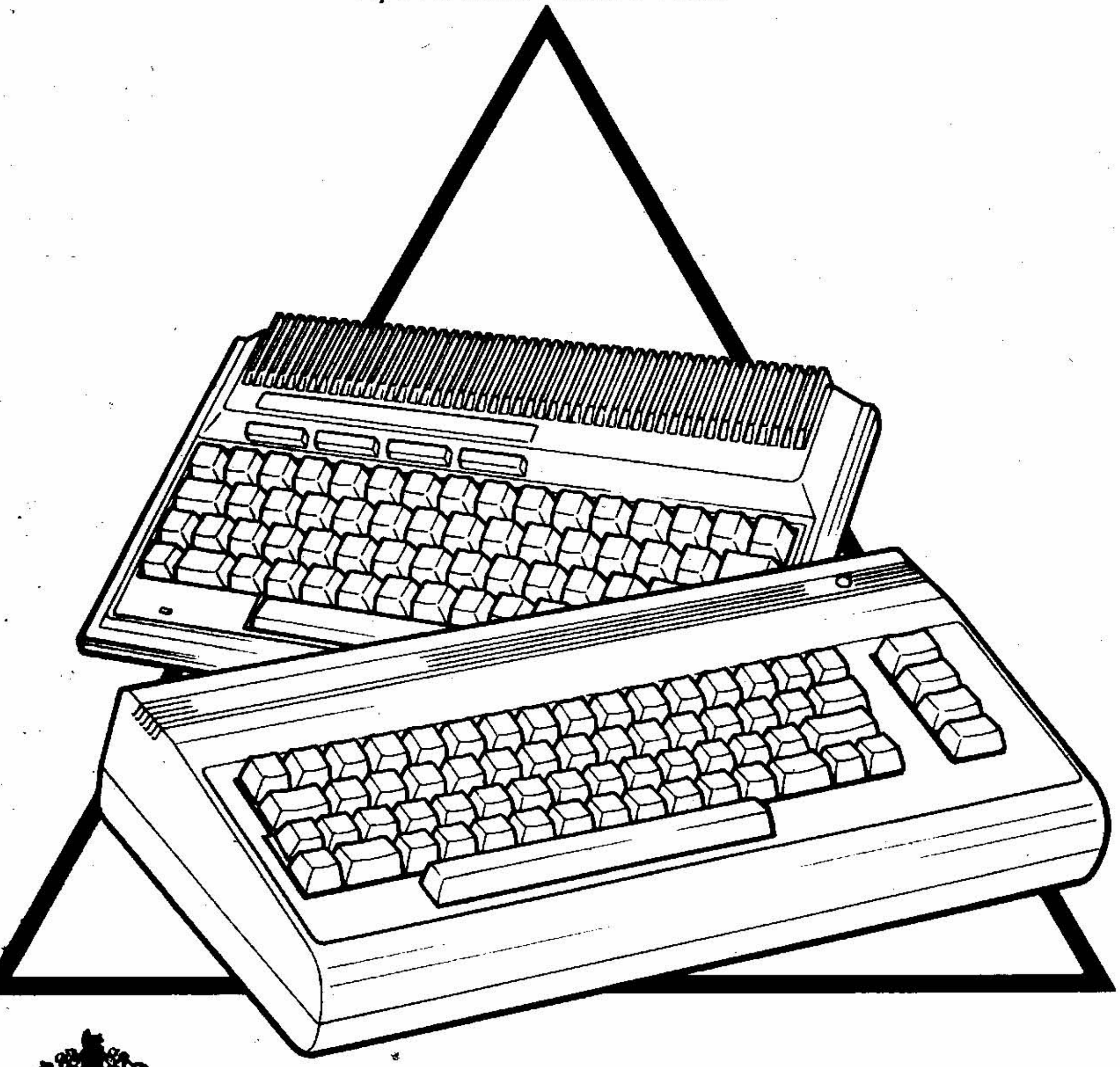
C16 plus/4

AN INTRODUCTION TO BASIC - PART 1

By Professor Andrew Colin



commodore

CONTENTS

This course is Part 1 of a series designed to help you learn about every aspect of programming the COMMODORE 16 and Plus/4 computers. The present course covers the principles of programming and all the elementary facilities of the BASIC programming language. It has three constituent parts:

1. A self-study text divided into 15 lessons or 'units', each of which deals with an important aspect of programming.

2. 2 cassette tapes or 1 disk with a collection of programs, which help you study the units.

3. A flow-chart stencil like the ones used by professional Computer Scientists. This stencil will help you design programs to be correct, efficient and robust.

Please note that this course teaches you to write useful and entertaining programs for your computer, but it does *not* cover the whole of the BASIC language. The more advanced features of BASIC are fully explained and discussed in the second course of the series.

CONTENTS LIST

INTRODUCTION

Welcome to the programming course for the COMMODORE 16 and PLUS/4. Both these machines are superb for playing games and producing brilliant and exciting pictures and sounds on your TV set; but they are also complete modern computers in their own right.

Computers are extraordinarily versatile; more so, in fact, than anything except a human. Your computer for instance, can be switched to be a teaching machine, a calculator, an aid to the handicapped, a word processor, a machine for financial records and stock control, a monitor for a patient in an intensive care unit, a controller for an industrial process, or a scientific computer used by engineers to design buildings, power stations and aircraft.

Computers and the systems they control are steadily entering into our everyday lives. Already many devices such as traffic lights, cash registers, and banking terminals have computers behind the scenes. This trend will continue for most of our lifetimes. The world is passing through a computer revolution, which will be as profound in its effects as the Industrial Revolution was in its own time.

The Computer Revolution can't be stopped; but all of us can, if we like, have some influence on the way it goes. The world is becoming divided into two sorts of people — the passengers and the pilots. The passengers let it all just happen; they may enjoy using computer-based products, or they may hate computers, or both. They often make their views known, but without any real effect — they can't reach the controls, and wouldn't know how to use them if they could.

The pilots, on the other hand, are in control of the whole revolution. They invent new types of computers, and think up original and useful ways of using them. The pilots have a heavy responsibility, since it rests on them fo steer the world towards peace, freedom and plenty, and away from the nightmare society often depicted in Science Fiction.

What sets apart a pilot from a passenger? Only one thing: understanding the way a computer works. Of course there are different levels of understanding. Most people understand how to use an arcade machine even though they couldn't explain the mechanism to you. (Yes — there is a computer inside.) The level I am thinking of is much deeper. It is so thorough and complete that you can make a computer do anything *you* want it to, in the way of teaching activities, industrial or medical applications, or even games to amuse you.

To have this power over your computer, to make it into a fast, accurate obedient and willing slave, you must be able to *program* the machine. Programming is the key to becoming a pilot.

This course is all about programming. It relates to the 16 and PLUS/4, but once you have mastered programming for these machines you will find it simple to transfer to any other computer, large or small.

The more programming you do, the easier it becomes. Most people can learn how to program if they give themselves a fair chance, and so can you. You do not need to know much about mathematics, but you will find it useful to have a quiet place to read, think and use the computer, and it is best to give yourself plenty of time to complete the course. Don't rush!

The course is split into fifteen 'units'. Each unit will take you one or two solid evenings' work, on average. Most of the units include some reading, some practical work on the machine, some programming, and a 'self-test' questionnaire to measure how well you have understood the unit. Every unit contains some 'experiments' which you should tick off as you do them.

When the units ask you questions, they generally give you spaces to write your answers. Use them. Write with a soft pencil, and have a rubber handy, so that your answers can be rubbed out if you pass the computer course on to someone else. If your copy of the course already has the answers written in, go through it and erase them before you start studying.

Programming is a tight-knit subject in which ideas depend closely on each other. Topics you learn about in earlier units are mentioned and used in the later ones without any further explanation. For example, you won't be able to make head or tail of unit 10 unless you have read and understood *all* of units 1 to 9. This makes it important that you follow the units in the order they are given.

When you start work on a new unit, begin by reading quickly right through it from beginning to end. You won't get much of the detail, but you will form an idea of the kind of topics you are going to study.

Next, work through the unit in detail. Every part matters, and the parts which seem the hardest matter the most. Don't skip anything, but try to understand every point. When you feel you've learned something, repeat it to yourself in your own words. Don't be upset if you find you have to read parts of the unit several times over, or even go back to an earlier unit to clear up some awkward point. This is quite usual with a technical subject.

Programming is like playing a musical instrument: you can only learn it by practice. You must therefore complete all the programming problems in the course. As soon as you can, start making up and solving problems of your own.

When you complete the course, you'll be able to use the computer for many different purposes. For instance, you can have it administer tests or quizzes, you can make it play games which you invent yourself, and you may find it useful for sums and accounts. The games or other applications can include coloured moving pictures to your own design, and sounds to emphasize your meaning — beautiful tunes or rude noises!

Programming is, however, a very large subject, and no one could do it full justice in a single course. After a while you will probably want to take your programming further. You may, for instance, be interested in solving more complicated problems, or in using the machine as a controller for a model railway or private telephone exchange. The second volume in this series, entitled An Introduction to Basic: —Part 2 will help you achieve these complex and sophisticated aims.

Well — enough talk. It is time you started on Unit 1. Good luck!

UNIT:1

This book is about two of the computers in Commodore's range. One of them is the Commodore 16, and the other is the more sophisticated Plus/4. The two machines have distinctly styled cases, different store sizes and keyboard arrangements, but in all important matters of programming they are the same. One book will serve for both.

Where the differences matter (mainly in the early units) the machines are decribed separately. The sections are distinguished like this:

This is about the COMMODORE 16 (and doesn't apply to the Plus/4).

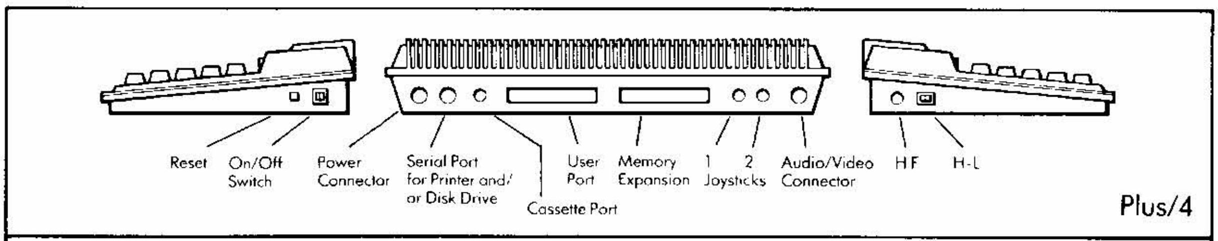
Here is something which refers only to the Plus/4 and has nothing to do with the 16.

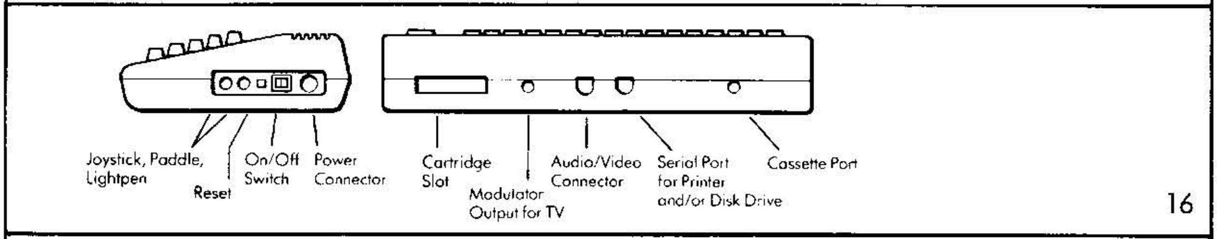
The first unit is designed to get you started using your computer. It explains a number of rather ordinary matters; practical questions which often raise serious problems when someone buys their first computer.

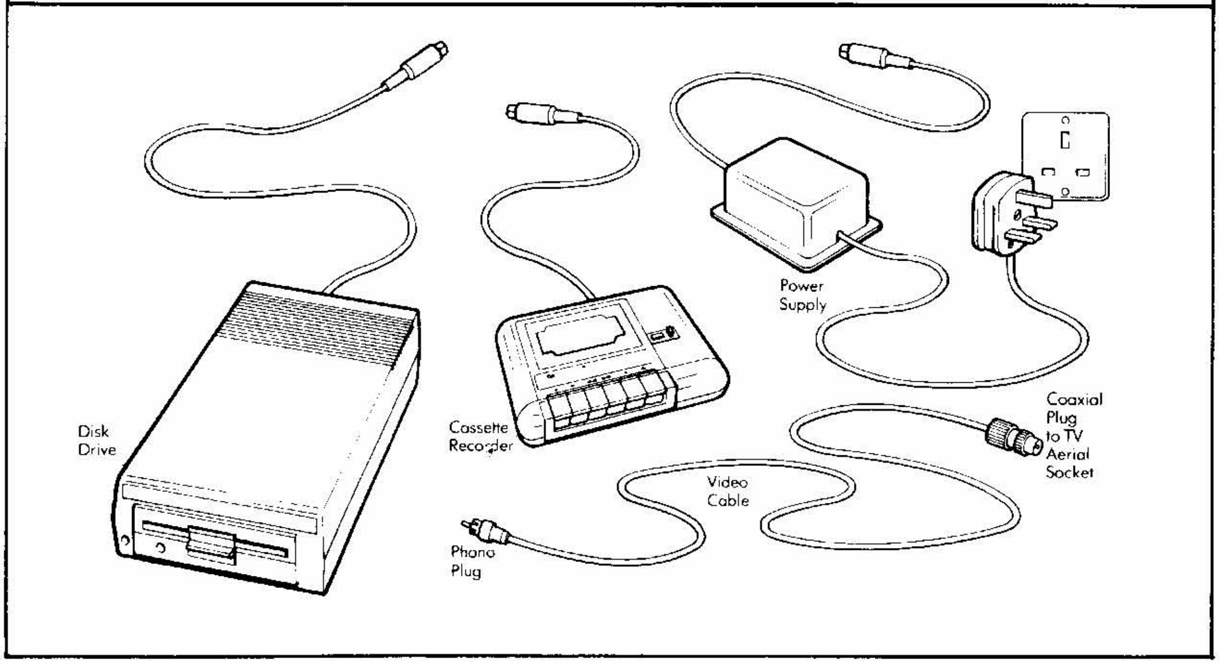
To learn programming you need the right surroundings. Find a quiet comfortable place, and timetable yourself long periods (at least 2 hours) at a time when you are not too tired to concentrate. Do everything you possibly can to avoid disturbance — put a notice on the door, take the telephone receiver off the hook, and tell everyone in your family that you are busy: there is nothing that makes programming more difficult than constant interruptions!

If you have already installed your computer and used it, you can skip straight through to Experiment 1. Otherwise, read quickly through the unit even if you know what it's all about; you may still find it useful.

First, arrange your equipment and connect it to the mains. The computer and cassette or disk drive go on the table in front of you, the power supply can be put on the floor under the table, and the TV should be about 6 feet (2 metres) away if it is a small portable set. Sets with larger screens should be put even further away. The pictures and text you can expect to see are quite large enough to be read at normal viewing distance, and if you try it, you will find that working with a screen close to your face is uncomfortable and tiring.







The various units are connected together as shown in the diagrams. All plugs should slide into their sockets with gentle steady pressure. Never use force, but look carefully at the pin arrangements of the plugs and sockets before you try to join them. Computers are robust, but plugs and sockets will get worn and damaged if they are connected and disconnected too many times. Once your computer is set up, aim to leave it undisturbed as long as you can.

If your TV set has to do double duty as a broadcast receiver, get an aerial switch which lets you keep the computer and the ordinary aerial both connected all the time.

Both the computer and the TV can be run from a mains extension cable. The longer the cable, the more freedom you have to arrange the computer just as you want it. If you're using a cassette recorder twin outlets are enough, but if you have a disk drive we suggest you get a 4-way adaptor. The spare power socket will be useful if you ever buy yourself a printer.

If you do decide to get an extension cable, we advise you not to fit the plugs yourself unless you really know how to do it. If you don't understand why we are giving this advice, then you definitely need someone to help! A qualified electrician is by far the best choice.

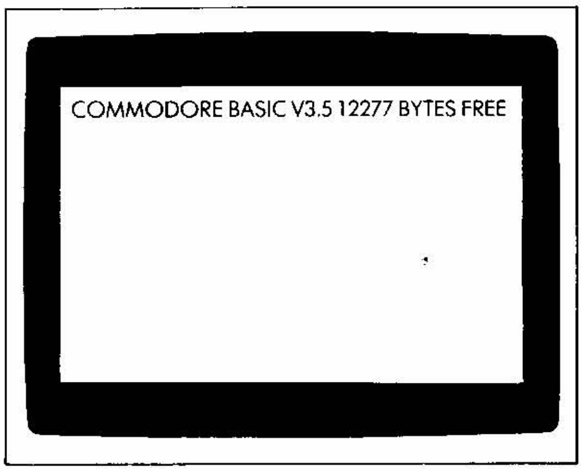
Now you are ready to switch on.

Turn on the TV. If it has a simple tuning knob, turn it to "36" or thereabouts. If it has push-button tuning, select a channel which is not normally used for broadcasts. The set will show you a blank screen and probably hiss loudly. Turn down the volume if you like.

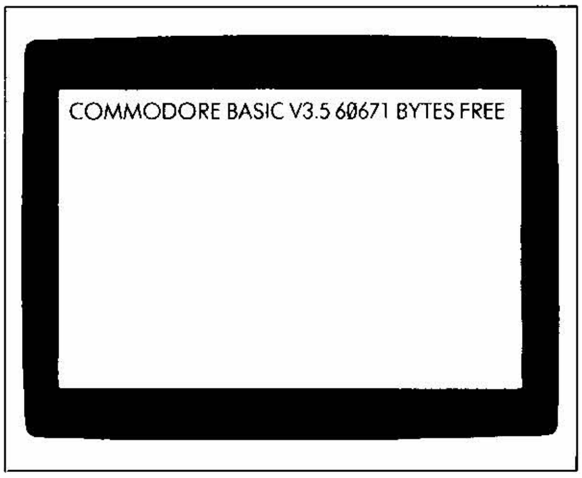
Next, power up the computer, using the switch on the right hand side. If all is well the red power lamp will glow, but unless you are lucky the TV set will still not show a picture.

Now go back to the TV and tune it. The exact method of tuning varies with the make of the set, and is always explained in the manufacturer's instructions. In most cases there is either a knob or a small screw associated with each channel. Sometimes the tuning controls are hidden behind a small panel. If you have to use a screwdriver, don't poke it inside the set, as you could easily get a nasty electric shock.

As you turn the tuning control, a picture will suddenly appear:



```
COMMODORE BASIC V3.5 12277 BYTES FREE
```



```
COMMODORE BASIC V3.5 60671 BYTES FREE
```

The central area is white, with a cyan (light blue) border. You may have to adjust the line hold and frame hold controls to get a steady picture.

If you don't get this picture, turn the computer off for a few seconds and try again.

If you have any difficulty, check the following points:

- Is the TV set working? Try it on ordinary broadcast reception, and have it repaired if necessary.
- Is the computer power light on? If not, check:
  - a) That there is no general power failure.
  - b) That some other device (such as a table-lamp or hair-dryer) will run from the socket you are using. If not, try changing the fuse in the extension plug lead.
  - c) That the fuse in the computer power supply plug is intact (try a new fuse).
  - d) That the power supply is firmly plugged in to the computer.
- Is the computer properly connected to the aerial socket on the TV?

If your system still doesn't work (a most unlikely event), take it back to your dealer for advice and repair.

The message now on your screen consists of a number of 'characters' including letters and numbers. These characters are always the same size, and when the screen is full it can hold 1000 of them.

The first line on the screen identifies the product: a BASIC system designed and manufactured by Commodore Business Machines. The 'V3.5' is a version number which may be changed from time to time.

The message in the last part of the line tells you how much memory there is in your machine. Every computer needs a 'memory' to store details of the job it is doing for you. Memory is measured in 'bytes', each of which can hold just one symbol or character of information. The more memory, the more complex the task the machine can handle.

The correct figures for available memory are 12277 for the COMMODORE 16, and 60671 for the Plus/4. If you expand your computer by plugging extra memory, the figures will be larger.

If the number on the screen is less than you expect, this is a clear sign that the computer is broken. It should be returned to your dealer for repair.

The next line tells you that the computer is now READY to obey commands typed on the keyboard.

The following line shows a flashing black square. This is called the cursor. When you type a command on the keyboard, the cursor indicates, in advance, exactly where each character will be displayed. For example, try the following:

P R I N T 5 + 8 RETURN

(This takes 9 key depressions:

P R I N T 5 + 8 and the RETURN key, which is at the right of the keyboard.)

Before you start typing, touch the SHIFT LOCK key to make sure that it is not locked down. As you type each symbol (except RETURN) it appears on the screen and the cursor moves on by one place. The main job of the RETURN key is to make the computer carry out an instruction: in this instance to PRINT (that is, to display) the result of adding 5 and 8!

To do anything useful, the computer must have a program.

Programs are stored on cassette tapes or floppy disks, and this first experiment will give you practice in loading a program from a tape or disk into the machine. If you have a cassette unit, follow the instructions immediately below. If, on the other hand, you are fortunate enough to be equipped with a disk drive, skip to Experiment 1.2, which is designed specially for you.

EXPERIMENT 1·1

Loading programs from tape:

1. Make sure that the cassette unit is plugged in to the computer.

2. Press EJECT on the cassette unit.

3. Open the holder on the tape unit, and put in the first of the two tapes provided with the course. It is labelled TAPE 1. You can load it either way up since both sides are the same. In any case, the tape window should be facing towards you. Close the holder. If it doesn't close flat do not force it but make sure you have put the tape in the right way.

4. Press the REWIND key on the recorder. Watch the cassette through the window, and if you see it spinning, wait till it stops. Please make sure you are at the beginning of the tape.

5. Press the STOP key on the recorder.

6. Now type the following message:

LOAD"TESTCARD" RETURN

This takes 15 keystrokes in all, counting " as a single stroke. To produce the " symbol, you will need to find one of the two SHIFT keys (either will do) and hold it down while you hit the key marked " 2

Remember to release the SHIFT key as soon as (but not before) the " appears on the screen.

You have to get the message right. Some common faults which you should avoid are:

Typing with the SHIFT LOCK key down. You will get a strange pattern with lines, hearts and spades, and nothing will happen.

Using two single primes ' ' instead of a double quote ". The computer will reply:

?SYNTAX ERROR

READY.

and you can try the command again on the next line.

Putting a space between " and T, TEST and CARD, or D and ".

Typing the letters R E T U R N instead of using the RETURN key.

Nothing will happen.

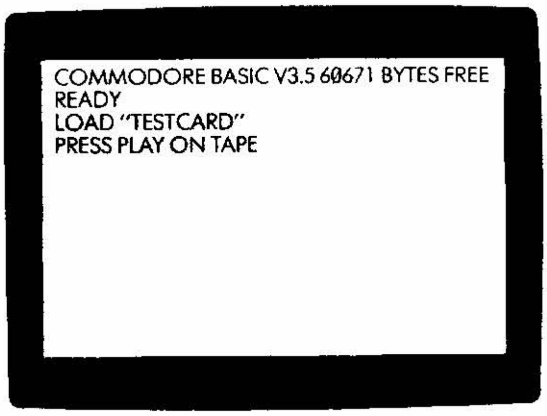
Using digit Ø instead of letter O in the word LOAD.

If you make a mistake, you can always 'rub out' by tapping the INST DEL key. Each depression erases one character and moves the cursor back one place.

7. If you give the message correctly (or even if you make a mistake in spelling the word TESTCARD) the machine will reply

PRESS PLAY ON TAPE

giving a picture like this:



```
COMMODORE BASIC V3.5 60671 BYTES FREE
READY
LOAD "TESTCARD"
PRESS PLAY ON TAPE
```

Press the PLAY key on the cassette unit. The screen will go completely blank and you will see the tape spin. After about half a minute the message will come up:

FOUND TESTCARD

and the tape will stop. This means that the cassette unit has wound the tape up to the beginning of the section on which the TESTCARD program is stored. To load the program quickly, hit any letter key on the keyboard. (The program will actually start to load itself if you do nothing for long enough.) When you have pressed the key the screen will go blank again, and the program will take about three minutes to be transferred from the tape to the computer.

If, after waiting for all this time, the computer comes up with the message

FOUND HANGMAN

this means that you misspelled the name TESTCARD when you typed the original message. Stop the computer by pressing the RUN STOP key and go back to step 4.

In most cases this procedure will work perfectly. If it doesn't and the tape just runs on and on without anything happening, it is possible that the tape has been damaged in some way. Turn it over and try loading the program from the other side. If you still can't manage to load the TESTCARD program, take the tape, the cassette unit and computer back to your dealer for a check-up. This is hardly ever necessary, as all Commodore computers use a special method of recording which is much more reliable than most other makes of machine.

When the TESTCARD program is loaded, the machine will say

READY.

Start the program by typing

RUN RETURN

(4 key depressions).

Alternatively, you can just hold down the SHIFT key and press the long key marked f3 f6, and the program will start by itself.

Experiment 1.1 Completed	

EXPERIMENT 1·2

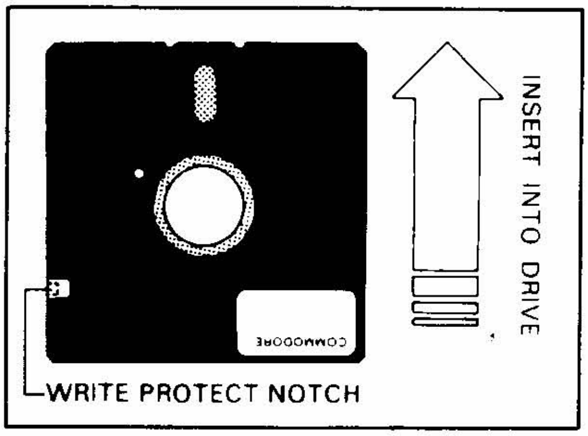
(For readers with disk drives.)

Important: Read these instructions all the way through before you follow any of them.

1. Make sure that the disk drive is connected to the mains supply and plugged in to the computer. The back of the 1541 disk unit has two identical sockets for the connection and you can use either of them. At this stage, don't try putting a disk into the machine!

2. Switch on the disk drive, using the switch at the back. Both the green and red lights at the front will come on. For a second or so, the drive tests itself to make sure that it is working correctly. If all is well the red light goes off, leaving only the green one glowing. If the red light doesn't turn itself off, or starts flashing, this is a sign that the disk drive may be broken. Try switching off and on again, and if the trouble doesn't go away take the disk drive back to your dealer for repair.

3. Open the drive by pushing the lever in and up. Push your program disk firmly but gently into the slot, making sure it goes all the way in. The COMMODORE label should be uppermost, towards you and on the right.



Press down the lever until it springs forward. Your disk is now loaded.

4. Find the long key labelled f2 f5 and press it. Then type the word

TESTCARD

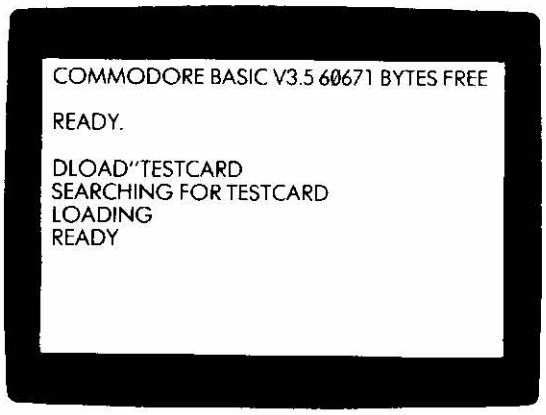
The screen should show the message

DLOAD"TESTCARD

If it doesn't you can rub out any characters which are wrong by pressing the INST DEL key. Each depression removes a character and moves the cursor back one place.

When you're sure the message is right, hit the RETURN key.

If you type everything correctly, the disk drive will clatter gently for a few moments with the red light glowing. Eventually the light will go out, and the screen will be left with the message



```
COMMODORE BASIC V3.5 60671 BYTES FREE

READY.

DLOAD"TESTCARD
SEARCHING FOR TESTCARD
LOADING
READY
```

If this process doesn't work, there are two likely reasons:

a) If you have not connected the disk drive properly or forgotten to turn it on, you will get the message

DEVICE NOT PRESENT

b) If you have forgotten to load the disk, or loaded it the wrong way round, or loaded the wrong disk, the message will be

FILE NOT FOUND

In both cases find out what is wrong, correct it and try again.

Once the program has been loaded correctly, you can start it by typing

RUN RETURN

(4 key depressions). Alternatively you can just hold down SHIFT and strike the f3 f6 key and the program will start.

Before going on, we should draw your attention to some important points about handling disks:

- Never switch the disk drive on or off if there is a disk inside it. Always load after switching on, and unload before switching off.
- Handle the disks as little as possible; keep them away from dust, heat, cold and damp, and never touch the surface which shows through the slot in the paper cover.
- Keep disks away from all electrical machinery such as TV sets, electric motors or metal detectors. If you take a disk through an airport security check, don't let it be X-rayed, but insist on being hand-searched.

(Readers with cassette recorders rejoin here.)

The first program shows you something of the range of colours and sounds your computer can produce. It also lets you adjust your TV set to get the best colour. Remember to turn up the volume control so that you can hear the program!

When you have watched TESTCARD for long enough, you can stop the program by pressing the RUN STOP key.

When you hit this key (or whenever a program stops for any reason) the screen shows a message like

BREAK IN 56Ø

READY.

The 56Ø in the example could be any number. BREAK doesn't mean your computer is broken: it just tells you there has been a break in the sequence of commands which makes up the program.

Experiment 1.2 Completed	

EXPERIMENT 1·3

Experiment 1.3 is a word-guessing game designed to help you get the feel of the keyboard. The program is called HANGMAN. You can load it either from tape or disk, using the commands

LOAD"HANGMAN" RETURN

for tape, or

DLOAD"HANGMAN RETURN

for disk. Remember that the f2 f5 key automatically gives you the sequence

DLOAD"

When the program is loaded and the READY message comes up, type

RUN RETURN or

hit the f3 f6 key while holding down SHIFT, and the game will start by itself. If you don't know how to play, just keep trying letters and watch (and listen) to what happens. You will quickly pick up the idea.

Play the game as long as you like, and use the opportunity to get accustomed to using the letters on the keyboard.

Experiment 1.3 Completed	

UNIT:2

Welcome back. This unit is about the keyboard, and tells you how to use it to write messages and draw pictures on the screen.

If you have ever used an ordinary typewriter, the computer keyboard will look familiar. You will find the letters, the numbers and most of the signs in their accustomed places, and there are the usual SHIFT and SHIFT LOCK keys — although they work a little differently on the computer.

On the other hand, don't be put off if you have never done any typing. You may need a little more time to get used to the keyboard, but that is all the difference it makes.

For this unit only, please don't use the RETURN key unless we say you should. As you saw in Unit 1, this key is the one which makes the machine actually carry out a command for you, such as loading a program or adding up some numbers. At present, just to use the screen, you don't need the help of the computer. If you do press RETURN the computer will only try to obey the message or picture you have just typed, misunderstand it and spoil its appearance.

Another symbol you should avoid just now is the double quote mark ("). This sign has a special meaning, and alters the way the screen reacts to many of the other keys on the keyboard. If a double quote is showing on the screen it can be much more difficult to draw useful pictures. You will learn all about this character in a later unit; but for now, keep off!

You may find this list of "don'ts" quite alarming. Here is another one: Don't worry! Unlike computers in Science Fiction, your machine has no 'self destruct' command. It is absolutely impossible to damage the machine by typing on the keyboard. Some patterns of characters which contain " or RETURN will make it behave quite strangely, and a few sequences, which you might hit by chance if you are careless, will stop the computer from responding to you at all. These troubles are only temporary: you can always cure them by switching the computer off and on again, or by pressing the reset button beside the main switch.

EXPERIMENT 2·1

The COMMODORE 16 has 66 keys in its keyboard, and the Plus/4 has 67. Both keyboards are illustrated below.

The keys are divided into two categories:

- 48 'symbol' keys which make the computer show characters on the screen. ('Space' counts as a symbol key.)
- 18 (or 19) 'function' keys, which control the way characters are drawn.

On the COMMODORE 16, the function keys are

ESC ← → ↑ ↓ INST DEL CTRL

CLR HOME RUN STOP SHIFT LOCK RETURN C=

SHIFT SHIFT f1 f4 f2 f5

f3 f6 HELP f7

On the COMMODORE Plus/4, the functions keys are

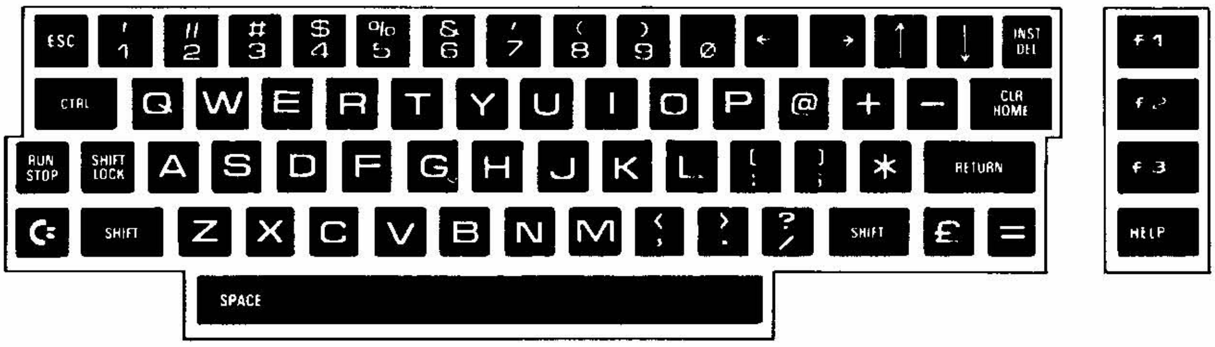
f1 f4 f2 f5 f3 f6 HELP f7

ESC CLR HOME INST DEL CTRL CTRL RUN STOP

SHIFT LOCK RETURN C= SHIFT SHIFT

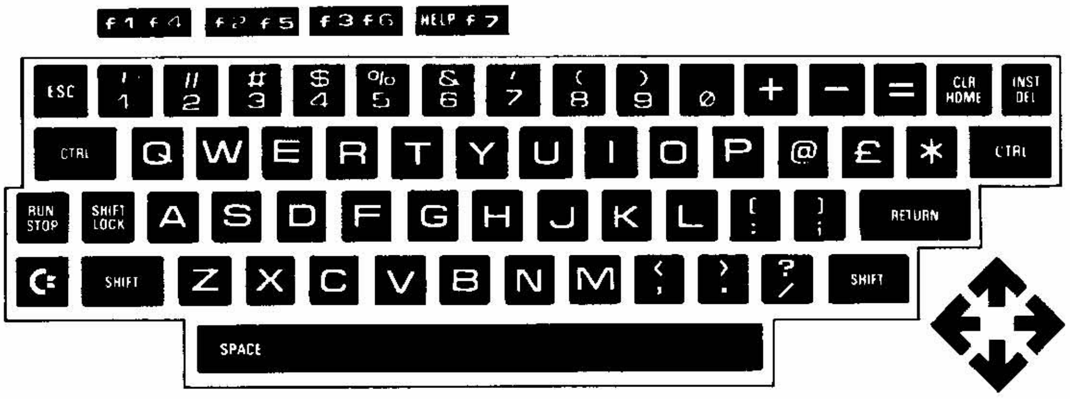
↑ ← → ↓

COMMODORE 16



ESC ! 1 " 2 # 3 $ 4 % 5 & 6 ' 7 (8) 9 0 ← → ↑ ↓ INST DEL
CTRL Q W E R T Y U I O P @ + − CLR HOME
RUN STOP SHIFT LOCK A S D F G H J K L [:] ; * RETURN
C= SHIFT Z X C V B N M < , > . ? / SHIFT £ =
SPACE
f 1 f 2 f 3 HELP

COMMODORE Plus/4



f 1 f 4 f 2 f 5 f 3 f 6 HELP f 7
ESC ! 1 " 2 # 3 $ 4 % 5 & 6 ' 7 (8) 9 0 + − = CLR HOME INST DEL
CTRL Q W E R T Y U I O P @ £ * CTRL
RUN STOP SHIFT LOCK A S D F G H J K L [:] ; RETURN
C= SHIFT Z X C V B N M < , > . ? / SHIFT
SPACE

The only difference between the two keyboards (apart from the layout) is that on the Plus/4 the Control key is duplicated.

Press the SHIFT LOCK key several times and note that it has two positions — up and down. Finally, make sure that it is in the 'up' position.

Now start your machine in the normal way, or press the reset button if the computer is already on. Just below the READY. message you will see the flashing cursor.

In this experiment we examine how the cursor moves when symbols are drawn on the screen. The purpose of the cursor is to show you where the next typed character will appear. Type a few letters, and watch the cursor. Notice that every character *replaces* the cursor, which then shifts to the next position.

Now fill up the whole line with letters, until the cursor is at the extreme end of the white area. Type one more letter and watch what happens: the cursor jumps to the beginning of the next line, all by itself.

Before going on, count the number of letters across the screen, and fill in the box:

There are ☐ spaces for characters on each line on the screen.

Next, type some more lines, and keep going until you reach the bottom of the screen. This isn't difficult, because if you hold down any symbol key it will 'repeat' and fill up the screen quite quickly. Count the number of lines showing and write the number in the box below. Remember to include the blank lines above and below the initial message

COMMODORE BASIC . . .

There are ☐ lines in a screenful of characters.

Now fill in the last line until the cursor reaches the lower right hand corner of the screen. Type one more character and watch the cursor. The whole screen moves up and the cursor moves to the beginning of the next blank line which appears at the bottom. Every new blank line is bought at the expense of the top-most one, which vanishes. The top line has gone for good, and there is no way or bringing it back unless it is stored somewhere else.

Fill in a few more lines, and confirm that the system always gives you room at the bottom of the screen for more text.

Experiment 2.1 Completed	☐

EXPERIMENT 2·2

(For COMMODORE 16)

The computer has 48 symbol keys, but it can display a much larger number of different symbols. They include letters, numbers, punctuation and mathematical symbols, and a wide range of 'graphics' or simple shapes which can be combined to make up different pictures. All these different characters can be selected by using either of the two SHIFT keys (they are connected together inside the computer) and the special 'Commodore' key labelled .

Restart your machine and type the line

1234567890QWERTYUIOP @ +−ASDF GHJKL:;★ZXCVB

(These are all symbol keys, taken row by row.)

Now hold one of the SHIFT keys down and type the line again. You will get an almost completely different line of symbols (including a " but this will not trouble you if you follow the instructions). Copy the symbols into the second row of the table below, and notice how some of the graphics (for example those on U and I) fit together. Note that the graphic symbols usually reach the edges of the little squares they occupy, so they can be made to touch each other.

Next, type the line a third time, but this time holding down the key. Many of the signs are different again. Copy the line into the third row of the table.

To examine the other graphics, repeat the experiment with the line NM,./£=, followed by 33 spaces, which will take you to the edge of the screen.

Note and remember that digit 0 is different from letter O. You should always use 0 (with the bar through it) to show you mean the number, not the letter.

Press and SHIFT down together. The capital letters on the screen will change to lower case letters, whilst some of the graphics will change into upper-case letters. Press the keys together again (or hold them pressed down) and the capitals come back. In general, you can use either a full set of graphics, or a restricted set and lower case letters, but not both at the same time. The use of small letters will be discussed in the second volume in this course.

SYMBOL	1	2	3	4	5	6	7	8	9	0	Q	W	E	R	T	Y	U	I	O	P	@	+	−	A	S	D	F	G	H	J	K	L	:	;	★	Z	X	C	V	B
SHIFT																																								

SYMBOL	N	M	,	.	/	£	=																																	
SHIFT																																								

(For COMMODORE Plus/4)

The computer has 48 symbol keys, but it can display a much larger number of different symbols. They include letters, numbers, punctuation and mathematical symbols, and a wide range of 'graphics' or simple shapes which can be combined to make up different pictures. All these different characters can be selected by using either of the two SHIFT keys (they are connected together inside the computer) and the special 'Commodore' key labelled C= .

Restart your machine (or press the RESET button) and type the line

1234567890+−=QWERTYUIOP@£★A SDFGHJKL:;ZXC

(These are all symbol keys, taken row by row.)

Now hold one of the SHIFT keys down and type the line again. You will get an almost completely different line of symbols (including a " but this will not trouble you if you follow the instructions). Copy the symbols into the second row of the table below, and notice how some of the graphics (for example those on U and I) fit together. Note that the graphic symbols usually reach the edges of the little squares they occupy, so they can be made to touch each other.

Next, type the line a third time, but this time holding down the C= key. Many of the signs are different again. Copy the line into the third row of the table.

To examine the other graphics, repeat the experiment with the line VBNM,./ followed by 31 spaces, which will take you to the edge of the screen.

Note and remember that digit Ø is different from letter O. You should always use Ø (with the bar through it) to show you mean the number, not the letter.

Press C= and SHIFT down together. The capital letters on the screen will change to lower case letters, whilst some of the graphics will change into upper-case letters. Press the keys together again (or hold them pressed down) and the capitals come back. In general, you can use either a full set of graphics, or a restricted set and lower case letters, but not both at the same time. The use of small letters will be discussed in Part Two of this course.

SYMBOL	1	2	3	4	5	6	7	8	9	Ø	+	−	=	Q	W	E	R	T	Y	U	I	O	P	@	£	★	A	S	D	F	G	H	J	K	L	:	;	Z	X	C
SHIFT																																								
C=																																								

SYMBOL	V	B	N	M	,	.	/																																	
SHIFT																																								
C=																																								

Experiment 2.2 Completed	

EXPERIMENT 2·3

So far you have been limited to displaying characters strictly in sequence, left-to-right and from the top down. This is a tedious way to draw a picture, and it would be far more convenient if you could place your text and graphic symbols at any position you choose.

This can be done with the cursor keys, of which there are five:

CLR HOME, ←, →, ↑ and ↓. On the Plus/4 (not the 16) the arrowed keys are actually shaped like arrows.

When you type CLR HOME by itself, it moves the cursor back 'home', which is the top left-hand corner of the screen. Reset your computer and strike this key. You will see the cursor move to the corner of the screen.

The four arrowed keys are used to move the cursor in the direction of the arrow. If you now strike ↓, the cursor moves one line down so that it is over the C of COMMODORE. The C remains visible because the cursor is transparent. Try moving the cursor along the line, using the → key, and then move it back to the start with ←. None of the characters in the line will change as long as you use only the cursor keys.

If you hit a symbol key, the character under the cursor is replaced by the new character, and the cursor advances one place. For example, if you put the cursor on the C of COMMODORE and type the letters EXCELLENT, the line will read

EXCELLENT BASIC . . .

(Of course, any other 9-letter word will do equally well for this experiment.)

Now use the → key to move the cursor to the B in BYTES, and change this word to GULPS.

Like all the other keys, the cursor control keys repeat, so that if you keep one pressed down the cursor will move continuously at about 1Ø character places a second. This is useful if you want to move around quickly.

If you press the → key when the cursor is at the end of a line, it moves on to the beginning of the next line. Similarly, ← at the beginning of a line moves the cursor back to the end of the previous line.

The ↑ and ↓ keys move the cursor directly up and down, with no sideways motion.

The top and bottom of the screen are handled in a special way, and it is worth noticing what happens.

Fill up the screen with a few lines and move the cursor to the end of the last line. When you move the cursor past the end of this line, using → or ↓, the whole screen moves up just as if you had added another character. The top line disappears

Now go back 'home' and try to move the cursor backwards. The screen does not move down as you might have expected; nothing happens at all and the cursor stays in the same place.

Now press CLR HOME whilst holding down one of the SHIFT keys. The cursor moves home and the screen is cleared, giving you a fresh screen to work on.

Practice making drawings on the screen using the graphics symbols and cursor control keys. Start with some simple geometrical shapes like squares, oblongs, triangles and small circles. If you make a mistake, move the cursor back and type the right character. SPACE will get rid of characters in the wrong place.

When you have got the feel of using the graphics, draw a box, like this, with your name in it.

CHRIS
CHARNOCK

Now draw some playing cards, with curved corners and the right symbols (we suggest you keep to black cards worth 1Ø or less).

Finally, if your artistic talent is up to it, try something like an animal, a space-ship or a human face.

Plan your picture first, using the grid below:

Experiment 2.3 Completed ☐

EXPERIMENT 2·4

Everyone makes mistakes when typing. If you get a single letter wrong in the middle of a word, you can correct it with the cursor controls. For example, if you type AUSTRAPIA when you mean AUSTRALIA, you can move the cursor back over the P and change it into an L. Try it!

Unfortunately, if you get the wrong number of letters (too few or too many) this method won't help you. A more powerful facility is provided by the INST DEL key, which lets you insert or remove characters from the screen.

When you type INST DEL by itself, it rubs out the character to the left of the cursor and shuffles all the other characters on the line one place left so as to fill in the empty space.

For example, suppose that you mistakenly type INXDIA when you mean INDIA. You want to get rid of the X, so put the cursor over the D (not the X!) and hit INST DEL. The X disappears, and DIA all move up one place, leaving INDIA — without a space in the middle.

Now try using the INST DEL key to make some corrections, as follows:

CHAINA to CHINA

EEGYPT to EGYPT

FINLANDIA to FINLAND

AUSTRALIA to AUSTRIA

In practice, the most common use of the key is to get rid of characters you have just typed. The key will remove the last symbol and reposition the cursor, all in one movement. You will soon get accustomed to hitting INST DEL whenever you make a typing mistake.

The other function of the INST DEL key can be called up by typing it as a shifted character: that is, holding down the SHIFT key when INST DEL is struck. This function is used to insert spaces into the middle of words or lines. These spaces can then be filled up with characters in the ordinary way.

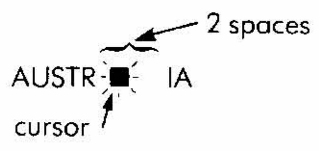
Try the following example, which involves changing AUSTRIA into AUSTRALIA.

Clear the screen (SHIFT and CLR HOME) and type

AUSTRIA

Move the cursor back over the I.

Hold down the SHIFT key and strike INST DEL twice. Each time the IA moves one place to the right. The cursor stays in the same place, so after two moves you get



Now finish by filling in the AL. Move the cursor past the end of the word, using the → key (not the space bar, since this would rub out characters you still need).

To practise using the INST DEL key, clear the screen, fill it up with the list of words on the left, and then change each one to the corresponding word on the right:

HOTEL	MOTEL
MICROPHONE	MICROCOMPUTER
PLYWOOD	WOOD
ANGLE	ANGEL
CHAP	CHEAP
WRITER	WRITTEN
ACTOR	AUTHOR
BALL	BARREL
WIRE	REWIRE
FLOWER	FLOUR
MOON	MORON
PIDGIN	PIGEON
TACT	TACIT
HORSE	HOARSE
WING	WARRING
TAXI	TAX
READ	READY
MERRY	MERCURY
JOVE	JUPITER
PAL	PASCAL
BACK	BASIC
JAVA	JAMAICA

And now change them all back again.

Experiment 2.4 Completed ☐

The Unit 2 program is entitled SPEEDTYPE. It helps you get familiar with the keyboard. Load it by typing

```
LOAD"SPEEDTYPE"
```

or

```
DLOAD"SPEEDTYPE
```

Start the program with the RUN command and practice using it as much as you feel is necessary.

UNIT:3

The COMMODORE 16 and the Plus/4 are both colour computers. This unit introduces you to some of the ways you can get the machine to draw many-coloured pictures on your TV screen.

If your TV set is a black-and-white model, do not expect brilliant results from Unit 3! You should work through it just the same.

EXPERIMENT 3·1

Use the TESTCARD program (Unit 1) to make sure that your TV receiver is properly adjusted.

Stop the program by hitting the RUN STOP key.

You will see that the cursor at this stage is black. Now the cursor can change colour, and as well as telling you *where* the next symbol will be placed, it also indicates *what colour* it is going to be. Try typing a sequence of ▄ symbols (C= and I keys) and note that they appear in black, which is the present colour of the cursor.

The colour of the cursor can be changed at any time by typing one of the eight colour keys while holding down the CTRL key. The colour keys are marked 1 to 8, and also carry abbreviations of the colours they control.

Hold down CTRL and strike the colour keys in succession.
When you hit 1 (BLK) the cursor changes to *black* (unless it's black already).
When you hit 2 (WHT) the cursor turns *white*.

The other keys change the cursor to *red, cyan, purple, green, blue* and *yellow* respectively. When the cursor is white, it is invisible against the white background: it just seems to disappear.

You can get another range of eight colours by using the C= key instead of CTRL.

They are the less brilliant pastel shades:

C= and 1 gives *orange*
C= and 2 gives *brown*
C= and 3 gives *yellow-green*
C= and 4 gives *pink*
C= and 5 gives *dark green*
C= and 6 gives *light blue*

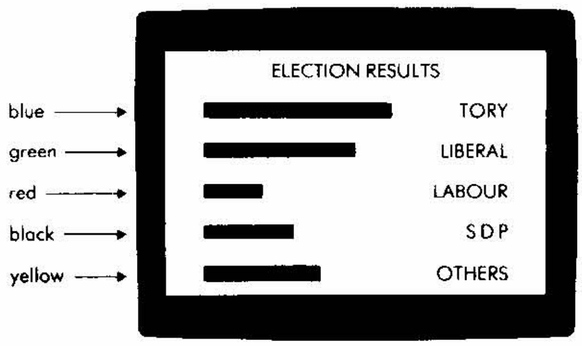
and 7 gives *dark blue*

and 8 gives *light green*.

Now try making some coloured pictures. A good way to start is to make some coloured bars of various lengths. Use the graphic (and U) to build up each bar. For example, to make a red bar 5 symbols long, first hold down CTRL and type 3 (RED); this will change the cursor to red. Then hold down and strike U five times.

You can change the colour of the cursor any time you like. When you have got the feel of the colour keys, try drawing an "election results chart" as shown in Figure 1. Keep the lettering black to emphasise the colours of the bars.

Figure 1



Experiment 3.1 Completed	☐

EXPERIMENT 3·2

You have seen how the computer looks after the colours of individual symbols on the screen. It can also control the shades of the outer frame and the background against which the symbols appear. The machine doesn't have any dedicated keys to control these colours, and the way to change them is to give special commands.

Type COLOR 4, 5. Check it carefully (you must use the North American spelling for the command) and correct if necessary, and then hit the RETURN key. The frame of the TV screen immediately turns purple.

This special command has three parts:

COLOR: This is called a *keyword*.

4: This number tells the computer which part of the screen is to be affected by the change. Thus

COLOR Ø, . . . would alter the background of the screen,

COLOR 1 would refer to the cursor, and

COLOR 4 is used for the frame.

5: This is a colour code. The full table is

Colour	Black	White	Red	Cyan
Code	1	2	3	4
Colour	Purple	Green	Blue	Yellow
Code	5	6	7	8
Colour	Orange	Brown	Yel.-green	Pink
Code	9	1Ø	11	12
Colour	D.Green	L.Blue	D.Blue	L.Green
Code	13	14	15	16

You can now see why the command "COLOR 4,5" turned the frame purple. Try some other commands of the same sort, like

COLOR 4, 2 RETURN

COLOR 4, 14 RETURN

The screen background can be changed in the same way, using COLOR Ø, . . . For example, try

COLOR Ø, 8

If you make a mistake when typing a COLOR command, you can expect one of the messages

?SYNTAX ERROR if the command is not spelled right

or ?ILLEGAL QUANTITY ERROR if the numbers are not in the right range (Ø-4 for the first one, and 1-16 for the second).

If you are drawing a picture, a good plan is to start by setting the frame and background colours by suitably chosen COLOR commands, and then clearing the screen with a CLR HOME and SHIFT.

If you set the background colour and the cursor colour to be the same, the cursor will be invisible. Certain other combinations (like yellow and orange) make the characters impossible to distinguish from the background. If you 'lose' the cursor, you can always regain it by pressing the RESET button.

Experiment 3.2 Completed	

EXPERIMENT 3·3

You may have noticed some strange gaps in the graphics characters; for instance we have ▚ but not ▞; we have ◤ and ◥ but not ◢ or ◣. Furthermore it seems impossible to fill a complete square with colour and therefore to build up large areas of the same hue.

The *reverse field* facility comes to our help.

When a character is displayed in reverse field, the colours of the characters and of its background are swapped. Try the following experiment:

Reset the computer, and type a few characters, including

▚, ◤, ◥ and a space. Now hold down CTRL and the 9 key (also labelled *RVS ON*).

Then release CTRL and type a few more characters. They will appear in white on a black background, instead of black on white. In particular ▚ comes out as ▞, ◤ and ◥ are changed to ◢ and ◣, and a space appears as a solid block of black. We say the the machine is in *reverse* mode.

To bring the following characters back to normal mode, hold down CTRL and type the Ø key (labelled *RVS OFF*).

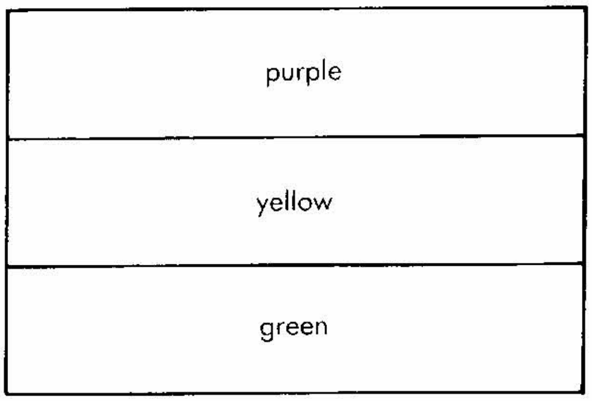
The cursor does not show whether the machine is reverse or normal mode. If you are using many reversed symbols it is easy to forget, and to be in some doubt as to how the next character will appear. This difficulty can be resolved in two ways:

(a) Type the next character and look at it. If it is wrong, erase it, change the mode, and try again.

(b) Type the RVS ON or RVS OFF key, as appropriate, before you type the next symbol. If the machine is already in the right mode this will not make any difference.

A useful way of filling up the screen with blocks of colour is to use reversed spaces. The space bar is a 'repeating' key, and if you hold it down it will generate a sequence of spaces at about 10 per second.

Drawing national flags makes a good way of getting practice in the use of colour. The easiest type of flag to reproduce is one with horizontal stripes, such as that of Luxembourg.



To paint this flag, set the frame to a suitable colour (say black) and the background colour to the same as the bottom right hand corner of the flag (green). This is done by

COLOR 4, 1 RETURN

COLOR Ø, 6 RETURN

Next move the cursor home and clear the screen, select purple and reverse mode (hold down CTRL and type 5 PUR and then RVS ON). Then hold down the space bar and fill up 8 lines with reversed purple spaces.

Next, select yellow and fill up 9 lines with yellow reversed spaces.

Lastly, change the colour to green. This will make the cursor disappear and leave an 8-line green area at the bottom of the flag.

It is important to set the background colour to one of the colours which appear on the picture, otherwise you will be unable to hide the cursor when the drawing is complete. Likewise the frame colour should be different from any colour which appears in the flag itself.

When you have mastered horizontal flags, try one with vertical stripes, such as Italy. Those with crosses (Switzerland or Iceland) are also worth drawing.

Even more difficult are flags with diagonal elements such as Czechoslovakia. The parts of the flag near the sloping lines must be made with the graphics ◢ or ◣ suitably reversed if necessary. If you think about it you'll see that the background colour must be chosen so that it is on one side of every sloping line. In the case of the Czechoslovakian flag, a suitable background colour is blue; red — say — wouldn't do because there would be no way of drawing a blue and white element on the upper diagonal.

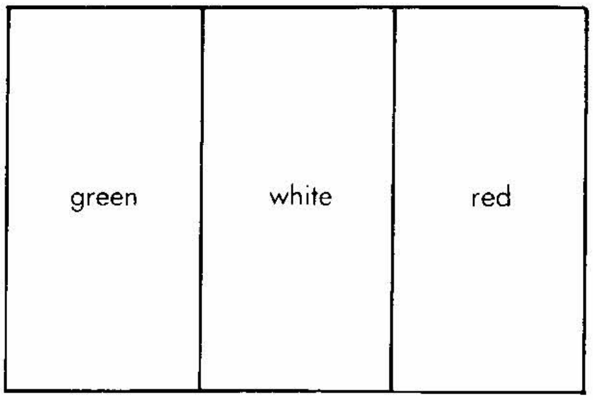
If a flag has a diagonal bar right across it (such as the flag of Tanzania) it is better to use only 25 of the 40 columns, and to fill in the rest with the 'frame' colour. Assuming a black frame, a typical line halfway down the Tanzanian flag would be entered as:

CTRL and GRN, CTRL and RVS ON
SPACE SPACE
CTRL and RVS OFF, SHIFT and £, SPACE,
CTRL and BLACK, CTRL and RVS ON
SHIFT and £, SPACE SPACE,
CTRL and RVS OFF, SHIFT and £, SPACE,
CTRL and BLUE, CTRL and RVS ON,
SHIFT and £, SPACE ...... SPACE

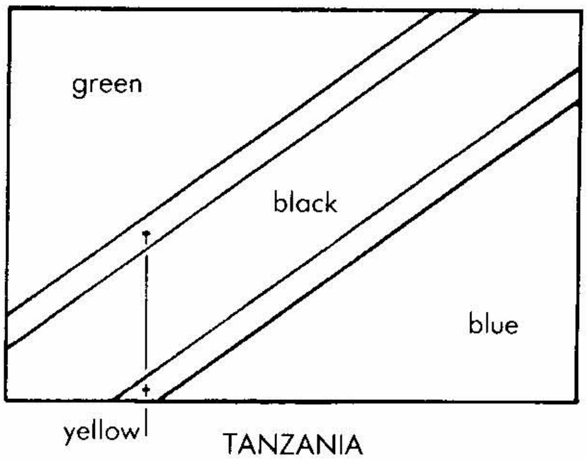
The commas as well as the word "and" shown above are simply to show you the different commands so that they are easier to follow. They should, of course, not actually be used when drawing the flag.

If you feel sufficiently patriotic, you might try the flags of St. Andrew, St. David, St. George or St. Patrick. The Union Jack is formidably difficult to draw (even without a computer) and could well be omitted.

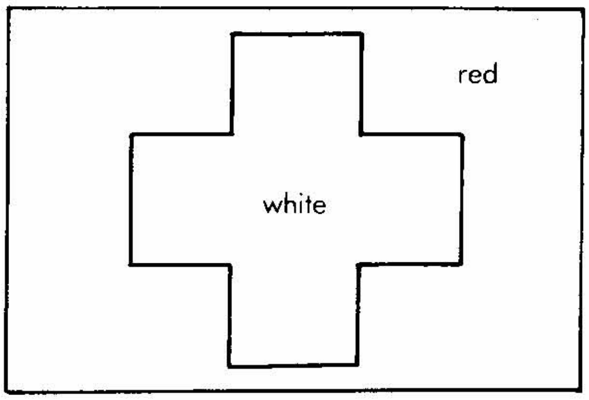
Experiment 3.3 Completed	☐



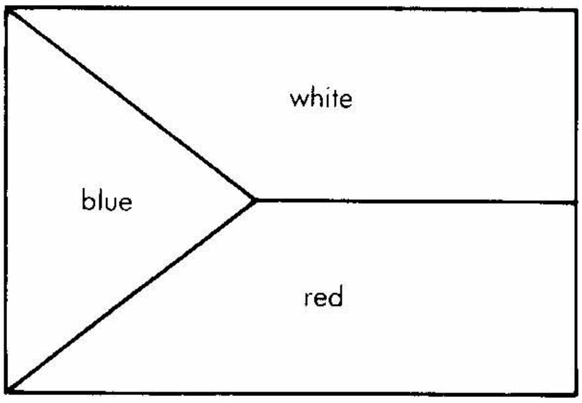
ITALY



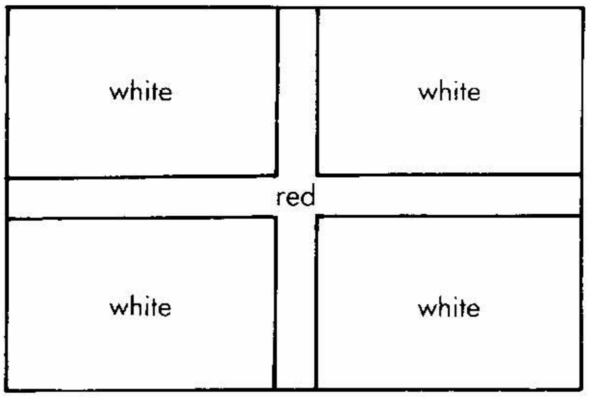
TANZANIA



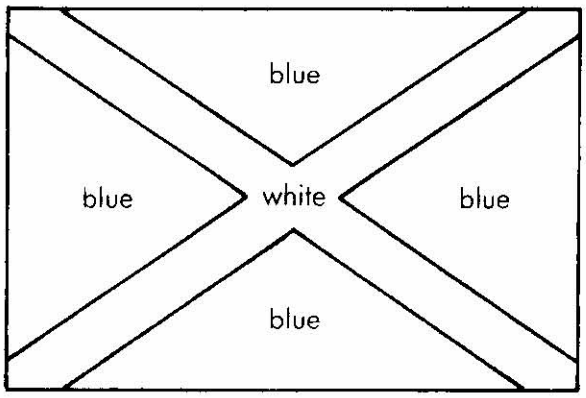
SWITZERLAND



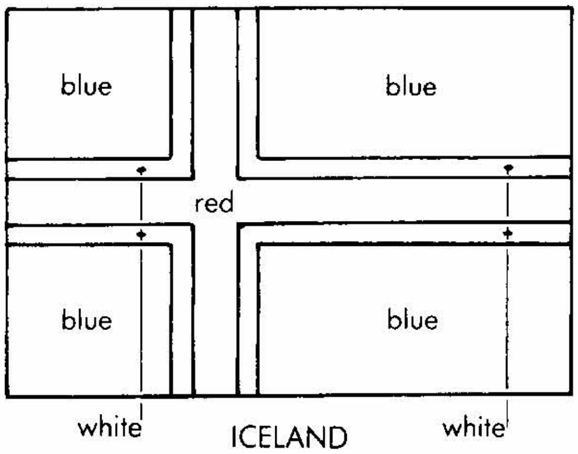
CZECHOSLOVAKIA



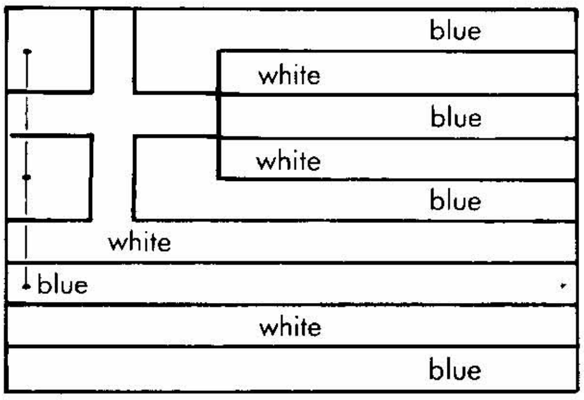
ST. GEORGE



ST. ANDREW



ICELAND



GREECE

EXPERIMENT 3·4

Another useful facility in the COMMODORE 16 and Plus/4 is that of flashing characters. Any character can be made to flash on and off, at the same rate as the cursor. Flashing is controlled by the two keys marked "Flash on" and "Flash off" near the space bar.

These keys are used in much the same way as RVS ON and RVS OFF. To see an example, try typing the sequence

VOTE Flash on and CTRL FLASH GORDON
Flash off and CTRL FOR PRESIDENT

Draw the best coloured picture you can, using the full scope of the computer. You may find it useful to plan your picture first, using a sheet of squared paper and some coloured crayons.

Experiment 3.4 Completed	

The program which goes with Unit 3 is a quiz, and can be loaded by typing

```
LOAD "UNIT3QUIZ"
```

or

```
OPEN 1, 8, 15, "I"
LOAD "UNIT3QUIZ",8
```

UNIT:4

In the first three units of the course we have concentrated almost wholly on the computer keyboard, and on using it to display text and paint pictures on the TV screen. This is sound preparation for the next part of the course, where we look at some of the functions the computer can do on your behalf.

As you already know, the computer will do various jobs when it is commanded to do so. The necessary commands are written in BASIC, a simple and popular computer language first devised by Kemeny and Kurtz at the Dartmouth College, USA. BASIC has its own rules of grammar just like any other language, but you may be glad to hear that they are simple to learn, and that you will easily memorise them through practice, without any special effort.

Every BASIC command starts with a 'keyword' such as LOAD or POKE or PRINT. This tells the computer what type of command is meant.

Similarly, every command ends with the RETURN key. This has two different meanings:

If the keyword is the first word on the line, RETURN is a kind of starting gun: "Now go and do it". For example, if you type

LOAD "TESTCARD"

the actual loading starts when the RETURN key is pressed. The other interpretation of RETURN is discussed in the next unit, but here is a short preview:

If the keyword in a BASIC command has a *number* in front of it — such as the "3" in

3 PRINT 5 + 7

then the RETURN key is a signal not to obey the command, but to store it away for later use.

In this unit we concentrate on the first interpretation. Our commands will not have numbers in front of them, and RETURN will be a cue for the computer to take immediate action.

EXPERIMENT 4·1

One of the most useful and flexible commands is PRINT. It makes the computer work out something for you and display the result on the screen. The word PRINT is used because the original BASIC system at Dartmouth relied on mechanised teleprinters which really did print the answers on rolls of paper.

Experiment 4.1 is arranged in three stages. First, we try out a number of different PRINT commands and make careful notes of the results. Next, we discuss the features of the command which have shown up in the examples; and finally, we examine some new PRINT commands, and try to predict what the computer will do with them. The answers can be checked by using the computer itself.

Begin by typing these commands, ending each one with the RETURN key. Be sure to get the commands right, using the cursor control keys to correct your typing if need be. Make a careful note of the responses in the boxes provided. The first two boxes are already filled in for you:

PRINT 999	999 READY.
PRINT "HELLO"	HELLO READY.
PRINT —56	
PRINT 3+4+4	
PRINT 5★7	

PRINT 27/7	
PRINT "COMPUTER"	
PRINT VIC	
PRINT 3,5	
PRINT 3;5	
PRINT "RABBIT", "DOG"	
PRINT "CAT"; "FISH"	
PRINT "3+5"	
PRINT 29–12; "LIONS"	
PRINT 1;2;3;4	

Before reading on, study your notes carefully and see how many different features of PRINT you can pick out. One common aspect of the answers is that they are followed by READY, but this is true of any command obeyed directly from the screen, so it hardly counts as a special property of PRINT!

Here then are the most important points of the PRINT command;

1. The command can handle both *numbers* and *strings*, and it does so in different ways:
   A *number* can either be given explicitly (like 999) or in the form of an expression or "sum" which the computer works out. The expressions in our trials were 3+4+5, 5★7, 27/7 and 29–12, so we see that the computer can add, take away, multiply and divide. The signs ★ and / mean multiplication and division, respectively. (If you are interested in using the computer for more advanced mathematics calculations, you will be glad to hear that these expressions can be as complicated as you need. They can include brackets, and all the special functions you would expect to find on a scientific calculator. You are advised to look at Appendix A, which is an extra unit designed specially for you.)
   A *string* is *any* sequence of characters enclosed in double quote marks. The PRINT command simply regurgitates such a string exactly as it was given, without trying to process it in any way. The strings in our tests were
   "HELLO", "COMPUTER", "RABBIT", "DOG", "CAT", "FISH", "3+5" and "LIONS".
   Notice that "3+5" is *not* an expression, even though it looks like one; it is enclosed in quotes, so it must be a string.
   If you want to display a string, but forget to put quotes round it, you will probably get 0 (although you might sometimes get something else).
2. The PRINT command can handle two or more quantities or strings at the same time. If the two are separated by commas, then the second result is spaced well across the screen (just over halfway). If a semicolon is used, the separation is less. In particular, strings are not separated at all (this is how we get "CATFISH"). Numbers displayed by the computer seem to be separated because every number is always preceded by a space (or by a — sign if it is negative) and always followed by another space. If your records don't show this very clearly, repeat the command

   ```
   PRINT 1;2;3;4
   ```

   and 'measure' the result by moving the cursor over it.
3. Next we look at spaces inside the command itself. The keyword PRINT must be compact (that is, its letters must not be separated by spaces), and any space which is *inside* a string belongs to that string and will be reproduced. Otherwise, spaces between strings or numbers are ignored. Thus

   ```
   PRINT   3       +5;7;           8
   ```

   will give exactly the same result as

   ```
   PRINT3+5;7;8
   ```

This rule — that spaces in commands are ignored everywhere except inside keywords and in strings — is generally true for the whole of BASIC.

4. If you make a mistake in the keyword itself, or if you supply an expression which doesn't make sense (such as 5★★7) the computer will reject your command with the comment

?SYNTAX

ERROR

This is computer jargon for saying that you have broken the rules of BASIC. There is nothing for it but to correct the command and try it again.

Now run through the following list of PRINT commands and predict what the computer will do with each one. Show how the results will be spaced: this is just as important as getting the results right in themselves. On the other hand, don't bother writing down READY. each time.

Some of the commands *may* contain deliberate errors.

Check your answers on the computer. If you have made any mistakes, and can't see why you've made them, go back and repeat the whole experiment, until the ideas become clear in your mind.

Note that the computer always carries out calculations involving multiplication and division before addition and subtraction. As examples:

PRINT 5+2★7 will give 19

and

PRINT 5+2+6/3−3 will give 6.

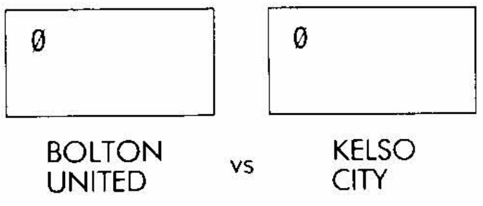
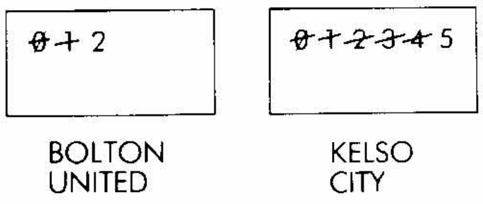
Command	*Your prediction*	*The computer's result*
PRINT 94		
PRINT 8 — 5		
PRINT 3 ★ 2 + 5		
PRINT "OH DEAR"		
PRINT ENOUGH		
PRINT "1★/3"		
PRINT 3; 47		
PRINT 2+2;2−2;2★2;2/2		
PRINT "CLOUD"; "BURST"		
PRIN 8 — 7		
PRINT "18", "MICE"		
PRINT 53+★7		
PRINT −1;−2;−3		

Experiment 4.1 Completed	

EXPERIMENT 4·2

If a computer could only do one command at a time, it would not be specially valuable to anyone. At best, it would be about as good as a (non-programmable) calculator. Most useful computer jobs consist of whole sequences of commands, controlled by sets of instructions called "programs". As the commands in the sequence are obeyed, there has to be some way of 'keeping the score', of remembering how far the job has gone, and of passing the results of one command on to the next. The memory which serves to link commands is provided in the form of *variables*.

Before discussing BASIC variables, we shall give you a human analogy. Suppose you are the score-keeper at a football match. Your instructions could be as follows:

> Before the match starts, draw two boxes, label them with the names of the teams, and write zeros inside them, thus:
>
> 
>
>
> Whenever either side scores a goal, replace the most recently written number in the corresponding box by the same number, *plus one*. Rub out (or cross out) the old number.
> Part-way through the match, the state could be
>
> 
>
>
> When the final whistle blows, use the scoreboard to display the names of the teams together with the (latest) numbers inside the boxes, thus:

BOLTON UNITED	2
KELSO CITY	5

In this example the boxes are *variables*; the numbers in the boxes serve to remember the current state of play, and change from time to time as necessary; but the *labels* remain the same for the duration of the match. The instructions for using them are very simple, but foolproof.

The memory of the computer (it probably has 60671 or 12277 bytes — remember?) is a bit like a large blackboard. When the machine is first switched on the board is wiped completely clean. Then, whenever a variable is first mentioned the computer "draws a box" by setting aside part of the memory, and labels it with a name chosen by the human user. Then it "writes a number in the box" by storing the appropriate value in the memory which has been set aside.

The BASIC command which makes the computer do all this has the keyword LET. Let's examine such a command in detail:

```
LET X = 5
```

Here, the variable name is X. The computer will set up a box called X (if it has not already done so) and will put the number 5 inside it. If a variable X already exists, then no more space is set aside; the 5 merely *replaces* the previous value. Study the following cases:

	X does not exist *(Case 1)*	X already exists *(Case 2)*
Before	(Memory empty)	$\boxed{37}$ X
After	$\boxed{5}$ X	$\boxed{5}$ X

Result of LET X = 5

When you give a LET command, the computer just says

```
READY.
```

There is no evidence on the screen that the machine has done anything at all. Fortunately, we are helped by the PRINT command, which displays the value of a variable whenever it is mentioned by name. Try the following sequence of commands:

	Your results
LET Z = 14 PRINT Z	
LET Z = 31 PRINT Z	

If you keep the right order, the *first* value of Z to be printed will be 14, and the second, 31. The first LET command both *creates* a variable called Z, and gives it the value 14; the second one merely *changes* its value to 31.

At this stage we have to give you a few simple rules about variables and their names.

There are two kinds of variables in BASIC:

- Numeric variables, for storing numbers.
- String variables, for storing strings (e.g. words or phrases).

The choice of names for variables is quite restricted. A numeric variable can be called by a single letter, a letter followed by a digit, or by two letters. Some examples of possible names for numeric variables are

A, X, Z, B5, TX, PQ

Names for string variables always end in the $ sign, but otherwise the rules for string variables are the same as for numeric variables. Examples are

C$, Z$, P7$, DB$

To show the use of string variables, try typing

```
LET T$ = "GOOD  "

PRINT T$; "MORNING"
```

The value which follows the = sign in a LET command doesn't need to be a simple number or string; it may be an *expression*, and furthermore it can use the current values of variables by referring to their names. For example, look at the following sequence of commands:

```
LET Q = 5

LET S = Q+3
```

The first one creates a variable called Q (it is a number variable because of its name) and sets its value to 5. The second one makes a variable called S. It then takes the value of Q, adds 3, and puts the result into S. To illustrate the point, try running these two commands, and inspect the result by typing:

```
PRINT Q; S
```

Now look at the following sequences and *predict* the outcome of the PRINT statement in each case:

LET AA = 15 LET B = 33—AA PRINT AA,B	
LET D = 3 LET E = D★D+7 LET F = E—D PRINT F;E;D	
LET F=4 LET F=F+1 PRINT F	

Did you get the last one right? Some people might find it a bit tricky.

There is no limit to the number of different values a variable can hold, as long as it only holds one at a time. A command like

LET F = F+1

means: *First* work out the expression (by taking the value of F and adding 1)

Then put the result in box F, replacing the previous value.

In other words, the command makes the computer *add 1* to the current value of F.

The signs which allow us to combine numbers in various ways are called *arithmetic operators*. They are +, −, ★ and /. BASIC also allows strings to be manipulated in various ways by using *string operators*. Only one of them combines two strings; it is called "concatenation" and is written as a + sign. The operator simply attaches the second string to the end of the first, so that

"DOG" + "ROSE" = "DOGROSE"

Look at the following sequence of commands, and predict the outcome of the PRINT's. Then try the sequence on the computer; remember to put a space before each of the closing quotes:

```
LET B$="DOG SPACE "
LET C$="BITES SPACE "
LET D$="MAN SPACE "
LET E$= B$+C$+D$
LET F$= D$+C$+B$
PRINT E$
PRINT F$
```

PRINT and LET are the two most frequently used commands in BASIC. It is worth remembering that when you use the computer you are allowed to replace the word PRINT by a single symbol: the query (?). LET can be omitted altogether. A valid sequence is

```
A=5

B=17

?A,B
```

The program which comes with this unit is designed to give you plenty of practice with PRINT and LET commands. It is called UNIT4DRILL.

You can stop the program when you are sure you fully understand the use of numeric and string variables.

Experiment 4.2 Completed	

UNIT:5

EXPERIMENT 5·1

The time has come to look at stored commands. Let's begin by showing that the computer really can put commands away in its memory, and then fetch them out again later.

Start up your machine (or if it is already running another program such as SPEEDTYPE, stop it by typing RUN STOP) and give the command NEW (followed, as usual by the RETURN key).
This command makes the computer wipe its memory clean, just as a teacher cleans the blackboard at the start of a lesson. You won't see anything happen except the READY. response, because the memory is all inside the computer.

Next, type the *labelled* command

```
1Ø PRINT 13+59
```

(NOTE: "one zero", not "eye oh") and follow it by pressing the RETURN key.
The only visible result is that the machine moves the cursor to the beginning of the next line. The result of the sum 13+59* is *not* worked out or displayed on the screen. Instead, something invisible *has* happened: the computer has remembered the command and put it away in its internal memory.

To verify this, first clear the screen (using the SHIFT and CLR HOME keys) and then give the command

```
LIST
```

If you have done everything the right way, a copy of the labelled command reappears on the screen. This proves that it was in the machine all the time.

An alternative way to give the LIST command is to hold down the SHIFT key and hit the long key marked HELP f7 .

So far, our PRINT command has been stored and retrieved, but it hasn't actually been obeyed. The computer is still to tell us what 13+59 is! To find out, we type

```
GOTO 1Ø
```

remembering to use letter Oh's (not digit zeros) in GOTO.

This tells the computer to *execute* the command labelled "1Ø". It does so, and the answer finally appears. You can do this as many times as you like. It does not destroy a command to have it listed or executed.

The computer can remember many commands at the same time. (The limit is set by the size of the memory: it takes one byte to hold each character in a command, plus a little bit of overhead for the command as a whole.) Every command must have its own label in front of it, and all the labels must be different. The machine always stores and lists commands in increasing order of label, and obeys them in this order too unless it is commanded not to.

Try typing NEW

```
1Ø PRINT "FIRST LINE"

2Ø A=5

3Ø B=1Ø

4Ø PRINT A;B;A+B

5Ø STOP
```

Remember to end each command with RETURN

Now try a LIST, and then a GOTO 1Ø, and check that the results are those which you expect. The STOP will make the computer stop and display a READY when it reaches the end of the sequence of commands.

The sequence in which commands are stored is kept right even if you type them in a different order from their label numbers. For instance, if you had typed:

```
3Ø B=1Ø

1Ø PRINT "FIRST LINE"

4Ø PRINT A;B;A+B

5Ø STOP

2Ø A=5
```

these five commands would still have been listed and obeyed in the order 1Ø, 2Ø, 3Ø, 4Ø, 5Ø. Clear the machine with a NEW, and try it for yourself.

Always start by making numbers go up in steps of 1Ø. If you decide later to slide some extra

**There is nothing special about the sum 13+59. Any other PRINT command would have done equally well for this example.*

commands in between the others, this rule makes it much easier: you can then use intermediate label numbers such as 15 or 38.

Why bother storing commands at all? There are two good reasons:

- Commands which are fetched out from the computer's own internal memory are executed much faster than if they are typed in.
- Commands which have been typed in once can be obeyed many times over. Practically every useful job done by a computer involves repetition, and it is only sensible to put the commands into the computer's memory, where they are easy and fast to get at.

Perhaps the easiest way to get repetition is to *store* a labelled GOTO command. Consider the following program:

```
10 PRINT "NORTH"

20 PRINT "WEST"

30 PRINT "SOUTH"

40 PRINT "EAST"

50 GOTO 10
```

When it is started at label 10, the computer obeys the first four commands in sequence. The next command sends it back to label 10 so that it starts the sequence all over again. It just keeps going round and round, and only stops when you type RUN STOP or turn the machine off.

Now clear the machine and type in the program. Start it by giving the initial command

```
GOTO 10
```

You will see the machine obeying the lines of your program, much faster than you can read them. You can slow the machine down by pressing and holding C= (try it), or you can stop it in the usual way with the RUN STOP key.

At this point, we will actually show the advantage of using label numbers separated by 10. Suppose you want to alter your program so that it includes the diagonal direction

NORTH

NORTH-WEST

WEST

SOUTH-WEST

etc.

You need four new instructions *in between* the existing ones. If you number them 15, 25, 35 and 45 they will go in just the right places. Type the following:

```
15 PRINT "NORTH-WEST"

25 PRINT "SOUTH-WEST"

35 PRINT "SOUTH-EAST"

45 PRINT "NORTH-EAST"
```

Now LIST your program, and check that your new lines have been inserted between the old ones, in the right places. Run the program and see what happens.

Now write and test your own program on the same lines. If you use graphics characters in the strings instead of letters, you can get some interesting patterns on the screen.

The GOTO 10 command you have been using to start your program has a more convenient equivalent: RUN. RUN simply makes the computer start obeying commands at the one with the lowest number.

When you put a semicolon *after* a string, the computer doesn't take a new line between that string and the next one when it runs your program (but of course *you* must still end each command with the RETURN key). Instead, it starts a new line across the screen only when it reaches the right-hand edge. A simple program like the one below will quickly fill the whole screen with curious designs; try it, and explain its action.

```
10 PRINT "├┤┬┴ ┌┐└";

20 GOTO 10
```

Experiment 5.1 Completed	☐

EXPERIMENT 5·2

A sequence of commands which is repeated over and over again is called a loop. A loop may include many different sorts of commands, including a LET, which gives a new value to a variable. Look at this program, and try to predict its action:

```
1Ø LET A = 1

2Ø PRINT A

3Ø LET A=A+1

4Ø GOTO 2Ø
```

Pretend you are the computer and do exactly what the computer does, patiently, step by step. Write down what happens to the variable A and its values. Don't read on until you have thought hard and filled in your answer.

(and so on)

Now enter the program and run it, holding down the C= key to slow it down. (But don't touch C= until you have typed RETURN after RUN.)

I'm sure you found this problem quite easy, but here is an explanation of what you saw.

The program begins by obeying the command labelled 1Ø, which gives the variable A the value 1. The next command displays this value on the screen.

Command '3Ø' replaces A by A+1. This is the same as adding 1 to the old value of A, so the result (this time) is 2. The next command is a GOTO, and makes the computer return to command 2Ø! The value of A is displayed again, but now it is 2. The machine again works through the sequence 2Ø, 3Ø, 4Ø and again, and again, but each time round the value of A is increased by 1. This gives the sequence 1,2,3 . . .

To give you some practice, try predicting the first few lines displayed by these two programs (remember, ★ means "times"):

1Ø B=Ø	1Ø A = 1
2Ø PRINT B	2Ø B = A★A
3Ø B = B+3	3Ø PRINT A,B
4Ø GOTO 2Ø	4Ø A=A+1
	5Ø GOTO 2Ø

And now check to see if you were right.

You can do the same kind of thing with strings. Try this program:

```
1Ø X$ = "★"

2Ø PRINT X$

3Ø X$ = X$ + "#"

4Ø GOTO 2Ø
```

The successive values of X$ as the program goes round the loop will be ★, ★#, ★##, ★###, and so on. The string X$ gets longer and longer, and uses up more space on the screen each time it is printed. After some 45 seconds, the string gets so long that it won't fit in the machine's memory (the largest number of characters allowed is 255), and the machine reports a fault:

```
? STRING TOO LONG

ERROR IN 30
```

The line ERROR in 30 means that the command which tried to store the offending string was the one labelled 30.

Here are some more programs for you to predict.

10 A$ = "++"	10 A$ = "XY"
20 PRINT A$	20 PRINT A$
30 A$ ="A"+A$+"—"	30 A$ = A$+A$
40 GOTO 20	40 GOTO 20

Remember that if a letter comes *inside* a string, it is just a letter and not a variable name. So "X" has nothing to do with variable X or X$.

As a final exercise, write a program with a simple loop, and run it for exactly one minute, timing it with your watch. Then stop it, see how far it has gone, and *calculate* how many commands the machine has obeyed in the time. Reduce your figure to the number of commands *per second*, and write your answer here:

Experiment 5.2 Completed	

The self-test quiz for this unit is called UNIT5QUIZ.

UNIT:6

The purpose of this whole course is to help you learn how to design and build your own programs. To back up your growing knowledge of programming you will need a collection of techniques or "tools" to organise your work, and to help in putting things right if they go wrong. This unit is a tool kit and puncture outfit. It isn't about programming as such, but the contents will be useful in an emergency. Read the unit carefully, get to know the techniques it describes, and give it a permanent place in your mind as you go further into the course.

If you are using a cassette recorder to load your programs you can skip this section.

If you have a disk drive, we would like you to format a new disk before starting Experiment 6.1.

As you already know, a single disk can be used to store many different programs. Every disk has an extra item; it is an index or 'directory' which lists the names and sizes of the programs on that disk.

Try the following experiment. Load the program disk supplied with the course and type

```
DIRECTORY   RETURN
```

This command will fetch the directory from the disk into the computer and display it on the screen. It will read something like:

```
 0 "ITB PROGRAMS          " DT 2A
12 "TEST PROG"                PRG
23 "HANGMAN"                  PRG
13 "SPEEDTYPE"                PRG
```

and so on, down to

```
470 BLOCKS FREE
```

This list is worth studying. The top line, which appears in *reversed* characters, gives the identity of the disk itself. In this case the name, ITB PROGRAMS, was chosen by Commodore. The "DT 2A" is a serial number which also belongs to the disk.

Next there is a line for each program. The first entry gives the size of the program in *blocks*. Each block holds 256 bytes, so that you can see that — for example — HANGMAN is 23 x 256, or 5888 bytes long. The second entry gives the name of the program, and third, "PRG", is the same for every program.

The line at the end of the directory tells you how many blocks are still left unused. ITB PROGRAMS has room for another 20 HANGMAN-sized programs, or more if they are smaller. The capacity of a completely empty disk is 664 blocks.

Another, faster, way of getting a disk directory is to strike the f3 f6 key (by itself).

Now you understand how program disks are arranged, you can go on to the next step.

Get a new blank disk of the right type, preferably from your Commodore dealer. Make sure that the slot on the left as you load the disk is clear – not stuck over with a silver label* like the ITB PROGRAMS disk.

Turn on the drive, and load the disk. Then type

```
HEADER "disk name",D0,I01   RETURN
```

where "disk name" stands for any title you want to give your disk, such as SUE'S PROGRAMS. The title can be up to 16 characters long. D0 is the drive number. The sequence which follows the comma is a serial number. It must start with the letter I.

The actual line you would type might be

```
HEADER "SUE'S PROGRAMS",D0,I01
RETURN
```

Wait about a minute, until the red light on the disk drive stops glowing. Then type

```
DIRECTORY   RETURN
```

You will get an *empty* directory; it just says,

```
0 "SUE'S PROGRAMS         " 01 2A
664 BLOCKS FREE
```

You have just *formatted* a disk. Every new disk you buy has to be formatted just once in its lifetime, although it needs to be *initialised* every time it is loaded. If you format a disk which has already been used to store programs, you will destroy everything recorded on it. You have been warned!

Take your formatted disk out of the drive, write its title on the label and set it aside.

**The label makes it impossible for the disk drive to record anything on the disk, and is a way of preventing the course programs from accidentally being destroyed. Since you will eventually be recording your own programs on the new disk, you don't need protection!*

EXPERIMENT 6•1

Load and run the Unit 6 program, called SENTENCES. Take a look at the 'random' sentences it displays. These absurd statements are constructed by a form of internal 'consequences', where each word or phrase is selected by chance from a short list of possibles. Here we shan't worry about how the program works (although it is quite simple in principle) but we'll use it as an example in showing you how to list, alter and preserve large programs.

When you have seen enough of the sentences, stop the program with the RUN STOP key, and do a LIST. The program is far too long to fit on the screen, and as the listing runs most of it disappears from the top of the frame. At the end, only the last eleven commands can be seen.

The BASIC language includes some special versions of the LIST command to allow for this situation. There are five possibilities, which you should try out as you read about them:

- You can list the whole program by typing LIST or by hitting the SHIFTed f7 key. This, as is now clear to you, has certain drawbacks if the program is too long.
- You can list a selected command by giving its number. For example

  LIST 11ØØ

  will display command 11ØØ (and no other).
- You can list all the commands *up to* a given label number by putting a – sign in front of the number. Thus

  LIST – 8Ø

  shows all the commands from the beginning of the program up to the one labelled 8Ø.
- You can ask for all the commands *from* a given label number up to the end of the program by putting a – sign *after* the number:

  LIST 9Ø9Ø–
- Finally you can list all the commands *between* any two numbers by quoting both numbers:

  LIST 2ØØØ – 2Ø9Ø

Now use some of these types of LIST command to look at various parts of the program. You will quickly notice that the label numbers don't always go up in steps of 1Ø; this is because the program was altered many times after it was first written.

At the head of the program and in several other places you will see commands with the keyword REM, followed by descriptive statements in English. REM is short for "remark". These lines play no part in the program itself, but are included to make the program easier for people to read. When you begin to write complicated programs you should always use plenty of REMs to explain what you are doing.

When you are satisfied that you have fully understood the various forms of the LIST command, try some experiments to see what happens if:

(a) The command you refer to isn't there.

(try LIST 65Ø)

(b) The label numbers are in the wrong order.

(try LIST 11ØØ – 1ØØØ)

(c) The LIST command is included in a program. Type NEW, then type in this program and run it.

```
1Ø PRINT "LIST TRIAL"

2Ø LIST

3Ø GOTO 1Ø
```

Hold down the key so that you can see exactly what is happening. Eventually stop the program with the RUN STOP key.

Experiment 6.1 Completed	☐

EXPERIMENT 6·2

This experiment discusses how programs can be altered and modified. At present you will be making changes to a program originally written by someone else, but later most modifications you make will be to your own programs.

There are three kinds of change you can make to a program:

(a) Removing existing commands

(b) Adding new commands

(c) Amending or replacing existing commands.

Removing Existing Commands

There are eight ways to get rid of a labelled command in the computer's store, but five of them involve deleting or changing the entire program and are quite drastic in their effect.

A whole program can be deleted by

- Switching the machine off.

or
- Typing NEW

or
- Loading a new program from cassette tape or disk.

or
- Pressing the RESET button.

or
- Typing SYS 32768 RETURN

An individual command can be removed

- By typing its label number alone
- By typing another command with the same label number.

A group of commands can be removed by typing DELETE followed by the line numbers of the first and last lines you want to get rid of. For example,

```
DELETE 150-230
```

will remove line 150, 230 and all the lines in between.

DELETE has almost the same variants as LIST:

DELETE −100 will discard all lines up to 100.

DELETE 120− will throw away each line from 120 to the end of the program.

DELETE 200 will remove only line 200.

DELETE by itself does not erase the whole program; it is reported as a SYNTAX ERROR.

If you use DELETE at all, do so with great care. A single mistake can do a great deal of damage to your program!

Reload the SENTENCES program and delete a few lines which have the REM keyword. Check that the deletion has worked by LISTing an appropriate part of the program both before and after.

Adding New Commands

A new command can be added to a program by typing it, with a suitable label number. The command is inserted at the place determined by the label number.

We have already practised inserting commands in Unit 5, but you can take this opportunity to insert a few REM commands. Make sure you don't replace any existing statement, or the program won't work.

Altering Existing Commands

The most drastic way of altering a command is to retype it, using the same label number.

Let's try an alteration. Begin by replacing the copyright line (label 5) with a line containing your own name. The dialogue might go:

```
          ┌─► LIST 5
          │
          │   5 REM COPYRIGHT © ANDREW
          │     COLIN 1984
          │
You type ─┼─► 5 REM CHRIS BLOGGS
          │
          └─► LIST 5

              5 REM CHRIS BLOGGS
```

Try altering a few more lines, but keep to those with REM keywords, otherwise you will almost certainly damage the program and prevent it from working properly. A program is like a living cell; random mutations are nearly always bad and usually fatal.

When a line needs only a minor change, it is often easier to alter the original (which is already on the screen) than to type a new version. This is done with the cursor and possibly with the INST DEL key. When the changes are complete, the RETURN key will make the computer register the new command in place of the old.

Suppose you want to alter line 1ØØ, so that it reads

```
1ØØ REM MAIN STUPID SENTENCE GENERATOR
```

LIST command 1ØØ, put the cursor on the S of SENTENCE, and insert 7 blanks (use SHIFT and INST DEL). Then type the word STUPID, check that all is correct, and strike RETURN. Do another LIST 1ØØ as a check.

Try a few more alterations of this type, always keeping to REM commands. Remember if you don't strike RETURN after changing a line with the cursor, the machine won't register your changes!

Now give a RUN command. If the program doesn't work any more, you must have made a mistake in editing, such as erasing or altering a statement without noticing. Don't be upset — this is quite common. Just reload the program from the cassette tape or floppy disk.

You must have observed that the SENTENCES program makes statements about well-known figures. The lists of possible choices are very short: they are in commands 9Ø7Ø (for men) and 91ØØ (for ladies). For the final part of this experiment you are invited to alter these lists so that the program makes up sentences about your family and friends instead.

Each of the two commands 9Ø7Ø and 91ØØ has the keyword DATA. This is followed by the list of names, separated by commas. The last name is followed by a comma and the letter Z.*

There can be as many names as you like. If the names run to more than 2 lines, use a second DATA command (with a label number one up on the first one). Third and fourth commands can also be used. Only the *last* DATA command in each group needs the Z at the end.

**The use of Z at the end of a DATA command is a feature of this program only, not of BASIC in general. DATA commands in most other programs don't need the Z at the end.*

Some possible alternatives for lines 9Ø7Ø and 91ØØ could be:

```
9Ø7Ø DATABILL,GEOFFREY,PERCEVAL,MR.SOPHOC
LES,THE HEADMASTER,Z
91ØØ DATAGRANNY,SUSAN,VIOLET,MRS.PINKERTO
N,THE GYM MISTRESS, AUNTIE FLO, RACHEL
91Ø1 DATAPENNY,KATE,LAURA, FRANCES, NORAH, V
ICKY,Z
```

When you've made these changes, run the program again. If it comes up with complete nonsense check that you have put a comma between each name (but not two commas) and that the last name is followed by a Z.

Once you've got the feel of making changes, you can apply your imagination to the other lists of words in the program. They are:

9ØØØ	Actions that people do by themselves (intransitive verbs)
9Ø1Ø	Actions that people do with each other (transitive verbs)
9Ø2Ø	Actions that people do with clothes (transitive verbs)
9Ø3Ø	Items worn by men
9Ø4Ø	Items worn by ladies
9Ø5Ø	A list of adverbs and adverbial phrases, describing actions that people do with each other.
9Ø6Ø	A list of adverbs and adverbial phrases, describing actions that people do by themselves
9Ø7Ø	Men's names
9Ø8Ø	Adjectives describing men
9Ø9Ø	Various sorts of men
91ØØ	Ladies' names
911Ø	Adjectives describing ladies
912Ø	Various sorts of ladies

Alter the lists in any way you like. Remember to keep them consistent. If you alter 9Ø2Ø to actions dealing with food, you must alter 9Ø3Ø and 9Ø4Ø accordingly, otherwise you may get sentences like

SUSAN ATE HER WELLINGTON BOOTS

Experiment 6.2 Completed ☐

EXPERIMENT 6·3

When you have altered the SENTENCES program to say amusing things about your friends, you may want to keep the new version to show at parties, etc. This section explains how to preserve the program on a cassette tape.

If you possess a disk drive, skip this section and read the next one instead.

Get hold of a blank tape, or one with nothing on it that you need to keep. It should be of good quality, and as short as possible: you are only going to use up about one minute's worth of tape, and there is no point in paying for more. The special cassettes made by Commodore are ideal.

Load the new tape into the cassette unit in place of the SENTENCES cassette, and rewind it. Release all the keys on the cassette unit. Then stop the SENTENCES program, and type

SAVE "FAMILY" RETURN

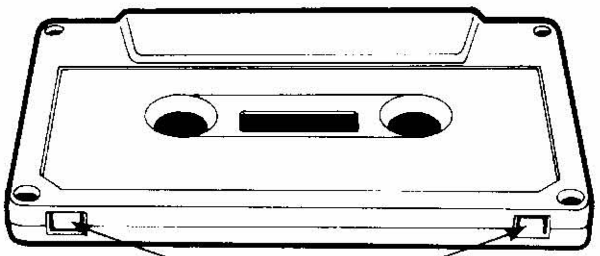
(you can use any name you like instead of FAMILY).

The machine replies

PRESS RECORD AND PLAY ON TAPE

Follow these instructions, pressing both keys on the recorder at the same time.

If the RECORD key won't go down, check that the tape you are using hasn't had its 'write permit' tabs taken away. These tabs are at the back of the cassette, like this:


TABS

If the tabs are broken off, it is almost impossible to put any new material on the tape. The idea is to protect important recordings which mustn't be destroyed, so get yourself another tape.

If all is well, the machine says

SAVING FAMILY

and a moment later,

READY.

In theory, your program is now recorded, but it is better to check. Various things may have gone wrong: you may have forgotten to rewind the tape, or to press the RECORD button, or the tape itself could have a small bald patch which prevents it from making a correct copy of the program. These things shouldn't happen, but in practice they do!

To check your tape, rewind it, and then type

VERIFY "FAMILY" RETURN

The machine replies

PRESS PLAY ON TAPE

Press the PLAY button (but *not* RECORD this time). The machine then looks for your program on the tape, and checks it against what is in the memory. Naturally you mustn't make any alterations between the SAVE and VERIFY commands.

If all is well, the messages you will see are:

```
VERIFY "FAMILY"
PRESS PLAY ON TAPE
OK
SEARCHING FOR FAMILY
FOUND FAMILY
VERIFYING
OK
```

If the machine finds an error, or doesn't get as far as FOUND FAMILY, you must go back to the beginning and try the SAVE command all over again. If the trouble persists, try another tape (or the other side of the first one). If you still can't make the system work, take the computer and the cassette unit back to your dealer for a check-out.

Once a program has been SAVEd, it can be stored and LOADed at any time, with a command such as

LOAD "FAMILY"

To save your program on a disk, remove the ITB PROGRAMS disk and load the one you formatted at the start of the unit.

Now record your version of the program with the command

```
DSAVE "program name" RETURN
```

where "program name" is a title you choose for yourself. For example,

```
DSAVE "FAMILY" RETURN
```

If all is well, the machine says

```
SAVING 0:FAMILY
```

and a moment later

```
READY.
```

To make quite sure that the program has been recorded correctly, type

```
VERIFY "program name",8 RETURN
```

(where "program name" is your program.)

Notice that in the VERIFY command (unlike DSAVE) the program name must be followed with the sequence ,8.

The machine replies

```
SEARCHING FOR program name
VERIFYING
OK
READY.
```

This process will practically always work as it is described; but if it doesn't try it again carefully, from beginning to end. If it still fails get advice from your Commodore dealer.

When you have DSAVEd the program and VERIFYed it, load and list the directory. Your program should now be included.

A quick way of typing the sequence DSAVE" is to hit the f2 f5 key while holding down SHIFT.

Once a program has been DSAVEd, it can be reloaded in the ordinary way, by a command like

```
DLOAD "FAMILY" RETURN
```

Now that you have a formatted disk, you can keep on storing more programs on it, until the space runs out. Occasionally you will want to get rid of a program and replace it by a newer version with the same name. If you just give the command

```
DSAVE "program name"
```

and a program of that name is already on the disk, the red warning light will flash and nothing else will happen. To *dispose* of the old program you must use an extra character in front of the program name: @ . The command now looks like

```
DSAVE " @ program name"
```

It is best to follow this kind of DSAVE with the special command

```
OPEN 1,8,15,"V"
```

and wait until the red light goes off. This gives the machine a chance to collect up the space freed by the removal of the old program so it can be used again.

A program doesn't need to be perfect to be SAVEd. If you are writing a very long program (or even copying one from a book) it pays to save your work every half-hour or so. This is because the computer's memory isn't as reliable as a tape or floppy disk in a drawer. The machine itself is unlikely to break down, but other accidents can happen. Thunder storms have been known to corrupt the information in a computer store: there may be a power cut, or your baby sister can trip over the mains lead and jerk the plug out of the wall. If you lose six hours of work through such an incident, you may feel a little upset. If you have been taking regular half-hourly dumps you can reload the most recent version and go on with only a small loss of your time.

To make the system absolutely safe, you should SAVE or DSAVE on two different tapes or disks alternately. Then even if the machine stops during a SAVE, with half the old version obliterated by half of the new one, you are still protected.

Experiment 6.3 Completed	

EXPERIMENT 6·4

In this section we point out a subtle and dangerous trap which lies in wait for COMMODORE computer programmers, and tell you how to clamber out if you do fall in.

To begin, we'll try to drop you straight in to a simple example of the trap. Type NEW to clear the store, and then enter the following program, inserting all the spaces shown carefully, and watching the screen as you type.

```
10  PRINT "A  FRIGHTFUL  AND  APPALLING  TRAP"
20  GOTO  10
```

Now do a LIST and check the program, which should appear exactly as shown.

You would expect the program, when typing RUN, to display the message

A FRIGHTFUL AND APPALLING TRAP

over and over again until it is stopped. Try it and see. It is likely to come up with

```
A FRIGHTFUL AND APPALLING TRAP 20

        ? SYNTAX ERROR IN 10

        READY.
```

Even if your program does work correctly, read on and find out just why you managed to avoid the pitfall.

The reason the machine failed to run your program (assuming that is what it did) is by no means obvious. You could show the program to the world's greatest experts on BASIC, and they wouldn't see anything wrong with it.

The difficulty arises because of the narrow screen width of the computer. Inside the machine, any BASIC command may be up to 75 characters long. The screen is only 40 characters wide, so the *displayed* version of a command can spread over 2 screen lines.

When you type a command and the cursor reaches the end of a screen line, the system moves it on to the beginning of the next line; but it still assumes that you are typing the *same* command. A command is only ended by the RETURN key.

If you fell into the trap (as you were supposed to), here is what happened:

You typed the first command (which was carefully designed to fill up the *whole* of a screen line). You then found the cursor at the beginning of the next line and naturally typed the next command, ending it with a RETURN. Since you didn't end the first line with a RETURN, the system thought that both lines were part of the same command, namely

```
10 PRINT "A FRIGHTFUL AND APPALLING
TRAP" 20 GOTO 10
```

This "command" is not correct BASIC, and gives rise to a syntax error when the machine tries to execute it.

This type of error is particularly difficult to find unless you know what you are looking for. You are most unlikely to notice the mistake as you type the program — even experienced programmers often forget to end their commands with RETURN if the cursor is at the beginning of a new line. If you LIST the program, or even the section which includes the error, the faulty command looks exactly like two correct ones, and the fault is invisible.

Fortunately, the error can be pinpointed by LISTing just the command in which the error is reported. If you type LIST 10, out come lines 10 and — apparently — 20! This must be wrong, since you only asked for 10. To correct the trouble, retype both commands completely, remembering to end each one with RETURN

In summary:

(a) Always end every command with RETURN, no matter where the cursor may be.

(b) If the computer reports an error in a command and you can't see anything wrong with it, LIST it out by itself and check whether it runs on into the next command in the program.

Experiment 6.4 Completed	

UNIT:7

The programs we wrote in Unit 5 were undisciplined. Once they were started, it needed drastic action to stop them and most, if left to themselves, would have gone on and on for ever. This unit is about how to control programs, and make them stop when they have gone far enough.

The topics described in this unit are fundamental to computing. When you master them, you take the biggest single step towards being a programmer. Read the unit slowly and carefully, and if you are in any doubt about some point, go back and read about it again. It is well worth doing because these ideas, once you understand them, will take you a long way forward.

The control of programs depends on a key concept, which may be new to you: the *condition*.

When people talk, they mostly make statements which are true, or which are at least supposed to be taken as true:

"My train broke down on the way in."

"I love you."

A condition is a special kind of statement which is not necessarily true, but might equally well be false. In English we use conditions after the word 'if'. In the following sentences the conditions are printed in bold type:

"If **the last train has left**, you'll have to spend the night in Aviemore."

"If **the program doesn't work**, find the fault and fix it."

The speakers of these sentences are not insisting that the last train has gone, or that the program really doesn't work; they simply don't know, and are making plans accordingly. In English, a condition can turn out to be true or false, without the speaker being called a liar.

In BASIC, conditions also come after the keyword IF. They involve the various 'objects' used in programs: number variables, string variables, numbers and strings. The conditions, which can be either true or false in any instance, are built round one of six relationships. This is best illustrated by example:

Consider the BASIC condition:

A < 5

(where < is a sign which means "is less than"). This condition is true if the value of the variable A really is less than 5 (say 0 or 3 or 4.98). It is *false* if A is worth 5 or more.

Another example, this time using strings, is

N$<>"JIM"

(where <> means "is different from").

This condition is true if the variable N$ has any value except "JIM"; thus it is true if N$="JACK" or N$="JIMMY". It is only false if N$ actually is "JIM".

The full set of relationships you can use in BASIC are these:

=	(is the same as)
<	(is less than)
>	(is more than)
<>	(is different from)
<=	(is less than or equal to)
>=	(is more than or equal to)

The relationships <>, <= and >= are each typed with *two* key depressions. These symbols may be more familiar to you in the forms ≠, ≤ and ≥, but the designers of BASIC had to accept the fact that computer keyboards don't usually have keys marked with these signs.

The relationships can all be used either between pairs of numbers, or between pairs of strings to make *conditions*. Numbers and strings can be represented by appropriate variables. Thus

5> 4 is *true* because 5 is greater than 4

7 <=6 is *false* because 7 *is* more than 6

if A = 10 and B = 7,

A >=B is *true*, and so is B<= 7.

When relationships are used between strings they imply alphabetical (dictionary) order, so that

"DOG" > "CAT" is true,

and "JIM" > "JIMMY" is false.

EXPERIMENT 7·1

Suppose that the computer has obeyed the following three statements:

```
LET A$ = "JOAN"

LET X = 5

LET Y = 7
```

Work down the following table, and mark each condition as false or true:

Condition	Value (false or true)
X < 7	
X >= 5	
A$<>"X"	
Y <> X	
A$ < "FRANCES"	
A$ > "JOAN"	
Y = 8	

The quantities on either side of the relationship can be expressions, just as in LET commands. The expressions can be as complex as you wish, but the important thing is to compare like with like: a condition which had a number on one side and a string on the other would make the computer stop and report a fault.

Assume the values of A$, X and Y are the same as above, and work out each of the following conditions:

Condition	Value (true or false)
A$+"NE"<>"JOANNE"	
5 > X	
X + Y <> 13	
X + 2 = Y	

Now check your answers, which are given in Appendix B.

Experiment 7.1 Completed	

EXPERIMENT 7·2

The chief instrument of control in BASIC is the IF command. It consists of the keyword IF, a condition, the word THEN, and a label number. It is very like a GOTO, but with one difference: the jump only happens if the condition is *true*.
An example of an IF command is

```
IF X$ <> "ABBBB" THEN 20
```

Here the condition is X$ <> "ABBBB" and the whole command tells the machine to jump to 20 if X$ is different from "ABBBB". If this condition is false, the machine continues obeying commands in their numerical order.
If you are like most people, your first reaction to this command is that it is a bit absurd. "Either X$ is different from that string with the A and B's" you say, "Or it isn't. It all depends on what comes before, but in any case when the programmer wrote that IF command, he must have known!"
Your view is understandable, plausible, but wrong. There could be two entirely different reasons:

- Suppose the IF command is in a loop where some variable has its value changed every time round. The condition could well be true for some of the values, but not others.
- Suppose again that you are writing a program for someone else to use. Then you won't know in advance what the *user* is going to do with it, but the actions of the program must still depend on what he or she actually does. If you want a good example, the author had to make the various quiz programs respond in a sensible way to your answers even though he had no idea how you were going to reply to any of the questions.

Putting an IF statement in a loop gives a good way of stopping it when it has gone round enough times. Type in and run the following:

```
10 X$ = "A"

20 PRINT X$

30 X$=X$ + "B"

40 GOTO 20

50 STOP
```

This program runs on, filling the screen with ever-lengthening strings of B's until it runs out of space. The STOP at line 50 is never reached.
Now stop it and replace line 40 with

```
40 IF X$ <> "ABBBB" THEN 20
```

When you run it, it displays

```
A

AB

ABB

ABBB
```

and stops!
The reason lies in the condition X$ <> "ABBBB". As the program goes round and round the loop, the condition is at first true (because X$ is AB, and then ABB, and then ABBB, all of which are different from ABBBB). In each of these cases the IF command behaves like a simple GOTO 20 and sends the machine round the loop another time. Eventually, X$ gets to ABBBB. The condition is now false; the jump doesn't happen and the machine drops through to the end of the program where it stops.
Now try altering the condition in various ways, and observe the effect when you run the program. Whatever string you use, make sure that the condition eventually becomes false, otherwise the program will never stop.
Possible conditions to try are:

```
X$<>"AB"

X$<>"ABBBBBBBBBBB"

X$<"ABBA"
```

The same control technique can be used with numerical variables.

```
Type in 10 P = 0

        20 PRINT P, P*P

        30 P=P+1

        40 GOTO 20

        50 STOP
```

Run this program, see what it does, stop it, and change line 40 to read

```
40 IF P < 11 THEN 20
```

Now run the program again. It displays two columns of figures which look familiar, and could be useful to someone who didn't know the squares of the numbers by heart. As a working program there is only one thing wrong: the display isn't labelled, and its meaning is not immediately obvious to anyone but you.

We can fix this defect by adding a heading, or line at the top which identifies each column, like this:

```
NUM        SQUARE

 0           0

 1           1

 2           4

 3           9

 ..        .....

etc.
```

Clearly the heading has to be displayed before any of the numbers or squares, so the command which displays it must come first. Since label 10 is already used, and it would be pointless to change all the labels in the whole program, a sensible decision would be to use label 5. The command itself is a PRINT, with two strings: "NUM" and "SQUARE". The comma between the strings ensures that the spacing corresponds to the spacing between the columns of figures.

The whole program now reads:

```
 5 PRINT "NUM","SQUARE"

10 P=0

20 PRINT P,P*P

30 P=P+1

40 IF P<11 THEN 20

50 STOP
```

Run the program in this form, and examine the output.

Do you want a blank line between the heading and the first row of figures? The command PRINT by itself (without any value or string to follow) will give you an empty line, so try adding the command

```
7 PRINT
```

In a few minutes you will be asked to write some programs of your own. Before you start, let's take a careful look at the programs we have already run, and draw some general conclusions. The example programs are:

(1)

```
10 X$="A"

20 PRINT X$

30 X$=X$+"B"

40 IF X$ <>"ABBBB" THEN 20

50 STOP
```

(2)

```
 5 PRINT "NUM","SQUARE"

 7 PRINT

10 P=0

20 PRINT P,P*P

30 P=P+1

40 IF P < 11 THEN 20

50 STOP
```

If we forget about the heading commands (5 and 7) in the second program, both programs seem to follow a set pattern. In each case,

1. There is a variable which changes regularly as the loop is repeated. You'll see that it is X$ in the first program and P in the second. In general, this is called a *control* variable, and it can be either a string or a number.

2. There is a command which gives the control variable its *starting value*. This command is outside the loop (that is, it is not repeated but only obeyed once).

3. There is a command which is obeyed for every value of the control variable. In our examples, these are the PRINT commands

   PRINT X$

   and PRINT P,P*P

   In practice, this part of the loop can be expanded to include any number of commands, *all* of which are obeyed for each value of the control variable. This group is called the *body* of the loop.

4. There is an *increment* or quantity by which the control variable grows each time round the loop. In our examples, X$ grows by adding a "B", and P is increased by 1. Other increments are possible; for instance a string could grow by 5 symbols at a time, or a number could go up in steps of 2 or any other number. It could also start with a high value and go *down*. The loop always includes a command which moves the control variable one step further each time it is obeyed.

5. There is a *final value* for the control variable. When the loop has been executed with this value, the repetition must cease. The last command in the loop is an IF command, with a condition which is *true* if the loop is still due to be executed, but *false* when the control variable has passed its final value.

In the table which follows, examine each program and fill in the name of the control variable, the starting value, the final value, the increment and the number of times the loop is obeyed. To work this out, it often helps to jot down the value of the controlled variable on the first, second, third .... time through the loop, and to see how many values there are until the final value is reached.

	Control variable	Starting value	Final value	Increment	No. of times round loop
1Ø X$="A" 2Ø PRINT X$ 3Ø X$=X$+"B" 4Ø IF X$<>"ABBBB"THEN 2Ø 5Ø STOP	X$	"A"	"ABBB"	"B"	4
1Ø P=Ø 2Ø PRINT P,P★P 3Ø P=P+1 4Ø IF P < 11 THEN 2Ø 5Ø STOP	P	Ø	1Ø	+1	11
1Ø Y$="Z" 2Ø PRINT Y$ 3Ø Y$=Y$+"XY" 4Ø IF Y$ <>"ZXYXYXY"THEN 2Ø 5Ø STOP					
1Ø R=5 2Ø PRINT R, R/8 3Ø R=R+3 4Ø IF R < 17 THEN 2Ø 5Ø STOP					
1Ø C=27 2Ø H=30−C 3Ø PRINT C,H 4Ø C=C−5 5Ø IF C > 2 THEN 2Ø 6Ø STOP					

When you have completed the table, check your answers against those given in the back of the book (Appendix B).

Experiment 7.2 Completed	

EXPERIMENT 7·3

When you construct a program, you should begin by doing some design, and then writing out the whole program on a piece of paper. Use pencil and *rubber*! Some people compose their programs directly on the computer keyboard, but this method is only for geniuses or morons — it is definitely not recommended for ordinary people. The reason is quite plain: if you start without plans you have about as much chance of success as a builder who puts up a house without any drawings, making up the architecture as he goes along. He might just produce an architectural jewel, but he is much more likely to end up with a leaky hovel which will blow open at the first storm.

When you design a loop for a program, you have to decide all the essential items for yourself. They include the name and type of the controlled variable, the starting and ending values, the increment, and the details of the body of the loop. When you have made up your mind on these points, you can put them together in the standard pattern.

Here is a worked example.

One pound sterling (£1) is worth 2350 Italian Lire at today's rate of exchange. We need a table which gives the Italian equivalent of Sterling amounts from £5 to £75, going up in steps of £5. The display is to start:

```
£     LIRE

 5    11750

10    23500

.............
```

and so on.

Let's think about the loop first. The control variable will clearly be a number, and we can call it PS (this stands for "Pounds Sterling" and is as good as any other name). The starting value will be 5, the final value 75, and the increment 5. The body of the loop is to print a value in £'s, and the corresponding value in Lire, which is 2350 times more.

The elements of our loop can now be jotted down. They are:

```
PS=5 ◄────────────────────── (Sets initial value)
PRINT PS, 235Ø★PS ◄──────────────────── (Body)
PS=PS+5 ◄────────────────── (Increments PS)
IF PS<8Ø THEN◄─ (Checks if final value passed)
STOP ───────────────────── (Stops program)
```

The label number following THEN is left blank because we don't know what it is going to be.

Before writing down the whole program, we should consider the heading. Suitable commands would be:

```
PRINT "£", "LIRE"
```

```
and PRINT ◄──────────── (To get a blank line)
```

Now we can assemble the parts and write out the whole program:

```
1Ø PRINT "£","LIRE"

2Ø PRINT

3Ø PS=5

4Ø PRINT PS, 235Ø★PS

5Ø PS=PS+5

6Ø IF PS < 8Ø THEN 4Ø

7Ø STOP
```

At the risk of becoming boring, let me repeat: *don't* short cut the design process: *don't* improvise your program straight into the computer. If you do, you'll never make a good programmer.

Now try these examples:

1. Write a program which displays a pattern of stars, thus:

```
★
★★
★★★
★★★★
..........
up to
★★★★★★★★★★
```

2. Write a program which gives the equivalent of $US for sums of British money between £10 and £30, going up in Steps of £2. (Take £1 = $1.43.)

3. The relationship between the Fahrenheit and centigrade scales is expressed by this formula

   F = 1.8 ★ C + 32

   Write a program which tabulates Fahrenheit equivalents of Centigrade temperatures between 15° and 30°, going up in steps of 1°.

   (HINT: the body of your program could be

   ```
   F = 1.8 ★ C + 32
   PRINT C, F
   ```

   This has implications for your choice of *name* for the controlled variable.)

When you have written and *run* all these programs, check your solutions against those in Appendix B.

Experiment 7.3 Completed	

People who like school Mathematics and are good at it sometimes get confused by the way that the "=" sign is used in BASIC. If this doesn't apply to you, you can safely skip this section.

In Mathematics, "=" is used in equations, to assert that two different expressions really have the same value. The equation tells you something which is true. For instance, if the Maths teacher writes on the board

"2x + 5 = 9"

you can be sure that for the particular x the teacher has in mind, the statement is right. If this weren't so, you can imagine the following conversation:

Pupil puts hand up.

Teacher: Yes?

Pupil :x is two

Teacher: No. The answer is seventy eight

Pupil :Eh? I don't understand.

Teacher: I lied when I said 2x + 5 = 9!

In BASIC "=" is used in two different senses, neither of which is the same as the mathematical one.

In a LET command, the sign means "becomes". It's an *instruction* to calculate the value of the expression on the right, and to put this value into the variable on the left. Instructions aren't statements, and it doesn't make sense to say that they are, or aren't true. (They may be wrong in a particular context, but that is a different matter.) The trouble is that if LET is left off, the command *looks like* an equation. It isn't. Let's make this clear:

In BASIC

Y=X+2

doesn't *inform* the computer that Y equals X+2; it *orders* it to work out the value of X+2 and put the result in variable Y. Here are some points to ponder:

● Q=Q+5	is a reasonable and useful BASIC command
● P=Q and Q=P }	Are not the same in their effects
● X+1=5	is *not* a legal BASIC command

In each case do you see why? Try to explain it to yourself in your own words.

The other use of "=" is in conditions. You'll remember that = is one of six possible relationships between quantities. Examples of its use are

```
IF X=Y+2 THEN 100

IF N$="YES" THEN 150
```

Again, there is no implication that the condition actually is true; instead the command is an order to work out *whether* the condition is true, and to take certain action if it is. In conditions "=" has the same logical force as any other relationship such as < or >=. It is best to avoid the word "equals" and to call the symbol "is the same as".

To summarise:

BASIC uses "=" in LET commands, where it means "becomes", and in conditions where it means "is the same as", but what it says isn't necessarily true. Got it?

The self-test program for this unit is called UNIT7QUIZ.

UNIT:8

At this point in the computer course you are just beginning to write your own programs. The first ones are short and simple. Later, as you develop your knowledge, experience and skill, you are certain to design and write programs of ever greater complexity and interest. The table gives you some idea of how far you can go:

Program	*Number of Commands*
Converting Italian Lire to £ Sterling (Unit 7)	7
Unit 3 quiz program	about 100
Chess playing program	about 5000
Program to control an industrial robot	about 25000
Program which runs a computerised airline booking system	about 5000000

Naturally, any program with more than about 5000 commands is always the result of a team effort (it would take too long for one person to write) but there is still plenty of scope for the individual programmer.

As you work at programming, you will often find yourself stuck. A program you have just written and keyed in with great care simply doesn't do what you expect. This unit describes some of the ways you can get over this difficulty. Read it now, and do the exercises; but remember that you can always refer back to it again when (not *if*) the need arises.

When people come to their first programming difficulty, they react in different ways. Some feel angry and insulted; some immediately give up in despair, and decide that programming is not for them; and some pretend that the program is "ninety nine percent right" and go on to the next problem! None of these reactions makes good sense. The only thing to do is to find the mistake and put it right. It can be a great comfort to remember that every programmer sometimes gets stuck, even those who have been working with computers for 25 years.

Program errors fall into three groups. The first and most common type is the one which comes up with SYNTAX ERROR when the computer tries to obey a particular command. This means that the command doesn't follow the rules of the BASIC language. For instance, it might have a spelling mistake in a keyword, or there may be too many (or too few) double quote signs. Most syntax errors are caused by typing mistakes and are obvious once you know they are there; but Appendix C gives a checklist of the kind of error to look for if you are in difficulty.

The second type of program error arises when the computer finds a particular command impossible to obey. Suppose the machine came to the command

```
130 GOTO 500
```

but there was no command labelled 500. This would make the machine stop and display an error message:

```
UNDEFINED STATEMENT IN 130
```

Unfortunately the error messages tend to be in programmer's jargon rather than plain English, but they are fully explained in Appendix C.

When you get an error message, a useful command is HELP. This command makes the computer display the program line which caused the error, with the faulty section is reversed characters.

To see the effect for yourself, type and run the program

```
10 PRINT 1; 2: 3; 4
```

(where the : is a deliberate error).

The program will fail after displaying 1 and 2. When you give the HELP command you will see

```
10 PRINT 1; 2: 3; 4
```

(with the 3; 4 flashing) which shows that the error was somewhere near the 3.

The HELP facility isn't always so clear (try

```
PRINT 1+2+3★★4)
```

but it is still worth using.

You can call the HELP command by hitting the HELP/F7 key, as well as by typing the letters H-E-L-P RETURN.

The third sort of program fault is the most difficult of all to find and put right. There are no error messages; instead, the computer simply displays the wrong answer to your problem or bogs down in a loop without displaying anything at all. The first and most obvious thing to do is to stop the machine, LIST the program and examine it carefully. This will usually help you pin-point the error. However, suppose it doesn't; let's imagine that you have spent a good few minutes examining each command, and you still can't find anything wrong.

At this stage you need a more powerful method of investigating the workings of your program. The method is called 'program tracing' and consists of pretending that you are the computer. You start at the beginning of the program and work through it, command by command, until you get a sudden insight into the cause of the trouble. You will need to be patient and methodical, and above all you'll need to *switch off* your intelligence, and work through the set of instructions like a stupid robot, without trying to make "plausible guesses", generalisations, or use any other type of short cut.

To imitate the computer, you must first have a good idea of how it works. Suppose you could somehow "freeze" the computer between two commands in the middle of running a program, open it up and look inside. You would discover*:

First, the program itself, stored in the memory in much the same form as it was originally typed.

Second, the variables the program has used up to this point. Each variable occupies some room in the memory, and has a *value*, which could be a number or a string.

Third, you find that the computer has kept track of its place in the program. Somewhere (actually, in a special variable called the "program counter") it remembers the label number of the next command it is due to execute.

Now let's unfreeze the computer just a little, long enough for it to execute one command. The command the machine chooses will naturally be the one remembered by the program pointer. When you look again, there will be certain changes, and they depend on the command which has just been obeyed. Here are some of the possibilities:

(a) a PRINT command will make something appear on the screen.

(b) a LET command will create a new variable if one is needed, and put a new value into it.

Both the PRINT and the LET commands will also move the program pointer on to the next command in sequence, so that when the computer is restarted it 'knows' which command to obey next.

(c) a GOTO will not display anything or alter any variables. It will simply reset the program pointer so that it indicates the command mentioned in the GOTO. For example, the command

```
13Ø GOTO 27Ø
```

will put 27Ø in the program counter.

(d) the IF command works in the same way, except that the condition is worked out first. If it is *true*, the program counter is set, just like in a GOTO. If it is *false*, the program counter is simply moved on to the label of the next command in sequence.

Look at:
```
12Ø IF X = 5 THEN 17Ø
13Ø PRINT "NO"
```

If X does have the value 5, the condition is true, and the program counter is changed to 17Ø. Otherwise, if X has some other value, the program counter is simply advanced to 13Ø.

(e) the STOP command indicates that the program has ended, by displaying a BREAK message. There is no point in continuing the program beyond this point.

To imitate the computer accurately, you'll need to see all these parts clearly: the program, the variables, the display and the program counter. A good method is to use a "program trace chart" which you draw on a piece of paper. Arrange it like this:

PROGRAM COUNTER 1Ø

VARIABLES

DISPLAY	*PROGRAM*
	1Ø A=5
	2Ø PRINT "ALPHA="; A
	3Ø A=A★3
	4Ø B=A+37
	5Ø PRINT "BETA="; B
	6Ø STOP

The program you plan to trace is filled in on the right, and the starting value of the program counter — that is, the label number of the first command to be obeyed — is at the top. Make sure that the program is an *exact* copy of the one which is giving you trouble: if it isn't, your trace will be a waste of time.

Now you are ready to start. The program counter says '1Ø', so take and interpret the command labelled '1Ø'. It says A=5, so it must be a LET command. Look in the box marked VARIABLES for an A. There isn't one, so write down an "A", a colon and the value 5. Finally, move the program counter on to the next command in sequence, putting a line through the previous value:

*If you took the cover off the computer you wouldn't actually see these things, but only a few silicon chips and other components. However, the appropriate electronic instruments would certainly show you the items we mention.

PROGRAM COUNTER ~~1Ø~~ 2Ø

VARIABLES A: 5

DISPLAY	PROGRAM
	1Ø A=5
	2Ø PRINT "ALPHA="; A
	3Ø A=A★3
	4Ø B=B+37
	5Ø PRINT "BETA="; B
	6Ø STOP

The next command is interpreted in the same way. You forget the 'purpose' of the program, or any knowledge you may have about sequencing, and take command 2Ø *only* because the program counter says so. The command is a PRINT, and you can work out that it will display "ALPHA = 5". Put this down in the DISPLAY section, and advance the program counter, giving:

PROGRAM COUNTER ~~1Ø~~ ~~2Ø~~ 3Ø

VARIABLES A: 5

DISPLAY	PROGRAM
ALPHA = 5	1Ø A=5
	2Ø PRINT "ALPHA="; A
	3Ø A=A★3
	4Ø B=A+37
	5Ø PRINT "BETA="; B
	6Ø STOP

The next command gives a new value to an existing variable A. You first work out the expression A★3 using the old value (5) and record it, crossing the old value out, like this:

A: ~~5~~ 15

The command after that creates a new variable. Continue tracing until you reach STOP. The final result is:

PROGRAM COUNTER ~~1Ø~~ ~~2Ø~~ ~~3Ø~~ ~~4Ø~~ ~~5Ø~~ 6Ø

VARIABLES A: ~~5~~ 15 B: 52

DISPLAY	PROGRAM
ALPHA = 5	1Ø A=5
BETA = 52	2Ø PRINT "ALPHA="; A
BREAK IN 6Ø	3Ø A=A★3
READY.	4Ø B=A+37
	5Ø PRINT "BETA="; B
	6Ø STOP

The next example involves a simple loop:

```
1Ø P=1

2Ø PRINT P; P★P★P

3Ø P=P+1

4Ø IF P<4 THEN 2Ø

5Ø STOP
```

The trace of this program as far as line 3Ø is straightforward:

PROGRAM COUNTER ~~1Ø~~ ~~2Ø~~ ~~3Ø~~ 4Ø

VARIABLES P: ~~1~~ 2

DISPLAY	PROGRAM
1 1	1Ø P=1
	2Ø PRINT P; P★P★P
	3Ø P=P+1
	4Ø IF P<4 THEN 2Ø
	5Ø STOP

The next command at 4Ø is an IF. To imitate the computer, evaluate the condition P<4. Since the current value of P is 2 (that's what it says in the VARIABLES section), and 2 is clearly less than 4, the condition is true. All you do, therefore is to put 2Ø as the new value of the program counter. You get

PROGRAM COUNTER ~~10~~ ~~20~~ ~~30~~ ~~40~~ 20

VARIABLES P: ~~1~~ 2

DISPLAY	PROGRAM
1 1	10 P=1
	20 PRINT P; P★P★P
	30 P=P+1
	40 IF P<4 THEN 20
	50 STOP

The trace continues this way, until at last the condition is false, and the program reaches STOP. The final result is:

PROGRAM COUNTER ~~10~~ ~~20~~ ~~30~~ ~~40~~ ~~20~~ ~~30~~ ~~40~~ ~~20~~ ~~30~~ ~~40~~ 50

VARIABLES P: ~~1~~ ~~2~~ ~~3~~ 4

DISPLAY	PROGRAM
1 1	10 P=1
2 8	20 PRINT P; P★P★P
3 27	30 P=P+1
BREAK IN 50	40 IF P<4 THEN 20
READY.	50 STOP

Enter this program into the computer and run it. Check that the results tally.

Now enter the program to print ALPHA and BETA again into your computer and run it. You should get the display

```
ALPHA =5
BETA =52
BREAK IN 60
READY
```

The computer has a remarkable facility of doing its own tracing. To try it out, type the command

TRON RETURN

(which stands for "TRace ON") and RUN your program again. The values of the program counter now appear in square brackets, like this:

```
[10] [20] ALPHA =5
[30] [40] [50] BETA =52
[60]
BREAK IN 60
```

This tells you that the program obeyed lines 10, 20, 30, 40, 50 and 60, in that order. The output

ALPHA =5

was produced by line 20, just as you expect.

To turn the trace off, type

TROFF RETURN

From now on, you can use TRON and TROFF to help with your tracing. You will soon learn to use these commands with care, because a program with loops generally obeys a very large number of commands, and the lists of line numbers can easily swamp your screen.

EXPERIMENT 8·1

Now practice your tracing with the following programs. Use a pencil, and have a rubber handy in case you make a mistake:

(a)

PROGRAM COUNTER 10

VARIABLES

DISPLAY	PROGRAM
	10 X=5 20 Y=7 30 Z=X+Y 40 W=Y−X 50 PRINT X;Y;Z;W 60 STOP

(b)

PROGRAM COUNTER 10

VARIABLES

DISPLAY	PROGRAM
	10 Q=1 20 PRINT "SHE LOVES ME" 30 PRINT "SHE LOVES ME NOT" 40 Q=Q+1 50 IF Q<3 THEN 30 60 STOP

When you have completed these two experiments check your answers by running the programs on your computer with the Trace switched on.

Experiment 8.1 Completed ☐

EXPERIMENT 8·2

How can tracing be used to find mistakes? It depends on switching between a state of robot obedience, and a state of human intelligence. First you become a robot and trace a command exactly as the computer would have executed it. Then you go back to being a person, and ask, "Is this what I expected?" If so, you carry on the trace. If not, you will have a good clue as to why the program is going wrong.

Here is a simple example. Suppose you've written a program to display the 12 times table. The display you expect is

```
TWELVE TIMES TABLE
 1 ★ 12 = 12
 2 ★ 12 = 24
 3 ★ 12 = 36
(and so on down to)
12 ★ 12 = 144
```

Your program has all the right parts: a loop, a command to display a heading, and a PRINT command to display each line of the table. It reads:

```
10 PRINT "TWELVE TIMES TABLE"
20 P=1
30 P=P+1
40 IF P< 13 THEN 30
50 PRINT P; "★12="; P★12
60 STOP
```

When you run this program, the results are a bit disappointing. All you get is

```
TWELVE TIMES TABLE
13 ★ 12 = 156
BREAK IN 60
READY
```

Not what you expected! The mistake may be perfectly obvious, but let's pretend you can't spot it. You begin to trace, and after a few steps you get

PROGRAM COUNTER ~~10 20 30 40 30 40~~ 30

VARIABLES P: ~~1 2 3~~ 4

DISPLAY	*PROGRAM*
TWELVE TIMES TABLE	10 PRINT "TWELVE TIMES TABLE" 20 P=1 30 P=P+1 40 IF P< 13 THEN 30 50 PRINT P; "★12="; P★12 60 STOP

and you suddenly realise that the value of P is working its way up to 12 without anything being displayed. It is now clear that the PRINT command ought to be inside the loop, not outside. The right place is between commands 2Ø and 3Ø. The IF command also needs to be changed to jump back to the PRINT. A quick edit produces

```
1Ø PRINT "TWELVE TIMES TABLE"

2Ø P=1

25 PRINT P; "★12="; P★12

3Ø P=P+1

4Ø IF P < 13 THEN 25

6Ø STOP
```

Program tracing is an extremely useful technique if you have the patience to do it step by step. If you make guesses about whole sections of program, you are likely to make the same mistake as you did when you wrote the program in the first place, and the trace won't reveal your error.

To save time you will often use the CBM's own trace mechanism to tell you the sequence of commands obeyed. You will still need to analyse the list of line numbers very carefully — TRON is not a substitute for thought!

Sometimes you will be sure that most of a program works correctly, and to avoid a flood of irrelevant line numbers you will only want the trace facility to be switched on for a small section of the program. This can be done by putting numbered TRON and TROFF commands into the program itself like this:

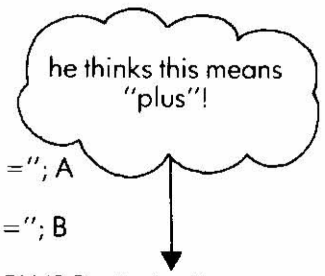
```
10 ——
20 ——
 .  .  .
105 TRON     )
110 ——       |
120 ——       }  Part of program traced
 .  .        |
185 TROFF    )
190 ——
200 ——
```

When you have debugged the suspicious part of the program, you can easily take the TRON and TROFF commands out again.

There are a few circumstances under which the tracing method as described doesn't work, and you should know what they are:

- If a program is very large, a straightforward trace would just take too long. More appropriate methods will be described later on in the course, at the time you may actually need them.
- If you simply don't believe that you *can* make a mistake, then tracing won't be much help. Most people, when they write down the last line of a program, experience a strong moral certainty that "This time it's right". The feeling only comes because you haven't been *conscious* of making any mistakes, and is extremely misleading. It is much better to say to yourself "This time it's wrong. Let's find the mistakes". But you'll need to swallow your pride!
- If you have misunderstood some fundamental aspect of BASIC, a trace will again be of little help. To take a crude example, imagine someone who believes, firmly but mistakenly, that in BASIC the sign "—" means "addition". He writes a program to add two numbers like this:

```
1Ø A=34

2Ø B=19

3Ø PRINT "A ="; A

4Ø PRINT "B ="; B

5Ø PRINT "A PLUS B ="; A—B

6Ø STOP
```



When he runs this program, it displays

```
A = 34

B = 19

A PLUS B = 15

BREAK IN 6Ø

READY
```

which is clearly wrong. On the other hand, when he traces it, he finds that command 5Ø gives

```
A PLUS B = 53
```

which is what he expects. As long as he really believes that "—" means "add", he will never find the error!

Of course most misunderstandings are much more subtle than this one. Nevertheless, if your trace comes out differently from the result displayed by the computer, and keeps coming out differently when you repeat the trace, this is clear evidence that there is something about the art of programming you haven't understood correctly. If you can, get advice from someone who knows BASIC better than you do*; but otherwise go back to the beginning of the text-book, and check every single item of your knowledge against what it says. This will nearly always bring the fault to light.

**There are now plenty of people who understand BASIC. If you don't know anyone personally, an advertisement in a local shop window will usually find help.*

Sometimes—very very rarely—your difficulty may be caused by a mechanical fault in the computer. Modern machines like the computer are extremely robust and reliable, and when they do break down, it is usually obvious: the cursor won't come up when you switch on, or you find it impossible to load programs. In practice you should never blame the computer for not running your program until you have examined every other possibility two or three times over. When you send your machine to be repaired, you must explain exactly *why* you think it is broken, and include a copy of the program which it refuses to run correctly.

Here are two programs with mistakes for you to find and correct.

(a) This program is supposed to display a conversion table for gallons to litres, starting at 1 gallon and ending at 10 gallons (1 gallon = 4.5 litres)

```
10 PRINT "GALLONS", "LITRES"

20 G=1

30 PRINT G, 4.5★G

40 G=G+1

50 IF G > 11 THEN 30

60 STOP
```

(b) This program is supposed to be a solution to problem 1 in Unit 7, to display a triangle of stars. It was actually written by someone learning BASIC:

```
10 A$ = "★"

20 PRINT A$

30 A$ = "★★"

40 IF A$<>"★★★★★★★★★★" THEN 20

50 STOP
```

Experiment 8.2 Completed	

EXPERIMENT 8·3

The program UNIT8PROG is supposed to display the 7-times table, but contains several errors. Load it, find and correct the mistakes. Check your answers in Appendix B.

Experiment 8.3 Completed	

UNIT: 9

Let's draw some more pictures. This time, we'll make the computer do all the hard work and drudgery for us.

If you think back to units 2 and 3 (look to remind yourself if you like) you'll remember that when you draw on the screen you can use a number of control 'functions':

- Cursor movement in four different directions
- Selection of sixteen different colours
- Colour and background reversal (on and off)
- Flashing characters
- Moving the cursor 'home' to the top of the left-hand corner
- Clearing the screen.

These functions share keys on the keyboard, so that you often have to use SHIFT or CTRL or C= to choose the function you really need.

You won't have forgotten that you can set the frame and background colours using 'COLOR' instructions and code numbers from the table on page 2Ø.

The computer can also make drawings on the screen under the control of a program. Every program has the use of all the screen control functions: it can select any colour for its characters, it can clear the screen whenever it needs to, and it can move its own cursor (which is invisible to you) to any position using the cursor control functions.

Of course the machine only does these things when obeying the commands you have given it. To put screen control functions into a command is easy: we simply include them in strings alongside the other characters to be displayed. You might find this a bit puzzling at first. Surely, if you type a string and include a screen-clear function in it, the whole screen will disappear as you type? In fact this does not happen, as the next experiment is designed to show.

EXPERIMENT 9·1

Do you remember that in Unit 2 we said, "Don't use the double quotes, they're *funny!*" Now you are going to find out what effect they really have, and why they're so useful.

When you start typing a command (say after a READY or a RETURN) the computer is in 'normal' mode. Control functions like colour selection or cursor movement work in the way you have come to expect. As soon as you type a double quote character to mark the beginning of a string, the machine changes to quote mode. Ordinary characters such as letters or graphics are still treated in the normal way, but control functions are *not* obeyed: instead they are put into the string as 'special' characters, mostly letters, signs or graphics on a reversed background. The machine switches back to normal mode when you type a second double quotes character (so ending the string) or if you give a RETURN .

Start up the computer, type a double quote and then give all the control functions, one by one.

See how each one looks on the screen, and fill in the table on the next page.

Function	*Key Struck*	*Symbol displayed*
Clear Screen	SHIFT and CLR HOME	▨
Cursor home	CLR HOME	S
Cursor up	↑	▨
Cursor down	↓	Q
Cursor left	←	▨
Cursor right	→	▨
Black	CTRL and 1 BLK	▨
White	CTRL and 2 WHT	▨
Red	CTRL and 3 RED	▨
Cyan	CTRL and 4 CYN	▨
Purple	CTRL and 5 PUR	▨
Green	CTRL and 6 GRN	▨
Blue	CTRL and 7 BLU	▨
Yellow	CTRL and 8 YEL	▨
Orange	C= and 1 ORNG	▨
Brown	C= and 2 BRN	▨
Yellow green	C= and 3 YL GRN	▨
Pink	C= and 4 PINK	▨
Blue green	C= and 5 BL GRN	▨
Blue	C= and 6 L BLU	▨
Dark blue	C= and 7 D BLU	▨
Light green	C= and 8 L GRN	▨
Reverse on	CTRL and RVS ON	R
Reverse off	CTRL and RVS OFF	▨
Flash on	CTRL and Flash on	▨
Flash off	CTRL and Flash off	▨

Now let's try some of these controls in action. First make sure your TV set is properly adjusted for colour, by using the TESTCARD program if need be. Make sure you have a white background by typing the command

COLOR 0, 2 RETURN

Next get the computer to display the word "EDINBURGH" in blue. Type in the command

PRINT " CTRL and 7 BLU EDINBURGH"

hold down

while you type

What actually appears on the screen (all still in black) is

PRINT " ← EDINBURGH". The reversed ←

reversed

symbol is the code for "blue".

Now hit the RETURN key. The word EDINBURGH appears on the screen, in blue.

This experiment illustrates the principle quite clearly: when a control function is typed inside a string, it is not put into effect when it is typed, but only obeyed when that string is displayed by the computer.

You will see that the flashing cursor has been left blue. Change it back to black by typing the correct control function — without quotes.

A PRINT command which gives you a colour change can be made part of a program, just like any other command. Key in and run the following:

```
10 PRINT " CTRL and 6 GRN GLASGOW"
20 PRINT " CTRL and 3 RED INVERNESS"
30 PRINT " CTRL and 5 PUR ST. CTRL
     and 8 YEL ANDREWS"
 40 STOP
```

Command 30 shows that you can put more than one control function into a string; also that yellow letters are hard to read on a white background.

Screen and cursor control functions can also be put into strings. Type the following:

PRINT " SHIFT and CLR HOME ↓ ↓ ↓ → → → CTRL and 3 RED PARIS"

On the screen this comes up as

PRINT " ♥QQQ]]]£ PARIS"

reversed symbols

When you strike RETURN, the control functions are actually obeyed. The screen is cleared, the cursor is moved three places down and three along, and the word PARIS appears in red half-way towards the middle of the screen. Try it for yourself.

In general, you can get the computer to paint words and symbols anywhere you like by including the right number of cursor shifts in a string.

When you get the computer to draw a picture on the screen, you don't want to spoil everything by displaying READY and the flashing cursor. A way out of this difficulty is to use a 'loop stop', or a GOTO which jumps to itself. Once the computer reaches this command, it will start chasing its own tail, and it won't display READY until someone hits the RUN STOP key. This program, for example, will display LONDON in white in the centre of a black screen:

```
                           12 times
10 COLOR 0, 1              ⏞
20 PRINT " SHIFT and CLR HOME ↓ . . . ↓
→ . . . . → CTRL and 2 WHT LONDON"
⏟
17 times

30 GOTO 30
```

Key this program in, run it, and then stop it with the RUN STOP key. The screen will still be black and the cursor white, but you can quickly get back to the normal state of affairs by holding down RUN STOP and pressing the RESET button on the side of the computer. When you see the word MONITOR, type X and RETURN. You'll remember you can always do this if the machine gets stuck for any reason; it is better than switching on and off because your program isn't lost when you do it.

As a short exercise, get the computer to display words and patterns of different colours at various positions on the screen. Remember that the SHIFT and CLR HOME function clears the screen, so if your program has a sequence of PRINT statements, only the first one should begin with this function — although some of the others could well start with CLR HOME by itself.

Experiment 9.1 completed

EXPERIMENT 9·2

You will know that modern clocks and watches are controlled by quartz crystals, and are extremely accurate over long periods of time. The computer also incorporates a quartz crystal vibrating several million times every second, and it is used — among other purposes — to control an internal digital clock. This clock doesn't have its own dial; instead, it is treated just like a string variable, so that you can display the time on the screen whenever you need. The name of the clock variable is TI$.

When you first start up any clock, you have to set it to the right time. The 16 and Plus/4 are no exception. You can adjust the clock from the keyboard, by typing a command like

```
TI$ = "193746" RETURN
```

This would set the clock to 7.37 and 46 seconds in the evening.

If you want to set the clock very accurately, it is best to wait for — say — the nine o'clock news on the radio. Just before it starts type

```
TI$ = "090000"
```

and then hit RETURN as you hear the last 'pip' of the time signal.

Once the machine's clock has been adjusted, it will keep time, to within a few seconds a day, until the machine is switched off or completely reset. There is no need for you to reset it or change the time from within a program.

To display the time, you simply mention TI$ in a PRINT command.

Now set up the computer's clock, using your own watch (it doesn't matter if the setting isn't very accurate). Then display the value of TI$ several times, using a PRINT command. See how the seconds change from one time to the next.

Now display the time continuously, by running the program.

```
10 PRINT TI$
20 GOTO 10
```

Stop this program, wait a few minutes, and restart it. You will see that the time is still correct, and that the clock has been running all the time.

This method of displaying the time is not attractive. You can make the computer into a respectable digital clock by a program as follows:

command 10:	Selects a purple frame
command 20:	Selects a yellow background
command 30:	Clears the screen
command 40:	Moves the machine's cursor home, then down 9 lines and across 6 spaces; no new line needed
command 50:	Displays TI$ in blue
command 60:	Jumps back to command 40.

Write down the code for this program in the box below; then enter it on the computer keyboard and try it out. If you get *really* stuck, look up the correct version in Appendix B, but don't go on until you have studied it carefully and found out how it works.

Experiment 9.2 Completed ☐

EXPERIMENT 9·3

Controlled loops are often useful in drawing shapes on the screen. Suppose you want a 10 x 10 block of red dots in the top left-hand corner. This can be done by displaying ten lines, each with ten ● graphics:

```
10 PRINT " SHIFT and CLR HOME ";

20 J=1

30 PRINT " CTRL and 3 RED ●●●●●
   ●●●●●"

40 J=J+1

50 IF J<11 THEN 30

60 GOTO 60
```

This program combines several of the ideas we have already met in previous units. The semicolon at the end of command 10 prevents the machine from starting a new line after clearing the screen, so that the first line of red dots appears at the top. Statements 20 to 50 form a controlled loop and 60 is a loop stop.

Enter the program and run it as it stands. Then stop it, and try for yourself the effects of

(a) removing the semicolon after " CLR HOME "

(b) changing the 11 in command 50 to some other value (say 15)

(c) removing command 60

You can of course make these changes by LIST ing and editing. Remember to get the cursor colour back to blue or black before you start!

To get a solid block of colour we use reversed spaces. Try changing line 30 to

PRINT " CTRL and 3 RED CTRL and RVS ON ← 10 spaces →"

and run the program again.

What happens if we want more than one block of colour in the same picture? The trick is to move the machine's cursor to the first line of the area, and then to fill it in, without interfering with the colour already on the screen. We'll look at two examples:

(a) To paint a blue 1Ø x 1Ø block just below the red one:

The lower half of the screen is empty, so we don't need to worry about spoiling anything else. Furthermore, after drawing the red block, the cursor will be in the right place. We can extend the program by adding

```
6Ø J=1

7Ø PRINT " CTRL and 7 BLU CTRL and
    RVS ON ← 10 spaces →"

8Ø J=J+1

9Ø IF J < 11 THEN 7Ø

1ØØ GOTO 1ØØ
```

Notice that the loop stop has been moved to the end of the program where it belongs. J is used as control variable in both the red and blue loops: this is perfectly alright because the red block is completely finished before the blue one is started, and J *isn't* asked to do two jobs at the same time.

(b) To paint a 1Ø x 1Ø black block *beside* the red one.

The starting line is the top one, so in drawing the black area we have to be careful not to damage the red block which is already there. This can be done by moving the cursor home, and displaying 1Ø lines, each of which begins with 1Ø "cursor right" movements to jump over the red. The program extension is

```
1ØØ PRINT " CLR HOME ";
                          10 times
11Ø J=1                $\overbrace{\qquad\qquad\qquad}$

12Ø PRINT " → .......... →
CTRL and 1 BLK CTRL and

RVS ON ← 10 spaces →"

13Ø J=J+1

14Ø IF J < 11 THEN 12Ø

15Ø GOTO 15Ø
```

Now assemble this program, type it in and try it out. Note that it has three separate loops which are executed one after the other.

Try extending the program to put a purple block under the black one. . .

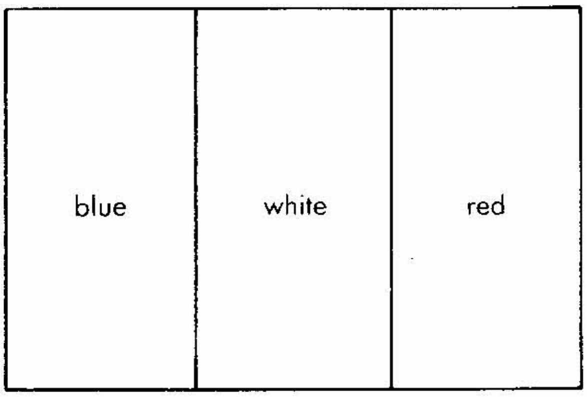
As a final exercise, try writing programs to display some simple flags, or other patterns which fill the whole screen. You will need your wits about you, because various pitfalls lie in wait.

- The normal meaning of a semicolon at the end of a PRINT command is "Don't start a new line". If the computer is made to put a character into the right-most position of a line, it *automatically* moves its cursor to a new line. Displays which are meant to fill complete lines should therefore be followed by semicolons unless you actually *want* a blank line to follow.

- There is no way of using a PRINT command to write a character into the lower right-hand corner of the screen without making the whole screen move up.
The way to get this square the right colour is to select the entire background colour accordingly.

You should plan your painting carefully, using squared paper as a guide. When you come to write your programs, be prepared to make plenty of mistakes, and don't be upset if it takes several tries to get things right. Remember that you learn by success — not by failure — so *don't just give up!*

To start you off, we'll give you a program for the French flag.



We'll make the central white stripe 14 characters wide, and the other two 13 each. 13 + 14 + 13 = 40

Appropriate starting colours are a red background and a black frame.

We can build up the flag by displaying 25 lines, each with thirteen white squares and fourteen blue ones. Remember that the last one must be different, because it mustn't be followed by a new line. We can put the first 24 lines into a controlled loop, but the last will need a command on its own.

We arrive at

```
1Ø COLOR 4, 1

2Ø COLOR Ø, 3

3Ø PRINT " SHIFT and CLR HOME ";

4Ø J=1

5Ø PRINT " CTRL and RVS ON CTRL and
  7 BLU ← 13 spaces → CTRL and
  2 WHT ← 14 spaces →"

6Ø J=J+1

7Ø IF J<25 THEN 5Ø

8Ø PRINT " CTRL and RVS ON CTRL and
  7 BLU ← 13 spaces → CTRL and
  2 WHT ← 14 spaces →"

9Ø GOTO 9Ø
```

Run this program and study it carefully until you understand every symbol. Now try some of your own flags, but keep off from ones with diagonal elements! Try the Iceland flag which is shown on page 23. You can check your answer with the one shown in Appendix B.

Experiment 9.3 Completed	

The self-test program for this unit is called UNIT9QUIZ.

UNIT:10

In the previous units we came across the idea that commands can be written once, but obeyed many times over. This happens whenever you put a command in a loop.

On a much larger scale, a similar thing occurs with complete programs. Most programs are designed to be useful, which means that they are stored and distributed on tapes, disks or ROM-packs and used many times by different people. If you want an example, look at the various programs which form part of this course.

Let's begin by considering all the programs which you personally have written so far. The drawback with every one of them is that no matter how many times you run it, it always produces the *same* result. Hardly very useful!

To give a specific example, let's go back to the program which calculates and displays a conversion table between £UK and Italian Lire. It was:

```
10 PRINT "£", "LIRE"

20 PRINT

30 PS=5

40 PRINT PS, 2350★PS

50 PS=PS+5

60 IF PS<80 THEN 40

70 STOP
```

On the day the Unit was written, the rate of exchange really was 2350 Lire to the Pound, so the program would have given correct results. By today, however, the rate has fallen to 2175. Any bank which used this original program to sell Lire in exchange for Pounds would be seriously out of pocket.

How could matters be improved? If you are a programmer one obvious approach would be to alter line 40 to read

```
40 PRINT PS, 2175★PS
```

Unfortunately this idea won't take you very far. Most people who use computers aren't programmers, or even if they are, they are just not interested in the guts of your program!

To make programs more flexible, more adaptable to everyday needs, we need a new facility: one which lets the *user* supply information which the programmer couldn't have known when the program was written. A program which allows this can be used by lots of different people, and lets each one solve their own particular version of a problem. For instance, suppose that the money conversion program allowed the user to tell it the current rate of exchange every time it was used; it would immediately become useful to banks all over the world, and it would work properly for any imaginable exchange rate.

Suppose you are designing a program for someone else to use. You begin by deciding which quantities you are going to leave undefined, and your program is going to ask the *user* to supply. In our example the rate of exchange is clearly one such quantity: it must be unknown to the programmer, but known to the user! You allocate the unknown quantities to variables, and give them *names* accordingly. For instance, a suitable name for the rate of exchange could be RE. You can then write your program using symbolic names instead of the actual values (which you cannot know in advance). Thus line 40 of the exchange program could read

```
40 PRINT PS, RE★PS
```

Of course there is something missing from this description. *You* may not know the values of the variables, but the *machine* must do so when it runs your program. The command which lets the *user* put in the missing information has the keyword INPUT. This is followed by the name (or names) of the variables needed. When the INPUT command is obeyed it waits for the *user* to type a value, which it then stores in the named variable. The rest of the program, which uses this variable, can now be obeyed.

Before giving an example, we stress one vital point: every program with an INPUT command must tell the user exactly what is wanted of him. This can usually be done with PRINT statements.

EXPERIMENT 10·1

Study the following program carefully:

```
  3 PRINT "TYPE TODAY'S"
  4 PRINT "RATE OF EXCHANGE"
  5 PRINT "BETWEEN £ AND LIRE"
  6 INPUT RE
 10 PRINT "£","LIRE"
 20 PRINT
 30 PS=5
 40 PRINT PS, RE★PS
 50 PS=PS+5
 60 IF PS<80 THEN 40
 70 STOP
```

Notice how the program doesn't assume any particular rate of exchange, but uses the variable RE to represent it wherever it is needed. The program begins by telling the user what is needed and asking him to supply a value.

Enter the program, check it carefully, and type RUN. Now *pretend* you are a user: a money-changer who knows nothing about programming. On the screen the machine is asking you to type something, so you enter the appropriate figure, and then strike the RETURN key.

As soon as you do this, the screen fills with a conversion table that lets you start business today.

Run the program many times, and notice how well it can handle different rates of exchange. Even if the Lira were to be revalued to a level of 23.7 to the £, the program would still produce sensible answers.

Now switch back to your personality as programmer. When the program was running, showing a cursor and waiting for the user to type his information, it was actually *obeying* the INPUT command.

The INPUT command comes in several slightly different forms. We'll look at some examples, and mention a few general rules.

1. Clear the computer by typing NEW and type in

```
10 PRINT "WHAT'S YOUR NAME"
20 INPUT N$
30 PRINT "HELLO SPACE "; N$;"!"
```

Run this program and see what happens. The example shows how the INPUT command works with strings as well as numbers. You could use this sequence — or something like it — near the beginning of any program where you wanted the computer to be 'friendly' to the user. If the program was a quiz of some kind, you could use the value of N$ in commands like

```
40 PRINT "NO SPACE "; N$;". YOU
   CAN DO BETTER THAN THAT"
```

(If you are in any doubt about what this command displays, tack it on to the end of the program already in the computer, and run the program again.)

2. Try

```
10 INPUT "NAME";N$
20 PRINT "GOODBYE SPACE ";N$
```

This example shows how a short piece of descriptive information can be included in the INPUT command itself. The information shows up on the screen as a guide to the user, just before the ?.
Command 10 in the example is equivalent to the *sequence*

```
PRINT "NAME";
INPUT N$
```

Notice that the string of descriptive words *must* be followed by a semicolon.

3. Lastly, try

```
10 PRINT "GIVE TWO NUMBERS TO
   BE ADDED"
20 INPUT A,B
30 PRINT "SUM="; A+B
40 STOP
```

The INPUT command now expects two

values, and the user must type them separated by a comma or by pressing the RETURN key. (That is, he or she could type either —

say 43, 19

or { 43
19

In general, the INPUT command may ask the user for any number of variables, but it is better to keep the number down to two to prevent confusion. In the command itself, the names of the variables are separated by commas.

When you have run this program a few times, pretend you are a really stupid user and try typing nonsense — for example

DONALD,DUCK

The computer will accept anything at all as a string, but if it is trying to input a number, and is given something which couldn't possibly be a number, it will display the message

REDO FROM START

and give you another try.

Sometimes you want to stop a program when it is obeying an INPUT command and displaying a cursor. Under these conditions the RUN STOP key by itself is disabled, and stopping the program is more complicated than usual. Hold down the RUN STOP key and press the RESET button. You will see the cursor come up on a screen which is blank apart from the word MONITOR and a few letters and figures. Now type X RETURN and the READY. message appears, showing that you are back in control.

Be careful not to press RESET by itself, as this will destroy your program.

Experiment 10.1 Completed	

EXPERIMENT 10·2

Writing useful programs is easy provided you remember that the programmer and the user are two *different* people. The user can't be assumed to understand programming (so he cannot be expected to LIST your program to find out what it does). In general the programmer may not 'talk to' the user except by making the computer display messages on its screen, and the user can't get back to the programmer at all, so the program had better not leave any questions unanswered!

When you are designing a program, pretend you are a fly on the wall watching someone trying to use it. Try to imagine everything that could go wrong, and try to prevent it by making sure the program gives the user plenty of guidance.

When your program is written, you can exercise it by pretending you are a user; later, as a final test, bribe a friend or relative to be a 'guinea-pig' and to try the program out for you. If your guinea-pig has to ask you any questions about what to do, or what the answers displayed actually mean, your test has failed and you should redesign your program accordingly.

Write programs to do the following jobs:

(a) To display any multiplication table selected by the user.

(b) To ask the user (who is assumed to be a married man) for his surname, and then for his wife's Christian name; and then to display his wife's full name.

Solutions are given in Appendix B, but don't look at them until you have done everything you can to write these programs by yourself.

Experiment 10.2 Completed	

The quiz for this Unit is called UNIT10QUIZ.

UNIT:11

One of the most interesting features of programming is its richness and variety. The same computer, if properly programmed, can be made to serve as a calculator, a teaching machine, a musical instrument, a monitor to look after a sick patient in hospital, or almost anything else useful you can think of. This power comes from the huge number of ways that a few basic types of command can be put together.

So far, our total vocabulary of commands used *within* programs is only seven:

PRINT, LET, GOTO, IF, INPUT, STOP and COLOR

Of course there are other BASIC commands you still have to learn about; but in this unit we'll explore the potential of the commands we already know.

The most flexible command of all is the IF. In previous units it's been used to control loops, but it is also useful in many other ways. For instance it can test *data* or items of information supplied by the user, so as to steer the computer along the right course of action.

EXPERIMENT 11•1

Let's imagine you are setting up a computerised marriage bureau, and the first facility you plan to provide is a program to advise on the ages of the partners your customers should look for. By tradition a man should marry a girl of *half* his age, plus seven. This implies, if you think about it, that a girl should look for a husband who is *double* her age, less 14.

Clearly, the advising program must begin by asking for the client's age. Then, to give the right advice, it has to find out whether the client is a man or woman. The program will be used both by men and women, so it must include a separate group of commands to give advice to each of the two sexes. Finally there must be an IF command to select the group actually needed on a particular occasion.

A first version of the advising program is given below. Study it carefully and work out exactly why each command is included:

```
10 INPUT "WHAT IS YOUR AGE";AG

20 INPUT "MALE OR FEMALE";SX$

30 IF SX$="MALE" THEN 70

40 PRINT "YOU SHOULD LOOK FOR"

50 PRINT "A MAN OF";2★AG-14

60 STOP

70 PRINT "YOU MUST FIND"

80 PRINT "A GIRL AGED"; AG/2+7

90 STOP
```

You will have spotted that the variable AG is used to hold the client's age, and SX$ his (or her) sex. The condition SX$="MALE" is true if the client answers MALE to the question "MALE OR FEMALE?". The expression AG/2+7 is BASIC's way of saying "half your age plus seven", and 2★AG-14 means "twice your age less fourteen".

When you have looked at the program, test your understanding by *predicting* as accurately as you can what will appear on the screen (a) for a man of 20, and (b) for a girl of 22. Use the boxes below. The first box is partly filled in for you.

```
RUN

WHAT IS YOUR AGE? 20

MALE OR FEMALE?
```

(a)

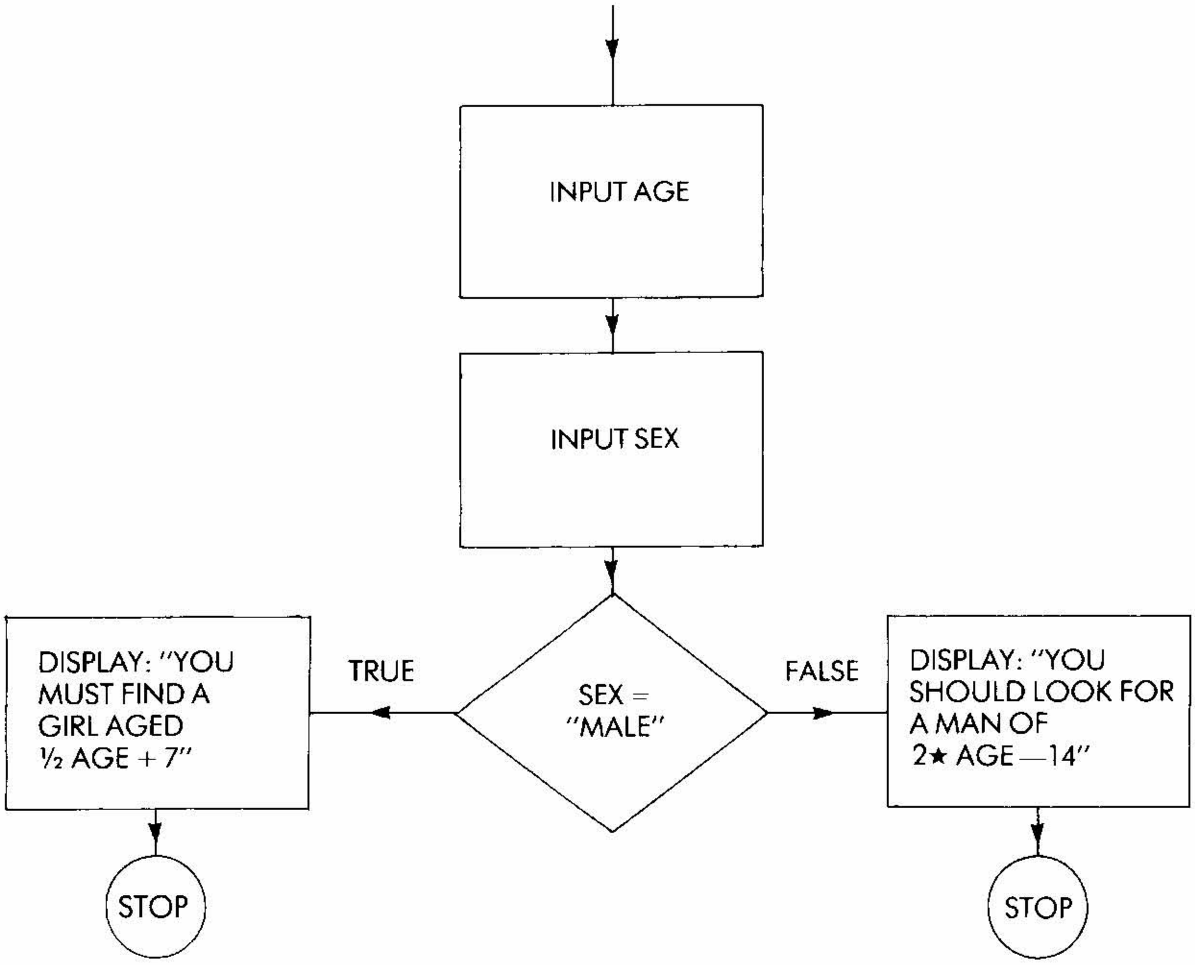
```
RUN
```

(b)

Now enter the program into the computer. Try it out, on behalf of various sorts of client, and check that both your predictions are right.

This simple example shows you that the action of the computer needn't be fixed in advance by the programmer, but can be made to depend on the information supplied by the user.

Programs often have complicated sets of decisions to make, so to plan them we use a special type of diagram called a flow chart. The flow chart for the advising program is like this:


INPUT AGE
INPUT SEX
SEX = "MALE"
TRUE
FALSE
DISPLAY: "YOU MUST FIND A GIRL AGED ½ AGE + 7"
DISPLAY: "YOU SHOULD LOOK FOR A MAN OF 2★ AGE — 14"
STOP
STOP

A flow chart consists of a number of blocks connected by arrowed lines. There are four kinds of blocks:

(a) A square or rectangular *box*. The box holds the description of a simple action, which can later be translated into one or two BASIC commands. In our sample flow chart, the top two blocks are examples of this type. The arrowed lines show that the program starts by obeying the first block, and then goes on to the second one, in that order.

(b) A *diamond* holds a condition, which may be either true or false. The diamond has one line going into it, but two coming out, labelled TRUE and FALSE (or sometimes YES and NO). The diamond corresponds to an IF command. It instructs the computer to test the condition, and to follow either the TRUE or the FALSE line according to the result.

(c) The *terminal* block, which tells the computer to stop obeying the program. It is a small circle with the word STOP.

(d) The *cloud* (which doesn't appear in our example). This is a symbol for an action which is too complicated to be described in detail. Usually, the cloud can be expanded into another complete flow chart, just as a country-wide road map is backed up by detailed plans of different towns.

A flow chart is really a 'map' of a program. A computer running a program is a little like someone playing a board game. At the beginning the player's token (motor-car, top hat or whatever) goes on the first block. Whenever the action described in a block has been completed, the token is moved along the arrowed line to the next block.

When the token lands on a diamond, the player looks at the condition and decides whether it is true. If it is, then he moves his token to the box at the end of the TRUE line, but otherwise, he follows the FALSE line. Eventually he reaches a STOP block, which is the end of the game.

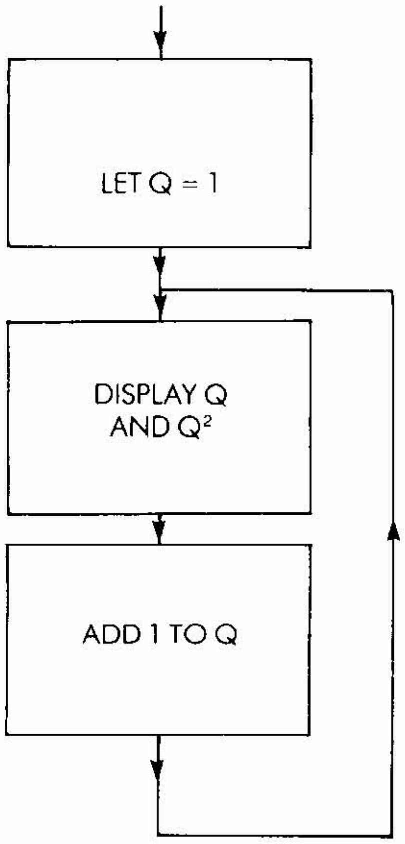
The point of this illustration is to help you see two very important things about computers:

- A computer can do only *one* thing at a time (not several)
- The order in which the computer does things is determined by the program.

It often surprises people that there is no flow chart symbol for a simple GOTO command. This is because the GOTO doesn't specify any action at all; it only affects the order in which commands are obeyed. It is well represented by a connecting line. For instance:

```
10 Q=1

20 PRINT Q; Q*Q

30 Q=Q+1

40 GOTO 20
```

has the flow chart



Now draw a flow chart for the following program. Use the plastic stencil for your blocks:

```
10 S=1

20 PRINT S,12*S

30 S=S+1

40 IF S < 13 THEN 20

50 STOP
```

(Check your answer in Appendix B.)

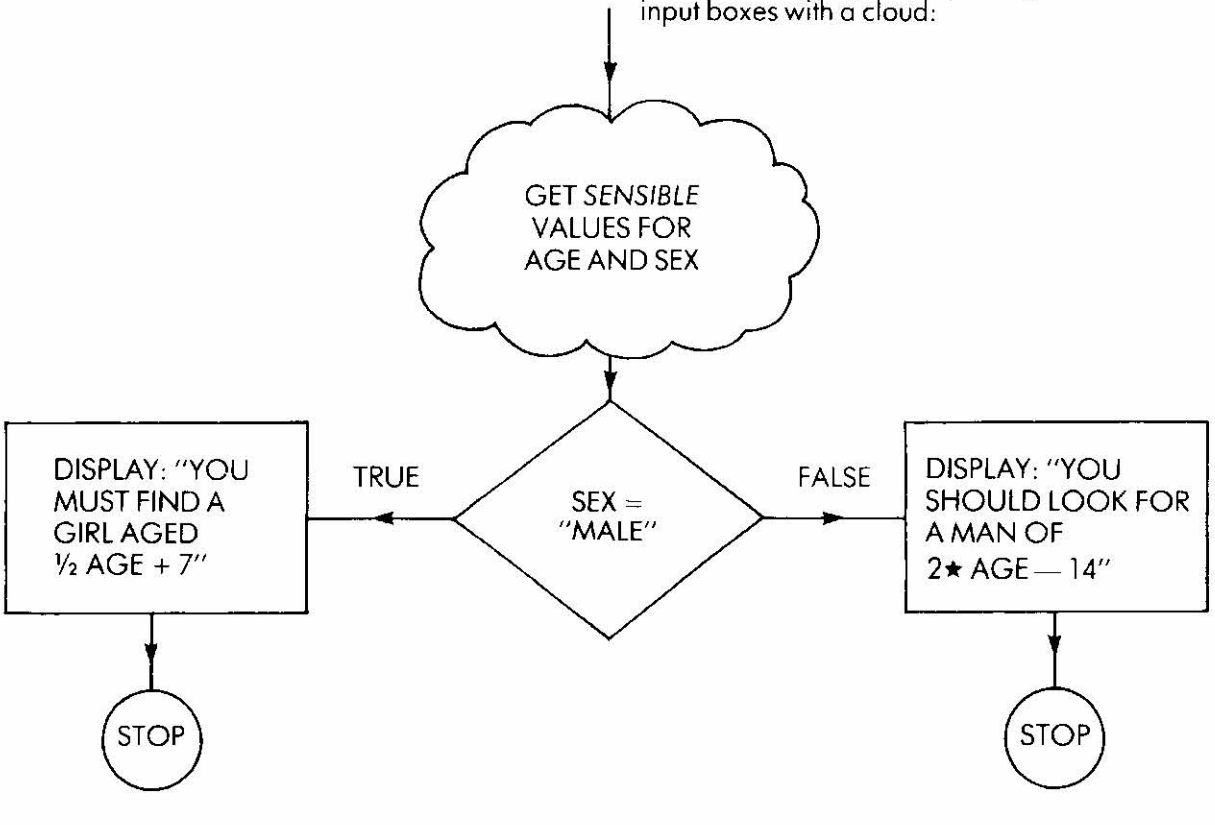
Experiment 11.1 Completed	

EXPERIMENT 11·2

Let's do some more exploring. One feature of our marriage guidance program was that if you give it incorrect data, it gives you silly answers. The name for this fact is "GIGO", which stands for "Garbage In, Garbage Out". For instance, a girl who gave her age as 6 would be told to find a husband aged −2: not even a gleam in his parents' eyes! Furthermore, if the user gives *any* answer other than MALE to the second question, the program assumes she is female. Someone who replies "M" or "MAN" or "MASCULINE" or "BOY" will be told to find a *man* as partner.

There are plenty of programs which do behave in this idiotic way, and they have given computing something of a bad reputation. In practice you can avoid the worst of these troubles by passing the user's information through a filter to make sure that it is at least *sensible*.

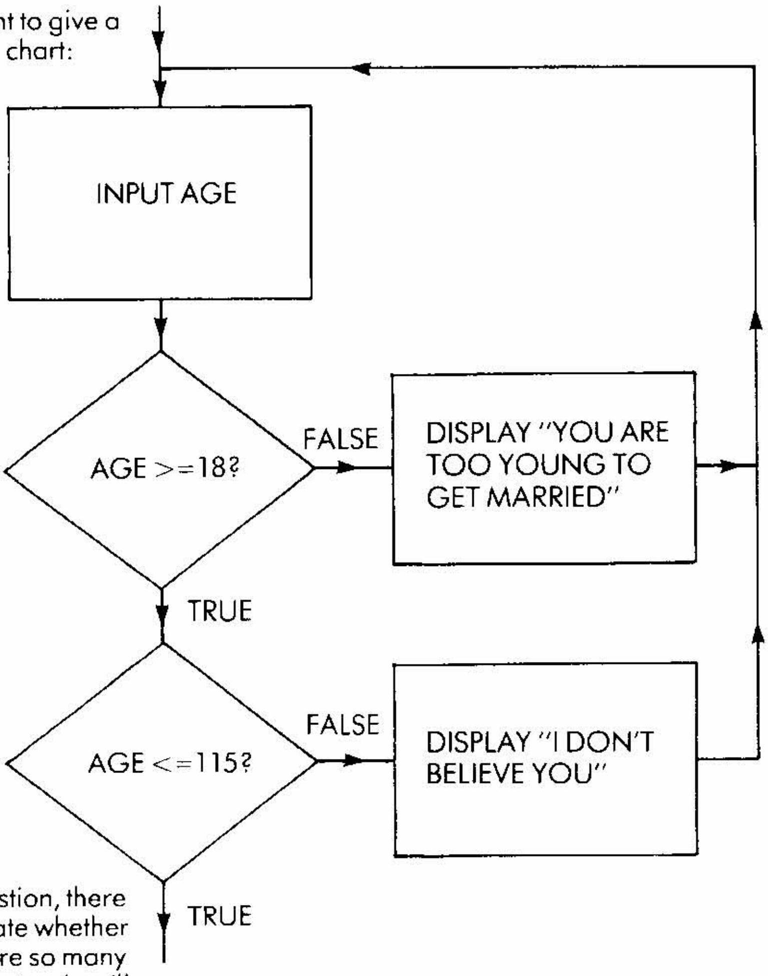
To begin with, we'll draw a new flow chart for the whole program, replacing the detailed input boxes with a cloud:


GET *SENSIBLE* VALUES FOR AGE AND SEX
SEX = "MALE"
TRUE
FALSE
DISPLAY: "YOU MUST FIND A GIRL AGED ½ AGE + 7"
DISPLAY: "YOU SHOULD LOOK FOR A MAN OF 2★ AGE — 14"
STOP
STOP

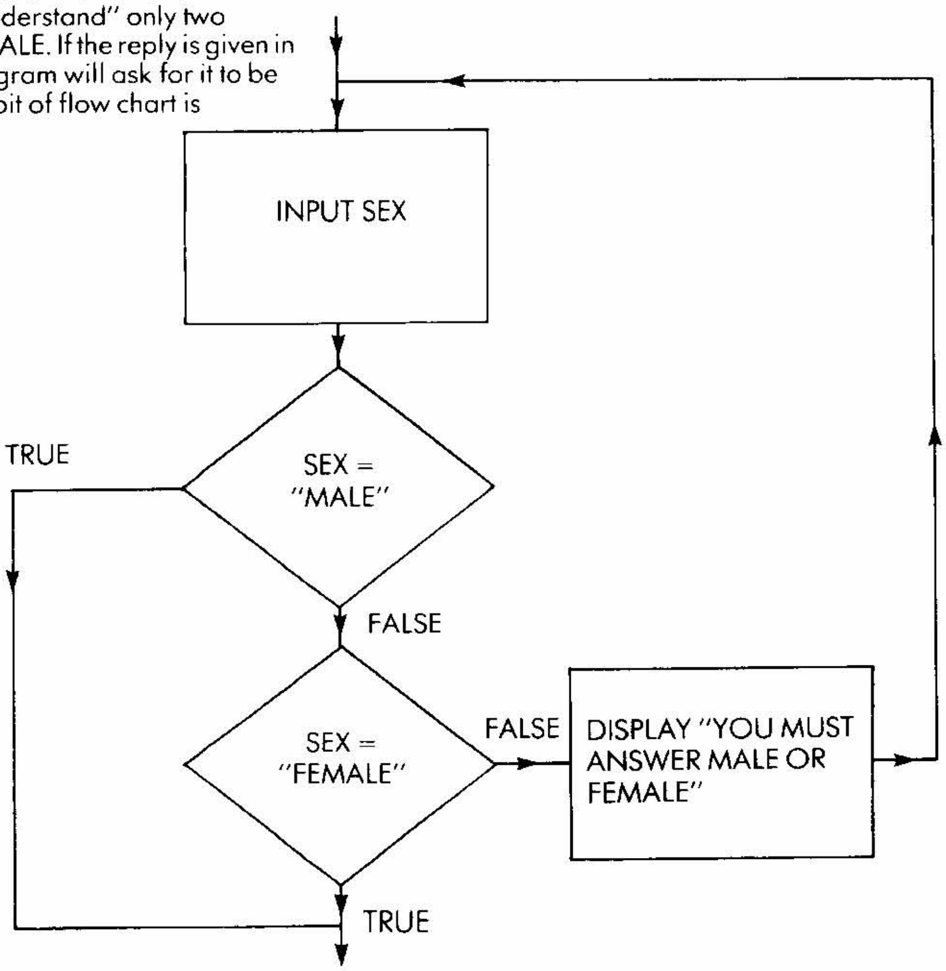
We use a cloud because we haven't yet fixed the details of what we actually mean by "sensible". The cloud is useful because it allows us to plan the program as a whole unit, but it involves an obligation to work out the action in greater detail. At the stage we have reached now the planning is not complete, but that doesn't mean that the main flow chart is useless or wrong!

Well, what does "sensible" mean? First let's think about the age of the user. The lowest likely value is 18, because people under 18 don't often come to marriage bureaux. The upper limit is harder to decide, but according to the *Guinness Book of Records* the oldest living person is 115. We'll take this figure as a guide.

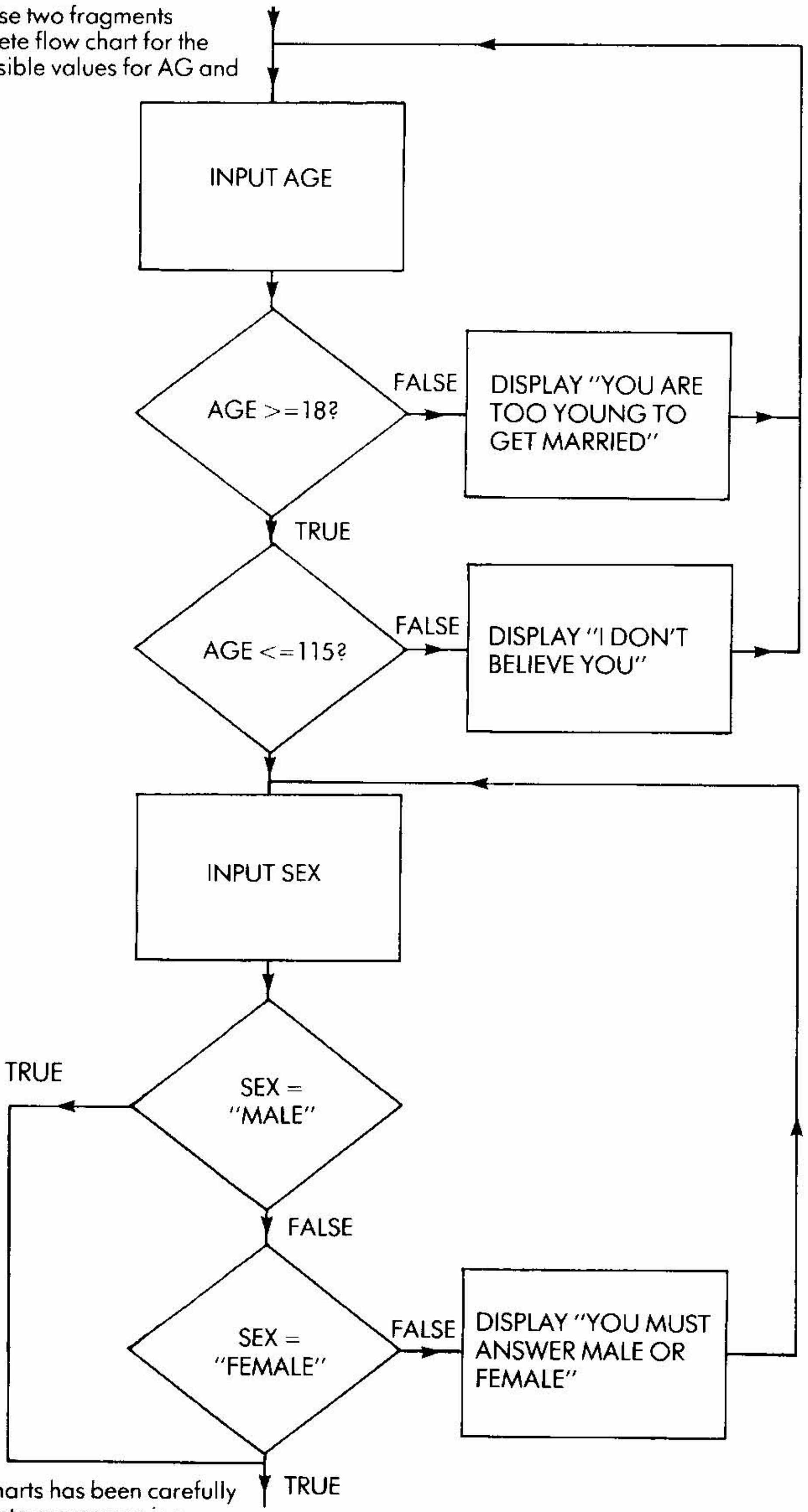
We'll design the program so that when the computer asks for the client's age, it decides whether to accept it as reasonable. If not, it displays a *reason*, and invites the client to give a more realistic figure. Look at this flow chart:


INPUT AGE
AGE >=18?
FALSE
DISPLAY "YOU ARE TOO YOUNG TO GET MARRIED"
TRUE
AGE <=115?
FALSE
DISPLAY "I DON'T BELIEVE YOU"
TRUE

When it comes to the second question, there are lots of ways the client could indicate whether they're male or female. In fact there are so many that we could never think of them all. Instead we'll make the program "understand" only two words: MALE and FEMALE. If the reply is given in any other way, the program will ask for it to be repeated. The correct bit of flow chart is



Now we can put these two fragments together to give a complete flow chart for the cloud which is to get sensible values for AG and SX$.


INPUT AGE
AGE >=18?
FALSE
DISPLAY "YOU ARE TOO YOUNG TO GET MARRIED"
TRUE
AGE <=115?
FALSE
DISPLAY "I DON'T BELIEVE YOU"
INPUT SEX
SEX = "MALE"
TRUE
FALSE
SEX = "FEMALE"
FALSE
DISPLAY "YOU MUST ANSWER MALE OR FEMALE"
TRUE

Once a set of flow charts has been carefully drawn, translating them into a program is a straightforward job. We start at the main flow chart, but the first block there is a cloud, so we refer to the subsidiary flow chart and translate it. Then we go back to the main chart for the rest of the program.

You will notice that two of the diamonds in the subsidiary chart have a TRUE line which goes straight to the end. The simplest way of filling in label numbers of the corresponding IF command is to put a REM at the end of the cloud and use its label number. The REM does nothing but act as a convenient anchor point:

We get:

```
 10 INPUT"WHAT IS YOUR AGE"; AG
 20 IF AG >=18 THEN 50
 30 PRINT "YOU ARE TOO YOUNG TO BE
    MARRIED"
 40 GOTO 10
 50 IF AG<=115 THEN 80
 60 PRINT "I DON'T BELIEVE YOU!"
 70 GOTO 10
 80 INPUT "MALE OR FEMALE"; SX$
 90 IF SX$="MALE"THEN 130
100 IF SX$="FEMALE" THEN 130
110 PRINT"YOU MUST SAY MALE OR FEMALE"
120 GOTO 80
130 REM AG AND SX$ HAVE SENSIBLE VALUES
```

This is followed by the rest of the program as before (but with adjusted label numbers).

```
140 IF SX$="MALE" THEN 180
150 PRINT "YOU SHOULD LOOK FOR"
160 PRINT "A MAN OF"; 2★AG-14
170 STOP
180 PRINT"YOU MUST FIND"
190 PRINT"A GIRL AGED"; AG/2+7
200 STOP
```

Enter this program into the computer and try it out. For sensible values of age it will behave just like the first version, but it will be much better at detecting and refusing silly answers. It has the important quality of *robustness*, or the ability to stand up to abuse.

To end this unit, you will write a program of your own. Before you start, here are some points of advice:

1. Get plenty of clean paper, a pencil and a rubber. Switch off your computer.
2. Study the problem carefully, and work out one or two simple examples yourself. Keep the answers to check against the computer.
3. Begin by deciding what variables you need. Jot down their names, types and purposes in a "glossary". For instance, the variables for the advisory program would have been noted down as:

Name	*Type*	*Purpose*
AG	Number	Age of Client
SX$	String	Sex of Client, as "MALE" or "FEMALE"

4. Draw a flow chart for the program. Be prepared to make lots of mistakes, and don't be surprised if you redraw the chart half a dozen times over. Keep on until you are satisfied. Programming is hard work, and this part of the job — flow charting — is where most of the effort comes.
5. Now translate your flow chart into BASIC. This should be easy. If it isn't, it means that you haven't done your flow charting properly, so go back and do it again.
6. Now — at last — switch on your computer, and enter the program. Apart from a few typing mistakes, it should run without any bother. Test it out on as many different examples as you can, including one you worked out earlier. Finally, preserve the program on tape or diskette (if you want to keep it) and file away your flow chart and variable glossary.

I have just described the way a good professional sets about programming. Lots of people don't do it that way at all — they sit down in front of their computers and compose their programs straight on to the keyboard. This method sometimes works for very small problems, but usually it leads to long, incomprehensible programs which only work some of the time, and which the programmer finds impossible to alter or put right. It also takes much longer to get anything working at all. However, this fact isn't at all obvious — it seems quicker to ignore all the planning and get on with the job. This, in truth, is why so many people program so badly.

You have a choice; you can either do as advised and quickly become a competent programmer, or you can learn the hard way, which will take you very much longer.

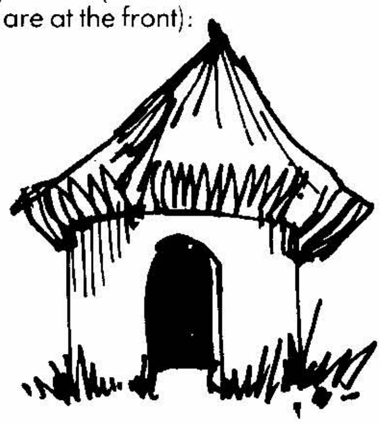
Now plan, flow chart, write and test a program for the following problem:

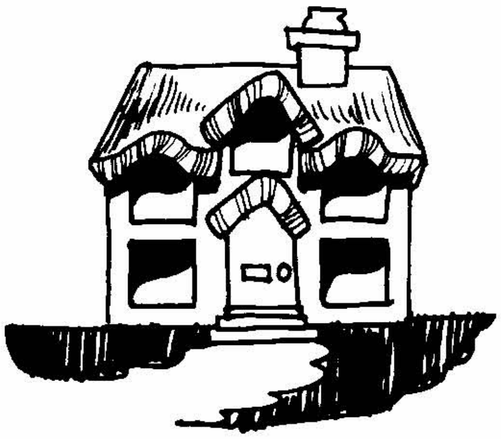
In Ruritania the house-tax is levied as follows:

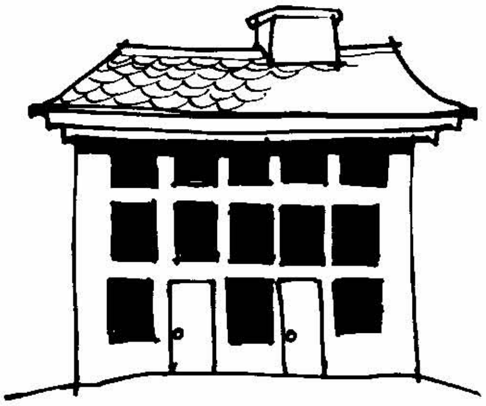
For each door:	£57
For each window:	£12
For each thatched roof:	£38
For a tiled roof:	£94

Assuming all houses must be either thatched or tiled, write a program to ask for the details of any house and display the house-tax payable. For instance, the right answer for a thatched cottage with one door and two windows would be £(38+57+2★12)= £119.

Get your program to display the rates for the following houses (assume all the doors and windows are at the front):







Check your answer in Appendix B.

Experiment 11.2 Completed	

EXPERIMENT 11·3

Load and run the program UNIT11PROG. When you have listed it, examine the code, and draw up a flow chart and a glossary for it.

Experiment 11.3 Completed	

UNIT:12

You don't have to look deep into any program to find a loop somewhere. Loops are so common, and so important, that the BASIC language gives you a short-hand method of writing down the essential details.

You'll remember that there are four vital parts to the control of any loop:

- The choice of control variable
- The starting value for the control variable
- The last or final value for the control variable
- The increment, or amount by which the control variable grows every time round the loop.

All these parts can be fitted into *one* special command which uses the keyword FOR. This is all that is needed to set up a loop except for a NEXT command to mark the end of the loop body.

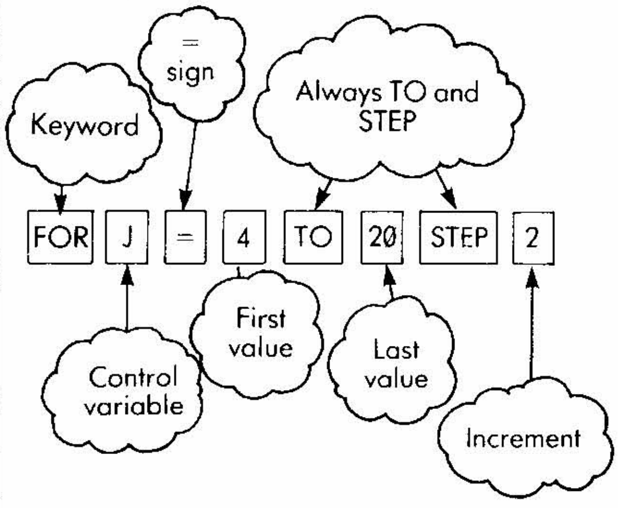
Compare the following two programs, which give exactly the same result:

```
10 J=4                  10 FOR J=4 TO 20 STEP 2
20 PRINT J,J★7          20 PRINT J, J★7
30 J=J+2                30 NEXT J
40 IF J<22 THEN 20

(Using IF.... THEN)     (Using FOR.... NEXT)
```

In both cases:

Control variable is J	
First value is	4
Last value is	20
Increment is	2

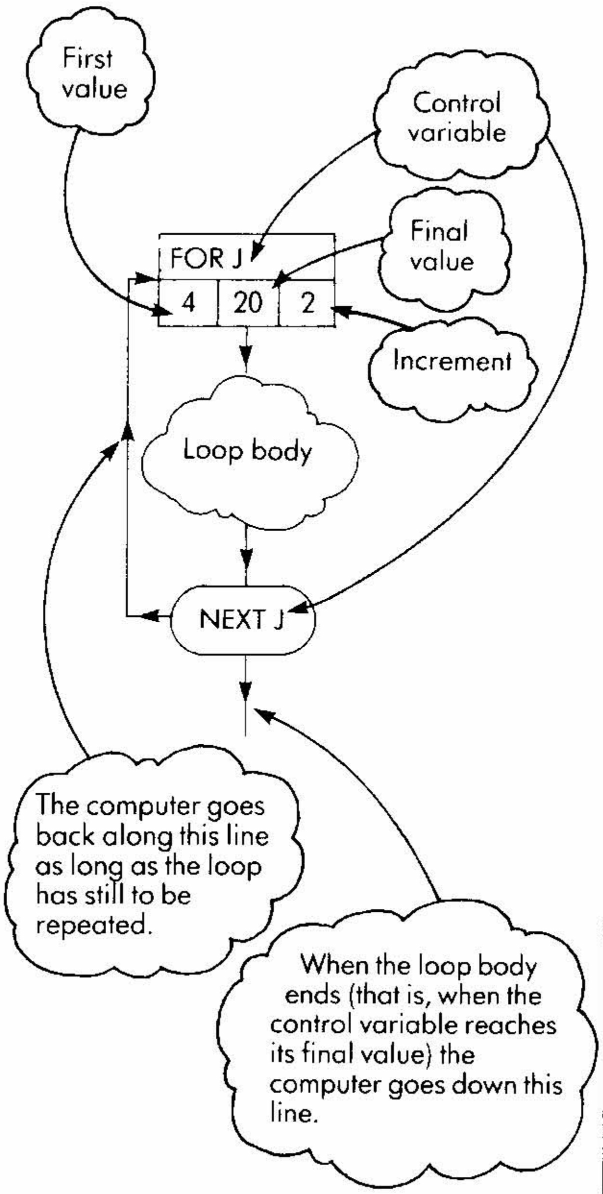
The example shows how the FOR command is built up



The NEXT command mentions the name of the control variable, as a check to help you read the program.

Every FOR command must have a corresponding NEXT, and between them they *enclose* the body of the loop.

In flow charts we show loops in a special way, using blocks which can't easily be mistaken for other kinds of action:



First value
Control variable
Final value
FOR J
4
20
2
Increment
Loop body
NEXT J
The computer goes back along this line as long as the loop has still to be repeated.
When the loop body ends (that is, when the control variable reaches its final value) the computer goes down this line.

EXPERIMENT 12·1

To help fix the details of the FOR command in your mind, look at the following short programs and write down what you think they will make the computer display. Then check your answers on the machine itself:

(i)
```
10 FOR Q=1 TO 16 STEP 5
20 PRINT Q;
30 NEXT Q
40 STOP
```

Your prediction:

(ii)
```
10 FOR R=38 TO 50 STEP 3
20 PRINT R; 50−R
30 NEXT R
40 STOP
```

Your prediction:

Now translate the following program into to FOR–NEXT notation. Check your answer by running both versions on the computer and ensuring that they give the same answers:

```
10 PRINT "NINE TIMES TABLE"
20 S=1
30 PRINT S; "TIMES 9 =";
40 PRINT 9*S
50 S=S+1
60 IF S<13 THEN 30
70 STOP
```

Your translation:

There are a few points about the FOR and NEXT commands which you ought to remember:

(a) If the increment or steps size is 1, the "STEP 1" at the end of the FOR command can be left off. The computer understands what is meant.

(b) The loop control can be made to count *backwards* by using a negative step size. The program

```
10 FOR X=10 TO 5 STEP −1
20 PRINT X;
30 NEXT X
```

will display:
```
10   9   8   7   6   5
```
in that order.

(c) The body of the loop is always obeyed at least once, even if the final value is less than the starting value. For example,

```
10 FOR R=5 TO 3
20 PRINT R
30 NEXT R
```

will display
```
5
```

(d) The values in the FOR command needn't be numbers but can be expressions which include other variables. For example, the following program will display the number of heart symbols requested by the *user*. Try it out and study it carefully:

```
10 INPUT "HOW MANY HEARTS"; H

20 FOR K=1 TO H

30 PRINT " CTRL and RED ♥ ";

40 NEXT K

50 STOP
```

(e) The control variable can't be a string. For instance, the "command"

~~FOR X$ = "A" TO "ABBBB" STEP "B"~~ ← NOT BASIC

would give a SYNTAX ERROR, and you aren't allowed to use this construction. Using this knowledge, predict the outcome of the following programs, and check your results on the computer:

(i)
```
10 FOR A=1 TO 4
20 PRINT A★A;
30 NEXT A
40 STOP
```

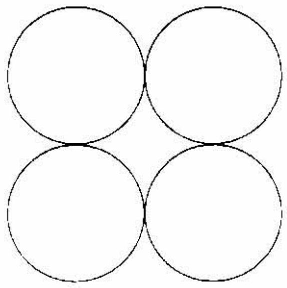
(ii)
```
10 FOR B=3 TO 0 STEP -1
20 PRINT B;
30 NEXT B
40 STOP
```

(iii)
```
10 FOR C = 5 TO 4
20 PRINT C;
30 NEXT C
40 STOP
```

(iv)
```
10 X=5
20 Y=9
30 Z=2
40 FOR W=X TO Y STEP Z
50 PRINT W;
60 NEXT W
70 STOP
```

So far we've been concentrating hard on the details of FOR and NEXT commands, so we have carefully chosen the bodies of the loops being controlled to be as simple as possible. In practice the body of a loop needn't be short and simple, but can be as complex as you like — the thing to remember is that it gets executed *every time* the computer goes round the loop.

Suppose you've been asked to build a square-based pyramid, out of cannon-balls. We'll number the layers 1, 2, 3, ... starting from the top. Layer 1, being the point, will need just one cannon-ball. Layer 2, the second one, will need four balls arranged like this:



Layer 3 will need nine balls, layer 4 — sixteen, and so on.

Clearly the number of cannon balls you need for the whole pyramid depends on how many layers you plan to build. A three-layer pyramid needs 1+4+9 or 14 cannon balls; one with four layers will require 1+4+9+16 or 30.

If you plan a very large pyramid, these sums will get rather long and boring, and you might decide to write a computer program to do them for you. This program will answer the question, "How many cannon balls will I need for a pyramid of 'so many' layers?".

In designing the program, a key factor is the number of cannon balls in each layer. The numbers 1 4 9 16 . . . . and so on look familiar, and in fact you soon spot that the number of balls in each layer is the *square* of the layer number. For instance, layer 7 will need 7 ★ 7, or 49 cannon balls.

Now for the details of the program. Let's begin by thinking about the variables we'll need.

Our overall plan will be to consider the layers one by one. We will get the computer to work out how many balls are needed for that layer, and add this number to a 'running total'. At the beginning the running total must be set to zero. At the end, when all the layers have been taken into account, the running total will show the number of cannon balls wanted for the whole pyramid. This is the answer to the problem.

A suitable name for the running total is RT.

We need two other variables:

(i) The number of layers in the pyramid. Remember that the *programmer* doesn't know this number; it is up to the *user* to supply any value he wants. A good *name* for this variable is L.

(ii) As the program runs it will deal with layer 1, then layer 2, then layer 3, and so on. We need a variable to indicate *which* of the L different layers the program is dealing with at any moment. A suitable variable name is V. Since V is going to take all the values between 1 and L, the number of the bottom layer, we can guess that it will be the control variable in a FOR command, thus:

```
FOR V=1 TO L

.......

NEXT V
```

The *glossary* for our program is thus:

Name	*Purpose*
RT	To keep running total of cannon balls
L	Number of layers in pyramid
V	Number of layer being dealt with at any moment.

Next, we'll write down some of the actions our program needs to take:

Add V squared to RT	(This adds in the number of cannon balls for layer number V)
Print RT	(Displays result)
Set RT=Ø	(Starts RT off from zero)
Input L	(Asks user how many layers there are in his pyramid)
FOR V=1 TO L NEXT V	(Loop control for taking every layer into account)
STOP	

These are all the fragments of program we need, but they have still to be put together in the right order. We have already decided that there must be a loop, and it will greatly help us if we can say, for each command, whether it should be executed

before the loop starts

or inside the loop (as part of the loop body)

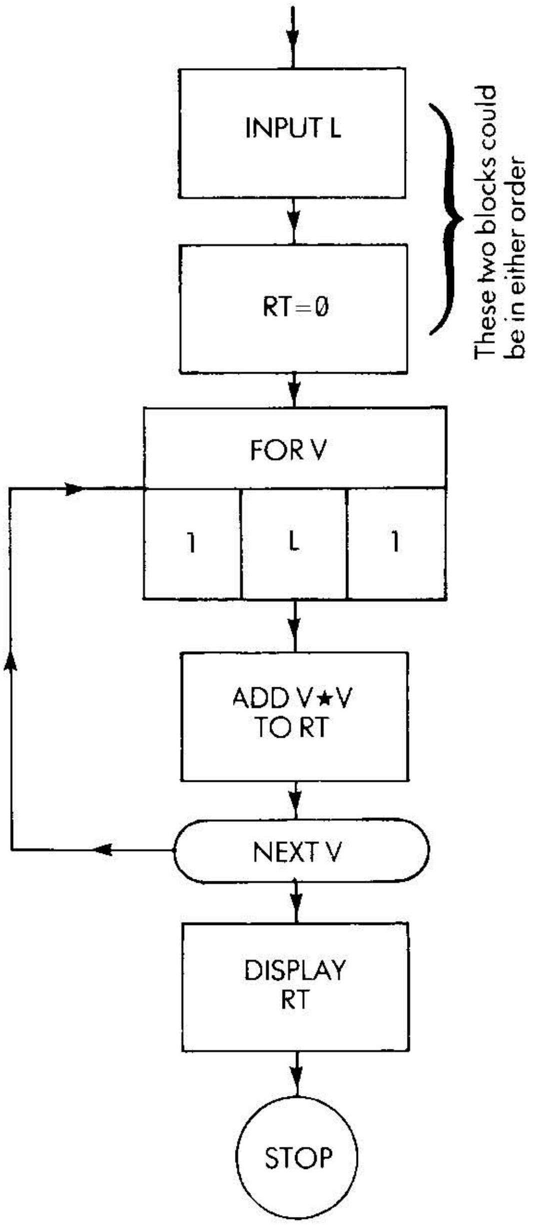
or after the loop has ended.

We can use various clues. The program has to know how many times to go round the loop before the loop itself can start, so input L must come *before* the loop. So must the command which sets RT to zero.

The total number of cannon balls for the whole pyramid includes at least some for each layer. The command to add a layer's worth to RT has to be repeated many times, and so it goes *inside* the loop.

Finally, the computer can't give you the right answer until it's taken all the layers into account, so the PRINT command can only come *after* the loop has ended.

Now we've got far enough to draw a flow chart. It is



And the corresponding program is

```
10 INPUT "NUMBER OF LAYERS"; L

20 RT=Ø

30 FOR V=1 TO L

40 RT=RT+V★V

50 NEXT V

60 PRINT RT;"CANNON BALLS NEEDED"

70 STOP
```

Enter this program and try it out.

Now here is a problem for you. In the game of cricket, a player can have a number of separate 'innings' during the season. Each time he scores some 'runs': many if he is a good player or lucky, or only a few (or even none) if he isn't so skilful. If you want to know how well someone has played over the whole season, you work out the *average* number of runs per innings. You get it by adding up all the runs he gains over the season and dividing by the number of innings. For instance, if he plays three times and scores 2Ø, 3Ø and 7Ø, his *average* is (2Ø+3Ø+7Ø)÷3 or 4Ø runs per innings.

Consider a program which does this calculation for you. It has to ask you for the number of innings, and then the score for each one, so that it can add them up together. The overall display would be like this:

```
RUN

NUMBER OF INNINGS? 3

SCORE? 2Ø                 Numbers

SCORE? 3Ø                 typed by

SCORE? 7Ø                 user

AVERAGE = 4Ø
```

Your job is to write the program for this problem. To make it easier, we'll give you a glossary and all the commands, but in jumbled order and with their labels stripped off. Begin by drawing a 'skeleton' with the loop commands, and then slot in the other commands in the right places. Finally, run the program on the computer and make sure that it works. If you get really stuck, look up the correct answer in Appendix B, but remember: this is an admission of failure!

The glossary and jumbled commands are:

Name	*Purpose*
J	Number of innings during season
Q	Control variable for loop
RS	Used to add up the total runs scored
S	Score for each separate innings

```
NEXT Q

INPUT"NUMBER OF INNINGS"; J

INPUT"SCORE"; S

PRINT"AVERAGE= ";RS/J

STOP

RS=Ø

FOR Q=1 TO J

RS=RS+S
```

Experiment 12.1 Completed	☐

EXPERIMENT 12·2

We end this section with a problem which you must solve without any help. If you go to the Post Office, you are quite likely to get stuck in a queue just behind someone buying a huge amount of stamps. You hear her saying:

"Eighty-three at 12½p

and One hundred and seventeen at 16p

and Thirty-five at 75p"

and so on. When all the stamps have been counted out, the clerk spends ages working out how much it all costs.

Write a program to help the clerk. The display should be something like this:

```
RUN

NUMBER OF BATCHES?      4

BATCH 1

  NUMBER OF STAMPS?     2Ø

  VALUE (EACH)?         15

BATCH 2

  NUMBER OF STAMPS?     5      Numbers typed by user

  VALUE (EACH)?         75

BATCH 3

  NUMBER OF STAMPS?     4

  VALUE (EACH)?         1

BATCH 4

  NUMBER OF STAMPS?     1ØØ

  VALUE (EACH)?         14

TOTAL DUE=2Ø79 PENCE
```

The program should be 1Ø commands long (including STOP). Four of these commands will form the body of a loop, obeyed once for each batch. However, don't try to write the program yourself until you have a proper design, with glossary and flow chart. Take plenty of time.

If, after spending a good deal of effort, you still can't get this problem right, go back a few units to a place where you feel confident, and work through the course material again.

Finally compare your answer with that given in Appendix B.

Experiment 12.2 Completed	

The self test quiz for Unit 12 is called "UNIT12QUIZ".

UNIT:13

EXPERIMENT 13·1

This unit is about a topic which is both easy and fun: using the computer to make sounds and musical notes.

Load the program entitled SOUND DEMO, turn up the volume on your TV set, and play through the selection of sound effects in the program. They are wide-ranging and give you some idea of what the computer can do in the way of sound production.

In due course you will want to design and program your own sounds, and this is what the unit is all about.

Sound production on the computer is controlled by two commands: VOL and SOUND.

Let's begin by explaining VOL, since it is the easier of the two. In a program, the keyword VOL is always followed by a number (or an expression) in the range 0 to 8. The command controls the loudness of sounds the computer makes; it is, in effect, a "volume control". The sounds are at their loudest when you have given

```
VOL 8
```

and reduced to silence when you give

```
VOL 0
```

Intermediate degrees of volume are selected by VOL 2, VOL 4, and so on.

When you write a program which includes sounds, you should always put

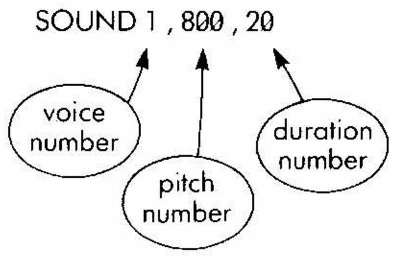
```
VOL 7
```

near the beginning. When you run the program you must also turn up the volume control on the television receiver (or monitor) otherwise you will hear nothing.

The command used to produce an actual musical note is SOUND. Every note has two important qualities:

- its pitch (whether it is high or low).
- its duration (how long it lasts).

In a SOUND command, the keyword is followed by three numbers or expressions, like this:



The computer has three different voices, which are numbered 1, 2 and 3. At present we shall only be using voice number 1. All our SOUND commands will start

```
SOUND 1 , . . .
```

The *pitch* of the note is controlled by the pitch number, which can have any value between 0 and 1023. In broad terms, the higher the value, the higher the note, but the spacing of the notes is not at all regular. The following short program plays every possible pitch between 0 and 1023, and lets you hear the limits of the machine's vocal range:

```
10 VOL 7
20 FOR J = 0 TO 1023
30 SOUND 1,J,1
40 NEXT J
```

The chart below shows you how the pitch numbers are related to the notes of the musical scale. This information will be useful when you come to write music for the computer.

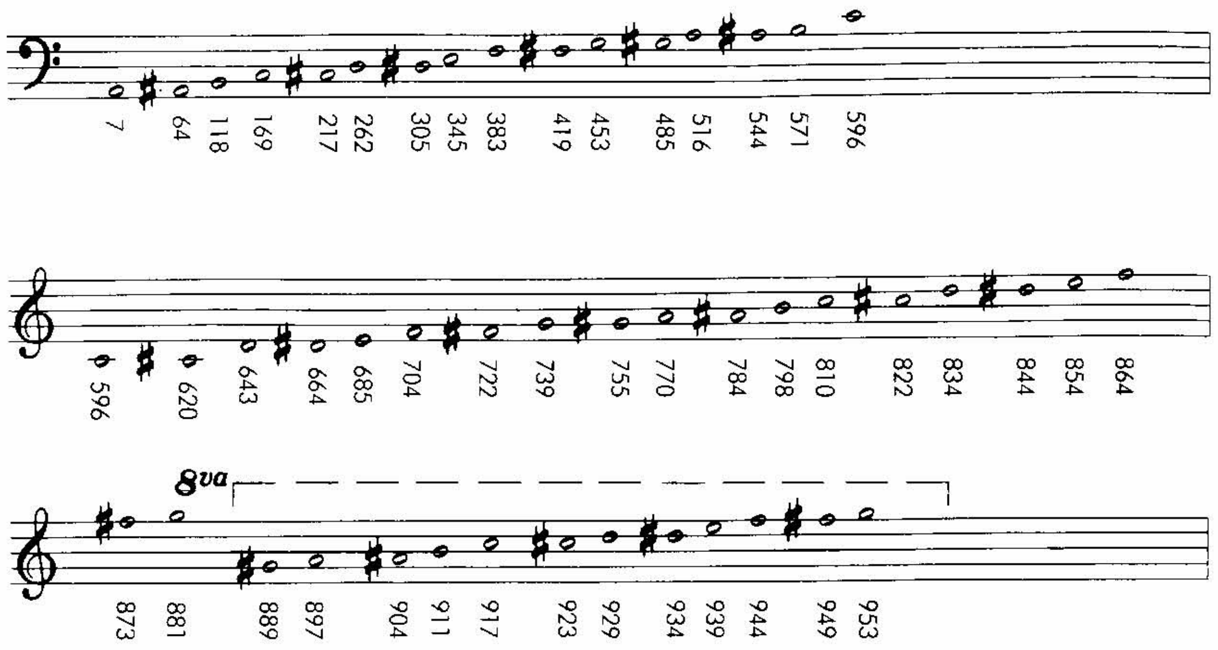


Figure 13.1

The length of the note, in jiffies (or 1/60ths of a second) is set by the duration number. For example, the note sounded by

```
SOUND 1,900,120
```

will last for exactly 120 jiffies or two seconds.

One of the simplest and most rewarding uses of sound is to inform a user whether the reply typed to some question is acceptable. Consider a program where the computer asks someone to say if they would prefer red wine (R), white wine (W), beer (B) or nothing (N). The user is invited to type the initial letter of his choice. If he types R,W,B or N (as he should) the machine gives a cheerful 'pip' and continues to the next question. If the user types something else, the computer sounds a low and mournful tone and repeats the choice again. The code for such a sequence would be as follows:

```
 10 VOL 7
 20 PRINT "RED WINE (R),
    WHITE WINE (W)"
 30 PRINT "BEER (B) OR NOTHING (N)"
 40 INPUT "PLEASE GIVE INITIAL LETTER";
    L$
 50 IF L$ = "R" THEN 140
 60 IF L$ = "W" THEN 140
 70 IF L$ = "B" THEN 140
 80 IF L$ = "N" THEN 140
 90 PRINT
100 PRINT "PLEASE ANSWER R, W, B OR N"
110 PRINT
120 SOUND 1,20,70 : REM LOW TONE
130 GOTO 20
140 SOUND 1,930,12 : REM PIP
150 PRINT "OK"
160 . . .
```

Another use for the SOUND command is in making your computer play tunes. Each note of a melody can be played by its own SOUND command. Here is a well-known Scottish tune, "The Road to the Isles".

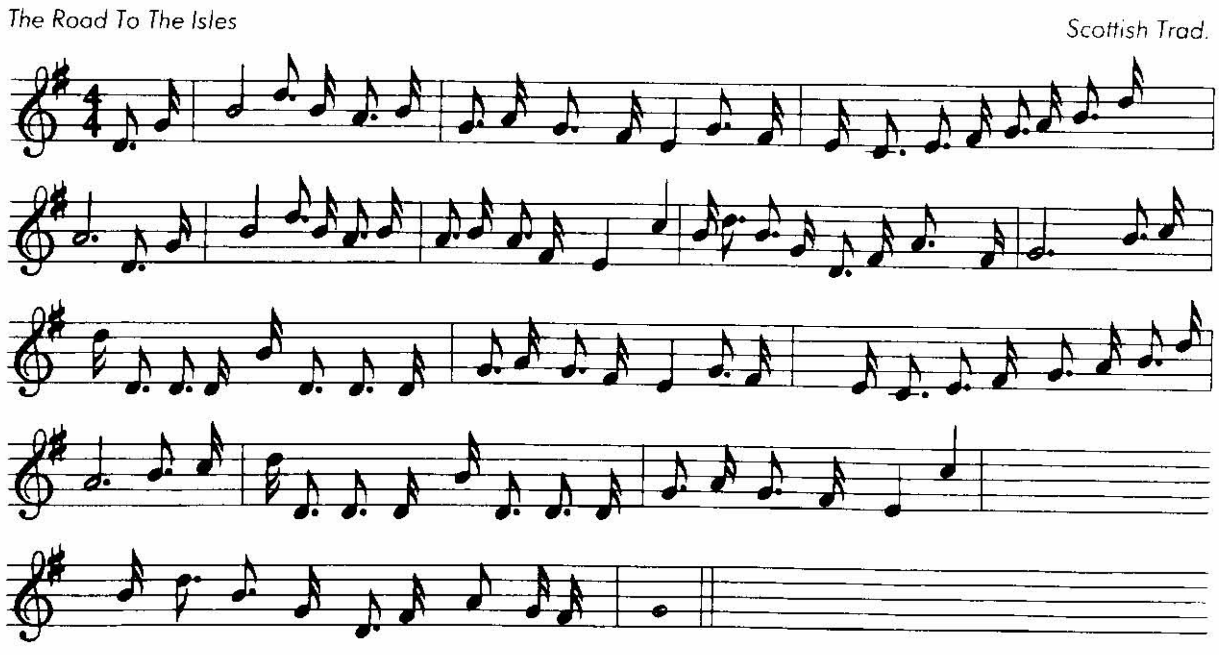


Figure 13.2

The first part of the melody has been coded up as program TUNE, which you should load and run. When you list the program, you will see how the coding has been done. Each note is translated into one SOUND command. Every beat of the tune is 32 jiffies long (this gives a metronome rate of 60★60/32 or 112 beats per minute). The first note is a dotted quaver or "three-quarters note" pitched at D, so it is coded as

```
SOUND 1, 643, 24
       ↗      ↖
pitch for D      three-quarters of 32
```

One of the most widely used devices in music is the 'rest' or beat of silence. Normally, you can make the machine pause for any length of time by giving it an empty loop, like this:

```
FOR J = 1 TO 1000 : NEXT J
```

With music, however, matters are not so simple. Clear the machine and type

```
10 VOL 7
20 SOUND 1,400,120
30 FOR J = 1 TO 1000:NEXT J
40 SOUND 1,400,120
```

When you run this program, the expected rest between the two notes isn't there! Why not? Because the SOUND command merely starts a note off. The voice then keeps on sounding the note while the machine goes on with the rest of the program. The computer is only held up when it tries to sound a note before the previous note is ended.

Now we can see exactly what happened when you ran your program. The machine started the first note, and immediately went on to the idle loop in line 30. After one second it completed the loop and tried to start the second note — but the first one was still sounding. When the first note ended the second one started immediately, without any gap.

The solution to the problem is now clear. If you want a note to be followed by a rest, the idle loop has to swallow up all the time the note lasts as well as the silent time. For example, if you want a note of a second followed by a second's rest, the loop must take up two seconds of computer time.

A similar problem arises if you want to play two notes of the same pitch. The sequence

```
SOUND 1, 400, 30

SOUND 1, 400, 30
```

is indistinguishable from

```
SOUND 1, 400, 60
```

because the two notes are run together without any gap. You need to put

```
SOUND 1,400, 30
FOR J = 1 TO 500: NEXT J
SOUND 1,400,30
```

As an exercise, complete the coding of "The Road to the Isles". Pay special attention to the repeated notes.

Experiment 13.1 Completed	

EXPERIMENT 13·2

The shortest note you can play on the computer is only one jiffy long. This makes it possible to produce some interesting effects by playing many short notes of variable pitch. The following example will make this clear:

```
10 VOL 7
20 FOR X = 700 TO 900 STEP 4
30 SOUND 1,X,1
40 NEXT X
50 GOTO 20
```

For another effect, try adding

```
35 SOUND 1,X+50,1
```

Now we'll consider the other two voices on the computer. Voice 2 by itself works in exactly the same way as voice 1. If you try to use both voices together, you'll get severe electronic interference between them; the effect is interesting but hardly harmonious! Voice 3 is used to produce a type of sound often called 'white noise' — a hiss with a poorly defined pitch. This type of sound is excellent for imitating steam engines, jet planes, rockets, and alien space-ships — if you know what they sound like. Try

```
10 VOL 7
20 FOR X=850 TO 600 STEP -1
30 SOUND 3,X,2
40 NEXT X
```

and a more complex example:

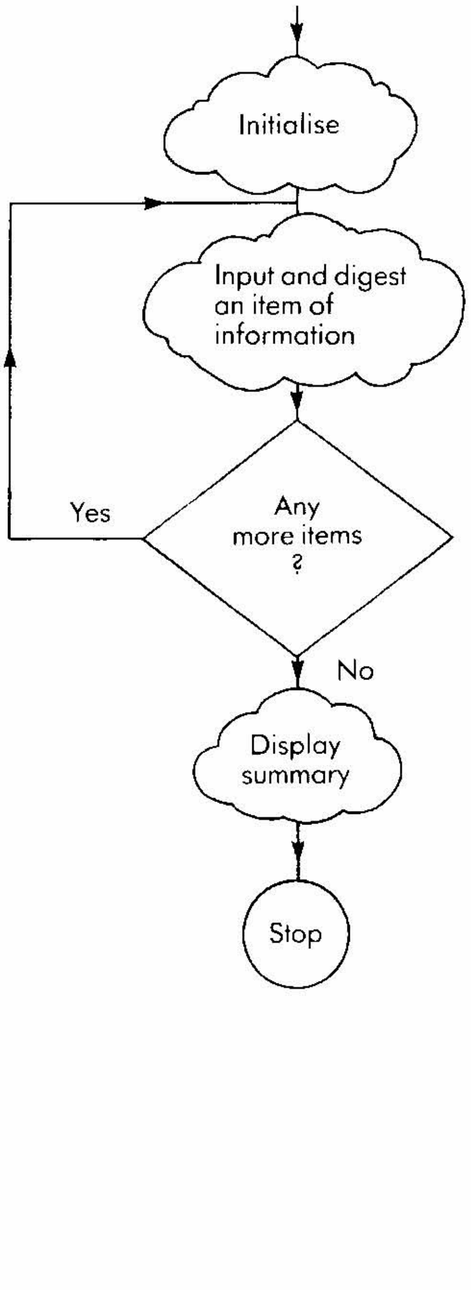
```
10 VOL 7
20 J=50
30 FOR L = 1 TO 100
40 J=0.97★3
50 M=J+15
60 SOUND 3, 860-M,M/3
70 FOR K=1 TO 4★M: NEXT K
80 NEXT L
```

Experiment 13.2 Completed	

UNIT:14

In this unit we'll look at an extremely common type of computer application: one where the machine is made to input and digest a large number of separate items of information, and to display a summary of its results. For instance, if you wanted to keep track of your bank account, you could feed in the details of every cheque you write, and every credit you pay in to the bank, and the machine would tell you your balance at the end of the week. To give another example, a school teacher could give the computer all the exam marks gained by the pupils in the class, and the computer would display the overall average mark.

All programs of this type conform to the same basic pattern, which has a flow chart something like this:



A very simple example is this program which inputs 10 numbers and finds their average value:

```
10 S=0  }
         } Initialise
20 P=1  }

30 INPUT X }
            } Read and digest an item
40 S=S+X   }

50 P=P+1          }
                   } Any more items?
60 IF P < 11 THEN 30 }

70 PRINT"AVERAGE=";S/10 } Display
                        } summary

80 STOP
```

Glossary

S: Used to add up values of items
P: Used to count the items
X: Used to input individual items

If you don't understand how this program works, *trace* it with the input values 3, 6, 2, 7, 0, 9, 8, 3, 12, 10.

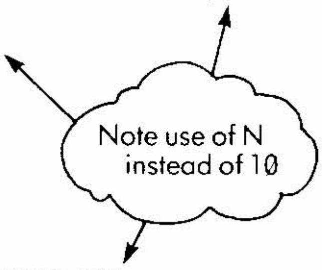
In this example, we've used an IF-THEN for the loop control to make the construction of the program more clear. In practice we would write the program with a FOR . . NEXT, like this:

```
10 S = 0

20 FOR P = 1 TO 10

30 INPUT X

40 S = S+X

50 NEXT P

60 PRINT "AVERAGE IS"; S/10

70 STOP
```

Let's think about the part of the program which says "any more items". In the first example the question was answered by keeping a simple count, and using the condition P < 11, which was true until the tenth item was input and added to the running total. This method depends on the *programmer* knowing in advance how many items there are going to be. The method is almost useless in practice because it is so inflexible: you would need different programs to find the average of 11, or 20 or any other number of numbers.

You can write a much better program if you assume that the *user* can tell the computer how many items to expect. The following program will work for any number of items:

```
10 S = 0

20 INPUT "HOW MANY NUMBERS"; N

30 FOR P = 1 TO N

40 INPUT X

50 S = S + X

60 NEXT P

70 PRINT "AVERAGE IS"; S/N

80 STOP
```



Glossary

S: Used to add up value of items

P: Used to count the items

X: Used to input individual items

N: Used to hold the *number* of items

So far we have been on familiar ground; but what about the case where the user has to feed in a large number of items (like a thousand or more)? It is unfair to make him count the items in advance, and unrealistic to suppose that he'll get the number right.

A different way of controlling a loop is not to use a predetermined count at all, but simply to tell the computer when the stream of items has ended. We could, for example, get the user to answer the question "any more items" each time round the loop. This would lead to a program like

```
10 S=0

20 N=0

30 INPUT "NEXT NUMBER"; X

40 S=S+X

50 N=N+1

60 INPUT "ANY MORE NUMBERS"; M$

70 IF M$ = "YES" THEN 30

80 PRINT "AVERAGE IS"; S/N

90 STOP
```

Glossary

S: Used to add up values of items

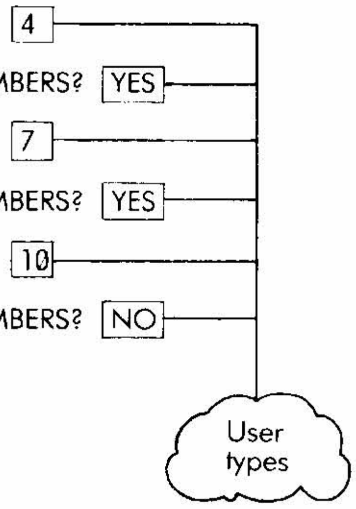
N: Used to count items

X: Used to input individual items

M$: Used to hold answer to question "Any more items"?

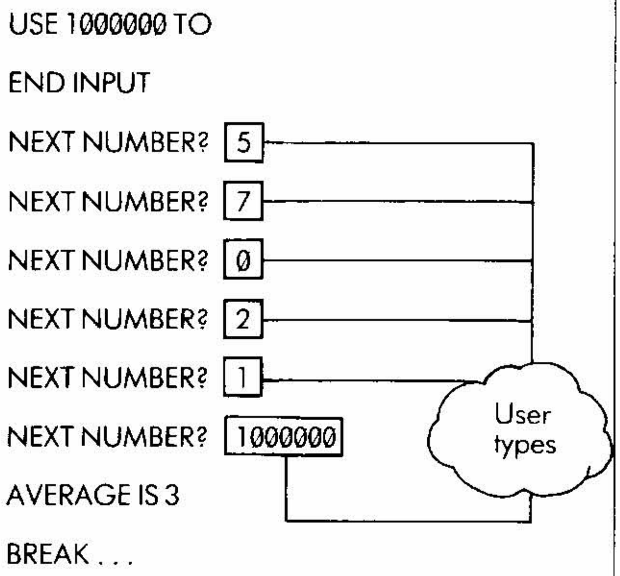
If you ran this program, the display might be:

```
NEXT NUMBER? 4

ANY MORE NUMBERS? YES

NEXT NUMBER? 7

ANY MORE NUMBERS? YES

NEXT NUMBER? 10

ANY MORE NUMBERS? NO

AVERAGE IS  7

BREAK IN 90

READY
```



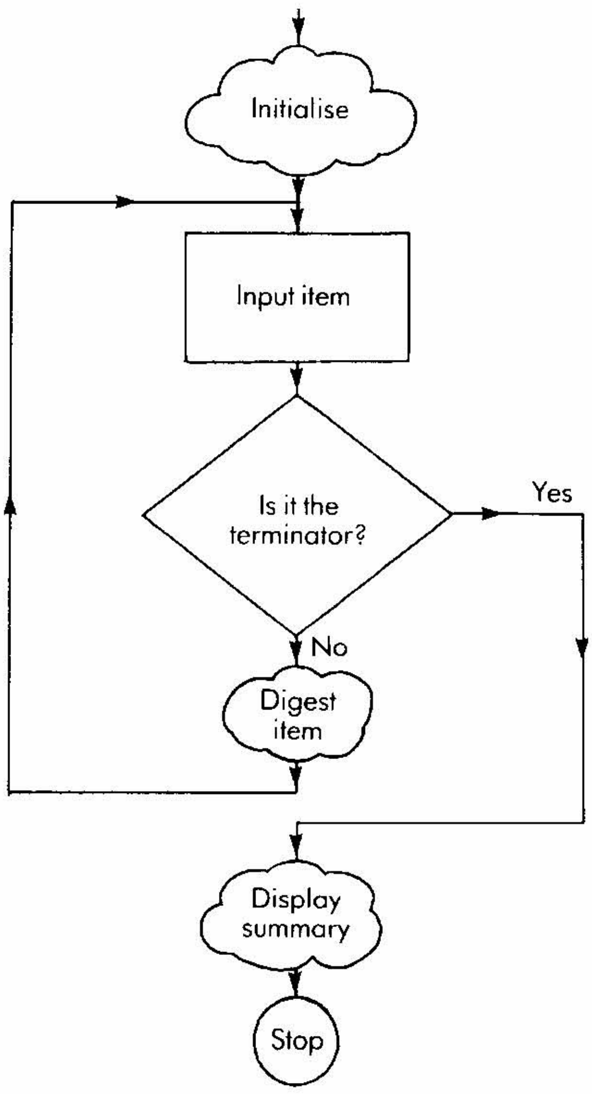
The drawbacks of this scheme are clear. The unfortunate user has to keep typing YES after every number except the last. This takes double the time, and doubles the risk of mistakes. A better method is to mark the end of the stream of items with a special value called a *terminator.* A good choice for a terminator is a value which couldn't possibly occur as one of the items. For instance, if you plan to use the program to average football scores, you could use the number 1000000, because you may be sure that no team can ever score a million goals in one match.

The display produced by a program written on these lines could be:



```
USE 1000000 TO

END INPUT

NEXT NUMBER? 5

NEXT NUMBER? 7

NEXT NUMBER? 0

NEXT NUMBER? 2

NEXT NUMBER? 1

NEXT NUMBER? 1000000

AVERAGE IS 3

BREAK . . .
```

To use this system we have to re-arrange the overall flow chart; in particular, the question "any more data" must come *before* the block which digests each data item — otherwise the terminating value would be treated as an ordinary item and would upset the summary.



The corresponding program for finding an average is quite straightforward:

```
 10 PRINT "USE 1000000 TO"

 20 PRINT "END INPUT"

 30 S=0

 40 N=0

 50 INPUT "NEXT NUMBER"; X

 60 IF X = 1000000 THEN 100

 70 S=S+X

 80 N=N+1

 90 GOTO 50

100 PRINT "AVERAGE ="; S/N

110 STOP
```

Glossary

S: Used to add up values of items

N: used to count items

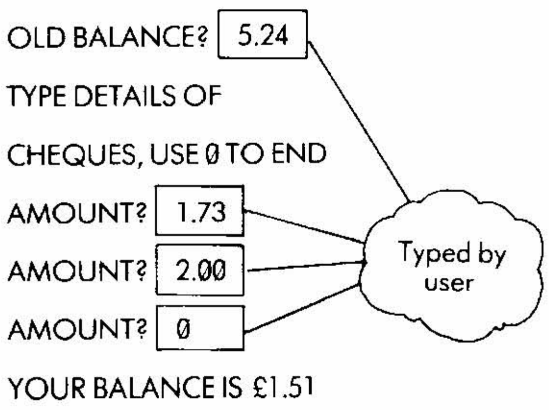
X: Used to input individual items

To summarise, we have looked at four different ways of indicating how many items of information are to be input by a program. They are:

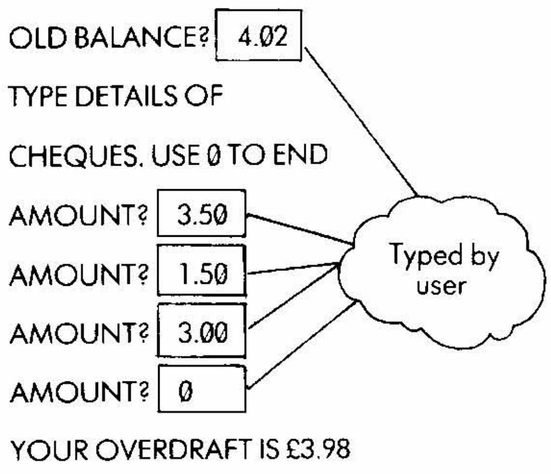
1. Number of items is specified by the programmer. Used only by beginners and useless in practice.
2. Number of items is specified in advance by the *user*. A good method if there are 20 items or less.
3. User indicates *after each item* if there are any more to follow. Intolerably tedious.
4. Stream of items ends with special value. A good method, generally better than the others.

EXPERIMENT 14·1

Write a simple banking program which inputs your old balance, and details of all the cheques you have written, and then displays your new balance or overdraft. Use the number zero as a terminator, because you will never write a cheque for £Ø.ØØ. Don't worry about credits. Design your program so that it could produce either of the two displays which follow:

(a)


```
OLD BALANCE? 5.24
TYPE DETAILS OF
CHEQUES, USE Ø TO END
AMOUNT? 1.73
AMOUNT? 2.ØØ
AMOUNT? Ø
YOUR BALANCE IS £1.51
```

(b)


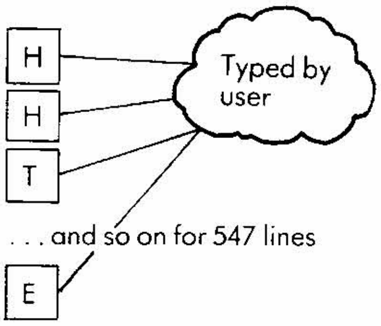
```
OLD BALANCE? 4.Ø2
TYPE DETAILS OF
CHEQUES. USE Ø TO END
AMOUNT? 3.5Ø
AMOUNT? 1.5Ø
AMOUNT? 3.ØØ
AMOUNT? Ø
YOUR OVERDRAFT IS £3.98
```

Hint: Your display section will be a little more complex than usual. If B is a variable which gives the current balance, then it will be *negative* (or less than zero) if you are overdrawn at the bank. The right condition to check this possibility is B < Ø. Your solution should include a flow chart and a glossary. Check it against the answer in Appendix B.

Experiment 14.1 Completed	

In some problems the various items in the stream have to be treated in different ways. The corresponding programs generally have 'IF' commands inside their main loops. For example, let's suppose that after a run of very bad luck in gambling you became suspicious that a coin was biased, so that it came up 'heads' much more often than 'tails'. You could follow up your hunch by tossing the coin a large number of times, and counting the number of heads and tails which came up. You might want the computer to help you keep the score, so you would write a program which produced a display like this one:

```
TYPE H FOR HEADS

T FOR TAILS

E FOR END

NEXT THROW? H

NEXT THROW? H

NEXT THROW? T

. . . . . . . . . . . . . . . . . and so on for 547 lines

NEXT THROW? E

OUT OF 547 THROWS

THERE WERE 49Ø HEADS

AND 57 TAILS

READY.
```



And you could draw your own conclusions about the bias of the coin.

Let's design and write this program, from glossary and flow chart down to BASIC commands.

The sample output shows that we use a special value, E, to terminate the stream of data items. The outline flow chart will be the same as the one on page 111, and all one need do is expand the clouds.

The program obviously needs three variables:

H: To count number of heads
T: To count number of tails
I\$: To input an item

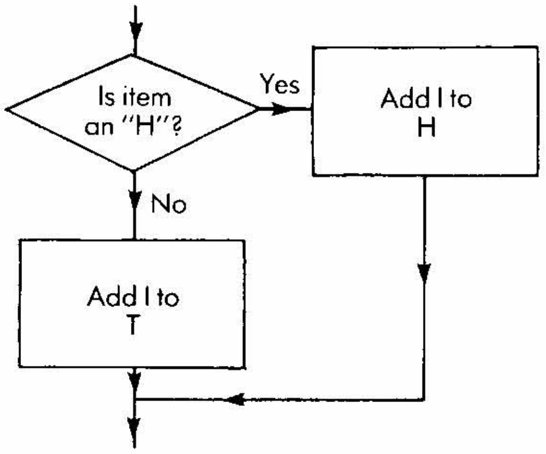
(As usual, the names H and T are freely chosen.)

Some people might be tempted to include a fourth variable to count the total number of tosses, but there is hardly any point; the total is always given by the expression H + T (the number of heads *plus* the number of tails).

Next we can work out the initialisation section of the program. There are two things to do:

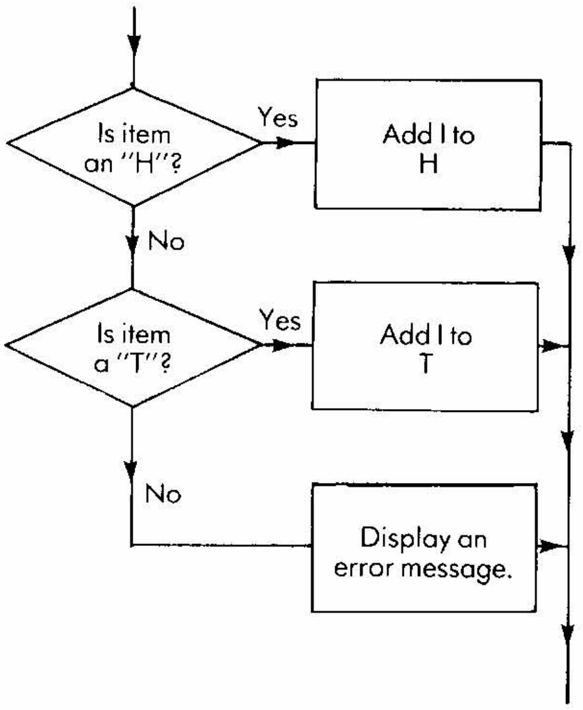
- Set variables H and T to zero
- Display the heading message.

Next we turn to the cloud inside the main loop, which digests each new item. By this stage, the 'E' will have been filtered out, and every item *ought to be* an H or a T. The basic job the cloud has to do is to add 1 either to the heads total, or to the tails total. One possible approach would use the argument "Is it an H? If not, it must be a T". This would result in a flow chart like



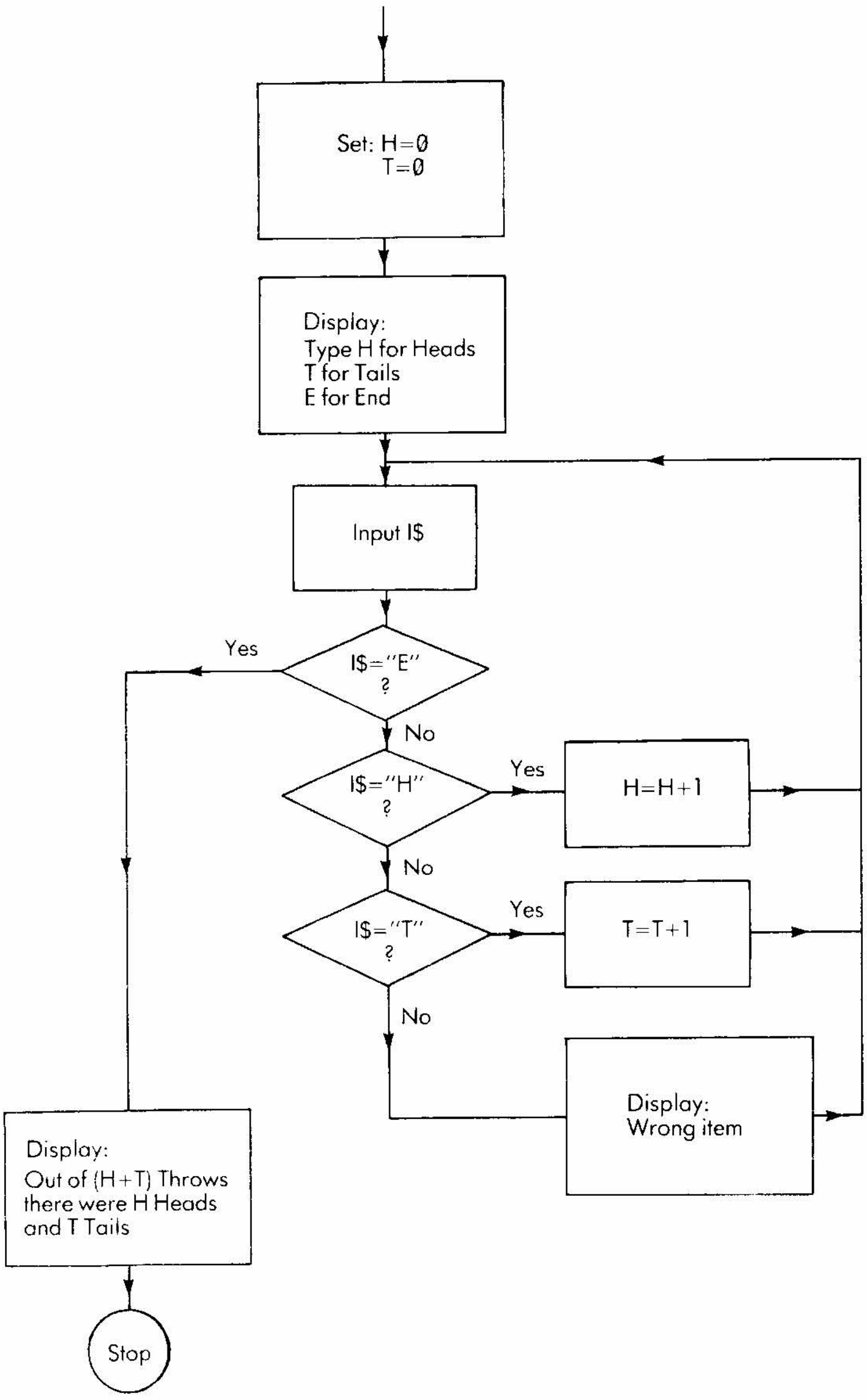
In practice, this method would never be used by a good professional programmer, because it doesn't allow for the user's typing mistakes. If the user hits a J instead of an H (they are next to each other on the keyboard) the program would count it as a T, which is most unlikely to have been what the user wanted.

It is much better to allow for the possibility of errors, like this:



A program which allows the user to make mistakes without disastrous consequences is called *robust*.

Finally, we can expand the "summary" cloud to give the three-line report at the end of the display. The expanded flow chart looks like this:



The corresponding program is written out below. Notice that the code for the main loop is a bit tangled. This is unavoidable since we have to force a two-dimensional flow chart into a single stream of instructions.

```
 10 H=0

 20 T=0

 30 PRINT "TYPE H FOR HEADS"

 40 PRINT "T FOR TAILS"

 50 PRINT "E FOR END"

 60 INPUT "NEXT THROW"; I$

 70 IF I$="E" THEN 160

 80 IF I$="H" THEN 120

 90 IF I$="T" THEN 140

100 PRINT "WRONG ITEM"

110 GOTO 60

120 H=H+1

130 GOTO 60

140 T=T+1

150 GOTO 60

160 PRINT "OUT OF"; H+T; "THROWS"

170 PRINT "THERE WERE"; H; "HEADS"

180 PRINT "AND"; T; "TAILS"

190 STOP
```

EXPERIMENT 14·2

(a) If a program has a great deal of input, the user may stop looking at the screen as he types. As we explained in Unit 13, it is a good idea to make the program react with sounds as well as displayed messages. You could, for instance, use a 'pip' for an item which is acceptable, and a rude noise for one which isn't.
Look at the heads and tails program. Every time the user types an H the machine obeys the commands at lines 120 and 130. We could insert a suitable noise by adding the commands:

```
123 VOL 7

125 SOUND 1,900,1
```

Load the HEADS program from the cassette tape (this saves you keying it in for yourself) and edit it so that it answers each input (right or wrong) with a suitable sound.

(b) At one time, clocks were liable to a curious form of tax, which was calculated as follows:
If the price of the clock was less than £12, the tax was one-third of the cost.
If the price was between £12 and £16, the tax was £4.

If the price was over £16, the tax was one-quarter of the cost of the clock.
Write a program which inputs a list of clock prices, ended by 0, and displays the total to be charged for each clock (including cost and tax).
Note that this program will have one or more PRINT commands inside the loop, and doesn't need a summary block. You will find a good flow chart indispensable.

(c) Write a program which inputs a stream of numbers ended by Ø, and displays the *largest*.
Hint: use a variable to record the *largest number so far*, and update it every time round the loop.

Experiment 14.2 Completed	

Now check your answers in Appendix B.

UNIT:15

This unit is about three important features of Commodore BASIC which are useful in games, quizzes and other programs where the machine and its user work closely together.

We'll begin by having a look at "REACTION", one of the programs you'll find on the cassette tape or diskette. A person's "reaction time" is a measure of how quickly they can respond to an unexpected event. A safe driver should have a fast reaction time, so that he or she can put the brakes on quickly when a child runs out into the road in front of the car. A good reaction time is also useful in most sports and many professions.

Most people, if they are paying attention, have reaction times between 0.2 and 0.3 of a second (twenty to thirty hundredths of a second). a time of less than 0.2 suggests someone who is quick on the uptake, whilst a reaction time of more than 0.3 is usually due to a few drinks too many!

EXPERIMENT 15·1

Load the REACTION program, and use it to measure your own reaction time. Run the program several times, and ignore the first two or three results, since they will not be typical. Keep trying the program until you are satisfied that you understand it thoroughly, and could confidently use it to measure the reaction time of a friend who had no knowledge of computing.

You may notice three aspects of the program which are not immediately obvious:

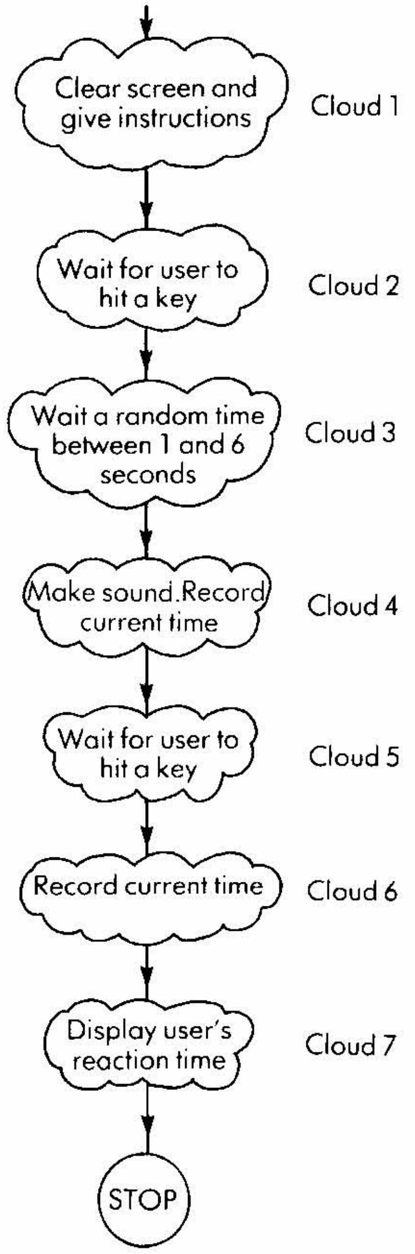
First, when the instructions say "any key", they really mean it. You will find that function keys like RETURN and INST DEL work just as well as letters or numbers, but C= SHIFT and CTRL have no effect.

The only ones you mustn't use are the long keys labelled f1 f4, f2 f5, f3 f6 and HELP f7, since they all produce several characters when pressed.
as letters or numbers.

Second, the time you must wait before hearing the tone is always different: it varies between 1 and 6 seconds in a way you cannot predict in advance.

Third, if you press a key *before* the tone starts, you get a message, "TOO SOON".

Now we'll examine the program in detail, and explain how it works. Let's start by examining the flow chart and BASIC program, which are shown below:


Clear screen and give instructions
Cloud 1
Wait for user to hit a key
Cloud 2
Wait a random time between 1 and 6 seconds
Cloud 3
Make sound. Record current time
Cloud 4
Wait for user to hit a key
Cloud 5
Record current time
Cloud 6
Display user's reaction time
Cloud 7
STOP

Cloud 1

```
10 REM REACTION TIME PROGRAM
20 PRINT" SHIFT and CLR HOME "
30 PRINT"TO MEASURE YOUR
   REACTION TIME HIT ANY"
40 PRINT"KEY, AND THEN WAIT FOR
   THE TONE."
50 PRINT"WHEN YOU HEAR IT,
   STRIKE ANY KEY AS"
60 PRINT"FAST AS YOU CAN. GOOD
   LUCK!"
```

Cloud 2

```
7Ø REM WAIT FOR ANY KEY
8Ø GET A$
9Ø IF A$="" THEN 8Ø
```

Cloud 3 (and see below)

```
1ØØ REM WAIT A RANDOM TIME
11Ø PRINT
12Ø PRINT"WAIT FOR IT!"
13Ø PRINT
14Ø Q=TI+INT(6Ø+3Ø1★RND(Ø))
15Ø GET A$
16Ø IFA$<>"" THEN 31Ø
17Ø IF TI<Q THEN 15Ø
```

Cloud 4

```
18Ø REM START TONE AND NOTE TIME
19Ø VOL 7
2ØØ SOUND1,95Ø,4
21Ø X=TI
```

Cloud 5

```
22Ø REM WAIT FOR ANY KEY
23Ø GET A$
24Ø IFA$="" THEN 23Ø
```

Cloud 6

```
25Ø REM GET RESULT
26Ø R=TI
```

Cloud 7

```
27Ø REM DISPLAY RESULT
28Ø PRINT"YOUR REACTION TIME IS"
29Ø PRINT(R-X)/6Ø; "SECONDS"
3ØØ STOP
```

Part of Cloud 3

```
31Ø PRINT"TOO SOON"
32Ø STOP
READY.
```

The program has been marked so that the commands which correspond to each cloud in the flow chart are clearly visible.

The first cloud (lines 10 to 60) consists entirely of PRINT commands and is quite straightforward.

The second cloud, lines 70 to 90, makes the program wait until the user types a key. The cloud uses a command with a new keyword:

```
GET A$
```

This command is in some ways like INPUT; it transfers information from the keyboard to the computer. However, there are some very important differences:

1. The keyword GET must be followed by exactly *one* variable name.
   For example

   GET Q

   GET X$ but

   GET PR$ GET Y$, Z$ ←*Not* BASIC

   (allowed) (forbidden)

2. The GET command doesn't wait for the user to do anything; it simply examines the keyboard at that instant and indicates which key has been typed since the last GET or INPUT command was obeyed. If a key has been struck, it is made into a one character or numeric value and put into the variable mentioned in the GET command. If no key has been newly struck, the variable is set to the *null* string or 0. This is a string with no characters, and is normally written as "" or 0 for numeric values.
   To illustrate this rule, imagine that we start off the computer on the following looped program, and watch what happens inside the machine:

   ```
   10 GET X$

   20 GOTO 10
   ```

   The computer will go round this loop about 50 times a second. As long as the user doesn't touch the keyboard, X$ will be set to the null string : " ".
   Now suppose the user presses down a key — say the one marked U. As soon as the GET command is obeyed (i.e. within a fiftieth of a second) X$ will be set to the string "U". However, this only happens once for each key depression; the next time round the loop X$ will again be set to " ", and this will continue until the U key is let go and another key (or possibly the same key) is pressed. The only exceptions to this rule are the so-called *repeating* keys like space.

3. The GET command doesn't treat certain control characters like INST DEL , RETURN , or cursor controls as special cases, but deals with them all in the same way, except for RUN STOP, which interrupts the program.

4. Any character which is detected by the GET command is *not* displayed on the screen.

With these points in mind, you can now begin to make some sense of line 80 and 90 in the REACTION program. Command 80 examines the keyboard and delivers a string in A$ which is null unless a key has been pressed. Command 90 tests A$, and makes the computer loop back to 80 until the user types any key, at which the program is allowed to drop through to line 100.

The point of this cloud is to hold the program up until the user shows he is ready to have his reaction time tested. Why do we use a loop with a GET, instead of a single command like

```
INPUT "READY"; A$   ?
```

There are two reasons. First, INPUT always expects a RETURN after the user's message. This implies a minimum of *two* characters to be typed.

Second, GET treats nearly all the characters in the same way, so there is much less chance of the program being spoiled if the user hits a control key instead of a letter or number.

Cloud number 3 makes the machine wait a random (that is, an unpredictable) time between the user's 'ready' signal and the tone. The waiting time must be variable, because if it were always the same, the user would soon learn how long to wait before the tone was due, and this would no longer be an 'unexpected' event.

The cloud uses *two* facilities which you haven't met before: the *random* function and the *internal timer*.

The random function is a way of making the machine produce an *unpredictable** number. Every time the machine works out the expression RND (0) it gets a different value somewhere between 0 and 1.

In most practical cases, we don't need a random *fraction* between 0 and 1, but a random *whole number* within limits which depends on the problem to be solved. For instance, if you make the machine imitate someone throwing a 6-sided die**, you expect a number between 1 and 6; or if you model a (European) roulette wheel, you need a number which is between 0 and 36.

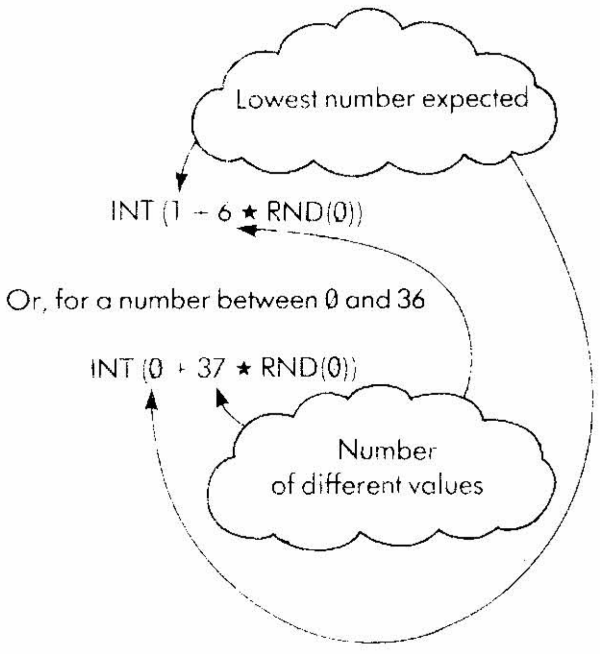
To get a whole number in any specified range, we use a slightly different expression:

INT(x + y ★ RND(0))

where x is the *lowest* number we need

y is the *number of different possibilities*

So, to get a number between 1 and 6, we would put



**The number isn't really unpredictable because everything which happens in a computer depends on what happened previously. However, each new 'random' number is derived from the previous one by a complicated process of squaring it and shuffling the digits of the result, and unless you know exactly how it is done you cannot tell what number is coming next.*

***That is: one of a pair of dice.*

An expression of this sort can be included in a loop, so that it is worked out many times. Type the following program, which imitates 120 throws of a die:

```
NEW

10 FOR J = 1 TO 120

20 S = INT(1+6★ RND(0))

30 PRINT S;

40 NEXT J

50 STOP
```

Run this program, and count the number of 1's, 2's . . . 6's which appear on the screen. Enter your results in the first row of the table below:

	1	2	3	4	5	6
No. of throws (1)						
No. of throws (2)						

Is the program a good imitation of a fair (or unbiased) die?

Now run the program again, and fill in the second row. Examine the results and note that they are different from the first run, just as you would expect with a real die.

The other important feature in cloud 3 is the internal timer, TI. We have already met the clock TI$, which keeps time in hours, minutes and seconds; but the special variable TI (which is not a string but a number) is intended to measure much shorter periods of time. TI is set to zero when the computer is started up, and from then on, no matter what else happens, it has 1 added to it every 660th of a second. This interval, a sixtieth of a second, is called one "jiffy". You can get the current value of the internal timer at any time in jiffies by using the name TI in an expression; but you can't alter the value in the way you can set TI$.

Give the command

PRINT TI

The machine will respond by displaying a fairly large number (60★60 or 3600 jiffies for every minute you've had the machine switched on). Now try the command again, and observe that the value has gone up by a few hundred or so. Finally, try to reset the value of TI and see what happens!

TI can be used to measure periods of time in two different, but related ways. In neither of them are we interested in the number of jiffies since the computer was switched on; instead, we use the fact that the duration of any length of time is given by the *difference* between TI at the end of it, and the value it had at the beginning. For instance, at the end of a period of 5 seconds, TI will be 5 ★ 60 or 300 *more* than it was at the beginning. This is true whether the machine has been switched on for 5 seconds or 5 hours.

In the first way of using the internal timer, we make the machine measure a period of time which is known in advance, and tell us when that time has elapsed. The method is simple. At the beginning of the period the program looks at TI and predicts what it should be at the end of the period; then it waits in a loop until TI reaches (or passes) that value. This is very like what you do in the kitchen, when you say, "These potatoes must boil for 25 minutes. Now it's ten past four, so I'll take them off at 4.35".

To illustrate the point, here is a general purpose timer program, which you could use in the kitchen, the laboratory, etc.

```
10 INPUT "HOW MANY MINUTES"; M

20 R=TI +M★3600

30 IF TI<R THEN 30

40 PRINT "TIME UP!"

50 STOP
```

If you try this program out, use a small number of minutes, otherwise you'll spend a lot of time waiting. As you study the program, remember that TI is moving up all the time, so that eventually, after M★3600 jiffies, the condition TI<R will be *false*.

In the second variant, we want the computer to tell us *how long* it takes from a given moment until some event occurs. We get the machine to record the value of TI at the beginning of the timing period. When the event comes, the difference between the value of TI now and the value recorded is a measure of the length of time, in jiffies. It is rather like the mountaineer who says, "I remember that I started climbing this hill at 5 o'clock. I have just got to the top at eleven o'clock, so it must have taken me six hours."

A program which measured time in this way would have commands something like this:

```
R=TI       (Stores value of TI at beginning of
            period)

                  and later

E=TI       Gets value of TI at end of period

D=E−R      Gets difference of times (in jiffies)

S=D/60     Gets time difference (in seconds)

PRINT"THAT TOOK"; S; "SECONDS"
```

Now we can piece together the commands in cloud number 3.

We want a waiting period of between 1 and 6 seconds. This is between 60 and 360 jiffies, to be decided by the *machine* in an unpredictable way. The appropriate expression is

```
INT(60+301★RND(0))
```

The waiting period is decided just before the period starts, so it *is* known in advance (although not to the user). We use the first method of timing, which involves predicting the value of TI at the end of the period. Command 140 makes this prediction and records the value in Q.

If this were all that were needed, the entire cloud could read:

```
140 Q = TI+INT(60+301★RND(0))

170 IF TI<Q THEN 170
```

As it is, we have to check that the user doesn't hit a key before the tone is sounded. Commands 150, 160, 310 and 320 are included simply to check for this possibility.

The rest of the program is now completely straightforward. The value of TI at the beginning of the reaction time period is stored in X, and commands 230 and 240 are used to wait for the user to hit a key.

Study the program carefully and make sure you understand every command.

Experiment 15.1 Completed	

EXPERIMENT 15·2

1. Write a 'stopwatch' program. When the user hits the 'B' key, the program starts timing. When he strikes 'S', it stops and displays the time taken, in seconds.
   Your program should display instructions, so that it can be used by anyone without further explanation.

   Hint: use GET and TI.

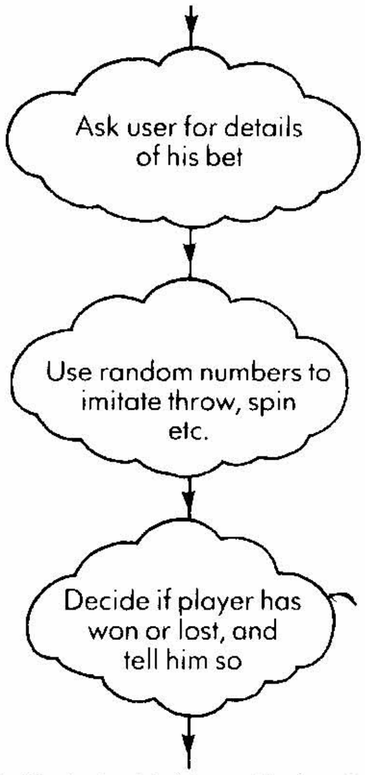
2. Write a program which imitates someone tossing a coin. Every time the user presses a key, the program displays either "HEADS" or "TAILS" at random.

Experiment 15.2 Completed	

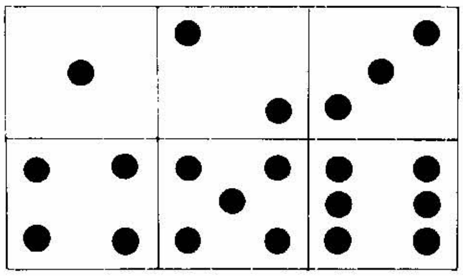
Now check your answers in Appendix B.

EXPERIMENT 15·3

Random numbers are useful in programming games of chance, such as dice, fruit machines, and so on. All these programs follow the same basic pattern, which for one 'throw' or 'spin' is like this


Ask user for details of his bet
Use random numbers to imitate throw, spin etc.
Decide if player has won or lost, and tell him so

Let's illustrate this idea with the old game of crown and anchor*. This is played with three dice and a board divided into six squares*:



Crown and anchor dice usually have different symbols, but this doesn't affect the principle of the game.

The player puts his bet on any one of the squares. For instance he might back ⚁ with £5. Then the banker throws all three dice. If one of them shows ⚁, the player gets back double his stake money: if two of the dice come up with ⚁, the player gets triple the original stake, and if ⚁ shows on all three dice, the player is rewarded with four times his stake. All these rewards include the original stake. On the other hand, if no ⚁ comes up, the player loses his stake.

The program for playing one throw of crown and anchor is given below. Using the glossary you should have no trouble in following it:

S: Player's stake

N: Number backed by player

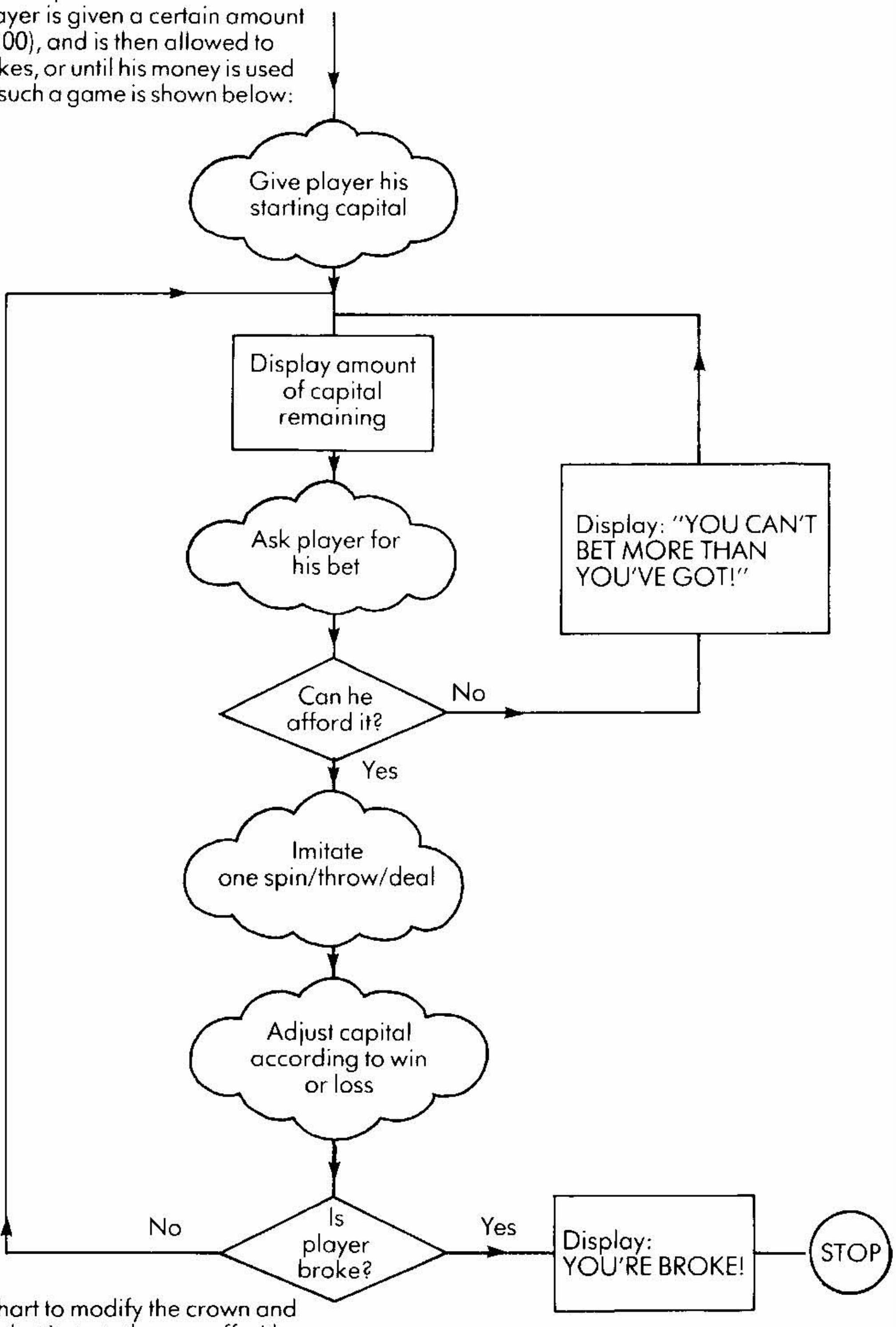
D1 }
D2 } Results of throwing 3 dice
D3 }

C: Number of dice showing N, the player's number.

```
 10 INPUT "STAKE"; S
 20 INPUT "NUMBER BACKED (1-6)"; N
 30 D1 = INT(1+6*RND(0))                  }
 40 D2 = INT(1+6*RND(0))                  } Throw 3 dice
 50 D3 = INT(1+6*RND(0))                  }
 60 C = 0                                 }
 70 IF D1<>N THEN 90                      }
 80 C = C+1                               } Count number of dice
 90 IF D2<>N THEN 110                     } showing number backed
100 C = C+1                               } by player
110 IF D3<>N THEN 130                     }
120 C= C+1                                }
130 PRINT "DICE THROWN:"; D1; D2; D3      }
140 IF C<>0 THEN 170                      }
150 PRINT "YOU LOSE"                      } Display results
160 GOTO 180                              }
170 PRINT "YOU RECEIVE"; S*(C+1);"POUNDS" }
180 STOP
```

Very few gamblers stop short at a single throw. Usually people start with a certain amount of capital and keep playing until they are broke or — very rarely — the banker runs out of money.

Gambling programs on the computer are better if they imitate complete sessions of this type. Initially the player is given a certain amount of "money" (like £100), and is then allowed to play as long as he likes, or until his money is used up. A flow chart for such a game is shown below:


Give player his starting capital
Display amount of capital remaining
Ask player for his bet
Can he afford it?
No
Display: "YOU CAN'T BET MORE THAN YOU'VE GOT!"
Yes
Imitate one spin/throw/deal
Adjust capital according to win or loss
Is player broke?
No
Yes
Display: YOU'RE BROKE!
STOP

Use this flow chart to modify the crown and anchor program so that it starts the user off with a capital of £100, and lets him play as long as he likes. When your program is complete, run it several times, and decide for yourself whether you would rather be a player or a banker!

Experiment 15.3 Completed	

EXPERIMENT 15·4

Write a program to imitate any other game of chance you know: craps, pontoon, etc. Embellish your program with pictures of dice or of cards, suitable sounds, and so on.

Experiment 15.4 Completed	☐

Your tape or disk contains a version of craps for you to try out. A complete listing of the program, called CRAPS, is given in Appendix B.

AFTERWORD

AFTERWORD

Congratulations on reaching the end of the course! By now you have gained a good knowledge of the principles of programming, and you'll be able to design and write programs for a wide range of interesting problems and computer applications. I hope that you've also cultivated the habit of careful, thoughtful design, of keeping and filing flow charts, glossaries and notes for your programs. It is this quality of planning and self-organisation that sets apart the really competent programmer from the others.

At this stage, you have reached a half-way point in your study of BASIC. There are many important problems which need parts of the language you haven't yet covered. For instance, you may want to program moving pictures on the screen, or to sort people's names into alphabetical order, or to store them on a cassette tape or diskette. These topics, and many others are fully explained in the second book of this series, entitled

INTRODUCTION TO BASIC (Part II)

This book is in the same style as the one you have just finished, and will complete your knowledge of the BASIC language.

Programming — as we said in the introduction — is a very broad subject. Now that you have made a start, you should broaden your knowledge in three ways:

(a) Read as widely as you can. Most of the popular computer magazines particularly Commodore Computing are worth looking at. Books on programming are also worth reading, even if they don't specifically refer to the computer.

(b) Join a local computer club. There are Commodore user groups being set up all over the country and details are given in the magazine Commodore Computing.

(c) Work at your programming. Practice constantly, and aim for perfection. Design your programs so that they are robust, and usable by anyone without special instruction.

Write them so that you can be proud, not ashamed, to display the inner workings to another computer expert.

One last point. You have found a fascinating hobby, and perhaps a life-long profession. Remember that with the advantages of knowing about computers, there also comes a responsibility to see that they are used humanely and wisely. No one wants a computer-controlled society with little work and no freedom, and it is now up to *you* — among others — to avoid it.

APPENDICES

APPENDIX A

Your computer is capable of large-scale mathematical calculations; as a matter of historical interest it can do arithmetic considerably faster than most large-scale computers installed before 1960!

This appendix outlines some of the mathematical facilities of the computer. You only need to read the appendix and understand the material in it if you plan to use the computer for calculations in Mathematics, Science or Engineering. Some of the features described are quite simple, and can easily be grasped by anyone who remembers the elementary arithmetic they learned at school. Other features need some more background knowledge, such as that covered by an A-level course in Mathematics. You need only go as far as your knowledge and confidence will take you, but you are expected to have read all the units in the body of the course.

1. Expressions

The expressions first mentioned in Unit 4 are very simple examples of a.more general facility. Thus in the commands

A = $\underline{34}$

B = $\underline{B+1}$

C = $\underline{((X + Y) - 34.7/(Q-3))\star(Z-3)\uparrow 2}$

the underlined portions are all expressions which the computer works out on your behalf.

Expressions are built up of three types of element:

Values: numerical variables or numbers such as

B, X, Y, 34, 34.7

Operators: the signs + − ★ / and ↑

(↑ means "raised to the power")

Brackets: (and)

Expressions in BASIC are written in the same way as in ordinary algebra, and have the same meaning. There are four minor differences:

- BASIC expressions are in capitals instead of small letters.
- Exponentiation ("raising the power") must be shown with the ↑ sign, because the computer screen doesn't let you write small numbers above the line. Instead of "3^2", you would put "3↑2".
- Multiplication must *always* be shown using the ★ sign. In BASIC, you would write "3★A", not "3A" as in conventional algebra. This rule can be a source of mistakes which are hard to find. If you put BA where you mean B★A, the machine will assume that you are talking about a new variable *called* BA. It won't report a syntax error, but it will produce the wrong answer!
- Division is written A/B, not $\frac{A}{B}$. If either the numerator or the denominator of the fraction is a complicated expression, you must delimit it with brackets. The correct way of writing $\frac{3+5}{7+8}$ in BASIC is (3+5)/(7+8). If you leave out the brackets and put 3+5/7+8 the rules of precedence (which are given in the next paragraph) will make the machine treat this expression as $3+\frac{5}{7}+8$.

When the computer works out an expression, it takes the ↑ signs first, then the multiplications and divisions, and lastly the additions and subtractions, working from left to right in each case. Anything in brackets is worked out first. These are called the *rules of precedence*, and they give the *same* results as ordinary school algebra.

The value of numbers in expressions do not have to be integers (i.e. whole numbers) but can be decimals. The computer works to an accuracy of about 8 decimal digits, which means that many fractions (such as 1/3 or 1/7) can't be represented exactly. You can expect small 'rounding' errors in some arithmetic commands, so that a result which you expected to be exactly 7 may come out as "6.99999998".

To test your understanding of expressions, work through the following examples, and predict what the computer will display in each case. Assume that X = 3 and P = 7.

COMMAND	*PREDICTED RESULT*	*ACTUAL RESULT*
PRINT 3 + 12 — 6 — 4		
PRINT 4 + 3 ★ 2		
PRINT X + P — 3		
PRINT 5 + 12 / 6 — 3		
PRINT 11 / 5 — 7 / 4		
PRINT 4 ↑ 2 — 2 ↑ 4		
PRINT 3 + 2 ↑ 3 — 3 ↑ 2		
PRINT 2 ↑ X — P		
PRINT 3 + 12 — (6 — 4)		
PRINT 5 + 12 / (6 — 3)		
PRINT (P + X) ↑ (1 — X)		
PRINT 4 ↑ 2 — 3 ↑ 0		
PRINT (P ↑ 2 — X ↑ 2) / 3		

Now check your results on the computer. Remember to set the values of P and X before you start.

In BASIC, expressions are most commonly used in PRINT and LET commands. Here is a simple program which inputs two numbers U and V, and displays a value F calculated according to the 'lens' formula:

$$\frac{1}{F} = \frac{1}{V} + \frac{1}{U}, \text{ or } F = \frac{1}{\frac{1}{V} + \frac{1}{U}}$$

```
10 INPUT "V"; V

20 INPUT "U"; U

30 PRINT "F="; 1/(1/V+1/U)

40 STOP
```

Example 1

Write a program which reads two values V and R, and which displays the value of the formula $A = \frac{V^2}{R}$

Example 2

Write a program which displays the values of the formula $y = \frac{1}{1+x^2}$ for values of x between 0 and 2, going up in steps of 0.2

(Hint: use a FOR loop like this:

```
FOR X = 0 TO 2 STEP 0.2

. . . . .

NEXT X      )
```

(The actual answers are given at the back of Appendix B.)

2 Standard Functions

Like most calculators, the computer has a set of 'scientific' functions. A useful one is the square root. This is abbreviated to SQR, and can be included in expressions like this:

```
        PRINT SQR(5)

     or PRINT SQR(B↑2+C↑2)
```

The quantity in brackets is called the *argument* of the function. In the case of SQR the argument must be zero or positive.

Here is a program which displays the square roots of all numbers between 100 and 115.

```
10 PRINT "N"; "SQR(N)"

20 FOR N=100 TO 115

30 PRINT N; SQR(N)

40 NEXT N

50 STOP
```

Example 3

If the lengths of three sides of a triangle are a, b and c, the area a of the triangle is given by the formula $a = \sqrt{s(s-a)(s-b)(s-c)}$ where s is the semi-perimeter, $(a+b+c)/2$.

Write a program which inputs three numbers. If they can be the sides of a real triangle the program displays the area of the triangle; otherwise (e.g. if the numbers are 1, 1, 10) the program displays an appropriate message.

(Hint: if the lines don't form a triangle the value of $s(s-a)(s-b)(s-c)$ is negative!)

Some of the more important mathematical functions are given below. Read through them, but do not feel obliged to learn them by heart — you can always refer back to the list later.

SIN(X)
COS(X)
TAN(X)
	Trigonometrical functions. The arguments must be in *radians*. (1 degree = $\pi/180$ radians)

ATN(X)	The arc-tangent of X. The result is in radians, between $-\pi/2$ and $\pi/2$.
LOG(X)	The natural logarithm of X (Log to the base e). X must be positive
EXP(X)	Equivalent to e^x
ABS(X)	The modulus of X (X if $X > 0$; otherwise —X)
INT(X)	The largest whole number equal to or less than X. Note that: INT (3.5) = 3 INT (−3.5) = −4

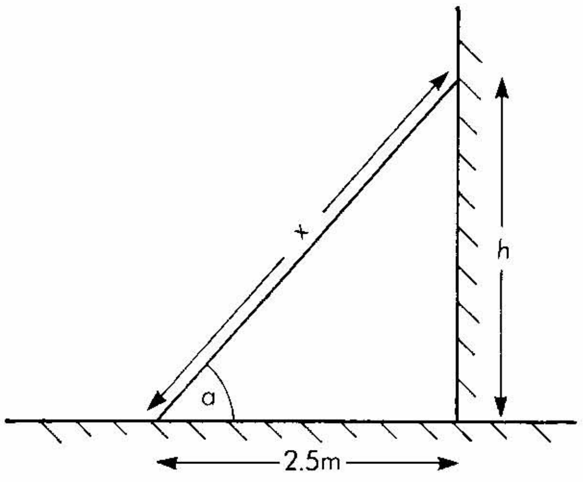
Now check your answer in Appendix B.

You can also use the keyboard symbol π instead of the number 3.14159265 . . . . . .

Here is an example to show the use of some of these functions.

A ladder can have its length changed from 4 metres to 5 metres in steps of 20 cms. It is placed with its base 2.5 metres from a vertical wall, and its top against the wall. Write a program to display the angle of the ladder with the horizontal for each of its 6 possible lengths.

First we do the mathematics, using a diagram. We use x to indicate the length of the ladder, and h to be the height of the top of the ladder, and a to be the angle with the horizontal.



$h = \sqrt{x^2 - (2.5)^2}$ (by Pythagoras)

$a = \arctan(h/2.5)$ (in radians)

or $a = (180/\pi) \star \arctan(h/2.5)$ in degrees.

Next we write the program, which has a simple looped structure:

```
10 PRINT "   LENGTH","   ANGLE"

20 FOR X=4 TO 5 STEP 0.2

30 H = SQR(X↑2—2.5↑2)

40 A = (180/π)★ATN(H/2.5)

50 PRINT X, A

60 NEXT X

70 STOP
```

One of the most useful functions is INT. We can use it to tell whether one number divides another exactly. If X is an exact multiple of Y, then the condition

X/Y = INT(X/Y)

will be true; otherwise it won't.

A number is a *prime* if it has no divisors except itself and 1. The following program calculates and displays prime numbers from 3 up to any value set by the user:

```
10 INPUT "HIGHEST VALUE"; H

20 FOR N=3 TO H

30 FOR J=2 TO N—1

40 IF N/J = INT(N/J) THEN 70

50 NEXT J

60 PRINT N;

70 NEXT N

80 STOP
```

Example 4

Study the prime number program (by tracing if necessary) and work out how it works. Run it, and time it for some value of H (say 500).

This method of calculating primes is actually very slow. Design and incorporate some improvements to make it run faster.

Hints: (a) No even numbers apart from 2 can be primes.

(b) In testing for possible factors, it is enough to go as far as the square root of the number.

Now check your answer in Appendix B.

APPENDIX B

UNIT:7

Experiment 7.1:

a) T,T,T,T,F,F,F
b) F,F,T,T

Experiment 7.3:

(1)
```
10 P$= "★"
20 PRINT P$
30 P$= P$+"★"
40 IF P$<> "★★★★★★★★★★★"
   THEN 20
50 STOP
```

(2)
```
10 PRINT "POUNDS", "DOLLARS"
20 PRINT
30 P= 10
40 PRINT P, 1.77★P
50 P=P+2
60 IF P< 32 THEN 40
70 STOP
```

(3)
```
10 PRINT "CENT", "FAHR"
20 PRINT
30 C= 15
40 F= 1.8★C+32
50 PRINT C, F
60 C=C+1
70 IF C < 31 THEN 40
80 STOP
```

Experiment 7.2:

Control variable	*Starting value*	*Final value*	*Increment*	*No. of times round loop*
X$	"A"	"ABBB"	"B"	4
P	0	10	+1	11
Y$	"Z"	"ZXYXY"	"XY"	3
R	5	14	3	4
C	27	7	—5	5

UNIT:8

Experiment 8.1:

a) PROGRAM COUNTER ~~1Ø 2Ø 3Ø 4Ø 5Ø~~ 6Ø

VARIABLES X: 5 Y: 7 Z: 12 W: 2

```
5 7 12 2            1Ø X=5
BREAK IN 6Ø         2Ø Y=7
READY               3Ø Z=X+Y
                    4Ø W=Y—X
                    5Ø PRINT X;Y; Z;W
                    6Ø STOP
```

PROGRAM COUNTER ~~1Ø 2Ø 3Ø 4Ø 5Ø 3Ø 4Ø 5Ø~~ 6Ø

VARIABLES Q: ~~1 2~~ 3

```
SHE LOVES ME        1Ø Q=1
SHE LOVES ME NOT    2Ø PRINT "SHE LOVES ME"
SHE LOVES ME NOT    3Ø PRINT "SHE LOVES ME NOT"
BREAK IN 6Ø         4Ø Q=Q+1
READY               5Ø IF Q<3 THEN 3Ø
                    6Ø STOP
```

Experiment 8.2:

a) Line 5Ø should be: 5Ø IF G<11 THEN 3Ø

b) Line 3Ø should be: 3Ø A$=A$+"★"

Experiment 8.3:

c) Line 2Ø: PRINT (not PR1NT)
No RETURN after line 4Ø
Line 6Ø: IF X<13 THEN 4Ø (NOT X>13)
Line 7Ø: STOP (not STØP)

UNIT:9

Experiment 9.2:

```
10 COLOR 4,5
20 COLOR 0,8
30 PRINT " SHIFT and CLR HOME ";
40 PRINT " CLR HOME ↓ ← 9 times → ↓
   → ← 6 times → → ";
50 PRINT TI$
60 GOTO 40
```

Experiment 9.3:

```
 5 REM FLAG OF ICELAND
10 COLOR 4,1
20 COLOR 0,15
20 PRINT " SHIFT and CLR HOME ";
30 J=1
40 PRINT " CTRL and RVS ON
   → ← 6 times → → CTRL and
   WHT ← 1 space → CTRL and RED
   ← 2 spaces → CTRL and WHT 1 space "
50 J=J+1
60 IF J <10 THEN 40
70 PRINT " CTRL and RVS ON CTRL and
   WHT ← 7 spaces → CTRL and RED
   ← 2 spaces → CTRL and WHT
   ← 13 spaces →";
80 J=1
90 PRINT " CTRL and RVS ON CTRL and
   RED ← 22 spaces →";
100J=J+1
110 IF J < 4 THEN 90
120 PRINT " CTRL and RVS ON CTRL and
   WHT ← 7 spaces → CTRL and RED
   ← 2 spaces → CTRL and WHT
   ← 13 spaces →";
130 J=1
140 PRINT " CTRL and RVS ON
   → ← 6 times → →
   CTRL and WHT ← 1 space →
   CTRL and RED ← 2 spaces →
   CTRL and WHT ← 1 space →"
150 J=J+1
160 IF J<9 THEN 140
170 PRINT " CTRL and RVS ON
   → ← 6 times → →
   CTRL and WHT ← 1 space →
   CTRL and RED ← 2 spaces →
   CTRL and WHT ← 1 space →";
180 GOTO 180
```

UNIT:10

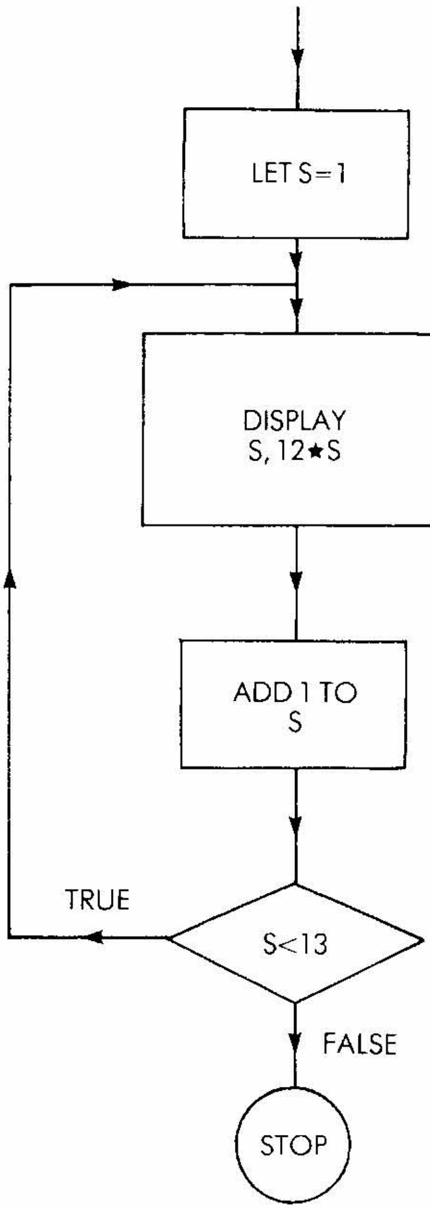
Experiment 10.2:

a)
```
10 PRINT "TABLE PROGRAM"
20 INPUT "TIMES"; N
30 X=1
40 PRINT X; "TIMES"; N; "IS"; N*X
50 X=X+1
60 IF X<13 THEN 40
70 STOP
```

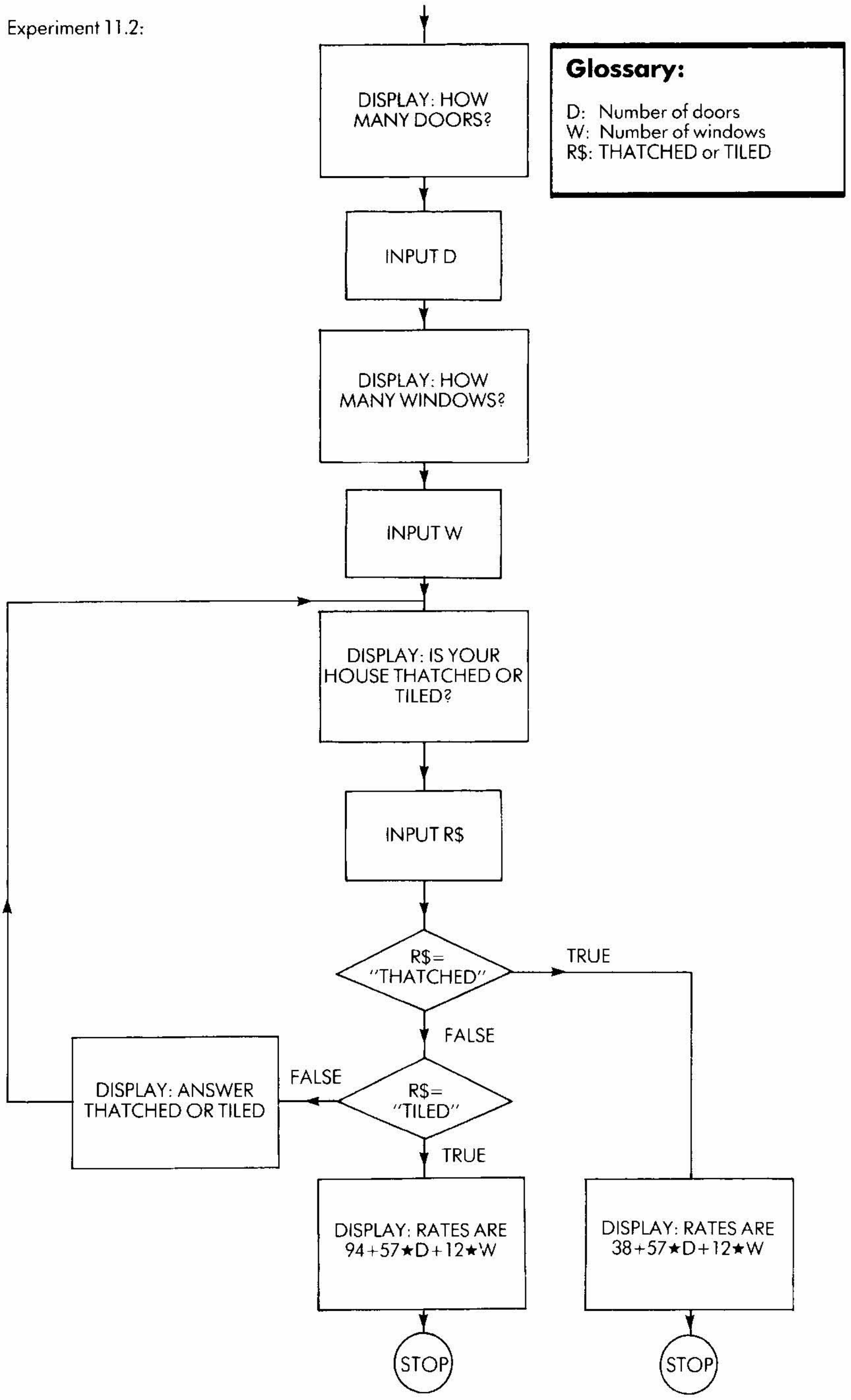
b)
```
10 PRINT "WHAT IS YOUR"
20 INPUT "SURNAME"; S$
30 PRINT "WHAT IS YOUR WIFE'S"
40 INPUT "CHRISTIAN NAME"; C$
50 PRINT "HER FULL NAME IS"
60 PRINT C$+"    "+S$
70 STOP
```

UNIT:11

Experiment 11.1:



Experiment 11.2:



DISPLAY: HOW MANY DOORS?
INPUT D
DISPLAY: HOW MANY WINDOWS?
INPUT W
DISPLAY: IS YOUR HOUSE THATCHED OR TILED?
INPUT R$
R$= "THATCHED"
TRUE
FALSE
R$= "TILED"
FALSE
TRUE
DISPLAY: ANSWER THATCHED OR TILED
DISPLAY: RATES ARE 94+57★D+12★W
DISPLAY: RATES ARE 38+57★D+12★W
STOP
STOP

Glossary:

D:	Number of doors
W:	Number of windows
R$:	THATCHED or TILED

```
 10 REM RURITANIAN RATES
 20 PRINT "RATING PROGRAM"
 30 INPUT "HOW MANY DOORS"; D
 40 INPUT "HOW MANY WINDOWS"; W
 50 PRINT "IS YOUR HOUSE"
 60 PRINT "THATCHED OR"
 70 INPUT "TILED"; R$
 80 IF R$= "THATCHED" THEN 140
 90 IF R$= "TILED" THEN 160
100 PRINT "PLEASE ANSWER"
110 PRINT "THATCHED OR"
120 PRINT "TILED"
130 GOTO 50
140 PRINT "RATES ARE"; 38+57*D+12*W
150 STOP
160 PRINT "RATES ARE"; 94+57*D+12*W
170 STOP
```

Correct answers to three sample problems are:

a) 95 b) 155 c) 364

UNIT:12

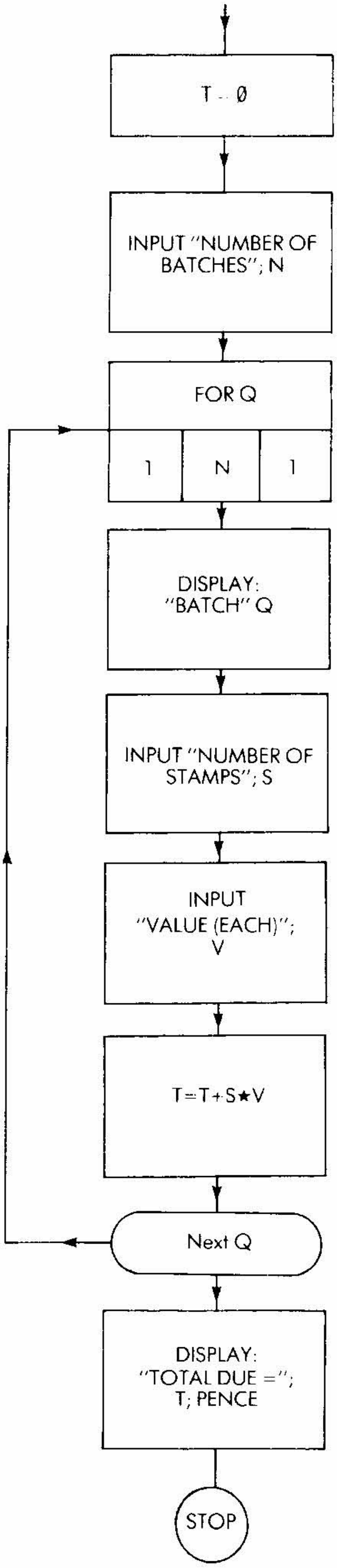
Experiment 12.1:

```
10 RS=0                              } either order
20 INPUT "NUMBER OF INNINGS"; J      } is correct
30 FOR Q = 1 TO J
40 INPUT "SCORE"; S
50 RS=RS+S
60 NEXT Q
70 PRINT "AVERAGE-"; RS/J
80 STOP
```

Experiment 12.2:

Glossary

N: Number of batches
S: Number of stamps in a batch
V: Value of each stamp in a batch
T: Running total due
Q: To count batch number

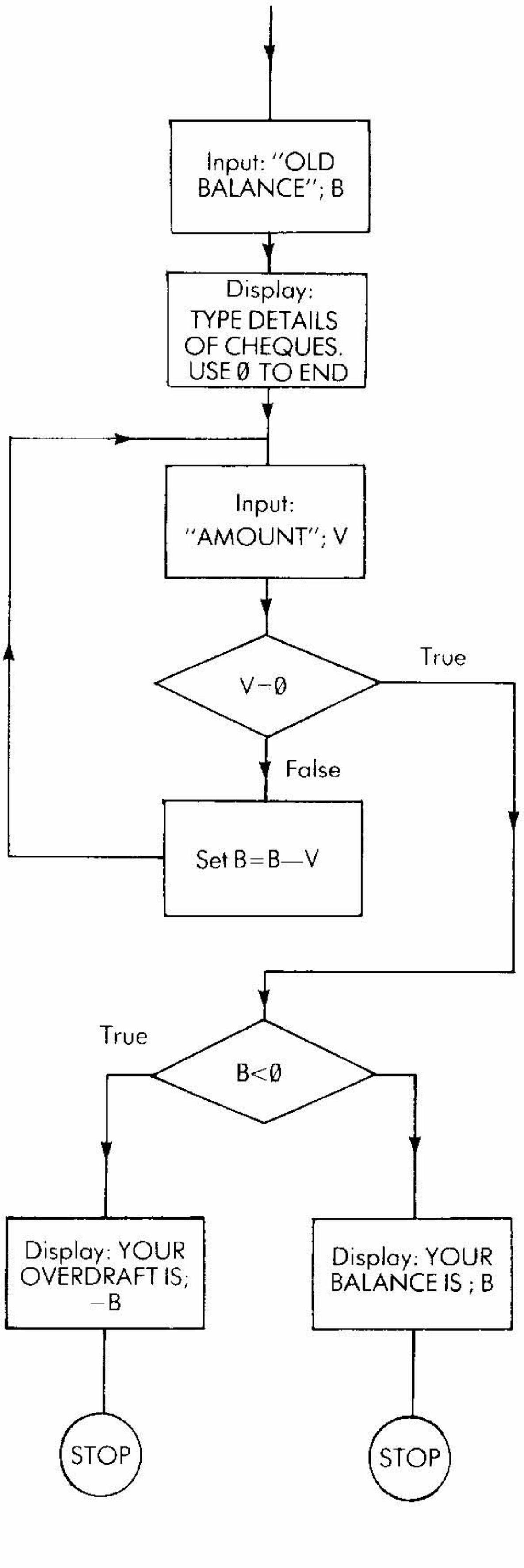


T = 0
INPUT "NUMBER OF BATCHES"; N
FOR Q
1
N
1
DISPLAY: "BATCH" Q
INPUT "NUMBER OF STAMPS"; S
INPUT "VALUE (EACH)"; V
T=T+S*V
Next Q
DISPLAY: "TOTAL DUE ="; T; PENCE
STOP

```
 10 T=Ø
 20 INPUT "NUMBER OF BATCHES"; N
 30 FOR Q= 1 TO N
 40 PRINT "BATCH"; Q
 50 INPUT "NUMBER OF STAMPS"; S
 60 INPUT "VALUE (EACH)"; V
 70 T=T+S★V
 80 NEXT Q
 90 PRINT "TOTAL DUE="; T; "PENCE"
100 STOP
```

UNIT:14

Experiment 14.1:


Input: "OLD BALANCE"; B
Display: TYPE DETAILS OF CHEQUES. USE Ø TO END
Input: "AMOUNT"; V
V=Ø
True
False
Set B=B—V
B<Ø
True
Display: YOUR OVERDRAFT IS; −B
Display: YOUR BALANCE IS ; B
STOP
STOP

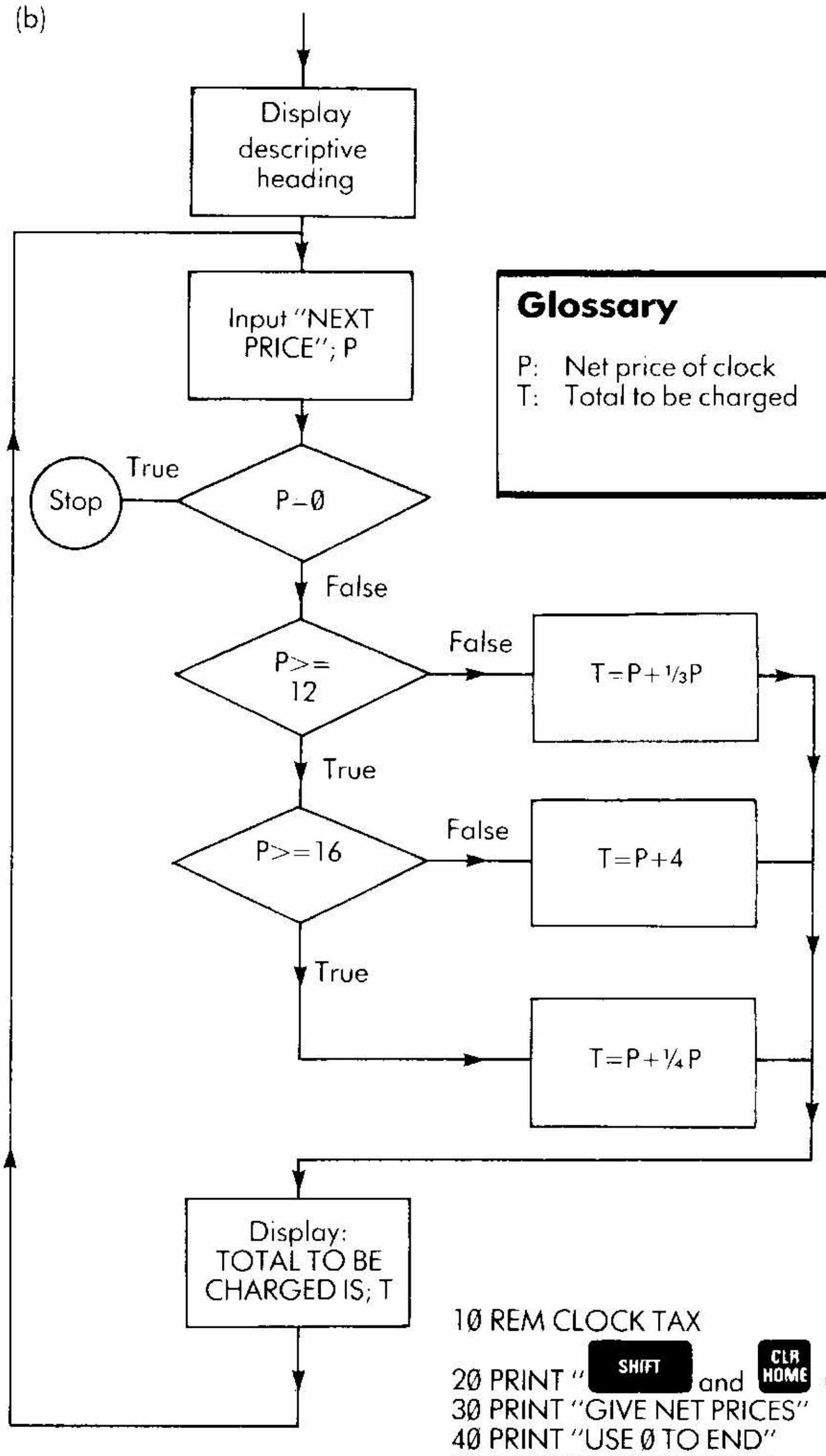
Glossary

B: Current balance
V: Each new transaction

```
 10 REM BANKING PROGRAM
 20 INPUT "OLD BALANCE"; B
 30 PRINT "TYPE DETAILS OF"
 40 PRINT "CHEQUES. USE 0 TO END"
 50 INPUT "AMOUNT"; V
 60 IF V=0 THEN 90
 70 B=B—V
 80 GOTO 50
 90 IF B<0 THEN 120
100 PRINT "YOUR BALANCE IS £"; B
110 STOP
120 PRINT "YOUR OVERDRAFT"
130 PRINT "IS £"; -B
140 STOP
```

Experiment 14.2:

(b)



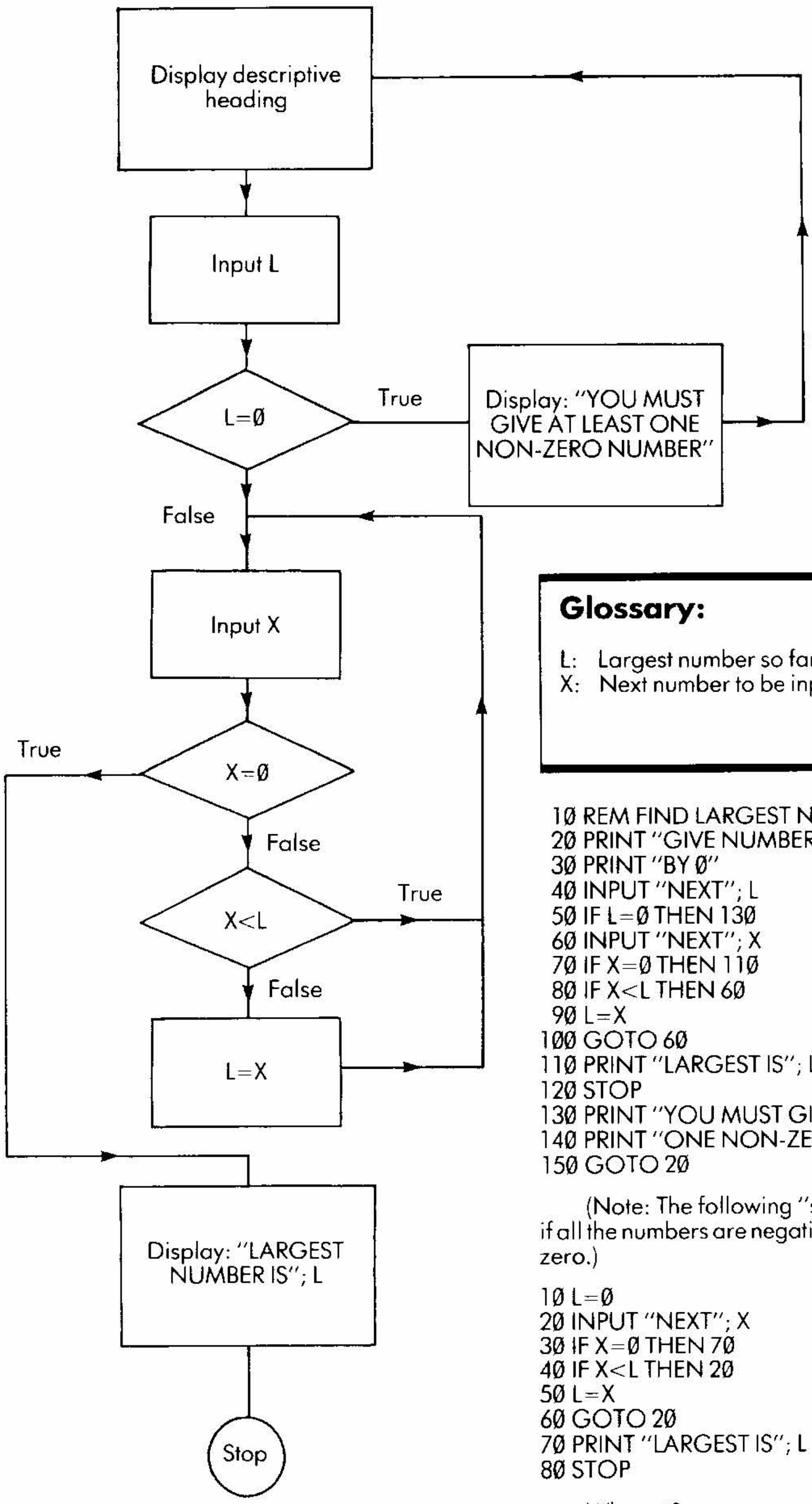
Glossary

P:	Net price of clock
T:	Total to be charged

```
 10 REM CLOCK TAX
 20 PRINT " SHIFT and CLR HOME  CLOCK TAX PROGRAM"
 30 PRINT "GIVE NET PRICES"
 40 PRINT "USE Ø TO END"
 50 INPUT "NEXT PRICE"; P
 60 IF P=Ø THEN 18Ø
 70 IF P>=12 THEN 1ØØ
 80 T= P+ (1/3)★P
 90 GOTO 14Ø
1ØØ IF P >=16 THEN 13Ø
11Ø T=P+4
12Ø GOTO 14Ø
13Ø T=P+ (1/4)★P
14Ø PRINT "TOTAL TO BE CHARGED"
15Ø PRINT "IS "; T
16Ø PRINT
17Ø GOTO 5Ø
18Ø STOP
```

Experiment 14.2:

c)


Display descriptive heading
Input L
L=Ø
True
Display: "YOU MUST GIVE AT LEAST ONE NON-ZERO NUMBER"
False
Input X
True
X=Ø
False
X<L
True
False
L=X
Display: "LARGEST NUMBER IS"; L
Stop

Glossary:

L:	Largest number so far
X:	Next number to be input

```
 10 REM FIND LARGEST NUMBER
 20 PRINT "GIVE NUMBERS ENDED"
 30 PRINT "BY Ø"
 40 INPUT "NEXT"; L
 50 IF L=Ø THEN 13Ø
 60 INPUT "NEXT"; X
 70 IF X=Ø THEN 11Ø
 80 IF X<L THEN 6Ø
 90 L=X
100 GOTO 6Ø
110 PRINT "LARGEST IS"; L
120 STOP
130 PRINT "YOU MUST GIVE AT LEAST"
140 PRINT "ONE NON-ZERO NUMBER"
150 GOTO 2Ø
```

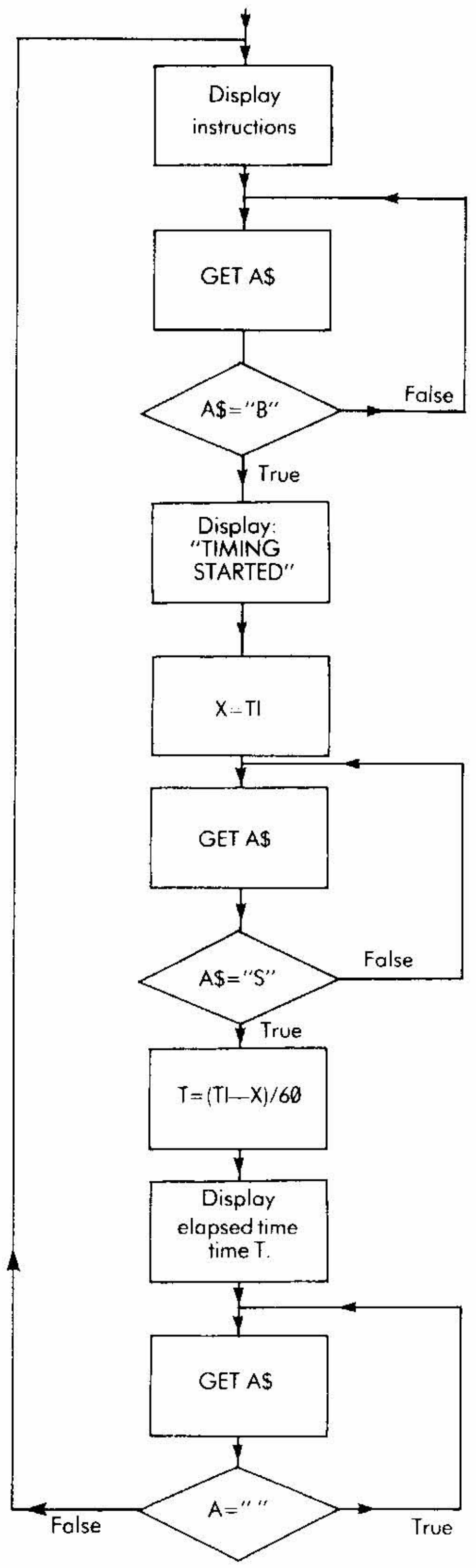
(Note: The following "solution" won't work if all the numbers are negative — that is, less than zero.)

```
1Ø L=Ø
2Ø INPUT "NEXT"; X
3Ø IF X=Ø THEN 7Ø
4Ø IF X<L THEN 2Ø
5Ø L=X
6Ø GOTO 2Ø
7Ø PRINT "LARGEST IS"; L
8Ø STOP
```

Why not?

UNIT:15

Experiment 15.2 (1):

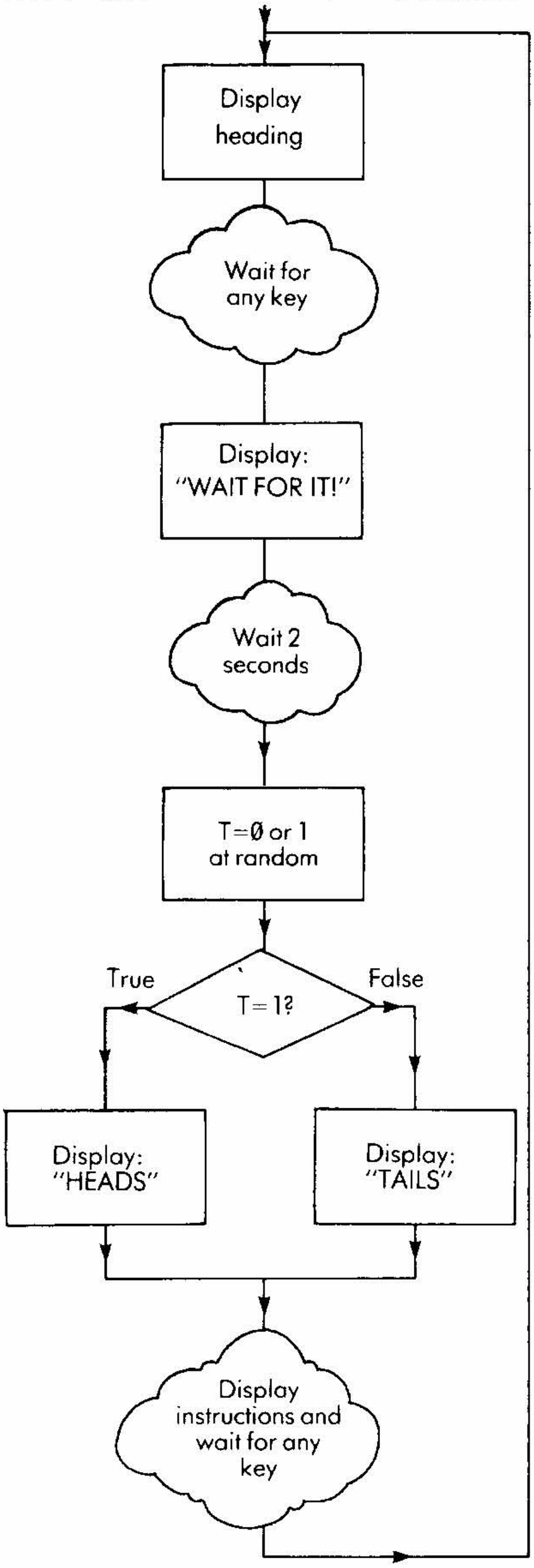


Glossary:

A$:	Keyboard character
X:	Internal time at start of interval (jiffies)
T:	Elapsed time (seconds)

```
  5 REM STOPWATCH
 10 PRINT " SHIFT and CLR HOME "
 20 PRINT "STOPWATCH PROGRAM"
 30 PRINT
 40 PRINT "TO START THE STOPWATCH"
 50 PRINT "HIT THE B KEY"
 60 PRINT "TO STOP IT, HIT S"
 70 GET A$
 80 IF A$ <>"B" THEN 70
 90 X=TI
 95 PRINT "TIMING STARTED"
100 GET A$
110 IF A$ <> "S" THEN 100
120 T= (TI—X)/60
130 PRINT "ELAPSED TIME WAS"
140 PRINT T; "SECONDS"
150 PRINT
160 PRINT "NOW HIT ANY OTHER KEY"
170 PRINT "FOR ANOTHER TIMING"
180 GET A$
190 IF A$= " " THEN 180
200 GOTO 10
```

Experiment 15.2 (2):



Glossary:

S$	Keyboard character
M:	Used in loop to wait 2 seconds
T:	Ø (for Tails) or 1 (for Heads)

```
 1Ø REM COIN TOSSING
 2Ø PRINT " SHIFT and CLR HOME "
 3Ø PRINT "HIT ANY KEY TO TOSS"
 4Ø PRINT "YOUR COIN"
 5Ø GET S$
 6Ø IF S$= "" THEN 5Ø
 7Ø PRINT "WAIT FOR IT!"
 8Ø PRINT
 9Ø FOR M=1 TO 2ØØØ
1ØØ NEXT M
11Ø T=INT (Ø+2★RND(Ø))
12Ø IF T=1 THEN 15Ø
13Ø PRINT "TAILS"
14Ø GOTO 16Ø
15Ø PRINT "HEADS"
16Ø PRINT
17Ø PRINT "HIT ANY KEY FOR"
18Ø PRINT "NEXT GO"
19Ø GET S$
2ØØ IF S$= "" THEN 19Ø
21Ø GOTO 2Ø
```

Experiment 15.4:

```
  5 REM CRAPS
 10 REM7
 30 PRINT" SHIFT and CLR HOME  CTRL
    and 1 BLK "
 40 PRINT"THE GAME OF CRAPS IS PLAYED
    WITH"
 50 PRINT"TWO DICE. FIRST YOU BET AND
    THEN YOU
 60 PRINT"THROW. IF YOU GET A SCORE OF
    7 OR 11,
 70 PRINT"YOU WIN. IF YOU THROW 2,3 OR
    12, YOU
 80 PRINT"LOSE. IF YOU THROW ANY
    OTHER NUMBER,
 90 PRINT"YOU DON'T WIN OR LOSE
    STRAIGHT AWAY:
100 PRINT"YOU KEEP ON THROWING UNTIL
    YOU
110 PRINT"THROW THE SAME AS YOU DID
    FIRST
120 PRINT"TIME (AND WIN)
130 PRINT"OR
140 PRINT"THROW A 7 (AND LOSE)
150 PRINT
160 PRINT"HIT ANY KEY TO CONTINUE
240 GET A$
250 IF A$=""THEN 240
255 REM SET A$,B$,C$ TO LINES OF DICE
    PICTURE
260 A$ = "← 4 spaces → ╭───╮
    ← 2 spaces → ╭───╮ "
270 B$ = "← 4 spaces → | ← 3 spaces →
    | ← 2 spaces → | ← 3 spaces → | "
280 C$ = "← 4 spaces → ╰───╯
    ← 2 spaces → ╰───╯ "
285 REM GET STARTING CAPITAL
290 PRINT " SHIFT and CLR HOME "
300 INPUT" STARTING CAPITAL";C
305 REM NOW START NEXT BET
310 PRINT" HIT ANY KEY FOR NEXT BET"
330 GET R$
340 IF R$=""THEN 330
350 PRINT" YOUR CAPITAL NOW IS";C
370 INPUT" HOW MUCH DO YOU BET";W
380 IF W > 0 THEN 390
385 PRINT" DON'T BE SO SILLY"
387 GOTO 310
390 IF W<=C THEN 420
400 PRINT" YOU CAN'T AFFORD IT"
410 GOTO 310
415 REM ORGANISE FIRST THROW
420 PRINT " SHIFT and CLR HOME ↓ ↓ ↓
    FIRST THROW (BET = "; W; ")"
430 PRINT " CLR HOME ↓ ← 7 times → ↓ "; A$
440 FOR J=1TO5
450 PRINTB$
460 NEXTJ
470 PRINTC$
475 REM SHOW 10-59 DIFFERENT FACE PAIRS
480 Q=INT(10+50★RND(0))
490 FORZ=1TOQ
500 A=INT(1+6★RND(0))
510 B=INT(1+6★RND(0))
515 REM SOUND A NOTE WHICH DEPENDS
    ON A AND B
520 SOUND 1,700+3★(A★A+B★B),4
540 PRINT " CLR HOME ↓ ← 10 times → ↓
    → ← 12 times → → ";A;
    " → ← 11 times → → ";B
560 NEXT Z
585 REM USE LAST VALUES OF A,B
590 T=A+B
595 REM JUMP IF PLAYER WINS OUTRIGHT
600 IF T=7 THEN 1000
610 IF T=11 THEN 1000
615 REM JUMP IF PLAYER LOSES OUTRIGHT
620 IF T=2 THEN 1100
630 IF T=3 THEN 1100
640 IF T=12 THEN 1100
650 PRINT
660 PRINT
670 PRINT
680 PRINT" YOU HAVE TO MAKE";T;"
    BEFORE 7"
700 PRINT" ↓ ← 8 times → ↓ HIT ANY
    KEY TO GO ON"
710 GET R$
720 IF R$= " " THEN 710
730 PRINT " SHIFT and CLR HOME NEXT
    THROW (BET= "; W; ")"
740 PRINT "MAKING"; T
750 PRINT " CLR HOME ↓ ← 4 times → ↓ "
760 PRINT A$
770 FORJ=1TO5
780 PRINT B$
790 NEXT J
800 PRINT C$
805 REM SHOW 10-19 DIFFERENT FACE PAIRS
810 Q=INT(10+10★RND(0))
820 FOR Z=1 TO Q
830 A=INT(1+6★RND(0))
840 B=INT(1+6★RND(0))
850 SOUND 1,700+3★(A★A+B★B),4
870 PRINT" CLR HOME ↓ ← 8 times → ↓
    → ← 12 times → → ";A; " →
    ← 11 times → → ";B
900 NEXT Z
925 REM IF A+B=T PLAYER WINS
930 IF A+B=T THEN 1000
935 REM IF A+B=7 PLAYER LOSES
940 IF A+B=7 THEN 1100
945 REM ELSE PLAYER THROWS AGAIN
950 GOTO700
```

```
 99Ø REM PLAYER WINS
1ØØØ PRINT" ↓ ←7 times→ ↓ YOU WIN"
1ØØ5 REM ADD WINNINGS TO CAPITAL
1Ø1Ø C=C+W
1Ø15 REM PAEAN OF PRAISE
1Ø17 FOR J=1TO 5ØØ:NEXTJ
1Ø2Ø SOUND1,834,32
1Ø25 SOUND1,798,24
1Ø3Ø SOUND1,81Ø,8
1Ø35 SOUND1,834,32
1Ø4Ø SOUND1,739,32
1Ø42 SOUND1,77Ø,8
1Ø44 SOUND1,798,8
1Ø46 SOUND1,81Ø,8
1Ø48 SOUND1,834,8
1Ø5Ø SOUND1,81Ø,16
1Ø52 SOUND1,798,16
1Ø54 SOUND1,77Ø,64
1Ø95 GOTO31Ø
11ØØ REM PLAYER LOSES
111Ø PRINT" ↓ ←10 times→ ↓ YOU
     LOSE"
1115 REM CHIRP OF VINDICTIVE TRIUMPH
112Ø FOR J=1TO5ØØ:NEXTJ
113Ø FOR X=8ØØTO1ØØØ STEP 4
114Ø SOUND1,X,1
115Ø SOUND1,X+23,1
116Ø NEXTX
1195 REM TAKE LOSSES FROM CAPITAL
12ØØ C=C−W
121Ø IF C> Ø THEN 31Ø
122Ø PRINT"YOU ARE NOW BROKE"
123Ø STOP
```

APPENDIX A PROBLEM SOLUTIONS

Example 1:

```
1Ø INPUT V
2Ø INPUT R
3Ø PRINT "A="; V↑2/R
4Ø STOP
```

Example 2:

```
1Ø PRINT "X  FORMULA"
2Ø FOR X=Ø TO 2 STEP Ø.2
3Ø PRINT X; 1/(1+X↑2)
4Ø NEXT X
5Ø STOP
```

Example 3:

```
 1Ø PRINT "GIVE THE THREE SIDES"
 2Ø INPUT "A"; A
 3Ø INPUT "B"; B
 4Ø INPUT "C"; C
 5Ø S= (A+B+C)/2
 6Ø X= S★(S—A)★(S—B)★(S—C)
 7Ø IF X <Ø THEN 1ØØ
 8Ø PRINT "AREA IS"; SQR(X)
 9Ø STOP
1ØØ PRINT "THESE ARE NOT THE"
11Ø PRINT "SIDES OF A TRIANGLE"
12Ø STOP
```

Glossary:

A, B, C:	Three "sides" of triangle
S:	Semi-perimeter
X:	Square of area (if any)

Example 4:

```
 1Ø REM SLIGHTLY FASTER VERSION
 2Ø INPUT "HIGHEST VALUE"; H
 3Ø FOR N=3 TO H STEP 2
 4Ø Q= SQR (N)
 5Ø FOR J=2 TO Q
 6Ø IF N/J= INT (N/J) THEN 9Ø
 7Ø NEXT J
 8Ø PRINT N;
 9Ø NEXT N
1ØØ STOP
```

APPENDIX C

Error Messages

This list covers errors which can arise if you use the BASIC facilities described in this book. Other errors can occur if you run programs of a more advanced nature.

Division by Zero

Dividing a number by zero is not allowed. The error may arise in commands like

```
    1Ø A = 5/Ø

or  2Ø B = Q/(J—J)
```

Extra Ignored

If you type too many items (numbers or strings) in reply to an INPUT command, the extra ones will be ignored. The program doesn't stop.

Illegal Quantity

A number used in a command is too large (or too small). For instance, any number you POKE into a location must be in the range Ø to 255.

This error can occur in commands like

```
    SOUND 1Ø, 5, 37

or  COLOR 3ØØ.2
```

Load Error

Your program is not loading correctly from the cassette recorder or the floppy disk. If using a cassette recorder, try cleaning the reading head. Alternatively, the program may not have been recorded correctly in the first place, or the tape may have been damaged by a magnetic field.

Next Without For

The FOR-NEXT structure of your program is wrong.

Out of Memory

The computer has run out of space in the memory. This only happens with very long programs, or ones which use large amounts of data.

Redo from Start

If an INPUT command expects a number, and you type something which isn't a number, the computer will display this message and let you try again.

String Too Long

A string formed by concatenation is larger than 255 bytes.

Syntax Error

A "command" has broken the rules of BASIC grammar. Possible causes are mismatched brackets, mis-spelled keywords, or elements of expressions in the wrong order.

Type Mismatch

This means that a number has been used instead of a string, or vice versa.

Verify Error

The verification process has failed. Try SAVE'ing the program again.

INDEX

INDEX

AN INTRODUCTION TO BASIC PART 1

TAPE 1	TAPE 2
TESTCARD	SENTENCES
HANGMAN	UNIT7QUIZ
SPEEDTYPE *	UNIT8PROG
UNIT3QUIZ	UNIT9QUIZ
UNIT4DRILL	UNIT10QUIZ
UNIT5QUIZ	UNIT11PROG
	UNIT12QUIZ
	SOUND DEMO
	TUNE
	HEADS
	REACTION
	CRAPS

Each program is recorded twice, once on each side of the tape.

commodore

Commodore Business Machines (UK) Limited
1 Hunters Road, Weldon, Corby, Northampton NN17 1QX, England.