefront of microcomputer languages.

This manual introduces the user to the Acornsoft implementation of COMAL. The first section takes the form of a tutorial course for the user who is new to programming. The second contains the information which a user already familiar with BASIC or Pascal will need to write software in COMAL. There is then a reference section containing a complete summary of the language by keyword as well as various tables, and finally a number of demonstration programs.

About the authors

Roy Thornton read Mathematics at Royal Holloway College, University of London. Following service in the Instruction Branch of the Royal Navy he became a schoolmaster and is currently Director of Computing at Portsmouth Grammar School.

Paul Christensen is currently studying Natural Sciences at Churchill College, Cambridge University, having previously attended Portsmouth Grammar School.

Acornsoft Limited, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, England. Telephone (0223) 316039

Note: British Broadcasting Corporation has been abbreviated to BBC in this publication.

5 012582 920008

SBD19