VIC 20
COLOUR COMPUTER

AN INTRODUCTION TO BASIC:~PART 1

THE COMPREHENSIVE TEACH YOURSELF PROGRAMMING SERIES FOR VIC 20

by Andrew Colin

commodore
COMPUTER

VIC 20 AN INTRODUCTION TO BASIC:~PART 1

1. "TESTCARD" 2. "HANGMAN" 3. "SPEEDTYPE" 4. "UNIT3QUIZ" 5. "UNIT4DRILL"
6. "UNIT5QUIZ" 7. "SENTENCES" 8. "UNIT 7QUIZ"

TAPE 1

100 50 0

TAPE 1

commodore
COMPUTER


VIC 20
AN INTRODUCTION TO BASIC:~PART 1
1. "UNIT8PROG" 2. "UNIT9QUIZ" 3. "UNIT10QUIZ" 4. "UNIT11PROG" 5. "UNIT12QUIZ"
6. "SOUND DEMO" 7. "PIANO" 8. "HEADS" 9. "REACTION"
TAPE 2
100 50 0
TAPE 2
commodore
COMPUTER
©ANDREW COLIN 1981 ©COMMODORE ELECTRONICS LTD.

CONTENTS

This course is Part I of a series designed to help you learn about every aspect of programming the Commodore VIC computer. 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 with a collection of VIC 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 VIC, 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

Congratulations, and welcome to the VIC programming course. VIC is a superb machine for playing games and producing brilliant and exciting pictures and sounds on your TV set; but it is also a complete modern computer in its own right.

Computers are extraordinarily versatile; more so, in fact, than anything except a human. The VIC, for instance, can be switched from game playing to be a teaching machine, a calculator, an aid to the handicapped, 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 to 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 a "Space Invaders" 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 games, teaching activities, or serious industrial or medical applications.

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 Commodore VIC, but once you have mastered VIC programming 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 VIC, 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 VIC, 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 VIC 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 VIC 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 pictures to your own design, and sounds to emphasise 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 VIC as a controller for a model railway or private telephone exchange. To make this possible, Commodore are producing a set of advanced programming courses.

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

UNIT:1

This unit helps you get started with your VIC. It explains a number of rather ordinary matters; practical questions which often raise serious problems when people buy their first computers.

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 of day 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!

1 If you have already installed your VIC and used it, you can skip straight through to experiment 1.1. Otherwise, read quickly through the unit even if you know what it is all about; you may still find it useful.

First, arrange your equipment and connect it to the mains. The VIC, power supply and cassette go on the desk or table in front of you, and the TV should be at least 6 feet (2 metres) away if it is a small one, or even further if it has a large screen. The pictures and text produced by the VIC are quite large enough to be read at normal viewing distance, and you will find — if you try it — that working with a screen close to your face is very irritating and tiring.

The various units connect together as shown in the diagram.

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.

VIC is an extremely robust machine, but plugs and sockets do get worn or damaged if they are plugged and unplugged too many times. Once your VIC is set up, aim to leave it undisturbed as long as you can.

If your TV set does double duty as a broadcast receiver, get an aerial switch unit which lets you keep the VIC and the ordinary aerial both connected all the time.

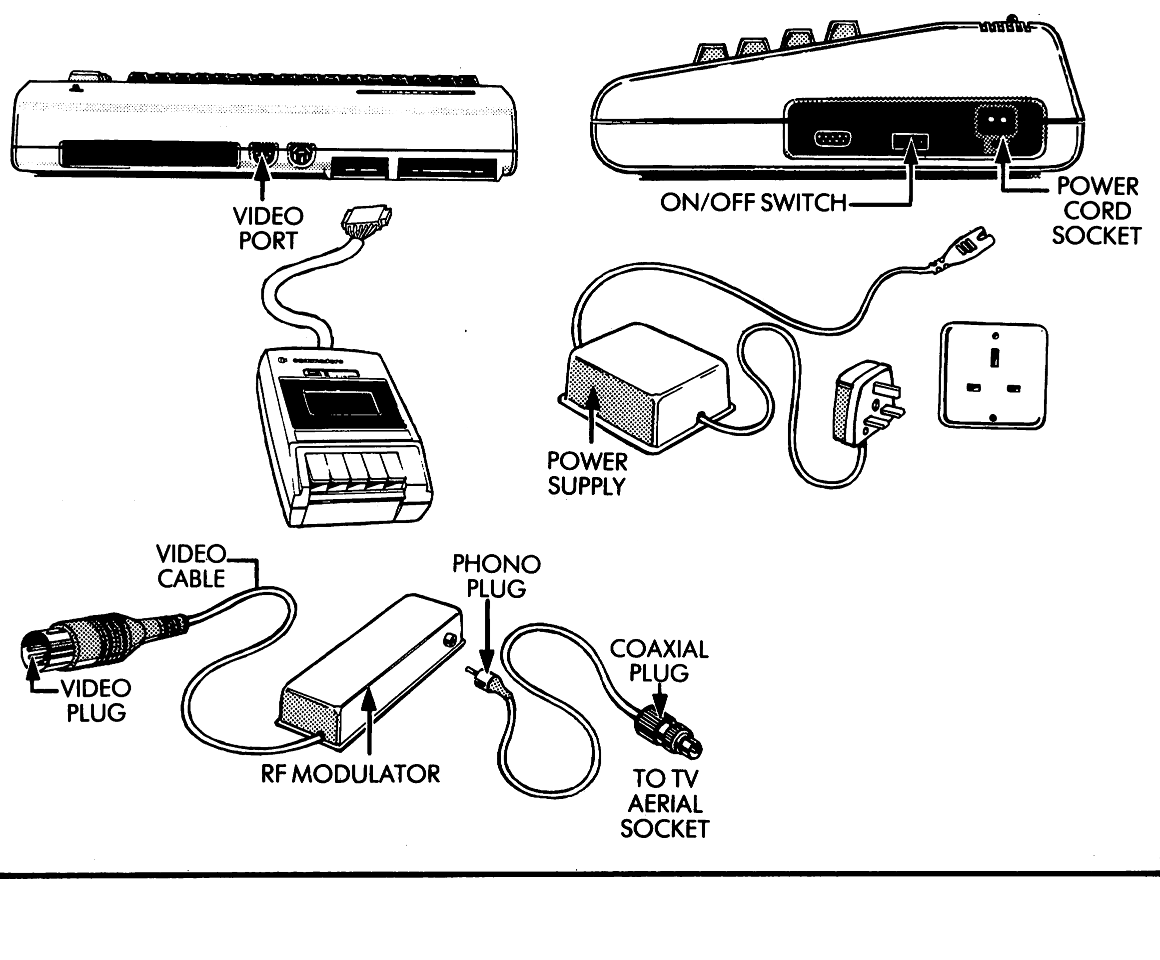
Both VIC and the TV can be run from a single extension power lead with twin power outlets. Such a lead gives you great freedom in deciding how to arrange your home computer system.

Now you are ready to switch on.

Turn on the TV, and select a channel which is not normally used for broadcast reception. (For example, if your set is tuned to receive BBC1, BBC2 and ITV on channels 1, 2 and 3, you could use channel 4.) The set will make a lot of noise, and you may turn down the sound.

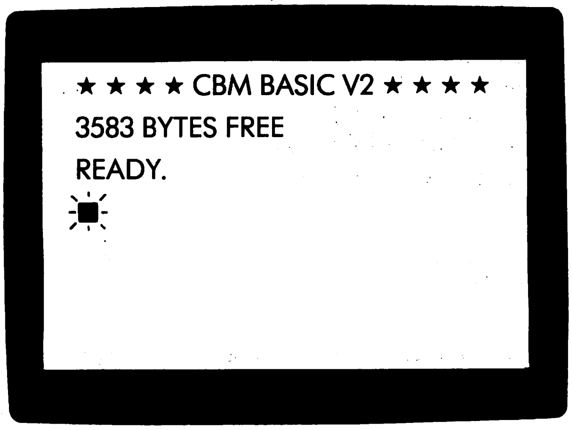
Next, power up the VIC, using the switch on the side at the right. If all is well, the red POWER lamp will glow, but unless you are very lucky, the TV set will still not show a picture.

Now go back to the TV, and adjust the tuning of the channel you have selected. The exact method of tuning varies according to the make of the set, and is always explained in the manufacturer's instructions; but in most cases there is either a small knob or a 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:



The central square 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, or if the picture comes up in black and white only, turn the VIC 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 need be.
- Is the VIC power light on? If not, check:
  - (a) That there is no general power failure
  - (b) That some other device (table-lamp or hair-dryer) will run from the socket you are using. If not, try changing the fuse in the extension lead plug.
  - (c) That the fuse in the VIC power supply plug is intact (try a new fuse).
  - (d) That the power supply is firmly plugged in to the VIC.
- Is the VIC properly connected to the aerial socket on the TV?

If your system still doesn't work, take it back to your dealer for advice and repair.

The message now on your screen consists of a number of 'characters' including letters, numbers and symbols such as ★. These characters are always the same size, and when the screen is full it holds about 500 characters.

The first line on the screen identifies the product: a BASIC system designed and manufactured by Commodore Business Machines. The 'V2' is a version number which may change from time to time.

The message on the next line of the screen 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.

If you are just starting you will probably have bought the smallest system, and the correct figure is 3583*. If you expand your VIC by buying and putting in extra memory, the figure will be larger.

If the number on the screen is less than 3583, or different from its usual value, it is a sign that the VIC is broken. It must be returned to your dealer for repair.

The third line tells you that VIC is now ready to obey commands which you type on the keyboard.

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

PRINT 5 + 8 RETURN

(This takes 10 key depressions:

P R I N T SPACE 5 + 8 and press the RETURN key. This is the large key on the right of the keyboard.) (Before you start typing, touch the SHIFT LOCK key to make sure 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 prime function 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!

**This figure may be a little different in later versions of the VIC.*

EXPERIMENT 1·1

To do anything useful, the VIC must have a program. Programs are often stored on cassette tapes, and this first experiment will give you practice in loading a program from a tape into the VIC. Follow these instructions carefully:

1. Make sure that the cassette unit is plugged into the VIC.
2. Press STOP on the cassette unit.
3. Open the holder on the cassette unit, take out any tape which might be there already, and put in the TESTCARD tape — label uppermost and with the tape window facing towards you. Close the holder. If it does not 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 key strokes 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 very 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 VIC 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:

```
**** CBM BASIC V2 ****

3583 BYTES FREE

READY.

LOAD "TESTCARD"

PRESS PLAY ON TAPE
```

Press the PLAY key on the cassette unit. Wait about a minute for the program to be loaded.

If the tape runs on and on, and the screen shows several messages like

FOUND TESTCARD
FOUND HANGMAN

you have probably mis-spelled the name TESTCARD. Stop the computer by pressing RUN STOP, and go back to step 1.

If the tape runs on and on, and *nothing* happens, make sure you aren't trying to play a blank tape. If not, suspect the cassette unit and take it (and the VIC) back to your dealer for a checkup.

When the program is finally loaded, the machine will say

READY.

Start the program by typing

RUN RETURN (4 key depressions)

The first program shows you the range of colours the VIC can handle, and helps you make fine adjustments to your TV set.

Turn up the volume, and adjust the channel tuning very gently until you hear music being played clearly and with as little background noise as possible. (You may recognise the piece, which is Offenbach's Cancan from the opera "Orpheus in the Underworld".) Then set the brightness and colour controls so that the colours correspond to their names and look right when seen from a reasonable distance.

You will see that the picture appears on a cyan frame. You can change the frame to any other colour by holding down the CTRL key and typing one of the 8 colour keys in the top row. They are labelled:

BLK WHT RED CYN PUR GRN BLU YEL

Eventually, you can stop the TESTCARD program by pressing the RUN STOP key.

When you press 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 the VIC is broken; it just tells you that there has been a break in the sequence of commands which makes up the program.

The READY is a sign that the VIC is ready to obey another command from the keyboard.

If the sound continues hold down RUN STOP and hit RESTORE, this will always silence the VIC and clear the screen.

Sometimes, when the TESTCARD program is stopped in the middle of a tune, the VIC goes on sounding the note it played last. You can stop the note playing by holding down the RUN STOP key and pressing RESTORE which is the key near the top right of the keyboard.

You may find it convenient to use the TESTCARD program whenever you need to readjust your set.

Experiment 1.1 Completed	

EXPERIMENT 1·2

4

Experiment 1.2 is a word-guessing game designed to help you get the feel of the keyboard. Load the program from the cassette; it is called HANGMAN, so you type
LOAD "HANGMAN" and press the RETURN key.

```
★ ★ ★ ★ CBM BASIC V2 ★ ★ ★ ★

3583 BYTES FREE

READY.

LOAD "HANGMAN"

PRESS PLAY ON TAPE
```

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

RUN RETURN

and the game will start. 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 1.2 Completed	

NOTE:
Each program supplied with this package is recorded twice. The duplicate block of programs follows the initial recording of the block on the cassette.

UNIT:2

EXPERIMENT 2·1

Welcome back. This unit is about the VIC's 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 VIC.

On the other hand, don't be put off if you have never done any typing. You will need a little more time to get used to the VIC 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 do something for you, such as loading a program or adding up some numbers. At present, just to use the screen, you don't need the computer's help. If you do press RETURN, the VIC 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, the VIC 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 for 30 seconds, and then on again.

The 66 keys on the keyboard are divided into two categories:

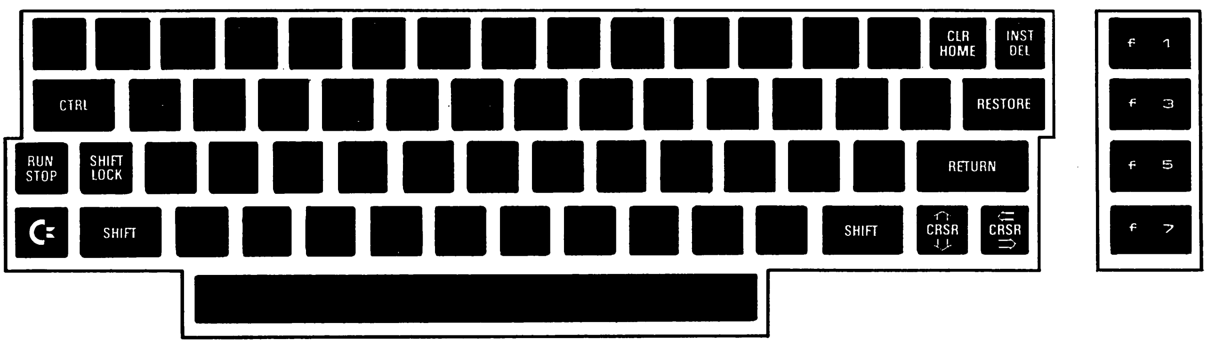
- 50 *Symbol* keys, which make the VIC draw characters on the screen.
- 16 *Function* keys, which control the way the characters are drawn.

The function keys are: CTRL, RUN STOP, SHIFT LOCK, C=, SHIFT, CLR HOME, INST DEL, RESTORE, RETURN, CRSR ⇕, CRSR ⇔, SHIFT and f1 to f8

The ten symbol keys marked 1 to Ø also have certain control functions.

Compare the keyboard with the chart below, and identify the various control keys.

Press the SHIFT LOCK 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. Just below the READY. message you will see the flashing cursor.

```
★ ★ ★ ★ CBM BASIC V2 ★ ★ ★ ★

3583 BYTES FREE

READY.

■
```

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 move across the screen. 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 right 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 in each line on the screen.

Next, type some more lines, and keep going until you reach the bottom line of the screen. Count the number of lines showing and write the number in the box below. Remember to include the blank lines above and below the message:

3583 BYTES FREE

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. Any blank lines are bought at the expense of the top-most ones, which have now vanished. The top lines have gone for good, and there is no way of bringing them back, unless copies are 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

VIC has some 50 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 and type the line
← 1 2 3 4 5 6 7 8 9 0 + − £ Q W E R T Y U I
(These are all the symbol keys in the top row and some of those in the second.)

```
★ ★ ★ ★ CBM BASIC V2 ★ ★ ★ ★

3583 BYTES FREE

READY.

←1234567890+−£QWERTYUI

■
```

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. If your TV picture is a bit smudged it may help you to look at the signs embossed on the keys themselves. Notice how the graphic symbols usually reach the edges of the little squares they occupy, so that they can be made to touch each other.

SYMBOL	←	1	2	3	4	5	6	7	8	9	Ø	+	−	£	Q	W	E	R	T	Y	U	I
SHIFT																						
C=																						

SYMBOL	O	P	@	★	↑	A	S	D	F	G	H	J	K	L	:	;	=	Z	X	C	V	B
SHIFT																						
C=																						

SYMBOL	N	M	,	.	/																	
SHIFT																						
C=																						

Next, type the line yet 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 lines
O P@★↑A S D F G H J K L : ; = Z X C V B
N M , . /

Fill up the last line with spaces.

Note and remember that the digit Ø is different from letter 'O'. You should always use the Ø to show that you mean the number, not the letter.

Press C= and SHIFT down together. Many of the capitals on the screen will change into lower-case letters. Press the keys together again 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 explained in the second volume of this course.

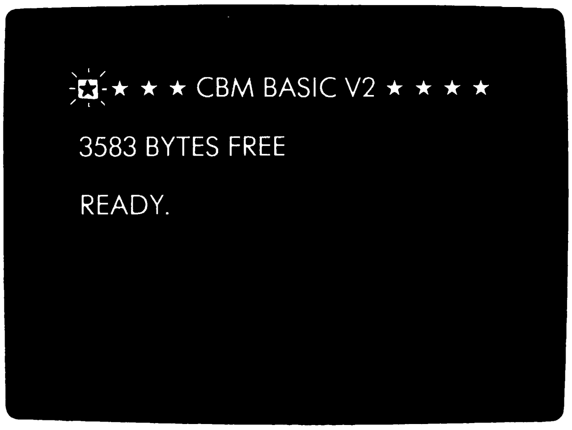
Experiment 2.2 Completed	

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 chose.

This can be done with the *cursor control* keys, of which there are three [CLR HOME] [CRSR ⇑⇓] and [CRSR ⇐⇒].

When you type [CLR HOME] by itself, it moves the cursor back 'home', which is the top left-hand corner of the screen. Restart your machine (just in case the previous experiment left it in a funny mood) and strike this key. You will see the cursor move to the ★ at the top left of the screen. The ★ remains visible because the cursor is transparent; but if you type another character (or



a space) the symbol under the cursor is replaced by the new one. Try putting an = instead of the ★. Type = three more times, giving you

= = = = CBM BASIC V2 ★ ★ ★ ★

as the top line of the screen.

You may now want to alter the ★ ★ ★ ★ on the right to = = = =, so as to keep the line symmetrical. If you move the cursor along by typing spaces, you will rub out the title in the centre of the line. The correct way is to use the [CRSR ⇐⇒] key. Every time you hit this key, the cursor moves one place right, but *without* spoiling anything underneath it.

Try moving the cursor to the first ★ on the right, and then putting in four = signs. The top line becomes

= = = = CBM BASIC V2 = = = =

and the cursor moves to the left of the next line.

If you hold the [CRSR ⇐⇒] key down continuously, then after a short pause the cursor moves by itself at a rate of about 10 places a second, line after line. This is useful to move around quickly.

When the [CRSR ⇐⇒] key is struck while [SHIFT] is held down, the cursor moves *backwards*. When it reaches the beginning of one line it moves *up* to the end of the previous one.

Next try going back to the first line, and changing the = signs back to ★'s.

Move the cursor down to the bottom line of the screen and watch what happens when you move past the end of the line: the whole screen moves up just as if you had added another character.

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.

The [CRSR ⇑⇓] key moves the cursor up or down a whole line at a time. Try some experiments with it, and make sure you understand how it works.

Next, fill up the screen with a few characters and graphics, and then press [CLR HOME] while holding down the shift key. The cursor moves home and the screen is cleared, giving you a fresh screen to work on.

Fill in the following table:

Key	Effect: No shift	Effect: Shift
CLR HOME	Moves cursor home	
CRSR ⇐⇒		Moves cursor 1 place backwards
CRSR ⇑⇓		

Now 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 which are 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
BLOGGS

Now draw some playing cards, with curved corners and the right symbols (we suggest you keep to black cards worth 10 or less).

Finally, if your artistic talent is up to it, trying 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 control. For example, if you type AUSTRAPIA when you mean AUSTRALIA, you can move the cursor back over the P and change it to 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*, and hit INST DEL. The X disappears, and DIA all move up to the left, 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 INST DEL key is to get rid of the character or 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

2 spaces

AUSTR■ IA

cursor

Now finish by filling in AL. Move the cursor past the end of the word.

To practice 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
RED	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 to get familiar with the keyboard. Load it (by typing LOAD "SPEEDTYPE"), start it with the RUN command and practice using it as much as you feel is necessary.

UNIT:3

VIC is a colour computer. 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

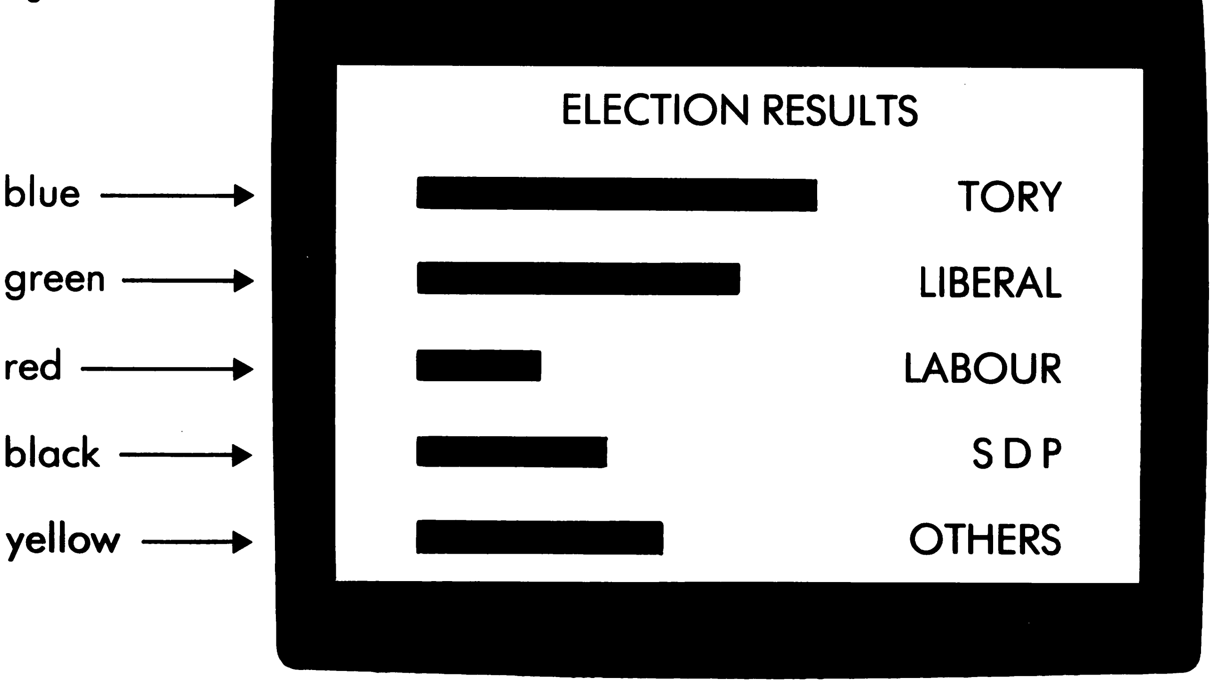
Use the TESTCARD program (unit 1) to make sure that your TV receiver is properly adjusted.

Stop the program by holding down RUN STOP and striking RESTORE .

You will see that the cursor at this stage is blue. 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 blue, which is the present colour of the cursor.

(Where a character has a lot of fine detail the apparent colour may not always be correct. This is a consequence of using an ordinary TV set, which has a narrow RF bandwidth. The difficulty can be cured by buying a more expensive *colour monitor*, but it is hardly worth it unless you plan to display a great deal of text in different colours.)

Figure 1



The colour of the cursor can be changed at any time by typing one of the 8 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*
When you hit 2 (WHT) the cursor disappears. This is because it is now the same colour as the background, and therefore invisible. This colour is called 'white'.
The other keys change the cursor to red, cyan, purple, green, blue and yellow, respectively.

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.

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.

Experiment 3.1 Completed	

EXPERIMENT 3·2

As well as looking after the colours of individual symbols, the VIC can control the colours of the outer frame and the background on which the symbols appear. VIC does not have any dedicated keys to control these colours, and the only way to change them is to give a special command.

Type POKE 36879,24 Check it carefully, correct it if necessary, and then strike the RETURN key. The frame on the TV screen immediately turns black.

This special command has three parts:

POKE: This is called a *keyword*.

36879: This is called an *address*. It identifies the part of the VIC which looks after the colours of the frame and background. Any special command to change either of these two colours must *always* refer to this address.

24 : This is a code, which indicates "Black frame, white background".

To select other colours and combinations, it is useful to know the 'colour codes'. There are 16 different colours, each with its own number. Any of the 16 can be used as the background colour, but only the first 8 can be selected colours for the frame or the symbols. The colours and their numbers are:

Colour	*Number*	*Comments*
Black	Ø	
White	1	Can be used for background, frame and symbols. The codes are *1 less* than the keys used to select the colour of the cursor.
Red	2	
Cyan	3	
Purple	4	
Green	5	
Blue	6	
Yellow	7	
Orange	8	
Light Orange	9	
Pink	1Ø	
Light Cyan	11	Can be used for background only.
Light purple	12	
Light green	13	
Light blue	14	
Light yellow	15	

The frame and background colours must always be set by the same command. The 'code number' can be worked out as

16 × background colour number + frame colour number + 8.

For example, a picture with a pink background and a purple frame could be called up by typing

POKE 36879, 172

because $172 = 16 \times \underbrace{\boxed{1Ø}}_{\text{pink}} + \underbrace{\boxed{4}}_{\text{purple}} + 8$

If you don't know your 16-times table too well (and few people do) you may find the following chart useful:

Select a few colour combinations, POKE them by using the appropriate special commands, and find one or two you really like.

Background Colour	Frame Colour							
	Black	White	Red	Cyan	Purple	Green	Blue	Yellow
Black	8	9	1Ø	11	12	13	14	15
White	24	25	26	27	28	29	3Ø	31
Red	4Ø	41	42	43	44	45	46	47
Cyan	56	57	58	59	6Ø	61	62	63
Purple	72	73	74	75	76	77	78	79
Green	88	89	9Ø	91	92	93	94	95
Blue	1Ø4	1Ø5	1Ø6	1Ø7	1Ø8	1Ø9	11Ø	111
Yellow	12Ø	121	122	123	124	125	126	127
Orange	136	137	138	139	14Ø	141	142	143
Light Orange	152	153	154	155	156	157	158	159
Pink	168	169	17Ø	171	172	173	174	175
Light Cyan	184	185	186	187	188	189	19Ø	191
Light Purple	2ØØ	2Ø1	2Ø2	2Ø3	2Ø4	2Ø5	2Ø6	2Ø7
Light Green	216	217	218	219	22Ø	221	222	223
Light Blue	232	233	234	235	236	237	238	239
Light Yellow	248	249	25Ø	251	252	253	254	255

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:

Restart the VIC, select purple as the cursor colour, 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 purple background, instead of purple on white. In particular comes out as , and are changed to and , and a space appears as a solid block of purple. We say the VIC is in *reverse* mode.

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

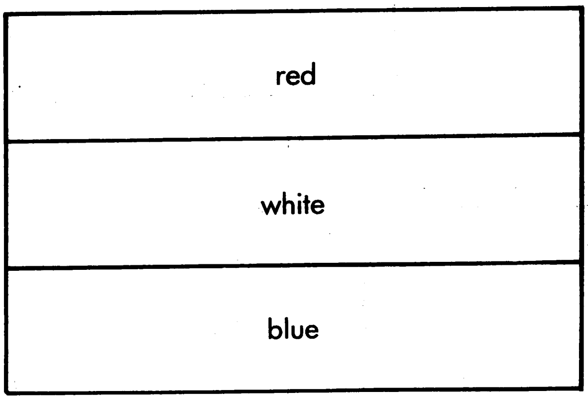
The cursor does not show whether the machine is reverse or normal mode. If you are using many reversed symbols it is easy to forget, and to be in some doubt as to how the next character will appear. This difficulty can be resolved in two ways:

(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.

The best way to fill 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 1Ø 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 left-hand corner of the flag (blue). This is done by

POKE 36879, 1Ø4 RETURN

(The '1Ø4' is copied from the table on page 17.)

Next move the cursor *home*, select red and reverse mode (hold down CTRL and type RED and then RVS ON). Then hold down the space bar and fill up 8 lines with reversed red spaces.

Next, select *white* and fill up 7 lines with white reversed spaces.

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

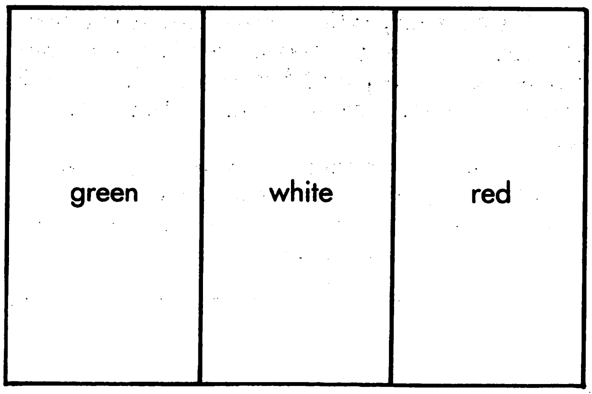
It is important to set the background colour to *one* of the colours which appear on the picture, otherwise you will be unable to hide the cursor when the drawing is complete. Likewise the frame colour should be different from any colour which appears in the flag itself.

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

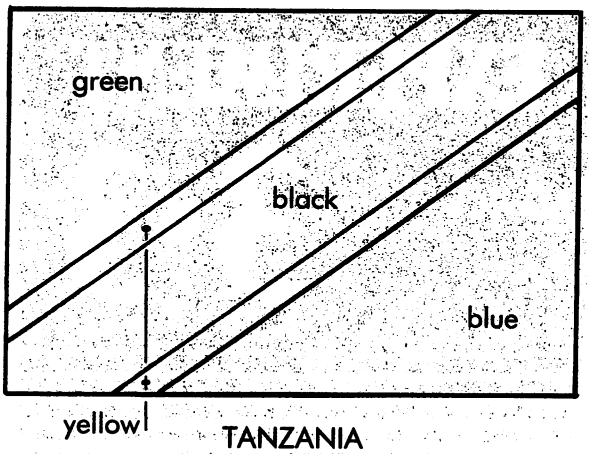
Even more difficult are flags with diagonal elements such as Tanzania or Czechoslovakia. The parts of the flag near the sloping lines must be made with the graphics or ,

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 Tanzanian flag, a suitable background is yellow; blue wouldn't do because there would be no way of drawing a green and

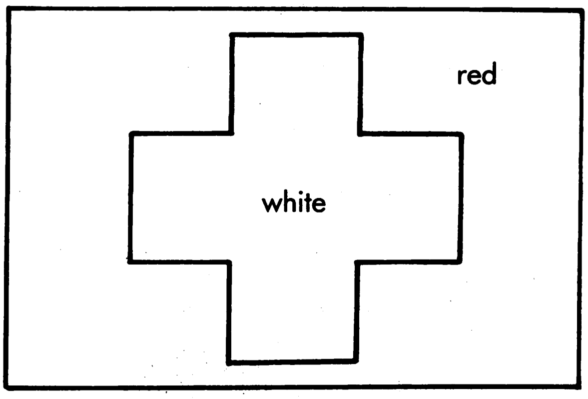
19



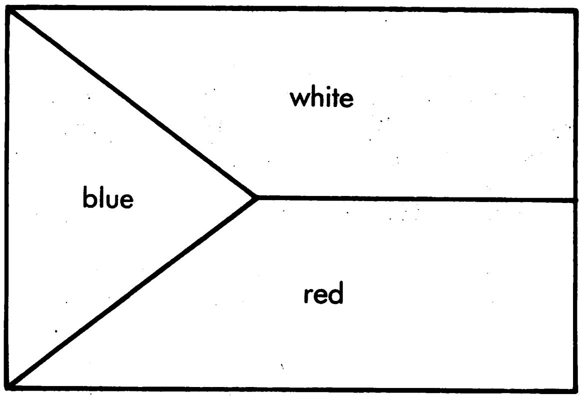
ITALY



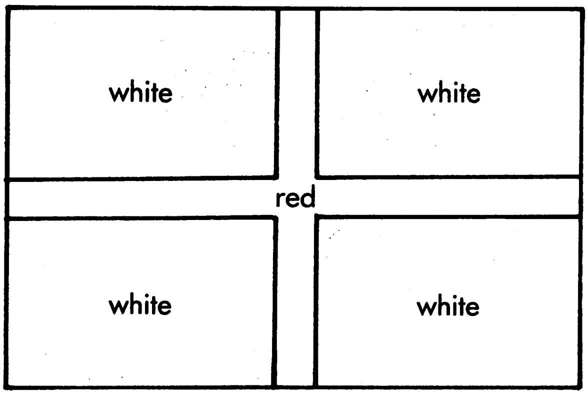
TANZANIA



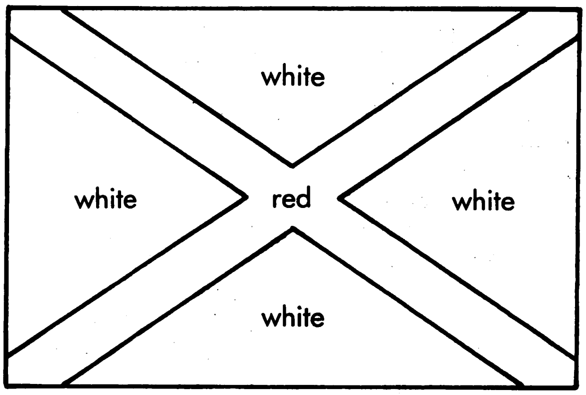
SWITZERLAND



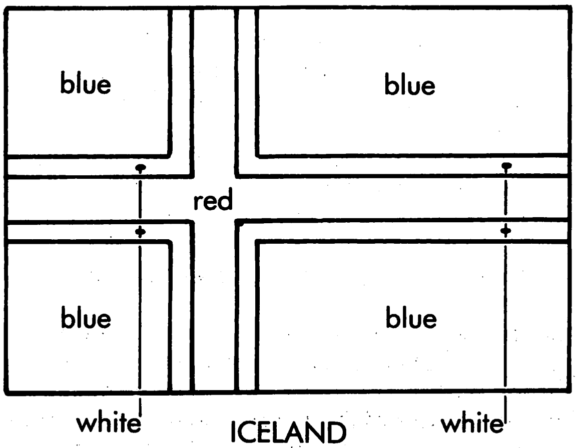
CZECHOSLOVAKIA



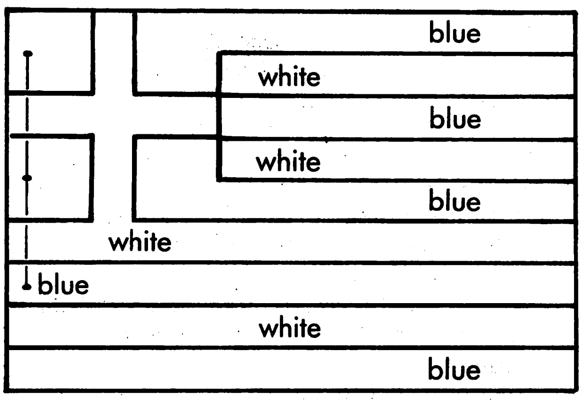
ST. GEORGE



ST. ANDREW



ICELAND



GREECE

yellow element.

A typical line about half-way 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	

EXPERIMENT 3·4

Draw the best coloured picture you can, using the full scope of the VIC. 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"

UNIT:4

In the first three units of the course we have concentrated almost wholly on the VIC 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 VIC can do on your behalf.

As you already know, VIC 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

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 VIC 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 "VIC 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 VIC 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", "VIC 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 Ø (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 VIC 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 VIC. 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 VIC 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*	*VIC'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:

Ø		Ø
BOLTON UNITED	vs	KELSO CITY

> Whenever either side scores a goal, replace the most recently written number in the corresponding box by the same numbers, *plus one*. Rub out (or cross out) the old number.
>
> Part-way through the match, the state could be

~~Ø 1~~ 2	~~Ø 1 2 3 4~~ 5
BOLTON UNITED	KELSO CITY

> 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 VIC (it probably has 3583 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 VIC 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)	37 X
After	5 X	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 VIC *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 VIC 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 VIC 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 VIC 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 VIC 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.

So far, our PRINT command has been stored and retrieved, but it hasn't actually been obeyed.

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

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 VIC 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 VIC 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 VIC 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 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 VIC'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:

```
1Ø PRINT "NORTH"
2Ø PRINT "WEST"
3Ø PRINT "SOUTH"
4Ø PRINT "EAST"
5Ø GOTO 1Ø
```

When it is started at label 1Ø, the VIC obeys the first four commands in sequence. The next command sends it back to label 1Ø, 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 1Ø

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 CTRL (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 1Ø. Suppose you want to alter your program so that it includes the diagonal directions

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 1Ø command you have been using to start your program has a more convenient equivalent: RUN. RUN simply makes the VIC start obeying commands at the one with the lowest number.

When you put a semicolon *after* a string, the VIC 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.

```
1Ø PRINT "├┤┬┴ ┌┐└ ";
2Ø GOTO 1Ø
```

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:

```
10 LET A = 1
20 PRINT A
30 LET A=A+1
40 GOTO 20
```

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 CTRL key to slow it down. (But don't touch CTRL 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 10, which gives the variable A the value 1. The next command displays this value on the screen.

Command '30' 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 VIC return to command 20! The value of A is displayed again, but now it is 2. The machine again works through the sequence 20, 30, 40 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"):

10 B=0	10 A = 1
20 PRINT B	20 B = A★A
30 B = B+3	30 PRINT A,B
40 GOTO 20	40 A=A+1
	50 GOTO 20

And now check to see if you were right.

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

```
10 X$ = "★"
20 PRINT X$
30 X$ = X$ + "#"
40 GOTO 20
```

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 40 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 3Ø
```

The line ERROR in 3Ø means that the command which tried to store the offending string was the one labelled 3Ø.

Here are some more programs for you to predict.

1Ø A$ = "++"	1Ø A$ = "XY"
2Ø PRINT A$	2Ø PRINT A$
3Ø A$ ="A"+A$+"—"	3Ø A$ = A$+A$
4Ø GOTO 2Ø	4Ø GOTO 2Ø

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

37

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.

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 six 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. 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 2000 — 2090

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 10; 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 REM's 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 650)

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

(try LIST 1100—1000)

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

```
10 PRINT "LIST TRIAL"

20 LIST

30 GOTO 10
```

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 five ways to get rid of a labelled command in VIC's store, but three 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 VIC off

or
- Typing NEW

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

An individual command can be removed

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

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 1981

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 VIC register the new command in place of the old.

Suppose you want to alter line 100, so that it reads

100 REM MAIN STUPID SENTENCE GENERATOR

LIST command 100, 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 100 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 type RUN at the end of the program. If it 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.

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 9070 (for men) and 9100 (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 9070 and 9100 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 4 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.

Some possible alternatives for lines 9070 and 9100 could be:

```
9070DATABILL,GEOFFREY,
PERCIVAL,MR.SOPHOCLES,
THE HEADMASTER,Z
9100DATAGRANNY,SUSAN,V
IOLET,MRS.PINKERTON,TH
E GYM MISTRESS,AUNTIE
FLO,RACHEL,PENNY
9101DATAKATE,LAURA,FRA
NCES,NORAH,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:

- 9000 Actions that people do by themselves (intransitive verbs)
- 9010 Actions that people do with each other (transitive verbs)
- 9020 Actions that people do with clothes (transitive verbs)
- 9030 Items worn by men
- 9040 Items worn by ladies

**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.*

9050 A list of adverbs and adverbial phrases, describing actions that people do with each other.

9060 A list of adverbs and adverbial phrases, describing actions that people do by themselves

9070 Men's names

9080 Adjectives describing men

9090 Various sorts of men

9100 Ladies' names

9110 Adjectives describing ladies

9120 Various sorts of ladies

Alter the lists in any way you like. Remember to keep them consistent. If you alter 9020 to actions dealing with food, you must alter 9030 and 9040 accordingly, otherwise you may get sentences like

SUSAN ATE HER WELLINGTON BOOTS

If your lists are very much longer than the originals, there is a danger of running out of space. The VIC uses its memory both to store the lists of names and as a 'scratchpad' for internal calculations. The overall capacity is 3583 bytes, and this is quite easy to exceed. In the extreme case, the machine will report an error as you actually enter a statement, but the computer can also run out while it is making up new sentences. The cure is to make the lists shorter.

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.

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

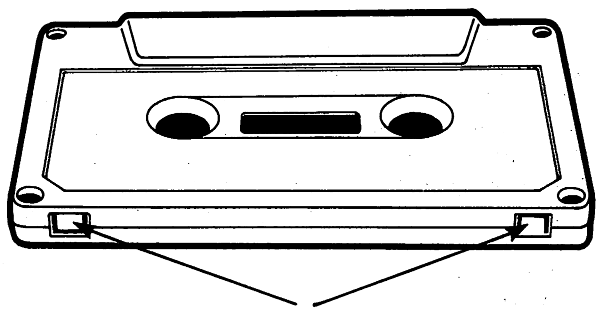
(you can use any name you like instead of FAMILY).

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 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 VIC and the cassette unit back to your dealer for a check-out.

Once a program has been SAVE'd, it can be stored away and LOAD'ed at any time, with a command such as

LOAD "FAMILY"

A program doesn't need to be perfect to be SAVE'd. 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 VIC memory isn't as reliable as a tape 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 on two different tapes 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.

It is amusing to play a VIC program tape through an ordinary (sound) recorder.

Experiment 6.3 Completed	

EXPERIMENT 6•4

In this section we point out a subtle and dangerous trap which lies in wait for VIC 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.

1	Ø		P	R	I	N	T	"	A		G	R	E	A	T		T	R	A	P	"
2	Ø		G	O	T	O		1	Ø												

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 GREAT TRAP

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

A GREAT TRAP 2Ø

? SYNTAX

ERROR IN 1Ø

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 screen width of the VIC. Inside the machine, any BASIC command may be up to 88 characters long. The screen is only 22 characters wide, so the *displayed* version of a command can spread over anything up to 4 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 type 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 GREAT TRAP" 2Ø GOTO 1Ø

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 LIST'ing just the command in which the error is reported. If you type LIST 1Ø, out come lines 1Ø and — apparently — 2Ø! This must be wrong, since you only asked for 1Ø. 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 VIC 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 *not* more than)
>=	(is *not* less than)

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 VIC 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 2Ø
```

Here the condition is X$ <> "ABBBB" and the whole command tells the machine to jump to 2Ø 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:

```
1Ø X$ = "A"

2Ø PRINT X$

3Ø X$=X$ + "B"

4Ø GOTO 2Ø

5Ø 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 5Ø is never reached.

Now stop it and replace line 4Ø with

```
4Ø IF X$ <> "ABBBB" THEN 2Ø
```

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 2Ø 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

```
1Ø P = Ø

2Ø PRINT P, P★P

3Ø P=P+1

4Ø GOTO 2Ø

5Ø STOP
```

Run this program, see what it does, stop it, and change line 4Ø to read

```
4Ø IF P < 11 THEN 2Ø
```

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
Ø          Ø
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 1Ø 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"
1Ø P=Ø.
2Ø PRINT P,P★P
3Ø P=P+1
4Ø IF P<11 THEN 2Ø
5Ø 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)

```
1Ø X$="A"
2Ø PRINT X$
3Ø X$=X$+"B"
4Ø IF X$ <>"ABBBB" THEN 2Ø
5Ø STOP
```

(2)

```
 5 PRINT "NUM","SQUARE"
 7 PRINT
1Ø P=Ø
2Ø PRINT P,P★P
3Ø P=P+1
4Ø IF P < 11 THEN 2Ø
5Ø 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+P 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, 2350*PS ◄──── (Body)
PS=PS+5 ◄────────── (Increments PS)
IF PS<80 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:

```
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
```

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
★★★★★★★★★★
```

53

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 = $1.77).

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

$$F = 1.8 \star C + 32$$

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 VIC 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 VIC 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.

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 VIC 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 pointer") 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

**If you took the cover off the VIC 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.*

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 pointer.

(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 POINTER 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 pointer — 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 pointer on to the next command in sequence, putting a line through the previous value:

PROGRAM POINTER ~~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 pointer 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 ~~1Ø~~ ~~2Ø~~ ~~3Ø~~ ~~4Ø~~ 2Ø

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

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

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

DISPLAY	PROGRAM
1 1	1Ø P=1
2 8	2Ø PRINT P; P★P★P
3 27	3Ø P=P+1
BREAK IN 5Ø	4Ø IF P<4 THEN 2Ø
READY.	5Ø STOP

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 1Ø

VARIABLES

DISPLAY	*PROGRAM*
	1Ø X=5 2Ø Y=7 3Ø Z=X+Y 4Ø W=Y−X 5Ø PRINT X;Y;Z;W 6Ø STOP

(b)

PROGRAM COUNTER 1Ø

VARIABLES

DISPLAY	*PROGRAM*
	1Ø Q=1 2Ø PRINT "SHE LOVES ME" 3Ø PRINT "SHE LOVES ME NOT" 4Ø Q=Q+1 5Ø IF Q<3 THEN 3Ø 6Ø STOP

When you have completed these two experiments check your answers against those given in Appendix B.

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:

```
1Ø PRINT "TWELVE TIMES TABLE"
2Ø P=1
3Ø P=P+1
4Ø IF P< 13 THEN 3Ø
5Ø PRINT P; "★12="; P★12
6Ø STOP
```

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

```
TWELVE TIMES TABLE
13 ★ 12 = 156
BREAK IN 6Ø
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 ~~1Ø~~ ~~2Ø~~ ~~3Ø~~ ~~4Ø~~ ~~3Ø~~ ~~4Ø~~ 3Ø

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

DISPLAY	*PROGRAM*
TWELVE TIMES TABLE	1Ø PRINT "TWELVE TIMES TABLE"
	2Ø P=1
	3Ø P=P+1
	4Ø IF P< 13 THEN 3Ø
	5Ø PRINT P; "★12="; P★12
	6Ø 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.

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
```

he thinks this means "plus"!

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 VIC, 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.

Sometimes—very very rarely—your difficulty may be caused by a mechanical fault in the computer. Modern machines like the VIC 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 from your cassette recorder. 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.

**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.*

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 1Ø gallons (1 gallon = 4.5 litres)

```
1Ø PRINT "GALLONS", "LITRES"
2Ø G=1
3Ø PRINT G, 4.5★G
4Ø G=G+1
5Ø IF G > 11 THEN 3Ø
6Ø 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:

```
1Ø A$ = "★"
2Ø PRINT A$
3Ø A$ = "★★"
4Ø IF A$<>"★★★★★★★★★★" THEN 2Ø
5Ø STOP
```

Experiment 8.2 Completed	

EXPERIMENT 8·3

The program on tape 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 VIC 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 eight different colours
- Colour and background reversal (on and off)
- 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 to choose the function you really need.

You won't have forgotten that you can set the frame and background colours using a 'POKE' instruction and a code number from the table on page 17.

The VIC 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 VIC 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 VIC 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 VIC, 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 below.

Function	*Key Struck*	*Symbol displayed*
Clear Screen	SHIFT and CLR HOME	
Cursor home	CLR HOME	
Cursor up	SHIFT and ⇑ CRSR ⇓	
Cursor down	⇑ CRSR ⇓	
Cursor left	SHIFT and ⇐ CRSR ⇒	
Cursor right	⇐ CRSR ⇒	
Black	CTRL and BLK	
White	CTRL and WHT	
Red	CTRL and RED	
Cyan	CTRL and CYN	
Purple	CTRL and PUR	
Green	CTRL and GRN	
Blue	CTRL and BLU	
Yellow	CTRL and YEL	
Reverse on	CTRL and RVS ON	
Reverse off	CTRL and RVS 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.

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

PRINT" CTRL and YEL EDINBURGH"

hold down

while you type

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

PRINT " π EDINBURGH". The reversed π

reversed

symbol is the code for "yellow".

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

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 yellow. Change it to black or blue, whichever you prefer, 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 GRN GLASGOW"

20 PRINT " CTRL and RED INVERNESS"

30 PRINT " CTRL and PUR ST. CTRL and YEL ANDREWS"

40 STOP

Command 30 shows that you can put more than one control function into a string.

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

PRINT " SHIFT and CLR HOME CRSR CRSR CRSR CRSR CRSR CRSR CTRL and 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 VIC 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:

10 POKE 36879,8

20 PRINT " SHIFT and CLR HOME $\overbrace{\text{CRSR} \ldots \text{CRSR}}^{\text{10 times}}$ $\underbrace{\text{CRSR} \ldots \text{CRSR}}_{\text{8 times}}$ CTRL and WHT LONDON"

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 hitting RESTORE. In fact, 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 VIC 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 VIC 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 VIC is 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$ = "Ø9ØØØØ"

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

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

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

Now set up the VIC'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

```
1Ø PRINT TI$

2Ø GOTO 1Ø
```

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 VIC into a respectable digital clock by a program as follows:

command 1Ø: Selects a purple frame with yellow background

command 2Ø: Clears the screen

command 3Ø: Moves the machine's cursor home, then down 9 lines and across 6 spaces; no newline needed

command 4Ø: Displays TI$

command 5Ø: Jumps back to command 3Ø.

Write down the code for this program in the box below; then enter it on the VIC 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 1Ø x 1Ø block of red dots in the top left-hand corner. This can be done by displaying ten lines, each with ten ● graphics:

1Ø PRINT " SHIFT and CLR HOME ";

2Ø J=1

3Ø PRINT " CTRL and RED ●●●●● ●●●●●"

4Ø J=J+1

5Ø IF J<11 THEN 3Ø

6Ø GOTO 6Ø

This program combines several of the ideas we have already met in previous units. The semi-colon at the end of command 1Ø 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 2Ø to 5Ø form a controlled loop and 6Ø 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 5Ø to some other value (say 15)

(c) removing command 6Ø

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 3Ø to

PRINT " CTRL and 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 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 ";

11Ø J=1

120 PRINT " $\underbrace{\text{CRSR} \ldots\ldots\ldots \text{CRSR}}_{\text{10 times}}$ CTRL and 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...

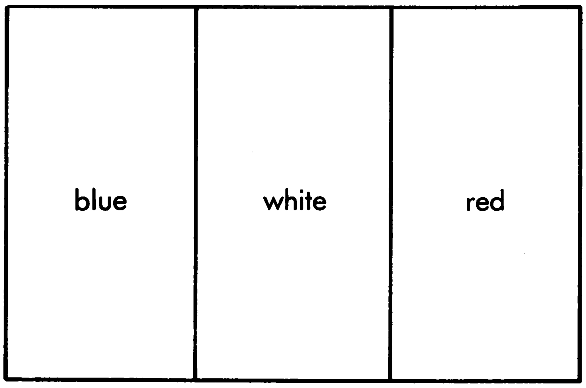
As a final exercise, try writing programs to display some simple flags, or other patterns which fill the whole screen. You will need your wits about you, because various pitfalls lie in wait.

- The normal meaning of a semicolon at the end of a PRINT command is "Don't start a new line". If the VIC 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 8 characters wide, and the other two 7 each. 7 + 8 + 7 = 22.

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

We can build up the flag by displaying 23 lines, each with seven white squares and eight 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 22 lines into a controlled loop, but the last will need a command on its own.

We arrive at

1Ø POKE 36879,4Ø

2Ø PRINT " [SHIFT] and [CLR HOME] ";

3Ø J=1

4Ø PRINT " [CTRL] and [RVS ON] [CTRL] and [BLU] ←7 spaces → [CTRL] and [WHT] ← 8 spaces →"

5Ø J=J+1

6Ø IF J<23 THEN 4Ø

7Ø PRINT " [CTRL] and [RVS ON] [CTRL] and [BLU] ←7 spaces → [CTRL] and [WHT] ← 8 spaces →";

8Ø GOTO 8Ø

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 19. 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 or ROM-packs and used many times by different people. If you want an example, look at the various taped 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:

```
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
```

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 4Ø to read

```
4Ø 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 4Ø of the exchange program could read

```
4Ø 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
 1Ø PRINT "£","LIRE"
 2Ø PRINT
 3Ø PS=5
 4Ø PRINT PS, RE★PS
 5Ø PS=PS+5
 6Ø IF PS<8Ø THEN 4Ø
 7Ø 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 VIC by typing NEW, and type in

```
1Ø PRINT "WHAT'S YOUR NAME"
2Ø INPUT N$
3Ø 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

```
4Ø 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 VIC, and run the program again.)

2. Try

```
1Ø INPUT "NAME";N$
2Ø 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 1Ø in the example is equivalent to the *sequence*

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

Notice that the string of descriptive words must be less than 22 characters long, and that it *must* be followed by a semicolon.

3. Lastly, try

```
1Ø PRINT "GIVE TWO NUMBERS TO
   BE ADDED"
2Ø INPUT A,B
3Ø PRINT "SUM="; A+B
4Ø 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 $\left\{ \begin{array}{l} 43 \\ 19 \end{array} \right.$

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. You must hold it down and strike the RESTORE key at the right of the main keyboard. Try it!

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 VIC 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 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 POKE

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? 2Ø

MALE OR FEMALE?

(a)

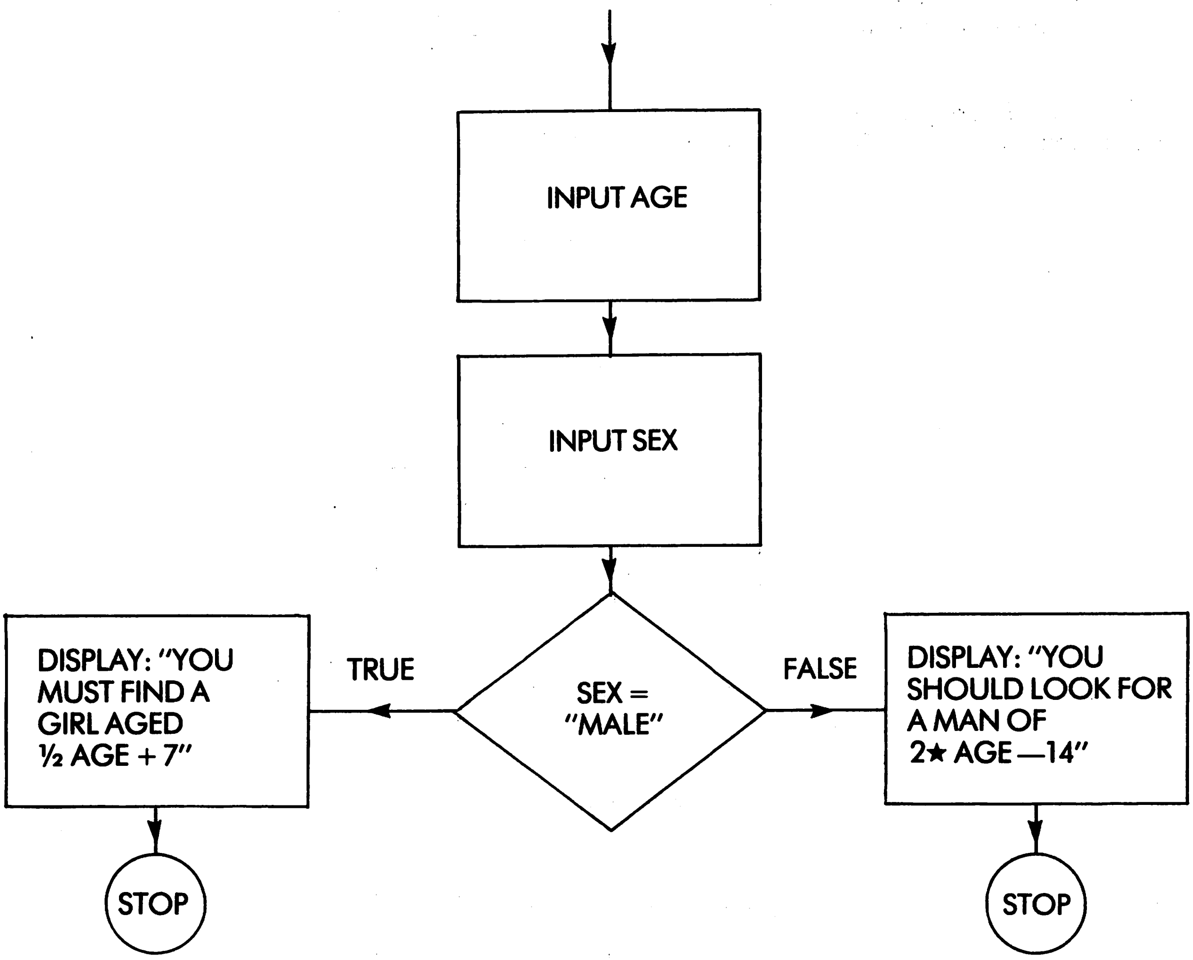
RUN

(b)

Now enter the program into the VIC. 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:



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.

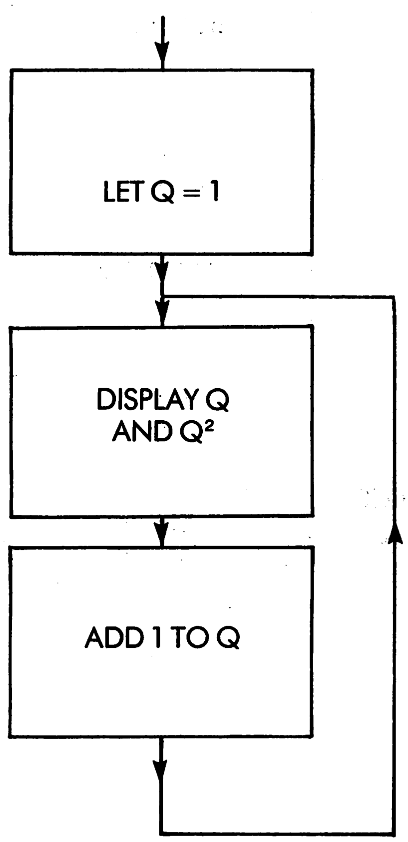
The point of this illustration is to help you see two very important things about computers:

- 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.)

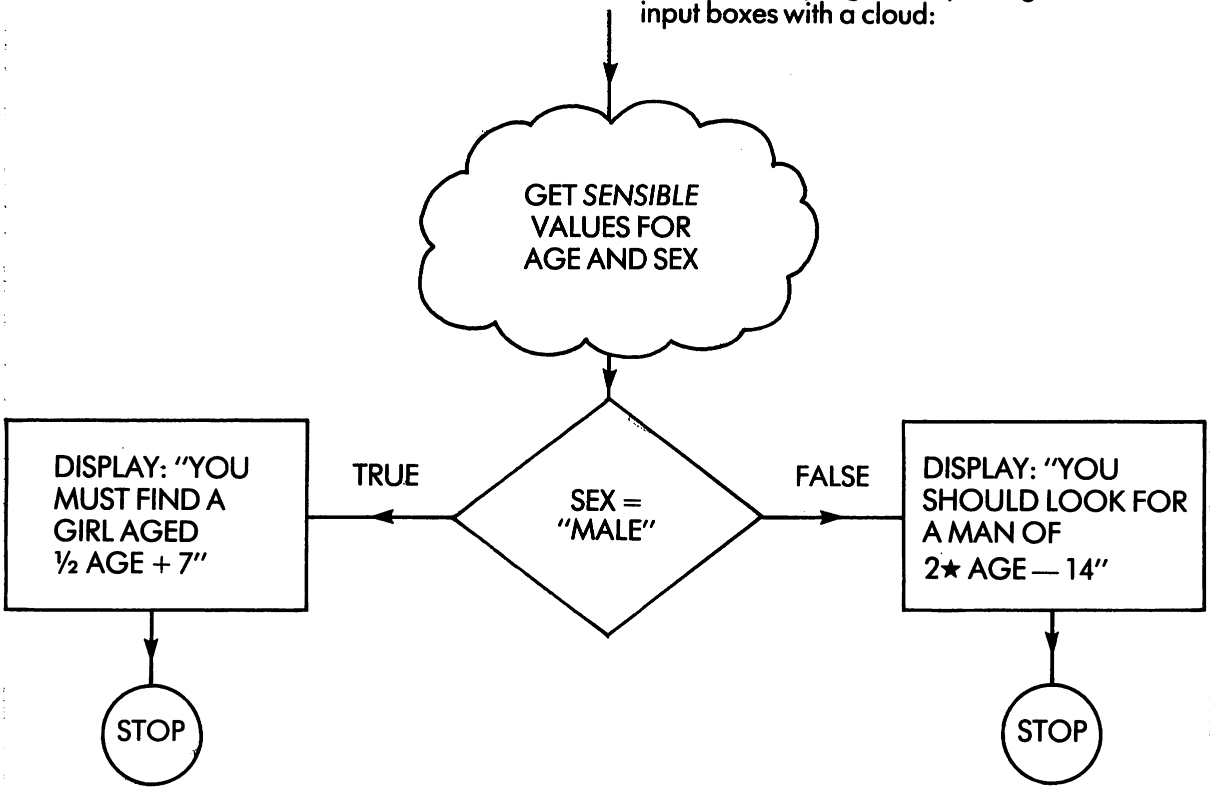
Experiment 11.1 Completed

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*.

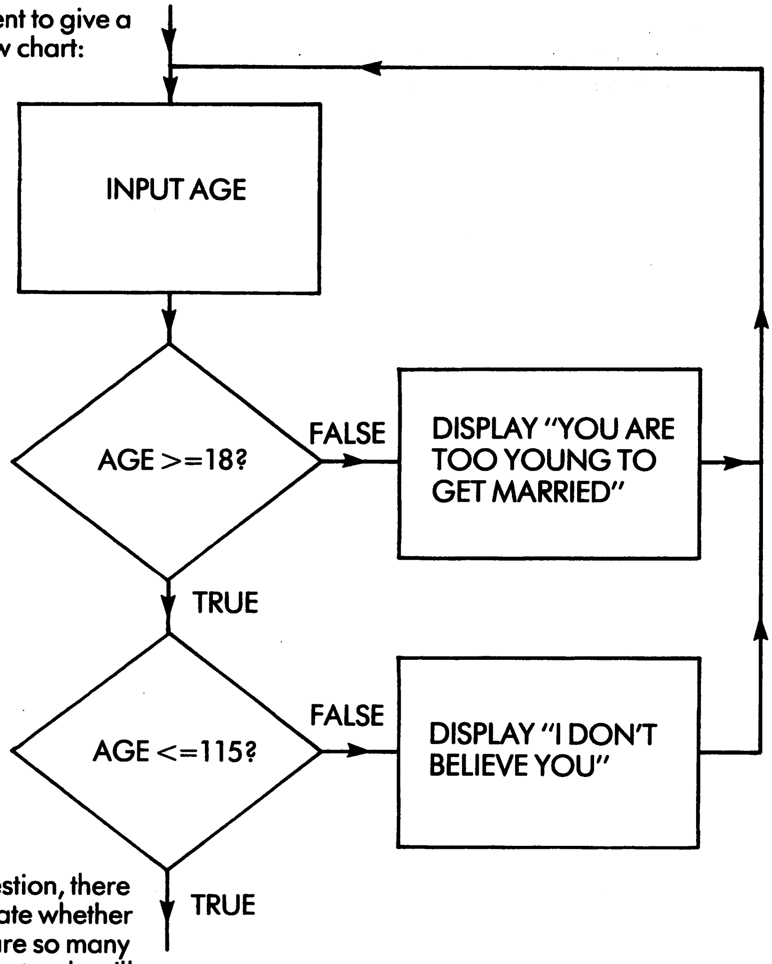
To begin with, we'll draw a new flow chart for the whole program, replacing the detailed input boxes with a cloud:



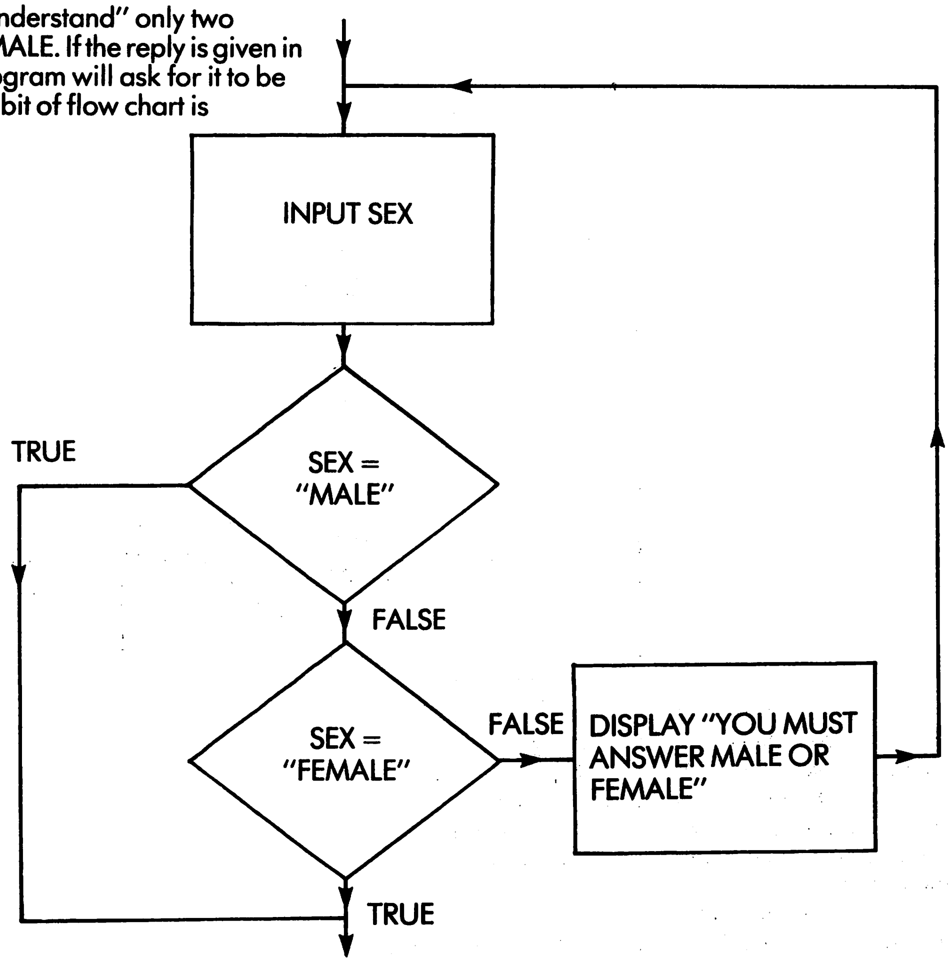
We use a cloud because we haven't yet fixed the details of what we actually mean by "sensible". The cloud is useful because it allows us to plan the program as a whole unit, but it involves an obligation to work out the action in greater detail. At the stage we have reached now the planning is not complete,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

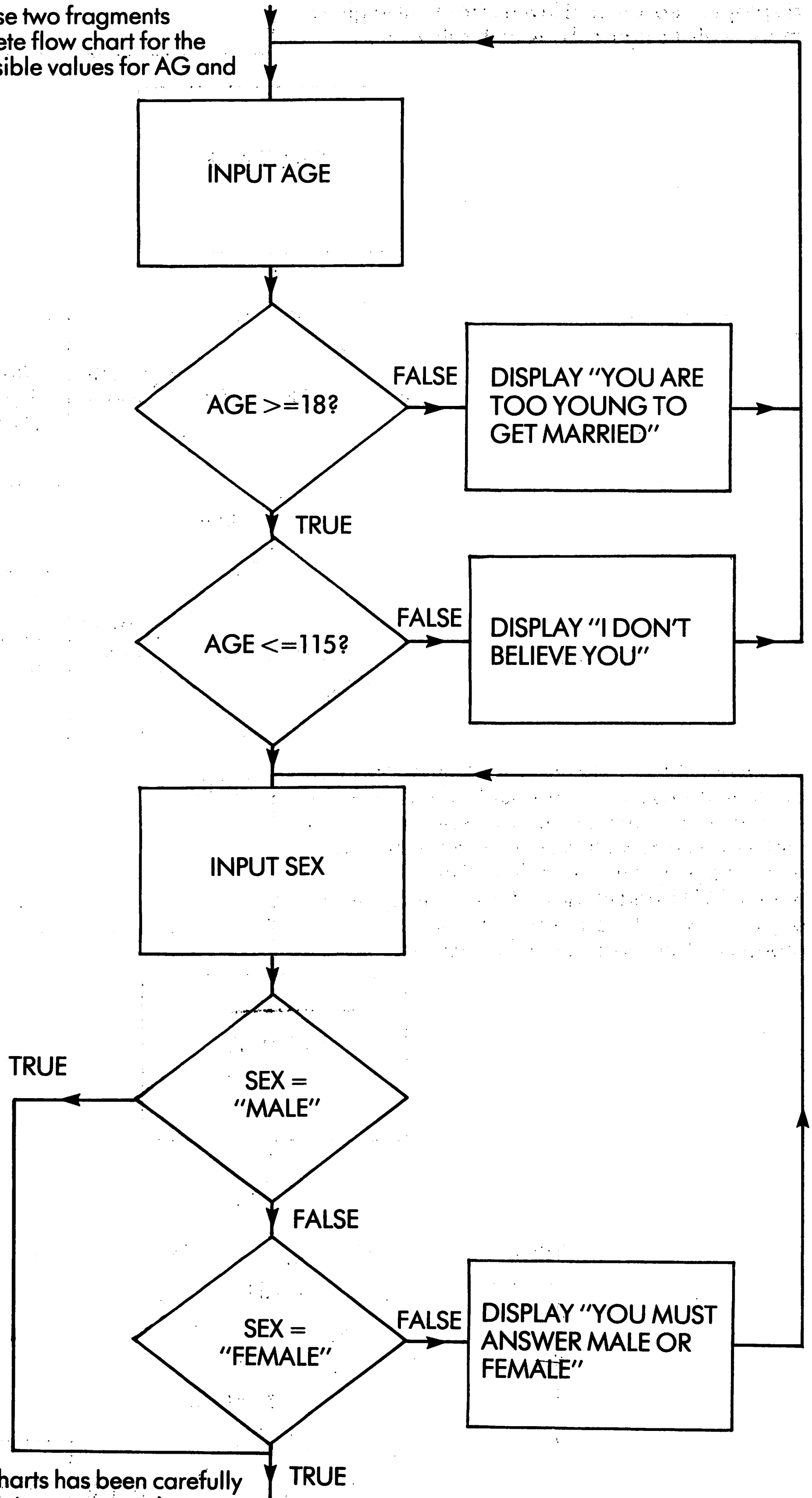
We'll design the program so that when the computer asks for the client's age, it decides whether to accept it as reasonable. If not, it displays a *reason*, and invites the client to give a more realistic figure. Look at this flow chart:



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$.



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 VIC 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 VIC, 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 (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, uncomprehensible 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.

90

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:

1Ø J=4	1Ø FOR J=4 TO 2Ø STEP 2
2Ø PRINT J,J★7	2Ø PRINT J, J★7
3Ø J=J+2	3Ø NEXT J
4Ø IF J<22 THEN 2Ø	
(Using IF . . . . THEN)	(Using FOR . . . . NEXT)

In both cases:

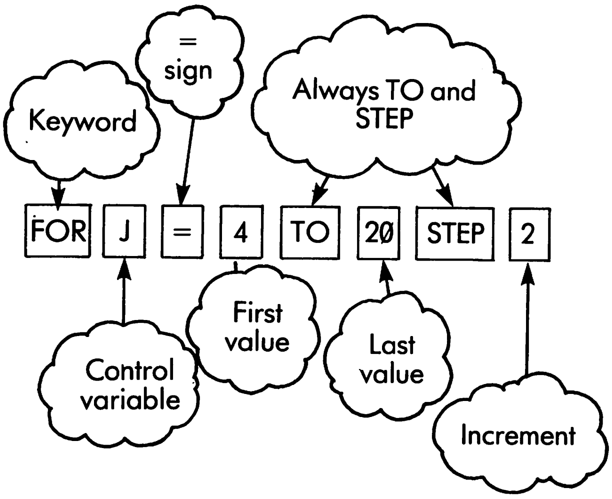
Control variable is J

First value is 4

Last value is 2Ø

Increment is 2

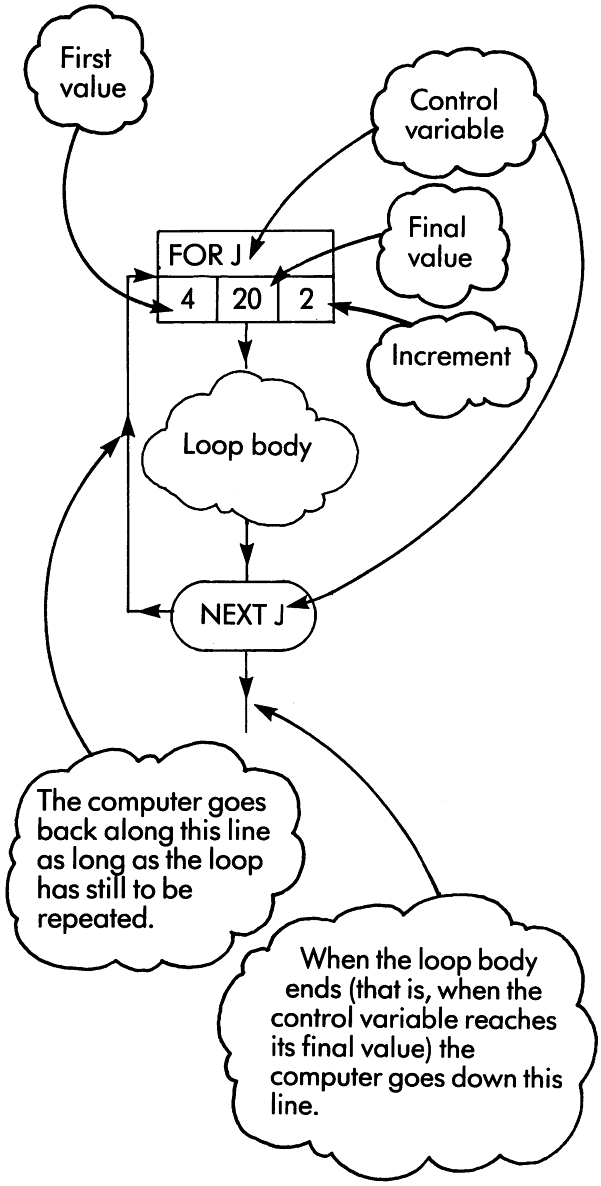
The example shows how the FOR command is built up



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:



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 VIC 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 VIC 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 step size is 1, the "STEP 1" at the end of the FOR command can be left off. The VIC 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
```

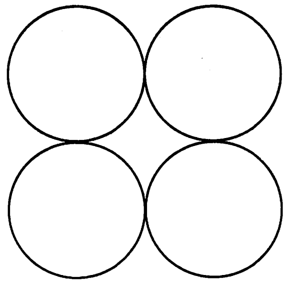
(ii)
```
10 FOR B=3 TO 0 STEP −1
20 PRINT B;
30 NEXT B
40 STOP
```

(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)

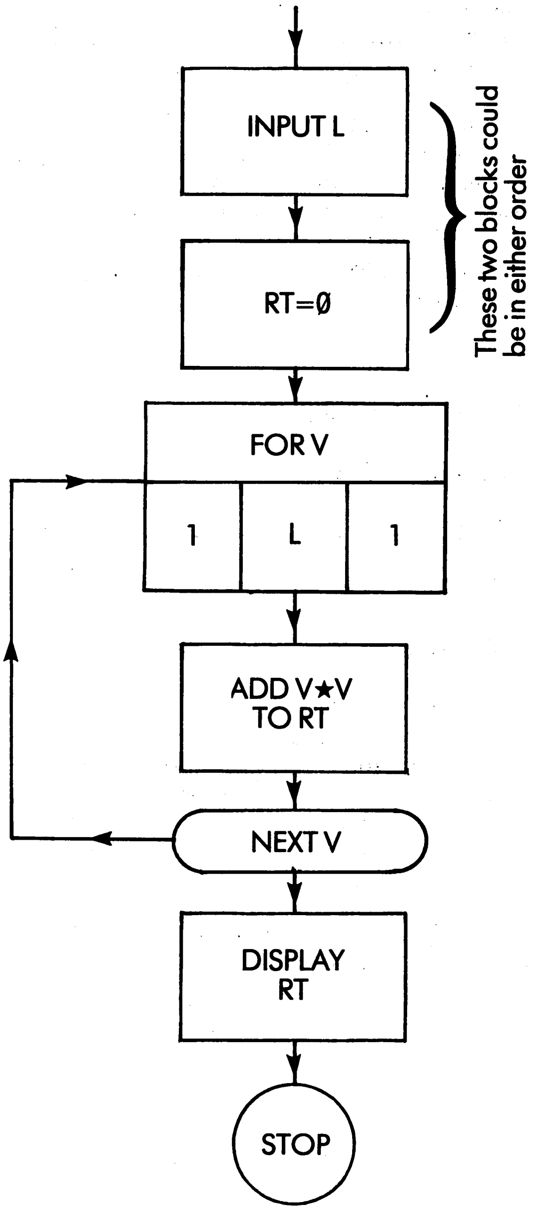
or after the loop has ended.

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
SCORE? 3Ø
SCORE? 7Ø
AVERAGE = 4Ø
```

Numbers typed by user

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 VIC 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 11p

and One hundred and seventeen at 14p

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
  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
```

Numbers typed by user

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

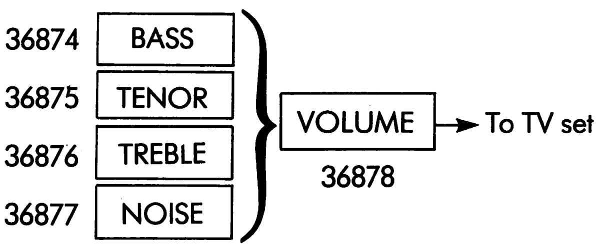
EXPERIMENT 13·1

This unit is about a topic which is both easy and fun: using the VIC to make sounds and musical notes.

Load the program entitled SOUND DEMO, turn up the volume control on your TV set, and play through the selection of sound effects the program provides.

As you will hear the sounds that can be made are wide ranging and give an indication of what the VIC can do. Undoubtedly you will want to design your own sounds, and this is what the rest of this unit is all about.

The sound production unit on the VIC is controlled by POKE commands. The unit has five special addresses, arranged like this:



On the left are four electronic 'voices'. Each of them except 'noise' will sing a clear musical note if a number is POKE'd into the right address. For instance, the treble voice will start singing middle C when you give the command

POKE 36876, 131

The pitch of the note depends on the number you poke. 128 gives the lowest note, and 254 the highest. The notes are *not* equally spaced, and the stave at the foot of the page shows how the pitches correspond to the notes on the piano.

To stop the treble voice, you command it

POKE 36876, Ø

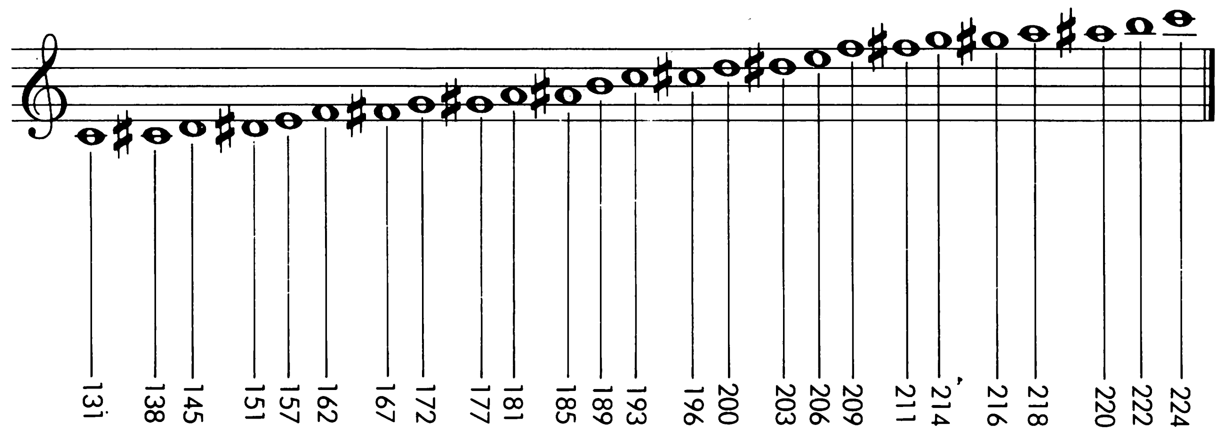
(although POKE'ing any number between Ø and 127 will have the same effect).

On the right of the diagram there is a mixer and volume control. It combines the sounds made by the four voices and sends them to the loudspeaker in the TV set. You can adjust the loudness of the sound by POKE'ing numbers into the volume control address:

POKE 36878, 15	is fortissimo, or full blast
POKE 36878, 1	is pianissimo or very quiet
POKE 36878, Ø	gives silence (even if the voices themselves are still singing).

Numbers between 1 and 15 give increasing degrees of loudness.

Note that the VIC's volume control is quite separate from the knob or slider on your TV set. The TV control should be set when you first switch on, and then left alone, leaving the VIC to change the loudness of the sound according to the program it is running.



Treble Voice

Now you know how to make the VIC produce any note, with any degree of loudness. For instance, this program will make it play a G at about half-volume:

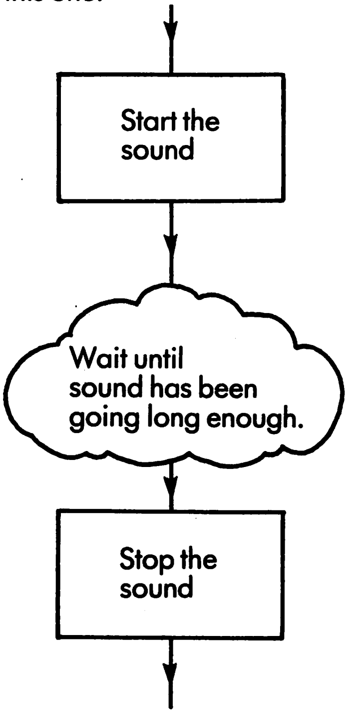
```
10 POKE 36876, 172

20 POKE 36878, 8
```

If you run this program you will see (or rather hear) a serious difficulty: the note continues to sound even when the program has ended and the READY message has appeared. To reduce the volume to zero type the command

```
POKE 36878, 0
```

A different way of stopping the sound is to hold down RUN STOP and strike RESTORE.

This experiment brings us to another vital point. One of the most important features of a sound is how long it lasts. A program running in the VIC must *time* the sound it makes, using a flow chart like this one:



A simple way to make the computer keep time is to make it obey an 'idle loop': that is, a loop without a body. While going round this loop, the VIC spends all its time counting and testing. Roughly speaking, the machine can get round an idle loop 1000 times every second, so that a one-second wait could be written as

```
FOR M = 1 TO 1000
NEXT M
```

note: *no* loop body

The control variable (in this case M) has been picked at random. In practice you can use any name, as long as it is different from the other names in your program.

Notes lasting any length of time can easily be programmed by choosing the right final value for the idle loop: 3000 for 3 seconds, 500 for half a second and so on.

Let's write a program which sounds a 'pip' like the ones on the radio. The right pitch for the treble voice is about 235, and the pip itself lasts for 1/10 of a second. Turning our flow chart into code we get

```
10 POKE 36878, 15

20 POKE 36876, 235

30 FOR M=1 TO 100

40 NEXT M

50 POKE 36876, 0
```

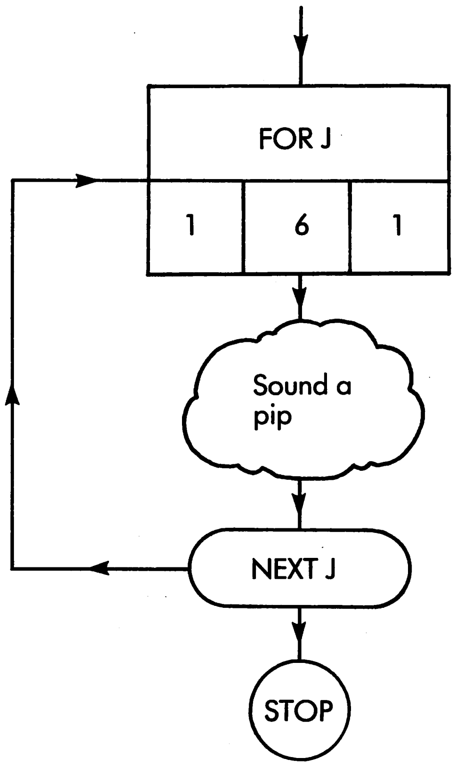
Type NEW and enter this program into the VIC. When you run it, the program will give a single 'pip' and then stop.

Next, you can modify the program and get it to give out a whole chain of pips one after the other. Basically you could make the program into a loop by adding a GOTO to jump from the end back to the beginning, but this by itself wouldn't be enough. The pips would come so close to each other that they would sound almost like a continuous note. To get the right pattern of sounds you must make the program wait in *silence* before going back. Each pip lasts one tenth of a second; so to get a pip every second, the right time to wait is *nine* tenths. Add the following commands to your program and try it out:

```
60 FOR Q=1 TO 900

70 NEXT Q

80 GOTO 20
```

When you've got this program running, try some experiments: change the pitch of the note, and the timing of both the pip and the silence, and see the effects for yourself. Finally, change the program back to its original form.

Now let's think about a very similar problem: how to get a program to give out a given number of pips (say 6) and then stop. If you take away the GOTO at line 80, you already have a program which sounds one pip, followed by a silence of the right length. To get 6 pips, all you need do is to make this program into the body of a loop which is obeyed six times. You get



The conversion is trivial:

5 FOR J=1 TO 6 (at the beginning)

and 80 NEXT J (at the end)

90 STOP

Try it!

Now list your program. You will see that inside the *body* of the main loop (with control variable J) there are two *inner* loops (with control variables M and Q). This means that the whole of each inner loop, 100 times round for M, and 900 times round for Q, is obeyed for *every* value of J in the outer loop. During the execution of the entire program (6 pips) the VIC goes round the M loop 6 x 100 or 600 times, and round the Q loop 6 x 900 or 5400 times.

This is our first example of a loop inside another loop. These 'nested loops' are extremely common, and they are easy to use if you remember to choose different control variables for each one.

You get interesting effects if you change either the pitch of a note or its volume while it is being sounded.

Try the following program:

NEW

RUN STOP and RESTORE

```
10 POKE 36878, 15
20 FOR C=180 TO 220
30 POKE 36876, C
40 NEXT C
50 POKE 36878, 0
```

As this program goes round its loop, 41 times in all, a higher number is POKE'd into the treble voice each time. The effect is to give a sound rising steadily in pitch. This sound could be the basis of a police siren, or a space-ship taking off. The trouble is that the sound is over much too quickly; you might like to slow it down and make it last a little longer.

One good way of stretching out the sound is to put a *short* idle loop into the main one. Try inserting

```
33 FOR M=1 TO 20
36 NEXT M
```

You now have a basic program for a 'rising tone'. You can 'tune' it in all sorts of ways by changing the starting and ending value for C, or the final value for M. For example, if you alter line 20 to read

```
20 FOR C = 130 TO 170
```

you'll get a howl of much lower pitch. If you put

```
33 FOR M = 1 TO 200
```

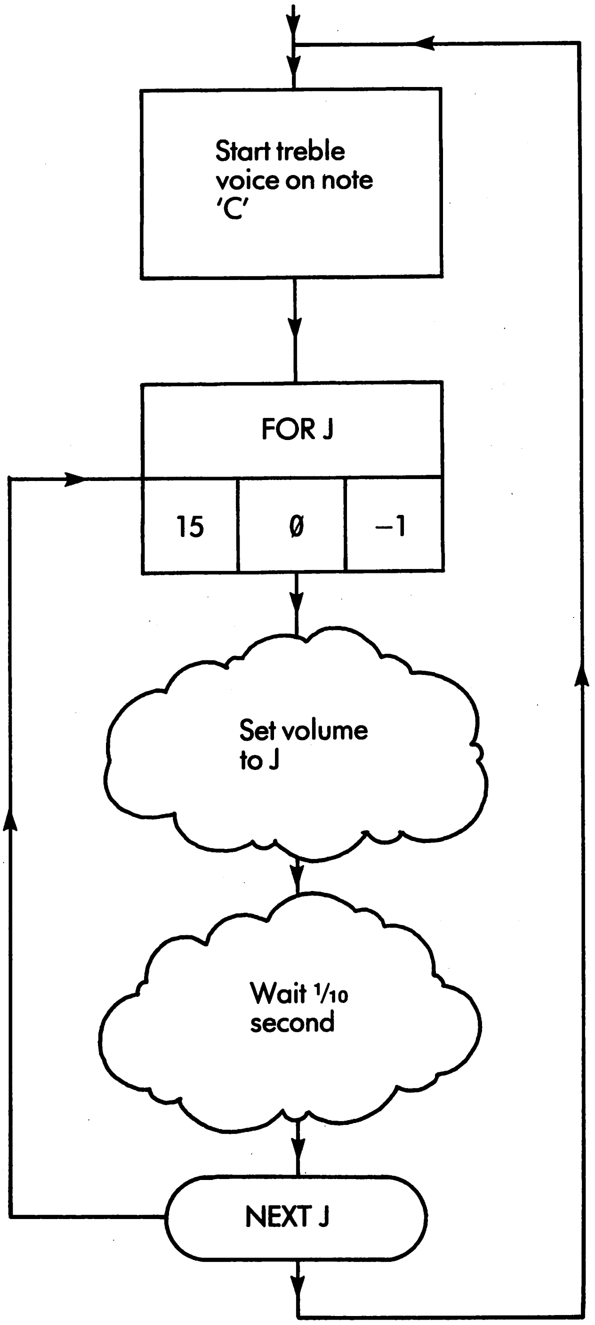
your tone will last about 10 times as long. Finally, if you use a negative step size, like this:

```
20 FOR C = 200 TO 150 STEP -1
```

the machine will sound a falling note instead of a rising one.

The effect of changing the *volume* of a note depends very much on how quickly it happens. A very slow change sounds like something approaching or going away, but a rapid reduction from full volume can make a note sound like a plucked instrument — a guitar, harp or harpsichord (but not a violin or 'cello).

Here is a program to let you try out the effect for yourself. Its flow chart is:



and the program itself is:

```
10 POKE 36876, 193
20 FOR J = 15 TO 0 STEP -1
30 POKE 36878, J
40 FOR M = 1 TO 100
50 NEXT M
60 NEXT J
70 GOTO 10
```

Now try your own hand at programming dramatic sounds. See how well you can imitate

(a) A fire engine

(b) A police siren

When you have done as well as you can, look up Appendix B and compare your answers to the programs you find there.

Experiment 13.1 Completed	

EXPERIMENT 13·2

So far we have only used the treble voice (address 36876). This section is about the other three voices.

The *tenor* voice (36875) sounds the same notes as the treble, but one *octave* lower. (An octave, as you will know, is a span of 8 white notes on the piano.) The bass voice (36874) is an octave lower again, and can be used to make low growling sounds which suggest robots, haunted houses, and so on. Try this program and see if you can work out why the pitch goes up and the pips come faster and faster.

```
10 FOR J = 130 TO 240 STEP 3
20 POKE 36874, J
30 POKE 36878, 15
40 FOR M = 1 TO 100
50 NEXT M
60 POKE 36878, 0
70 FOR Q = 1 TO 350—J
80 NEXT Q
90 NEXT J
100 STOP
```

You can play more than one voice at a time, but they blend best if they are each made to sing notes an octave apart.

The fourth voice is used to produce electronic *noise*, to imitate rockets or explosions. The 'pitch' of the noise depends on the number you poke into address 36877. It is low like a distant charge of dynamite if the number is small (say between 130 and 140). If you poke a high number (such as 250) the noise sounds much more like a jet engine.

Noises often sound more effective if they start loud and fade away. This can be arranged by starting the volume at 15 and bringing it down gradually.

Type NEW, RUN STOP and RESTORE, and try this program:

```
10 FOR K = 130 TO 250 STEP 5
20 POKE 36877, K
30 FOR J = 15 TO 0 STEP -1
40 POKE 36878, J
50 FOR M = 1 TO 100
60 NEXT M
70 NEXT J
80 NEXT K
90 STOP
```

It should give you a good idea of what the various pitches of noise sound like. This program has *three* nested loops, but that shouldn't worry you:

The outer loop (controlled by K) makes the machine work through the pitches 130, 135, 140 . . . . . . 250.

The middle loop (controlled by J) makes the program reduce the volume of each sound by working down from 15 to 0.

The inner loop (controlled by M) simply gets the machine to wait 1/10 second between each change of volume.

When you have run this program several times, insert a new command:

```
55 PRINT M;
```

The machine now displays every value of M, every time round the inner loop. This slows everything down quite a lot, and gives you an idea of how much is really going on!

To end this unit, experiment with the VIC sounds. Then choose one of the following and imitate it as well as you can:

(a) Jet flying overhead

(b) Ship's klaxon

(c) Horn on railway engine

(d) Morse code letter V (dot-dot-dot dash)

Write down your program for future use.

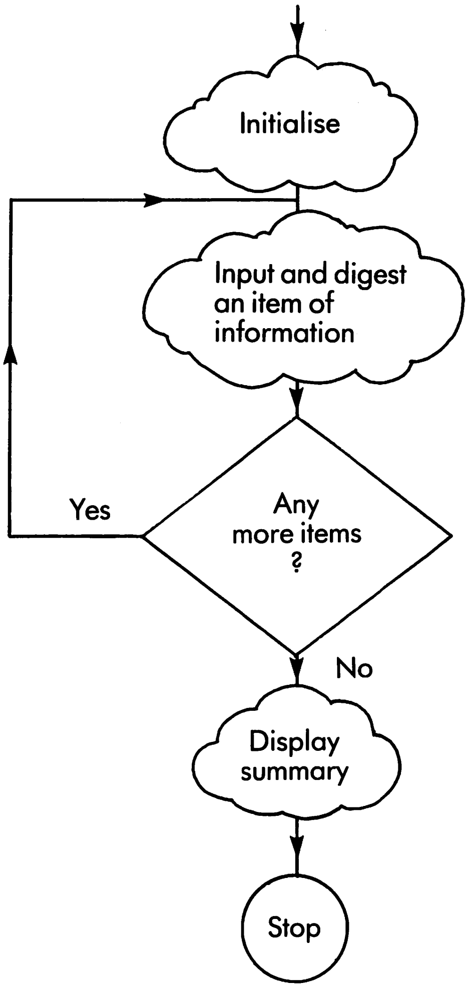
Experiment 13.2 Completed	

If you can use a musical keyboard, the PIANO program will let you play a novel musical instrument. Use the top two rows of keys on the keyboard to play the notes.

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 1Ø numbers and finds their average value:

```
1Ø S=Ø                  } Initialise
2Ø P=1

3Ø INPUT X              } Read and digest an item
4Ø S=S+X

5Ø P=P+1                } Any more items?
6Ø IF P < 11 THEN 3Ø

7Ø PRINT"AVERAGE="; S/1Ø } Display summary

8Ø 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, Ø, 9, 8, 3, 12, 1Ø.

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:

```
1Ø S = Ø

2Ø FOR P = 1 TO 1Ø

3Ø INPUT X

4Ø S = S+X

5Ø NEXT P

6Ø PRINT "AVERAGE IS"; S/1Ø

7Ø 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 2Ø 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
```

Note use of N instead of 10

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:

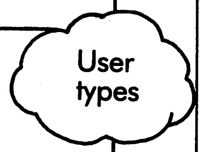
```
NEXT NUMBER? 4
ANY MORE NUMBERS? YES
NEXT NUMBER? 7
ANY MORE NUMBERS? YES
NEXT NUMBER? 10
ANY MORE NUMBERS? NO
AVERAGE IS  7
BREAK IN 90
READY
```

User types

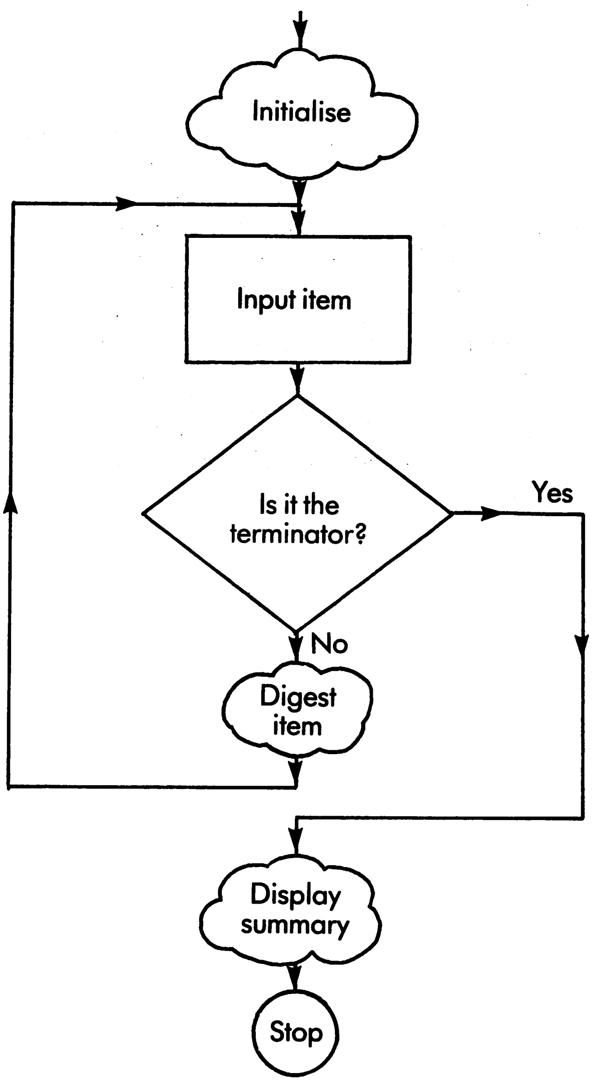
The drawbacks of this scheme are clear. The unfortunate user has to keep typing YES after every number except the last. This takes double the time, and doubles the risk of mistakes. A better method is to mark the end of the stream of items with a special value called a *terminator*. A good choice for 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

Typed by user

(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

Typed by user

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 VIC 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 490 HEADS

AND 57 TAILS

READY.
```

Typed by user

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 109, 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

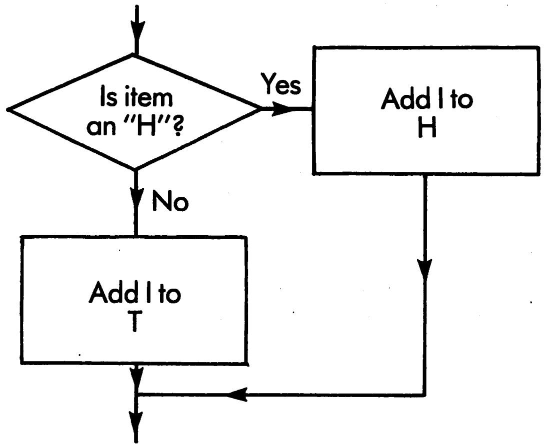
(As usual, the names H and T are freely chosen.)

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:

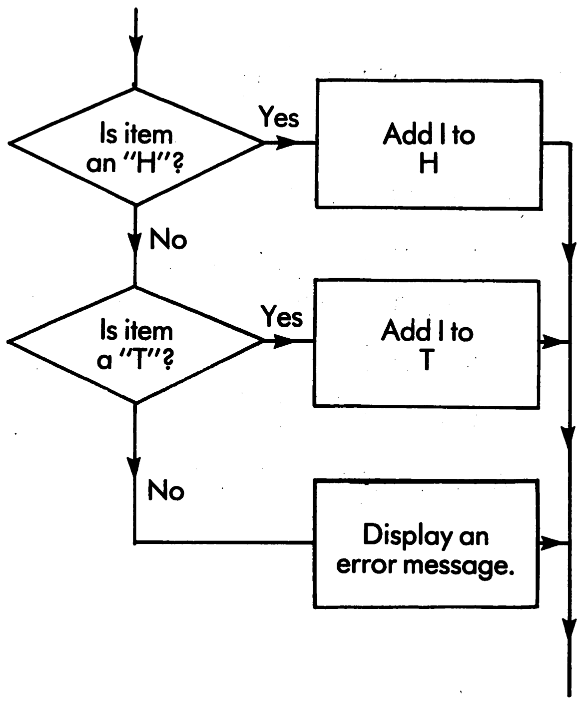
- Set variables H and T to zero
- Display the heading message.

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



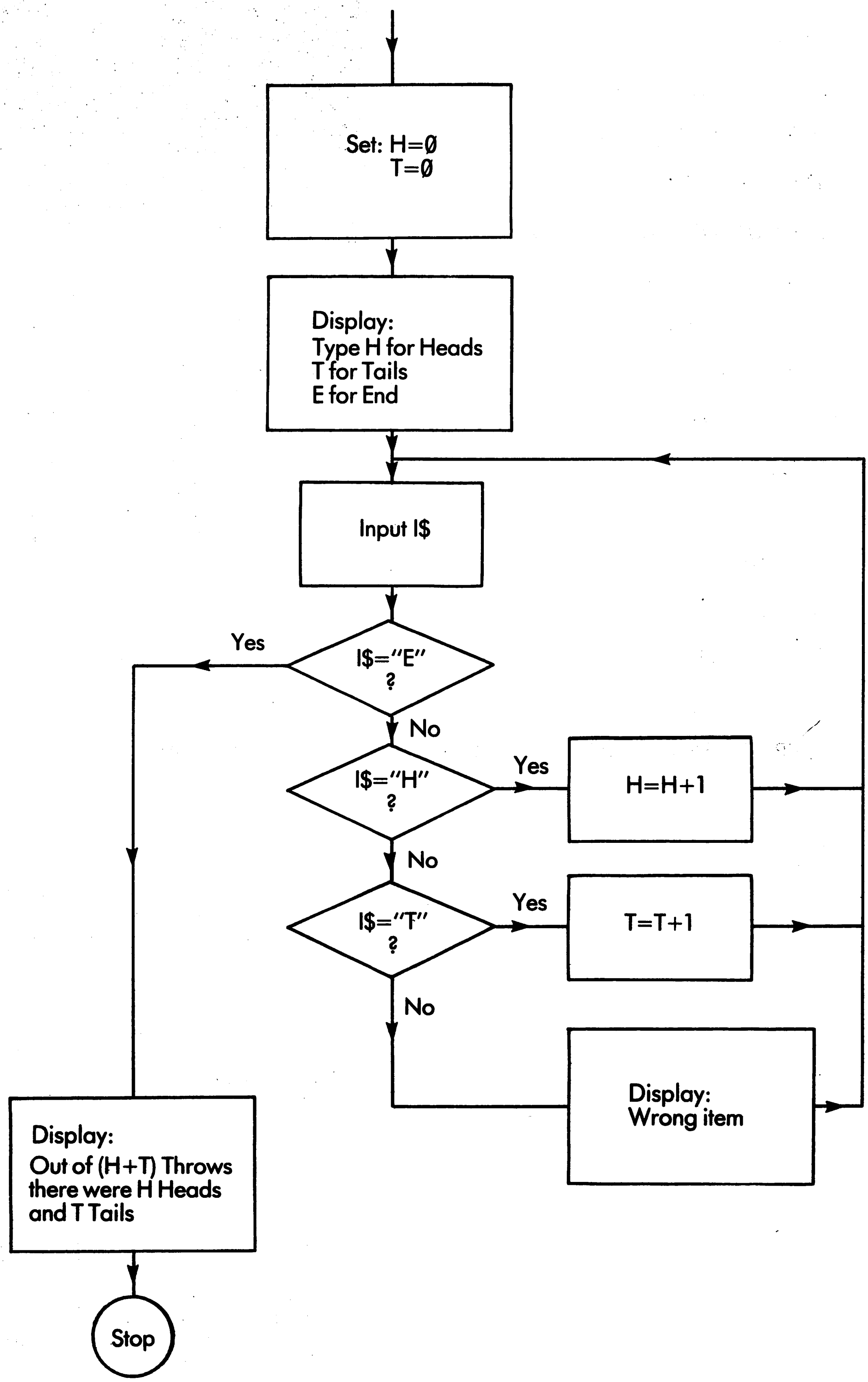
In practice, this method would never be used by a good professional programmer, because it doesn't allow for the user's typing mistakes. If the user hits a J instead of an H (they are next to each other on the keyboard) the program would count it as a T, which is most unlikely to have been what the user 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:



112

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=Ø
 20 T=Ø
 30 PRINT "TYPE H FOR HEADS"
 40 PRINT "T FOR TAILS"
 50 PRINT "E FOR END"
 60 INPUT "NEXT THROW"; I$
 70 IF I$="E" THEN 16Ø
 80 IF I$="H" THEN 12Ø
 90 IF I$="T" THEN 14Ø
100 PRINT "WRONG ITEM"
110 GOTO 6Ø
120 H=H+1
130 GOTO 6Ø
140 T=T+1
150 GOTO 6Ø
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. It is a good idea to make the program react with sounds as well as displayed messages. You could, for instance, use a cheerful '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 12Ø and 13Ø. We could insert a suitable noise by adding the commands:

```
121 POKE 36878, 15
122 POKE 36876, 23Ø
123 FOR M = 1 TO 1ØØ
124 NEXT M
125 POKE 36876, Ø
126 POKE 36878, Ø
```

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 Ø, 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. 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.

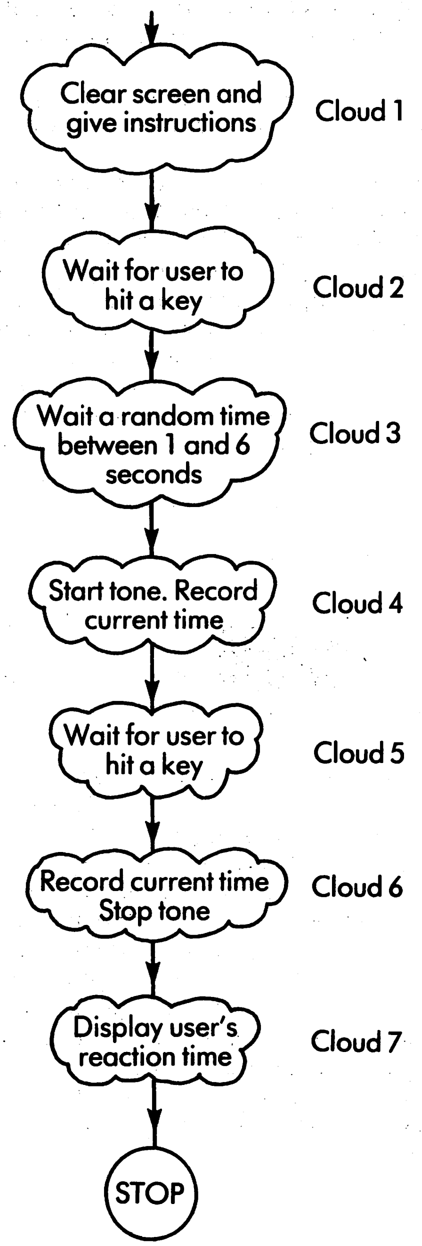
You may notice three aspects of the program which are not immediately obvious:

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.

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:



Cloud 1

```
10 REM REACTION TIME PROGRAM
20 PRINT " SHIFT and CLR HOME "
30 PRINT "TO MEASURE YOUR"
40 PRINT "REACTION TIME:"
50 PRINT "HIT ANY KEY"
60 PRINT "THEN WAIT FOR THE"
70 PRINT "TONE. WHEN YOU"
80 PRINT "HEAR IT, HIT ANY"
90 PRINT "KEY AS FAST AS"
100 PRINT "YOU CAN. GOOD LUCK!"
```

Cloud 2

```
110 REM WAIT FOR ANY KEY
120 GET A$
130 IF A$ = "" THEN 120
```

Cloud 3 (and see below)

```
140 REM WAIT A RANDOM TIME
143 PRINT
145 PRINT "WAIT FOR IT!"
148 PRINT
150 Q=TI+INT(60+301★RND(0))
160 GET A$
170 IF A$<>"" THEN 340
180 IF TI<Q THEN 160
```

Cloud 4

```
190 REM START TONE AND NOTE TIME
200 POKE 36876, 225
210 POKE 36878, 15
220 X=TI
```

Cloud 5

```
230 REM WAIT FOR ANY KEY
240 GET A$
250 IF A$= "" THEN 240
```

Cloud 6

```
260 REM GET RESULT AND STOP TONE
270 R=TI
280 POKE 36876, 0
290 POKE 36878, 0
```

Cloud 7

```
300 REM DISPLAY RESULT
310 PRINT "YOUR REACTION TIME IS"
320 PRINT (R—X)/60 ; "SECONDS"
```

```
330 STOP
```

Part of Cloud 3

```
340 PRINT "TOO SOON"
350 STOP
```

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 1Ø to 1ØØ) consists entirely of PRINT commands and is quite straightforward.

The second cloud, lines 11Ø to 13Ø, 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* string variable name. The names of number variables are not allowed. For example

GET X$	but	GET Q
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 string 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. This is a string with *no* characters, and is normally written as " ".
To illustrate this rule, imagine that we start off the computer on the following looped program, and watch what happens inside the machine:

```
1Ø GET X$

2Ø GOTO 1Ø
```

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 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 12Ø and 13Ø in the REACTION program. Command 12Ø examines the keyboard and delivers a string in A$ which is null unless a key has been pressed. Command 13Ø tests A$, and makes the computer loop back to 12Ø until the user types any key, at which the program is allowed to drop through to line 14Ø.

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 function 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 (Ø) it gets a different value somewhere between Ø and 1.

In most practical cases, we don't need a random *fraction* between Ø 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 Ø and 36.

To get a whole number in any specified range, we use a slightly different expression:

INT(x + y ★ RND(Ø))

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

Lowest number expected

INT (1 + 6 ★ RND(Ø))

Or, for a number between Ø and 36

INT (Ø + 37 ★ RND(Ø))

Number of different values

**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 12Ø throws of a die:

```
NEW
1Ø FOR J = 1 TO 12Ø
2Ø S = INT(1+6★ RND(Ø))
3Ø PRINT S;
4Ø NEXT J
5Ø 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$, which keeps time in hours, minutes and seconds; but the special variable TI (which is not a string but a number) is intended to measure much shorter periods of time. TI is set to zero when the VIC is started up, and from then on, no matter what else happens, it has 1 added to it every 60th of a second. This interval, a sixtieth of a second, is called one "jiffy". You can get the current value of the internal timer at any time in jiffies by using the name TI in an expression; but you can't alter the value in the way you can set TI$.

Give the command

PRINT TI

The machine will respond by displaying a fairly large number (6Ø★6Ø or 36ØØ 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 VIC 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 ★ 6Ø or 3ØØ *more* than it was at the beginning. This is true whether the machine has been switched on for 5 seconds or 5 years.

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★36ØØ 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/6Ø	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 5 seconds. This is between 6Ø and 3ØØ jiffies, to be decided by the *machine* in an unpredictable way. The appropriate expression is

INT(6Ø+3Ø1★RND(Ø))

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 15Ø makes this prediction and records the value in Q.

If this were all that were needed, the entire cloud could read:

```
150 Q = TI+INT(60+301★RND(0))

160 IF TI<Q THEN 160
```

As it is, we have to check that the user doesn't hit a key before the tone is sounded. Commands 16Ø, 17Ø, 34Ø and 35Ø 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 24Ø and 25Ø 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.

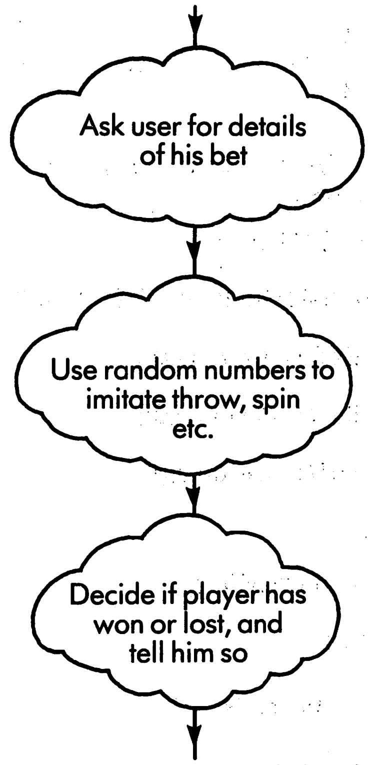
2. Write a program which imitates someone tossing a coin. Every time the user presses a key, the program displays either "HEADS" or "TAILS" at random.

Experiment 15.2 Completed	

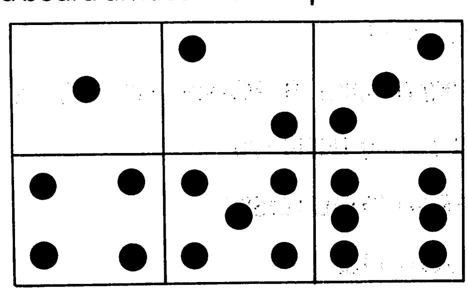
Now check your answers in Appendix B.

EXPERIMENT 15·3

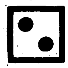
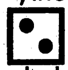
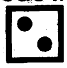
Random numbers are useful in programming games of chance, such as dice, fruit machines, and so on. All these programs follow the same basic pattern, which for one 'throw' or 'spin' is like this



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

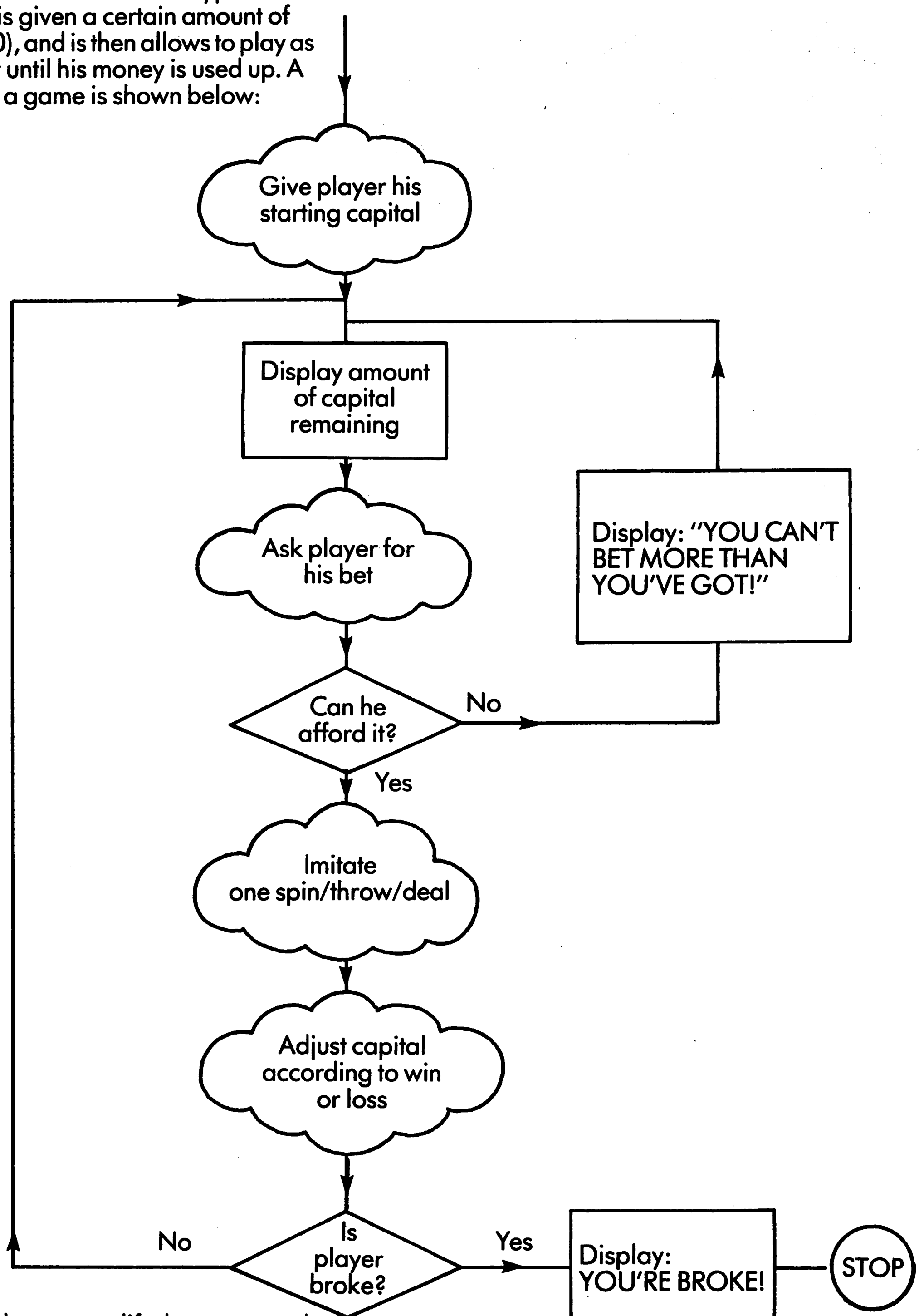
D1
D2 } Results of throwing 3 dice
D3

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 VIC 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 allows to play as long as he likes, or until his money is used up. A flow chart for such a game is shown below:



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	

Appendix B contains a 'craps' program for you to try out.

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. 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 VIC Computing are worth looking at. Books on programming are also worth reading, even if they don't refer specifically to the VIC.

(b) Join a local computer club. There are VIC user groups being set up all over the country and details are given in the magazine VIC 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

VIC is a computer 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 VIC. 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 = <u>34</u>

B = <u>B+1</u>

C = <u>((X + Y) − 34.7/(Q−3))★(Z−3)↑2</u>

the underlined portions are all expressions which the VIC 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 VIC 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 VIC 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 VIC 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 VIC 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 ↑ Ø		
PRINT (P ↑ 2 — X ↑ 2) / 3		

Now check your results on the VIC. 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 Ø and 2, going up in steps of Ø.2

(Hint: use a FOR loop like this:

```
FOR X = Ø TO 2 STEP Ø.2
.....
NEXT X
```
)

(The actual answers are given at the back of Appendix B.)

2 Standard Functions

Like most calculators, the VIC 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 these:

```
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 = π/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

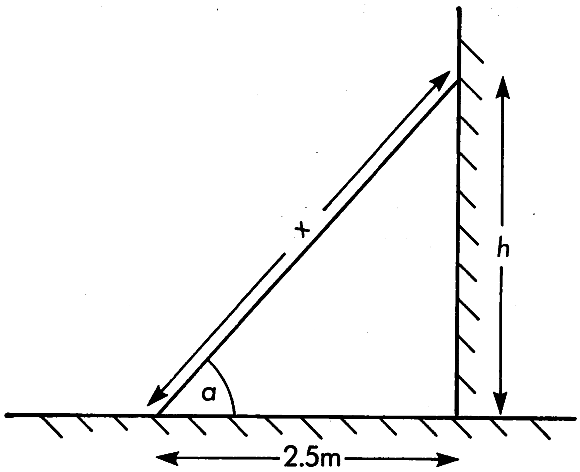
Now check your answer in Appendix B.

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.



$$h = \sqrt{x^2 - (2.5)^2} \quad \text{(by Pythagoras)}$$

$$a = \text{arc tan}\,(h/2.5) \quad \text{(in radians)}$$

or $a = (180/\pi) \star \text{arc tan}\,(h/2.5)$ 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 BREAK IN 6Ø READY	1Ø X=5 2Ø Y=7 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 SHE LOVES ME NOT SHE LOVES ME NOT BREAK IN 6Ø READY	1Ø Q=1 2Ø PRINT "SHE LOVES ME" 3Ø PRINT "SHE LOVES ME NOT" 4Ø Q=Q+1 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:

1Ø POKE 36879, 124

2Ø PRINT " [SHIFT] and [CLR HOME] ";

3Ø PRINT " [CLR HOME] [CRSR ⇑⇓] ←9 times→ [CRSR ⇑⇓]
[CRSR ⇐⇒] ←6 times→ [CRSR ⇐⇒]";

4Ø PRINT TI$

5Ø GOTO 3Ø

Experiment 9.3:

5 REM FLAG OF ICELAND
1Ø POKE 36879, 1Ø5

2Ø PRINT " [SHIFT] and [CLR HOME] ";

3Ø J=1

4Ø PRINT " [CTRL] and [RVS ON]
[CRSR ⇐⇒] ←6 times→ [CRSR ⇐⇒] [CTRL] and
[WHT] ←1 space→ [CTRL] and [RED]
←2 spaces→ [CTRL] and [WHT] 1 space"

5Ø J=J+1

6Ø IF J <1Ø THEN 4Ø

7Ø PRINT " [CTRL] and [RVS ON] [CTRL] and
[WHT] ← 7 spaces → [CTRL] and [RED]
←2 spaces→ [CTRL] and [WHT]
←13 spaces →";

8Ø J=1

9Ø PRINT " [CTRL] and [RVS ON] [CTRL] and
[RED] ←22 spaces→";

1ØØJ=J+1

11Ø IF J < 4 THEN 9Ø

12Ø PRINT " [CTRL] and [RVS ON] [CTRL] and
[WHT] ← 7 spaces → [CTRL] and [RED]
←2 spaces→ [CTRL] and [WHT]
←13 spaces→";

13Ø J=1

14Ø PRINT " [CTRL] and [RVS ON]
[CRSR ⇐⇒] ←6 times → [CRSR ⇐⇒]
[CTRL] and [WHT] ←1 space→
[CTRL] and [RED] ←2 spaces→
[CTRL] and [WHT] ←1 space→"

15Ø J=J+1

16Ø IF J<9 THEN 14Ø

17Ø PRINT " [CTRL] and [RVS ON]
[CRSR ⇐⇒] ←6 times → [CRSR ⇐⇒]
[CTRL] and [WHT] ←1 space→
[CTRL] and [RED] ←2 spaces→
[CTRL] and [WHT] ←1 space→";

18Ø GOTO 18Ø

UNIT:10

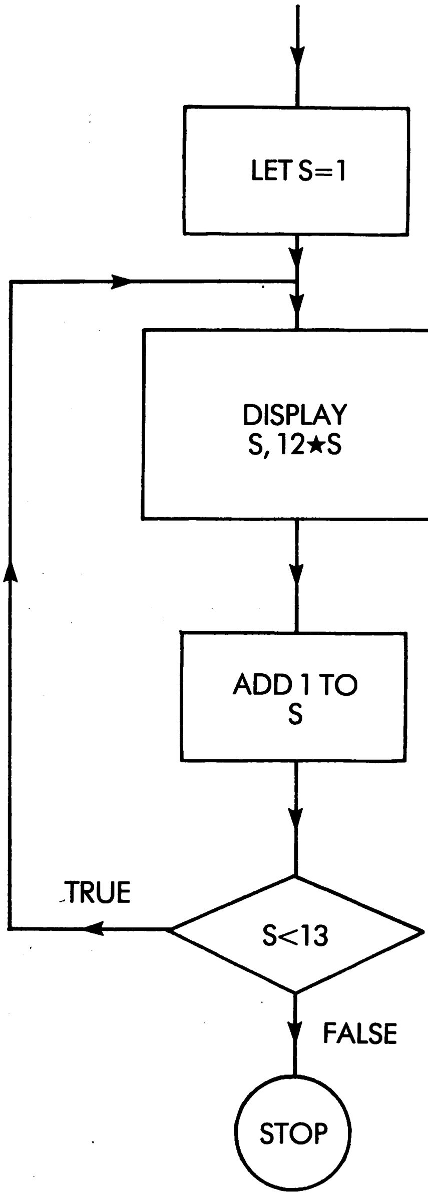
Experiment 10.2:

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
```

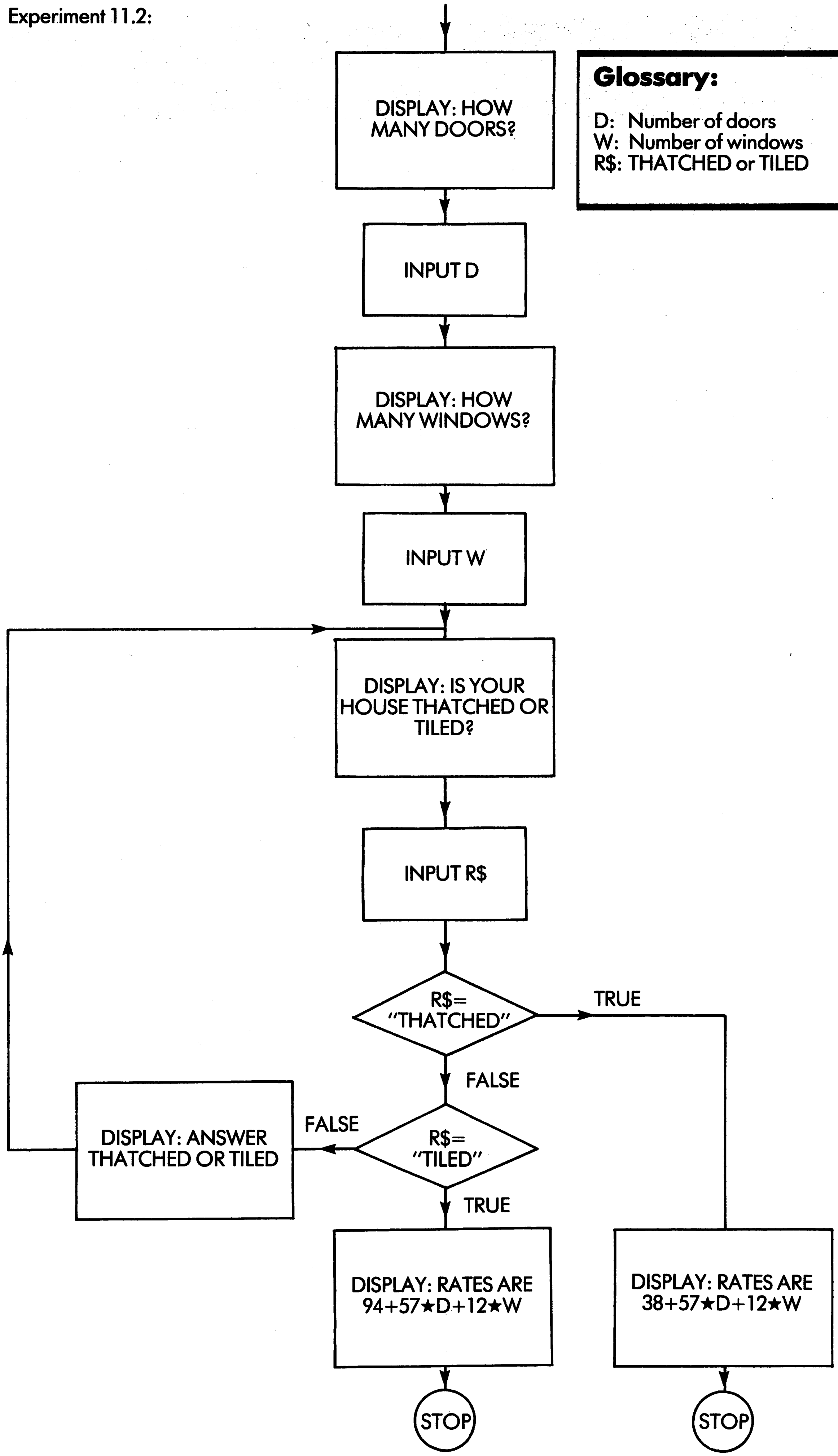
b)
```
10 PRINT "WHAT IS YOUR"
20 INPUT "SURNAME"; S$
30 PRINT "WHAT IS YOUR WIFE'S"
40 INPUT "CHRISTIAN NAME"; C$
50 PRINT "HER FULL NAME IS"
60 PRINT C$+"    "+S$
70 STOP
```

UNIT:11

Experiment 11.1:



Experiment 11.2:



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:

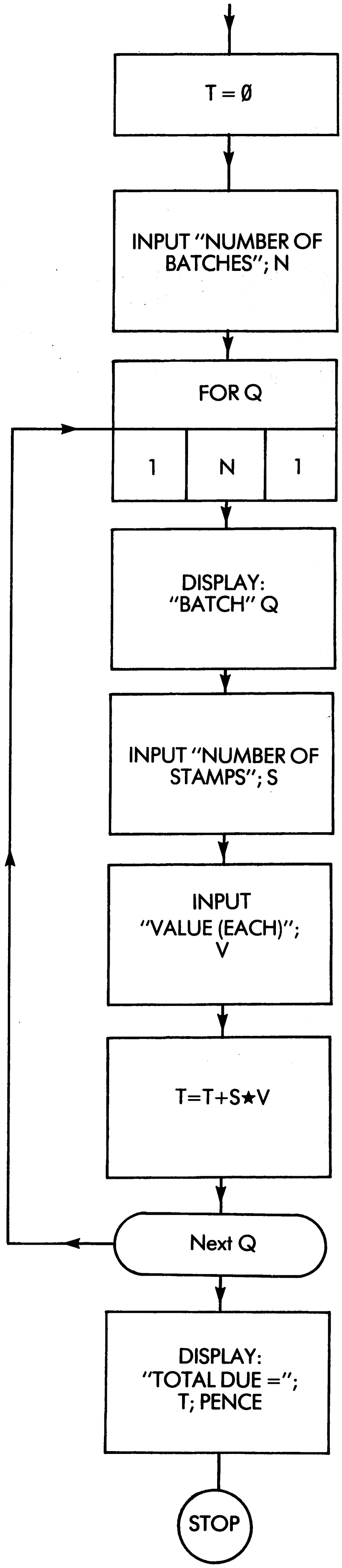
```
10 RS=0
20 INPUT "NUMBER OF INNINGS"; J
30 FOR Q = 1 TO J
40 INPUT "SCORE"; S
50 RS=RS+S
60 NEXT Q
70 PRINT "AVERAGE="; RS/J
80 STOP
```

either order is correct

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



```
 10 T=0
 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:13

141

Experiment 13.1:

```
 10 REM FIRE ENGINE
 20 POKE 36878, 15
 30 POKE 36876, 231
 40 POKE 36875, 232
 50 FOR M=1 TO 400
 60 NEXT M
 70 POKE 36876, 224
 80 POKE 36875, 223
 90 FOR M=1 TO 400
100 NEXT M
110 GOTO 30
```

```
 10 REM POLICE SIREN
 20 FOR J=0 TO 15
 30 POKE 36878, 3+J/2
 40 POKE 36876, 200+J
 50 FOR K=1 TO 20
 60 NEXT K
 70 NEXT J
 80 FOR J=1 TO 100
 90 NEXT J
100 FOR J=15 TO 0 STEP —1
110 POKE 36878, 3+J/2
120 POKE 36876, 200+J
130 FOR K=1 TO 20
140 NEXT K
150 NEXT J
160 GOTO 20
```

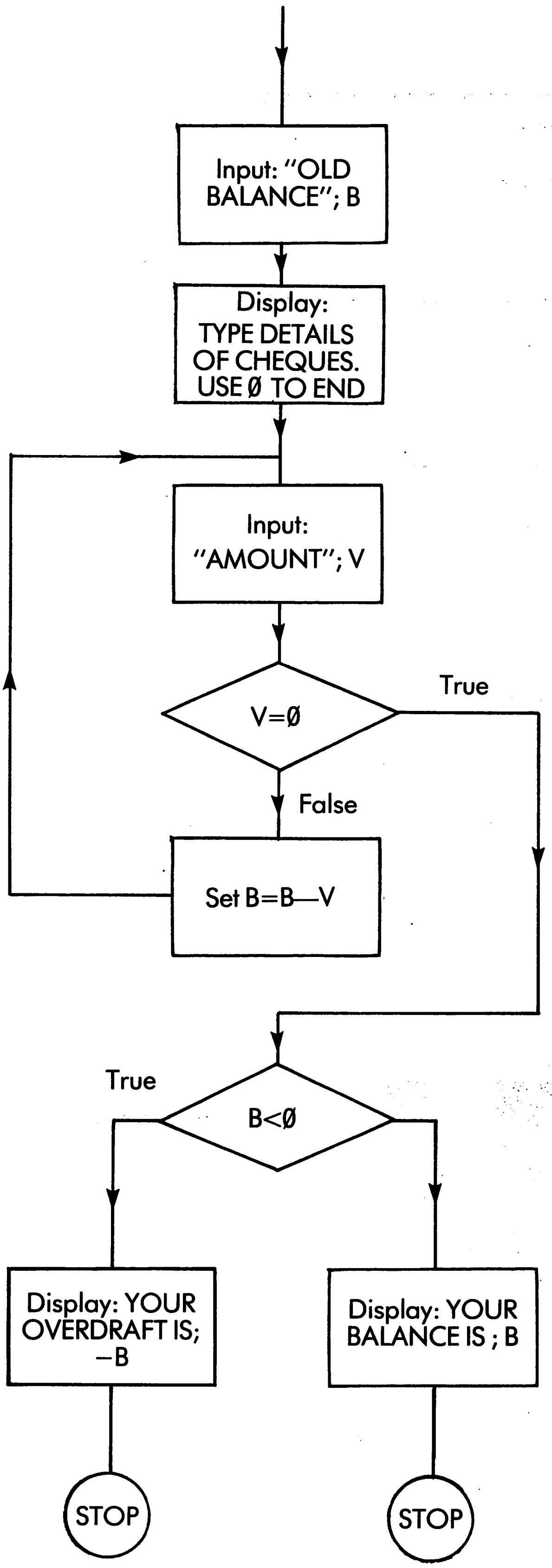
Experiment 13.2:

```
 10 REM DIESEL TRAIN HORN
 20 POKE 36878, 15
 30 POKE 36876, 235
 40 POKE 36875, 235
 50 FOR J=1 TO 1000
 60 NEXT J
 70 POKE 36878, 0
 80 FOR J=1 TO 100
 90 NEXT J
100 POKE 36878, 15
110 POKE 36876, 240
120 POKE 36875, 240
130 FOR J=1 TO 1000
140 NEXT J
150 POKE 36878, 5
160 POKE 36876, 235
170 POKE 36875, 235
180 FOR J=1 TO 1000
190 NEXT J
200 POKE 36875, 0
210 POKE 36876, 0
220 POKE 36878, 0
230 STOP
```

```
 10 REM JET FLYING OVERHEAD
 20 FOR J=110 TO 100 STEP -0.2
 30 V= 6000/(500+3*(J—50)↑2)
 40 POKE 36878, V+3
 50 POKE 36877, 225 +J/6
 60 FOR M=1 TO 50
 70 NEXT M
 80 NEXT J
 90 POKE 36877, 0
100 POKE 36878, 0
110 STOP
```

UNIT:14

Experiment 14.1:



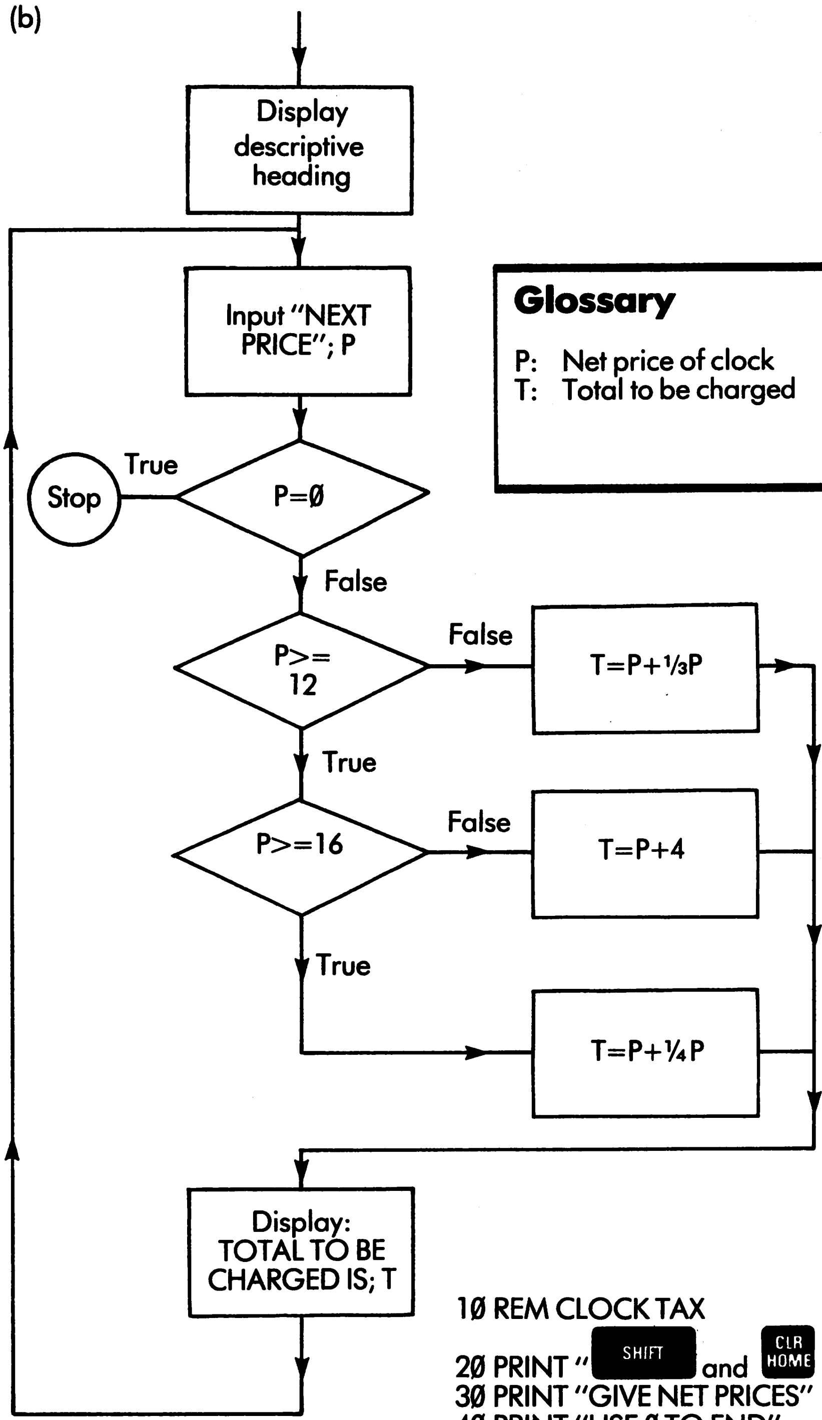
Glossary

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 Ø TO END"
50 INPUT "AMOUNT"; V
60 IF V=Ø THEN 9Ø
70 B=B—V
80 GOTO 5Ø
90 IF B<Ø THEN 12Ø
100 PRINT "YOUR BALANCE IS £"; B
110 STOP
120 PRINT "YOUR OVERDRAFT"
130 PRINT "IS £"; −B
140 STOP
```

Experiment 14.2:

(b)



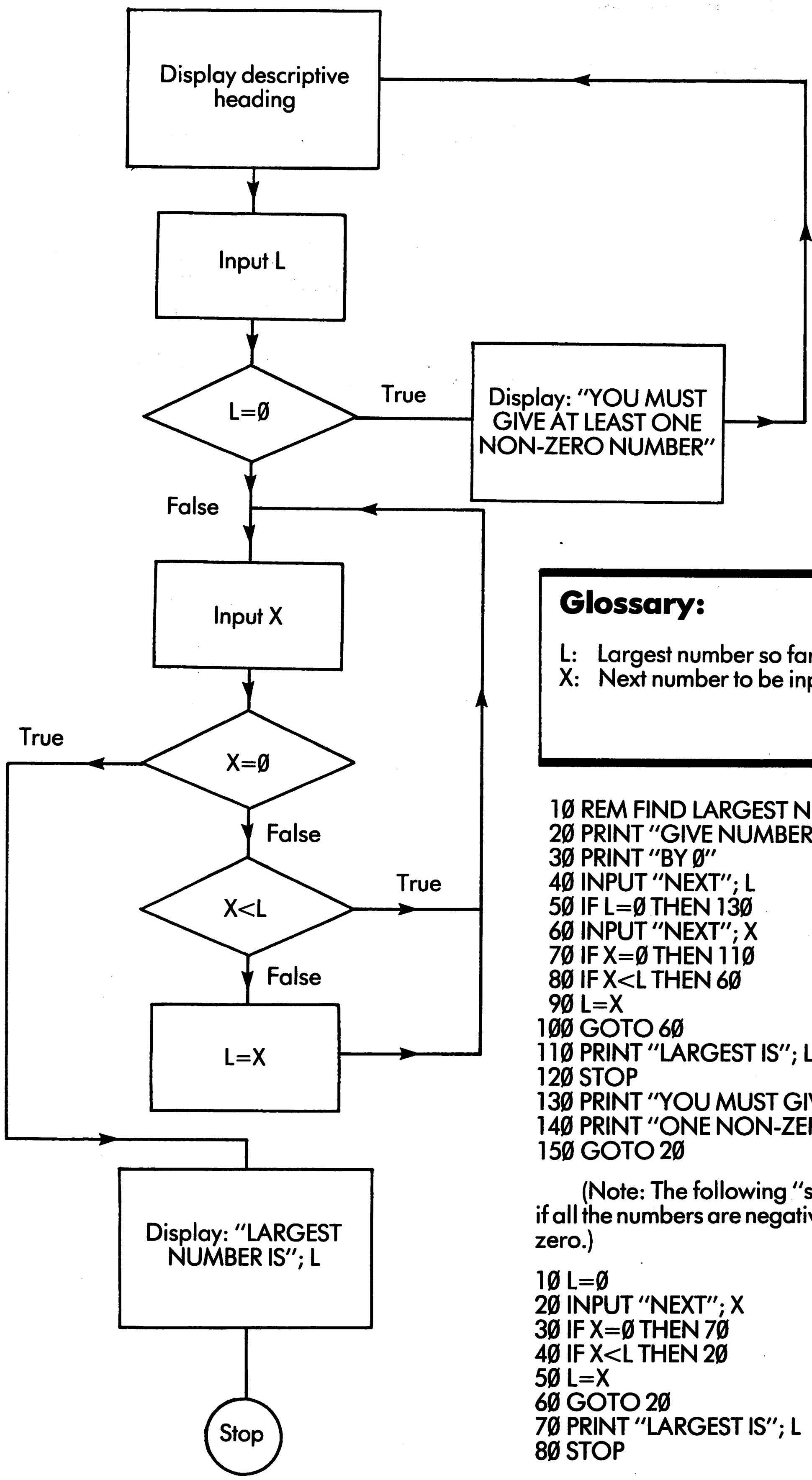
Glossary

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 10Ø
80 T= P+ (1/3)★P
90 GOTO 14Ø
100 IF P >=16 THEN 13Ø
110 T=P+4
120 GOTO 14Ø
130 T=P+ (1/4)★P
140 PRINT "TOTAL TO BE CHARGED"
150 PRINT "IS "; T
160 PRINT
170 GOTO 5Ø
180 STOP
```

Experiment 14.2:

c)



Glossary:

L: Largest number so far
X: Next number to be input

```
 1Ø REM FIND LARGEST NUMBER
 2Ø PRINT "GIVE NUMBERS ENDED"
 3Ø PRINT "BY Ø"
 4Ø INPUT "NEXT"; L
 5Ø IF L=Ø THEN 13Ø
 6Ø INPUT "NEXT"; X
 7Ø IF X=Ø THEN 11Ø
 8Ø IF X<L THEN 6Ø
 9Ø L=X
1ØØ GOTO 6Ø
11Ø PRINT "LARGEST IS"; L
12Ø STOP
13Ø PRINT "YOU MUST GIVE AT LEAST"
14Ø PRINT "ONE NON-ZERO NUMBER"
15Ø GOTO 2Ø
```

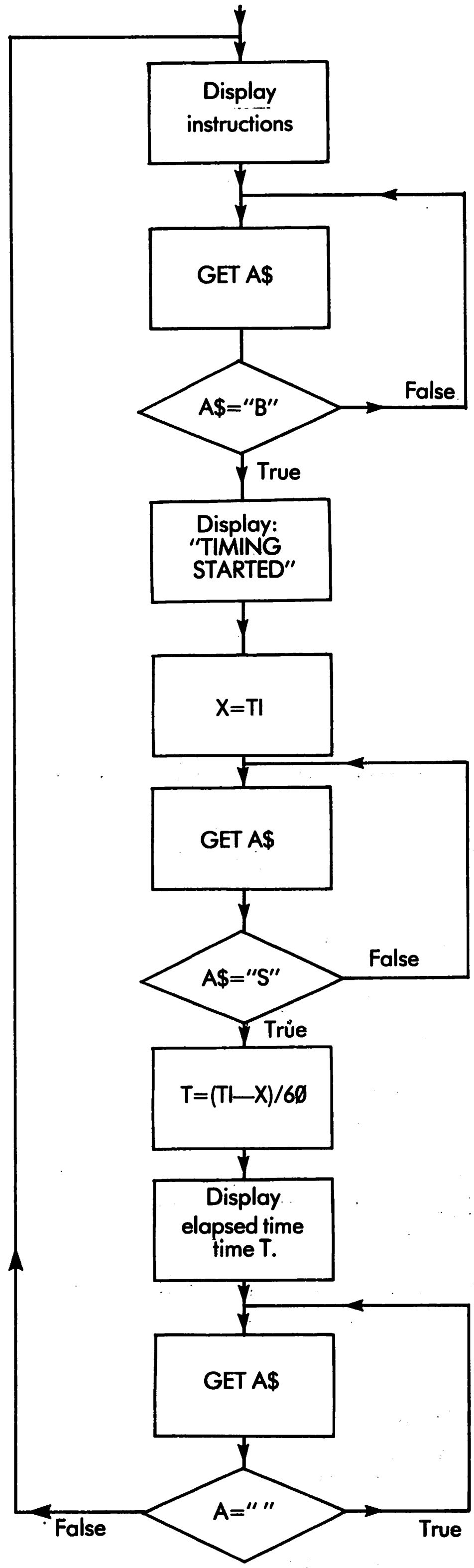
(Note: The following "solution" won't work if all the numbers are negative—that is, less than zero.)

```
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):

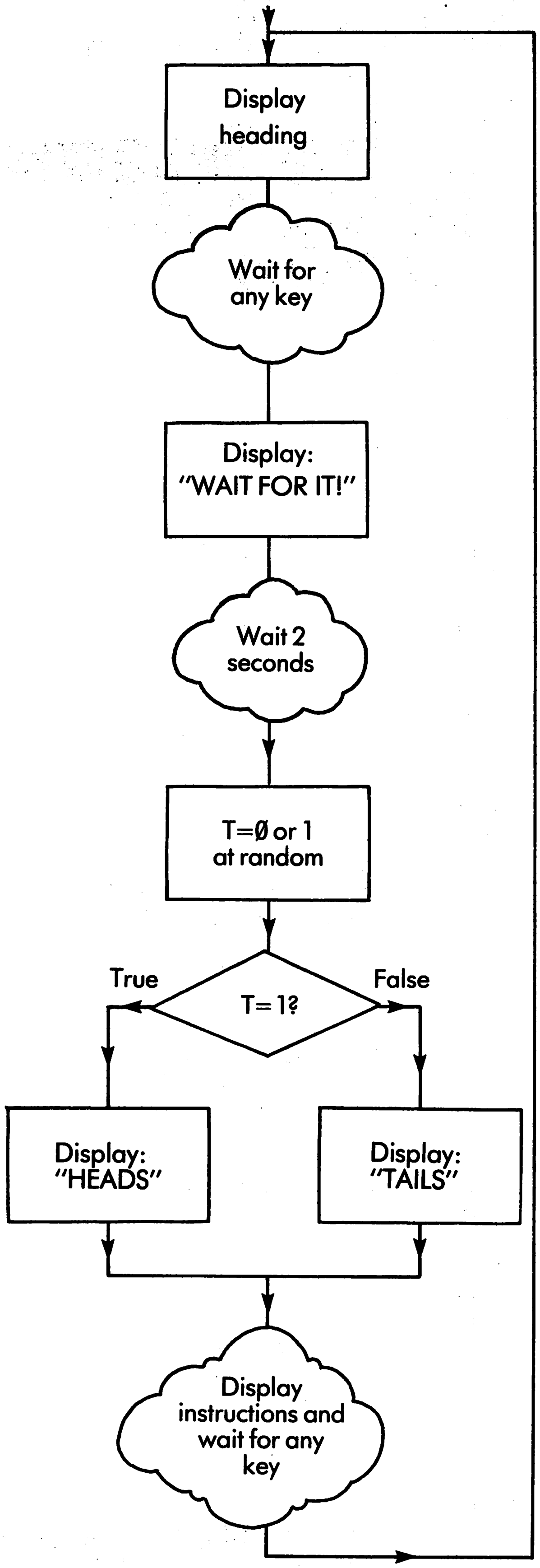


Glossary:

A$: Keyboard character
X: Internal time at start of interval (jiffies)
T: Elapsed time (seconds)

```
  5 REM STOPWATCH
 1Ø PRINT " [SHIFT] and [CLR HOME] "
 2Ø PRINT "STOPWATCH PROGRAM"
 3Ø PRINT
 4Ø PRINT "TO START THE STOPWATCH"
 5Ø PRINT "HIT THE B KEY"
 6Ø PRINT "TO STOP IT, HIT S"
 7Ø GET A$
 8Ø IF A$ <>"B" THEN 7Ø
 9Ø X=TI
 95 PRINT "TIMING STARTED"
1ØØ GET A$
11Ø IF A$ <> "S" THEN 1ØØ
12Ø T= (TI—X)/6Ø
13Ø PRINT "ELAPSED TIME WAS"
14Ø PRINT T; "SECONDS"
15Ø PRINT
16Ø PRINT "NOW HIT ANY OTHER KEY"
17Ø PRINT "FOR ANOTHER TIMING"
18Ø GET A$
19Ø IF A$= " " THEN 18Ø
2ØØ GOTO 1Ø
```

Experiment 15.2 (2):



Glossary:

S$ Keyboard character
M: Used in loop to wait 2 seconds
T: Ø (for Tails) or 1 (for Heads)

```
10 REM COIN TOSSING
20 PRINT " SHIFT and CLR HOME "
30 PRINT "HIT ANY KEY TO TOSS"
40 PRINT "YOUR COIN"
50 GET S$
60 IF S$= "" THEN 50
70 PRINT "WAIT FOR IT!"
80 PRINT
90 FOR M=1 TO 2000
100 NEXT M
110 T=INT (0+2*RND(0))
120 IF T=1 THEN 150
130 PRINT "TAILS"
140 GOTO 160
150 PRINT "HEADS"
160 PRINT
170 PRINT "HIT ANY KEY FOR"
180 PRINT "NEXT GO"
190 GET S$
200 IF S$= "" THEN 190
210 GOTO 20
```

Experiment 15.4:

```
10 REM CRAPS
20 PRINT " [SHIFT] and [CLR HOME] "
30 PRINT "THE GAME OF CRAPS"
40 PRINT "IS PLAYED WITH TWO"
50 PRINT "DICE. FIRST YOU BET"
60 PRINT "AND THEN YOU THROW. IF"
70 PRINT "YOU GET A SCORE OF 7"
80 PRINT "OR 11, YOU WIN. IF YOU"
90 PRINT "THROW 2, 3 OR 12, YOU"
100 PRINT "LOSE. IF YOU THROW ANY"
110 PRINT "OTHER NUMBER YOU DON'T";
120 PRINT "WIN OR LOSE STRAIGHT"
130 PRINT "AWAY: YOU KEEP ON"
140 PRINT "THROWING UNTIL YOU"
150 PRINT "EITHER"
160 PRINT " THROW THE SAME AS YOU"
170 PRINT "DID FIRST TIME (AND"
180 PRINT "WIN) "
190 PRINT "OR"
200 PRINT " THROW A 7 (AND LOSE)"
210 PRINT
220 PRINT "HIT ANY KEY TO"
230 PRINT "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"
320 PRINT "NEXT BET"
330 GET R$
340 IF R$= " " THEN 330
350 PRINT "YOUR CAPITAL NOW IS"
360 PRINT C
370 PRINT "HOW MUCH DO YOU"
380 INPUT "BET"; W
390 IF W <=C THEN 420
400 PRINT "YOU CAN'T AFFORD IT"
410 GOTO 310
415 REM ORGANISE FIRST THROW
420 PRINT " [SHIFT] and [CLR HOME] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓]
      FIRST THROW (BET = "; W; ")"
430 PRINT " [CLR HOME] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] "; A$
440 FOR J=1 TO 5
450 PRINT B$
460 NEXT J
470 PRINT C$
475 REM SHOW 10-59 DIFFERENT FACE PAIRS
480 Q=INT (10+50★RND(0))
490 FOR Z=1 TO Q
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 POKE 36878, 15
530 POKE 36876, 254-A★B
540 PRINT " [CLR HOME] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓]
      [CRSR ↑↓] [CRSR ↑↓] [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] "; A;
      " [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] "; B
545 REM WAIT A BIT
550 FOR M=1 TO 50
555 NEXT M
560 NEXT Z
565 REM STOP SOUND
570 POKE 36876, 0
580 POKE 36878, 0
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
645 REM ELSE ORGANISE MORE THROWS
650 PRINT
660 PRINT
670 PRINT
680 PRINT "YOU HAVE TO MAKE"
690 PRINT T; "BEFORE 7"
700 PRINT " [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓]
      [CRSR ↑↓] HIT ANY KEY TO GO ON"
710 GET R$
720 IF R$= " " THEN 710
730 PRINT " [SHIFT] and [CLR HOME] [CRSR ↑↓] NEXT
      THROW (BET= "; W; ")"
740 PRINT "MAKING"; T
750 PRINT " [CLR HOME] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] "
760 PRINT A$
770 FOR J=1 TO 5
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 POKE 36878, 15
860 POKE 36876, 254-A★B
870 PRINT " [CLR HOME] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓] [CRSR ↑↓]
      [CRSR ↑↓] [CRSR ↑↓] [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] [CRSR ⇐⇒] "; A;
```

```
      "[⇐CRSR⇒][⇐CRSR⇒][⇐CRSR⇒][⇐CRSR⇒]";B
 88Ø FOR M=1 TO 5Ø
 89Ø NEXT M
 9ØØ NEXT Z
 9Ø5 REM SILENCE
 91Ø POKE 36876,Ø
 92Ø POKE 36878,Ø
 925 REM IF A+B=T PLAYER WINS
 93Ø IF A+B=T THEN 1ØØØ
 935 REM IF A+B=7 PLAYER LOSES
 94Ø IF A+B=7 THEN 11ØØ
 945 REM ELSE PLAYER THROWS AGAIN
 95Ø GOTO 7ØØ
 99Ø REM PLAYER WINS
1ØØØ PRINT "[⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓]
     YOU WIN"
1ØØ5 REM ADD WINNINGS TO CAPITAL
1Ø1Ø C=C+W
1Ø15 REM PAEAN OF PRAISE
1Ø2Ø POKE 36878,15
1Ø3Ø FOR J=1 TO 2Ø
1Ø4Ø POKE 36876,24Ø
1Ø5Ø FOR M=1 TO 25
1Ø6Ø NEXT M
1Ø7Ø POKE 36876,Ø
1Ø8Ø FOR M=1 TO 25
1Ø85 NEXT M
1Ø9Ø NEXT J
1Ø95 GOTO 31Ø
11ØØ REM PLAYER LOSES
111Ø PRINT "[⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓][⇑CRSR⇓]
     YOU LOSE"
1115 REM CHIRP OF TRIUMPH
112Ø POKE 36878,15
113Ø FOR J=22Ø TO 127 STEP —1
114Ø POKE 36874,J
115Ø POKE 36875,J
116Ø FOR M=1 TO 5
117Ø NEXT M
118Ø NEXT J
119Ø POKE 36878,Ø
1195 REM TAKE LOSS 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

```
10 A = 5/0
```

or

```
20 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 0 to 255.

This error can occur in commands like

```
10 POKE 36878, 1234
```

or

```
20 J = 300
30 POKE 36876, J
```

Load Error

Your program is not loading correctly from the 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

Commodore International Limited
487 Devon Park Drive, Suite 200,
Wayne, PA 19087

PRINTED IN ENGLAND

commodore
COMPUTER