MASTERING THE COLOUR GENIE

IAN SINCLAIR

Mastering the Colour Genie

Ian Sinclair

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

British Library Cataloguing in Publication Data

Sinclair, Ian R.
Mastering the Colour Genie.
1. Colour Genie (Computer)
I. Title
001.64'04 QA76.8.C/

ISBN 0-246-12190-4

First published in Great Britain by Granada Publishing 1983
Reprinted 1984

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed in Great Britain by Mackays of Chatham, Kent

Contents

Preface

Among the many computers that compete in its price class, the Colour Genie stands out by its ancestry, its careful use of memory, and by its quite outstandingly good sound system. This book is intended as a guide for the complete beginner to computing who is learning by means of a Colour Genie. I have assumed that you, the reader, have no previous experience of computing or electronics. I have not even assumed that you have electrical knowledge or can tune a TV receiver. The ownership of a computer, after all, should not have to depend on any more technical knowledge than the ownership of a car.

Even if you are not a complete beginner to computing, you will still find much of this book very useful. The two Colour Genie manuals contain much that is of interest, but the information is not always easy to extract. The guide to the colour graphics and the sound system of the Colour Genie will, I believe, be of considerable help to anyone who is using the Genie for the first time, and will be a valuable reference guide thereafter.

A book such as this does not appear without a considerable amount of help from a lot of people. I am most grateful to Lowe Computers, distributors of the Genie range of computers, who lent me the Colour Genie. I am also indebted to Keith Bedford of Lowe Computers who answered questions over the telephone at odd hours of the day.

As always, the team at Granada Publishing worked hard and with astonishing efficiency. Richard Miles kept the project moving and exercised tender loving care over the manuscript at all stages. The editing work of Sue Moore also merits my gratitude, because her sharp eyes can always detect errors that the rest of us have overlooked.

Finally, the program listings in this book have been set directly from a printout produced immediately after running each program.

This ensures a standard of accuracy in listings which is unobtainable by any other methods, and which greatly adds to the usefulness of the book. All zeros have been slashed through in the listings to avoid confusion with the letter O. All examples have been kept as short as possible so as to ensure that they can be typed reasonably quickly.

Ian Sinclair

Chapter One
Setting Up the Colour Genie

By the time that you read this, you will have found that the Colour Genie is not the type of computer that can be carried away in a light breeze. It is a 'one piece' computer, so that it's ready to go into service whenever you connect a mains plug to it. A computer is a more complicated device than a kettle or a toaster, however, and connecting a mains plug is just the start of getting the Colour Genie working its magic for you.

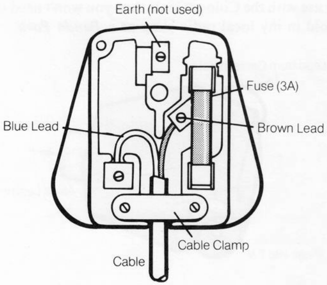

Fig. 1.1. Connecting the mains plug. Note that the earth pin is *not* connected. If you haven't wired a mains plug before, take it to an electrician.

The plug is connected as indicated in Fig. 1.1. There are only two leads, one blue and the other brown, and the cable should be tightly clamped. The fuse should be a 3 amp type, not the 13 amp variety which usually comes with the three-pin plug. If you are accustomed to fitting plugs for yourself then the diagram should be enough to remind you of what is needed. If you don't want to have anything to do with mains supplies, then take the Genie along to an electrician and get a plug, with a 3A fuse, connected or ask your Colour Genie retailer to fit a plug for you.

With that hurdle over, you are almost ready to work some Genie magic, but you need the use of a TV receiver. A computer is a device which is arranged so as to send signals to a TV receiver, and unless you connect a TV receiver to the Colour Genie you won't be able to see what it is doing. It will still compute for you just as well, but you won't see what is going on.

Unlike many other small computers, the Colour Genie comes with its TV cable ready-attached and with an aerial plug at the end of the lead. You could, of course, simply plug this lead into the TV receiver, but a better option is to use the type of 2-to-1 adaptor that is illustrated in Fig. 1.2. This allows you to keep an aerial cable plugged in, and to connect or disconnect the Colour Genie as you wish without disturbing the TV receiver. It's useful if you have to share a colour TV with the family. It also saves wear on the aerial connector of the TV receiver itself. If you have a TV that you can reserve for use with the Colour Genie then you won't need this device which is sold in my local radio shop as a *Panda Pack*.

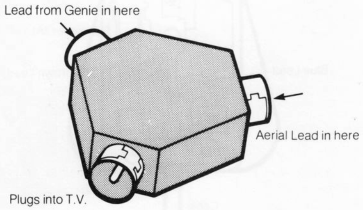

Fig. 1.2. A typical 2-to-1 TV aerial adaptor.

The TV that you use to display the Colour Genie's signals need not be a colour receiver – not to start with, at least. The skills of programming a Colour Genie do not require you to see the results in colour until you come to the colour instructions of the Genie in Chapter 7. Many colour computers produce a recognisable set of grey shades on a black/white TV, but I found that my Colour Genie did not show any noticeable differences on a portable B/W receiver, though the colours were most impressive on a standard colour television receiver.

The big switch-on

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have around. When you are in full control of your Colour Genie you will need three mains sockets. Two of these will be for the Colour Genie and the TV receiver, but you will need one more for a cassette recorder. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of a three- or four-way socket strip with a cable and a plug (Fig. 1.3). This avoids a lot of what the famous advert calls 'spaghetti hanging out the back'. Don't rely on the old-fashioned type of three-way adaptor – they never produce really reliable contacts. The Colour Genie has its own mains switch, so you can keep it plugged in if you like.

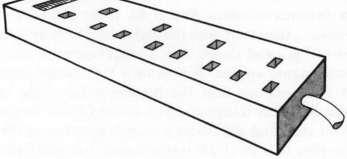

Fig. 1.3. A four-way socket strip. You'll find this essential unless you have a lot of sockets available.

The next step, then, is to switch on the TV receiver and the Genie. The sounds are played through the loudspeaker of the TV so that you have full control over the volume. You can also send these signals out to a hi-fi system so that you can hear them at full volume, or record them, as you please. Your Colour Genie dealer will be able to supply suitable connecting leads for this purpose.

An ordinary domestic TV is not ideal for viewing the Colour Genie signals, or those of any other computer. This is because the signals cannot be sent directly to the TV in the form that would give a clear picture. Instead, they have to be transmitted, using a miniature transmitter that is called a *modulator*. This is because most TV receivers cannot be safely connected to anything except by the aerial lead. Very much clearer pictures can be obtained by using what is called a *monitor*. This is a form of stripped-down TV which can't receive broadcast signals (no licence needed!), but which can be safely connected to the Colour Genie to show high-quality pictures. If you are lucky enough to see a demonstration of Genie signals displayed on a colour monitor you will get some idea of how much is lost when a modulator and an ordinary colour TV has to be used.

The second point is that a TV receiver has to be tuned to the signal from the Colour Genie. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked 'VCR' it's unlikely that you will be able to get the Colour Genie tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the Colour Genie's signals.

Figure 1.4 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Fig. 1.4(a). This is the type of tuning system that you find on black/white portables, and you only have to turn the dial to get the Colour Genie's signal on the screen. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then start with the dial turned fully anticlockwise as far as it will go, and slowly turn it clockwise until you see the Colour Genie signal appear. If you turn the volume control up slightly so that you can hear the rushing noise of the untuned receiver, you will hear things go quiet as the Colour Genie signal appears. You may find that there is some reduction in the sound level as you tune to a local TV transmission, but you'll notice the difference. The Colour Genie doesn't give you the sound of Coronation Street!

What you are looking for, if the Colour Genie hasn't been touched since you switched it on, is the phrase MEM SIZE? on the screen. When you can see these words, turn the dial carefully, turning slightly in each direction until you find a setting in which the words are really clear. On a TV receiver, particularly a colour TV, the words may never be particularly clear (Fig. 1.5), but get them steady at least and as clear as possible.

The older types of colour and B/W TV receivers used mechanical push-buttons (Fig. 1.4(b)) which engage with a loud clonk when you push them. There are usually four of these buttons, and you'll need to use a spare one which for most of us means the fourth one. Push

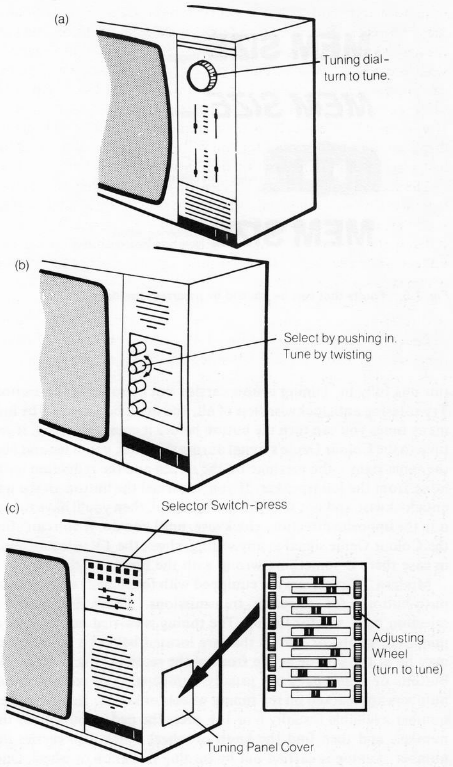

Fig. 1.4. The main methods for tuning TV receivers. (a) Single dial, as used on B/W portables, (b) four-button, (c) latest 12-switch type with tuning panel.

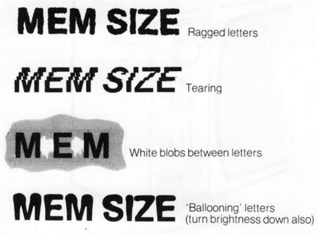

Fig. 1.5. Faults that can be caused by incorrect tuning.

this one fully in. Tuning is now carried out by rotating this button. Try rotating anticlockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the Colour Genie's signal during this time, you'll see and hear the same signs – the message on the screen and the reduction in the noise from the loudspeaker. If you've turned the button all the way anticlockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the Colour Genie signal at any setting, check the TV using an aerial in case there is something wrong with the tuning of the TV.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver (Fig. 1.4(c)). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen and silence from the loudspeaker. On this type of receiver, the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave it off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep re-tuning.

The mystery starts

Once you have achieved a tuned signal from your Colour Genie, the business of mastering the Genie magic begins. To start with, you have the message MEM SIZE? shining at you from the top left-hand corner of the screen. This doesn't mean that it has forgotten how much memory it has, simply that it wants to be told if you wish to reserve any of the memory for special purposes. Throughout this book, we shall not need to make any special use of the memory, so we don't need to answer this question. If you answer with a number, then the machine will partition off its memory, keeping a section in reserve. You could find, if too much has been reserved in this way, that you couldn't use the machine for normal purposes! The best reply, until you know a lot more about the machine, is simply to press the key that is marked RETURN, on the right-hand side of the keyboard. Do not press any letter keys before you press RETURN, as this can cause the machine to have a minor fit and refuse to obey you. If this happens, switch off and then on again after a few seconds. It's important to note that nothing that you can do by pressing keys on the keyboard can possibly damage the Colour Genie – the worst you can do is to lose a program that was stored in the memory. You can, however, damage the Colour Genie by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on.

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the Colour Genie. If we ignore the keys at the left- and the right-hand sides, most of the Colour Genie keys look like typewriter keys. The arrangement of letters and numbers is the same as that of a typewriter and if you've ever used a typewriter, particularly an electric typewriter, then you should be able to find your way round the keyboard of the Colour Genie pretty quickly.

There's one very noticeable difference, though. When you use a typewriter, pressing a letter key gives you a small letter (called *lower-case*), and pressing a letter key along with the SHIFT key produces a capital letter (called *upper-case*). On the Colour Genie, you will get upper-case (capitals) when you press letter keys by themselves, and lower-case (small letters) when you press letter keys with the SHIFT key also pressed. This is the opposite of the typewriter arrangement, and it's deliberately designed that way because instruction words for the Colour Genie should be typed in upper-case letters. There is a SHIFT LOCK key on the left-hand side of the keyboard, but I suggest that you leave it alone for now – we'll see why later.

As well as the ordinary typewriter keys, there are a number of special keys which are not found on any typewriter. There is a key at the left front of the keyboard which is marked MODSEL (Mode Selector), which is used to place graphics symbols (parts of pictures) on the screen in place of letters and numbers. There is also a pair of keys that are marked RST, one on each side of the back row of keys. These are 'panic buttons' which when pressed together will return the control of the Genie to you if it appears to have 'locked up' and refuses to obey instructions. Pressing just one of the RST keys has no effect, so you really have to intend to use these keys. Another key, marked BREAK (at the right-hand rear) will stop the Genie from carrying out program actions, and return it to awaiting your next command. The most important of these special keys, however, as far as we are concerned at the moment, is the key that is marked RETURN. This is in the position of the 'carriage return' key of an electric typewriter, but its action is not the same in all respects. Pressing the RETURN key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it. Perhaps OBEY might have been a good name for a computer!

If you are accustomed to using an electric typewriter, you will have to change some of your habits as far as this key is concerned. During the use of a typewriter, you would press the 'carriage return' key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The RETURN key of the computer does rather more than this. If the material that you are typing into the Colour Genie takes more than one line on the screen, the machine will automatically select the next screen line for you. The RETURN key must not be used for this purpose. The RETURN key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. It will always provide a new line for you, however, and select a position at the left-hand side. The position where a letter, or other character, will appear when you press a key is indicated by a flashing block on the screen. This flashing block is called the 'cursor', and it acts as a sort of signpost for you, as we'll see later.

Cassette try-out

You can obtain a lot of enjoyment from a computer system that consists only of the machine and a TV receiver. Each time that you switch the machine off, however, all the program and other information that has been stored in the memory of the computer will be lost. Since it might take several hours to enter a program into the machine by typing instructions on the keyboard, this waste just has to be avoided. We avoid the loss of programs by recording them on tape.

The computer has circuits which will convert the instructions of a program into musical tones, which can then be recorded on an ordinary cassette recorder. When these notes are replayed, another set of circuits will convert the signals back into the form of a program. In this way, the use of a cassette recorder allows you to record your program on tape and to replay them again. Before you tackle the rest of this book, then, it's important to check now that you can record and replay programs.

Almost any cassette recorder is suitable, but if you are buying a recorder specially for computer use, get one which is mains and battery operated, has a tape counter and automatic recording level control. Most small portable cassette recorders have these features, but hi-fi and other stereo recorders are usually unsuitable. I have used a Trophy CR100 recorder for years with very satisfactory results. Recorders by Sanyo, Hitachi and Sharp have also proved satisfactory. Lowe Computers have developed a specific computer cassette recorder, called the EG2016 available from your local Genie dealer.

Start work by switching everything off. Now find the cassette lead (Fig. 1.6) of the Colour Genie. This has a five-pin plug at one end, and two small plugs at the other end. These small plugs have colour-coded leads. The plug which is fitted at the end of the white lead engages into the socket on the cassette recorder that is marked EAR or has a drawing of an ear. This is where the signals come out of the recorder to be sent to the computer. The plug on the black lead fits into the socket that is marked MIC; this is the mircophone input of the cassette recorder. The Colour Genie uses only these two plugs.

The five-pin plug at the other end of the cable fits into a socket which is at the back of the Colour Genie, on the right-hand side as

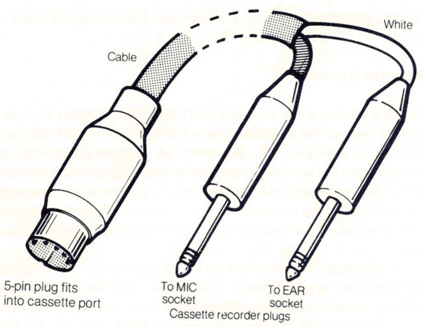

Fig. 1.6. The cassette lead of the Colour Genie. The small plugs are called *jack plugs,* and they plug into the cassette recorder.

you face the keyboard. Be careful how you push this plug in. It should fit only one way round, so don't force it. Look for the small notch in the plug and the corresponding part of the socket – these should engage together.

Once you have made this connection, the cassette recorder is ready for use. It's preferable to run the recorder from the mains because battery life can be unpredictable. If your batteries decide to fail while you are recording, you will probably lose the program. The next thing that you have to sort out is a supply of blank cassettes. There's nothing wrong with using reputable brands of C90 length cassettes (ordinary 'ferric' tape, not the hi-fi $Cr0_2$ type), but you'll find that the short lengths of tape that are sold as C5, C10 or C15 in computer shops and in most branches of W. H. Smith's, Boots and Currys are much more useful. Lowe Computers produce specific blank computer cassettes for data recording (see your local Genie dealer).

Put a fresh cassette into the machine, with the 1 or A side uppermost. The first part of the cassette consists of a 'leader' which is plain, not recording, tape. This has to be wound on before you can record. If your recorder has a tape counter, reset the counter to zero, and then fast-wind the cassette to a count of 5. If there is no tape counter, take the cassette out and insert the body of a BiC pen into the centre of the empty reel. Turn the pen so that the tape winds on to the reel, and keep turning until you see the brown recording tape replace the clear or brightly-coloured leader.

Now before you can make a recording to test the system, you need a program to record, and this involves some typing. This is easy if you have just switched the Colour Genie on, but if you have been pressing keys at random, then it's a good idea to switch off again, then on. Press the RETURN key when you see the MEM SIZE? question appear, and your Colour Genie is ready for testing.

Type the number 1∅ (1 and then ∅), and then the word REM. Check that this looks correct, and then press the RETURN key. The effect of this is to place the instruction line 1∅ REM into the memory of the Colour Genie. Now type the rest of the lines, as illustrated in Fig. 1.7, remembering the press the RETURN key after you have completed typing each line. The numbers are called *line numbers*, and they are there for two reasons. One is to remind the computer that this is a program, the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers.

```
10 REM
20 REM
30 REM
40 REM
```

Fig. 1.7. A test program for recording and replaying.

Check that your program looks on the screen like the printed version in Fig. 1.7, and make sure that the cassette recorder is ready. Now type CSAVE"A". The C stands for cassette, and CSAVE is the instruction to the computer meaning that you want to save (record) a program on a cassette. The "A" is a *filename* which the computer will use to recognise the program if it is asked to. You must put in the quotes (inverted commas, obtained by pressing SHIFT and 2 at the same time), otherwise the computer cannot carry out the instruction. Don't press RETURN yet! Now start the recorder by pressing its PLAY and RECORD keys. Press them firmly so that they lock in place, and you will see the reels of the cassette turning. A few machines, notably some Sharp models, require you only to press the RECORD key of the recorder, but this is unusual. When the recorder is working, press the RETURN key on the Colour Genie. After a very short time, the cursor of the Colour Genie will reappear on the screen with a READY message. This lets you know that the program has been recorded, and you can switch the recorder off. That's all. Now set the volume control of the recorder to half-way along its range, and the tone control, if any, to maximum treble.

Now comes the crunch. You have to be sure that the recording was O.K. The Colour Genie has a particularly useful command just for this purpose. Wind back the tape, using the rewind key of the recorder, and type:

```
VERIFY
```

then press RETURN. This command will cause the Colour Genie to compare what is stored in its memory with the program that you recorded. It can't do so, however, until you play the program back. Press the PLAY key of the recorder, and wait. After a time, you should see two stars appearing at the right-hand top part of the screen. One star will remain steady, the other will flash briefly. When the screen shows the message READY you can stop the recorder. Your cassette recording is O.K. and correct recordings are being made. Just to show that the program has indeed been recorded, wind back the tape again. Type NEW and press RETURN. This should have wiped your program from the memory. Now type LIST and press RETURN. Nothing should appear – LIST means put a list of the program instructions on the screen, and there shouldn't be any!

You can now load the instructions in from the tape. Type CLOAD and press RETURN. Now press the PLAY key of the recorder (did you rewind the tape?). The stars should appear again, with the READY message to show when the loading operation is complete. When this appears, the program is in place, and the recorder can be stopped. Type LIST now, then press the RETURN key. You should see your program appear on the screen.

Once you can reliably save programs on tape, verify them and re-load them, you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes' more work will save your effort on tape so that you won't have to type it again.

What can go wrong? Bad connections, mainly, and the Table in Fig. 1.8 should help you to trace the source of problems. There is one other problem, however, which you may find puzzling. Even though you can record and replay your own programs, you may find that when you buy a program on a cassette it refuses to load at any setting of the recorder's volume control. This is nearly always because you need the head of the cassette recorder looked at. There's advice on this problem in Appendix A.

Error check list

1. Are you sure that you have a program on the tape? Remove the EAR plug, and play the tape back. You should hear a musical tone which signals the start of the program, then a short burst of noise which is the program.
2. Have you tried several settings of the volume control? Some of the programs which I recorded would play back only at maximum volume control setting.
3. Check that you have the plugs the correct way round. They should be colour-coded, but the colour coding *might* possibly be wrong.
4. Check that your recorder is working properly by using the microphone to record a message, and then listening to the replay.
5. Always check by making a fresh recording and then verifying it. Be careful to separate your recordings from each other, because they take up only a small length of tape each.

Fig. 1.8. A table for fault-finding in record/replay problems.

Chapter Two

Getting Your Name in Lights

Chapter 1 will have broken you in to the idea that the Colour Genie, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the RETURN key is pressed. You will by now have used the command NEW which clears out a program from the memory; and LIST which prints your program instructions on to the screen. You will also have found that the CLEAR key at the top right-hand side of the keyboard has the effect of wiping the screen clear.

Now there are two ways in which you can use a computer. One way is called *direct mode*. Direct mode means that you type a command, press RETURN, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In program mode the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*.

The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again, and then press RETURN.

Let's take a look at the difference. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

```
PRINT 1.6 + 3.2 (and then press RETURN)
```

You have to start with PRINT because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want to see is the answer on the screen. It doesn't recognise instructions like 'GIVE ME' or 'WHAT IS', only a few words that we call its *reserved words* or *instruction words*. PRINT is one of these words.

When you press RETURN after typing PRINT 1.6 + 3.2, the screen shows the answer, 4.8, under the command, and the word READY appears under this answer. The READY is a 'prompt', a reminder that the computer is ready for another command. Once this command has been carried out, however, it's finished.

A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press RETURN. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the difference between your commands and your program instructions. On computers that use the 'language' called BASIC (Beginners All-purpose Symbolic Instruction Code), this is done by starting each program instruction with a number which is called a *line number*. This must be a positive whole number, the type of number that is called a positive integer. This is why you can't expect the computer to understand an instruction like 5.6 + 3 = ; it takes the 5 as being a line number, and the rest doesn't make sense.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results.

```
10 PRINT 2.4+3.7
20 PRINT 3.5-1.6
30 PRINT 2.8*4.4
40 PRINT 7.3/1.8
```

Fig. 2.1. A four line arithmetic program. The word PRINT is essential.

Take a close look at this, because there's a lot to get used to in these four lines. To start with, the line numbers are 1∅,2∅,3∅,4∅ rather than 1,2,3,4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 1∅ and line 2∅, then you can type the line number 15, or 11 or 12 or any other whole number between 1∅ and 2∅, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 1∅ and 2∅. If you number your lines 1,2,3 then there's no room for these second thoughts. The Colour Genie can, in fact, renumber your lines for you, but more about this later.

The next thing to notice is how the number zero is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the ∅ in place of O, nor the O in place of ∅, and the slashing makes this difference more obvious to you so that you are less likely to make mistakes. Some magazines, unfortunately, reprint computer programs with the slashmarks removed, so that it's very easy to make mistakes.

Now to more important points. The star or asterisk symbol in line 3∅ is the symbol that the Colour Genie uses as a multiply sign. Once again, we can't use the × that you might normally use for writing multiplication because this is a letter. There's no divide sign on the keyboard, so the Colour Genie, like all other small computers, uses the slash (/) sign in its place.

So far, so good. The program is entered by typing it, just as you see it. You don't need to leave any space between the line number and the P of PRINT, because the Colour Genie will put one in for you when it displays the program on the screen. The space that shows on the screen does not get stored in the memory, so by missing this space out, we save memory. You will have to press the RETURN key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration.

When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. There are two things that you need to know now. Once is how to check that the program is actually in the memory, the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can use the CLEAR key to wipe the screen first if you like, then type LIST and press the RETURN key. When you press the RETURN key, and not until, your program will be listed on the screen. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. To make the program operate, you need another command, RUN. Type RUN, then press the RETURN key, and you will see the instructions carried out. To be more precise, you will see:

```
6.1
1.9
12.32
4.∅555555
READY
```

When you follow the instruction word PRINT with a piece of arithmetic like 2.8*4.4, then what is printed is the result of working out that piece of arithmetic. The program doesn't print 2.8*4.4, just the result of the action 2.8*4.4.

Now this is useful, but it's not always handy to get a set of answers on the screen, especially if you have forgotten what the questions were. The Colour Genie allows you a way of printing anything that you like on the screen, exactly as you type it, by the use of what is called a *string*.

```
10 PRINT"2+2= ";2+2
20 PRINT"2.5*3.5= ";2.5*3.5
30 PRINT"9.4-2.2= ";9.4-2.2
40 PRINT"27.6/2.2= ";27.6/2.2
```

Fig. 2.2. Using quotes. Anything placed between quotes is printed just as it is; anything outside the quotes is worked out!

Figure 2.2 illustrates this principle. In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. Enter this short program and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

```
2+2=4
```

Now there's nothing automatic about this. If you type a new line:

```
15 PRINT "2+2=  ";5*1.5
```

then you'll get the daft reply, when you RUN this, of:

```
2+2=7.5
```

The computer does as it's told and that's what you told it to do. What loony thought that computers would take over the world?

This is a good point also to take notice of something else. The line 15 that you added has been fitted into place between lines 1Ø and 2Ø – LIST if you don't believe it. No matter in what order you type the lines of your program, the computer will sort them into order of ascending line number for you. Note also that the spaces in the program of Fig. 2.2 between the = and the " are useful – just see what happens if you miss them out!

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, as far as the Colour

```
10 PRINT"THIS IS "
15 PRINT"THE MAGICAL "
20 PRINT"COLOUR GENIE"
```

Fig. 2.3. Printing words – the words must be placed between quotes.

Genie is concerned, always means print on to the TV screen. For activating a paper printer (*hard copy*, it's called), there's a separate instruction LPRINT (and LLIST for program listings). The L once meant 'line' in the days when printers for computers were huge pieces of machinery that printed a whole line at a time. You must not use these instructions unless you have a printer connected and switched on.

Now try the program in Fig. 2.3. You can try typing the lines in any order that you like, to establish the point that they will be in line number order when you list the program. When you RUN the program, the words appears in twos, with two words on each line. This is because the instruction PRINT doesn't just mean 'print-on-the-screen'. It also means 'take a new line', and start at the left-hand side!

```
10 PRINT"THIS IS ";
20 PRINT"THE MAGICAL ";
30 PRINT"COLOUR GENIE"
```

Fig. 2.4. The effect of a semicolon. This keeps printing on one line, and it will also remove spaces that you have typed (outside quotes).

Now this isn't always convenient, and we can change the action by using punctuation marks that we call *print modifiers*. Start this time by acquiring a new habit. Type NEW and then press the RETURN key. This clears the old program out. If you don't do this, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. In Fig. 2.3, for example, the line 15 would be left in store even when you typed a new line 1ϕ and a new line 2ϕ.

Now try the program in Fig. 2.4. There's a very important difference between Fig. 2.4 and Fig. 2.3, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to prevent the next piece of printing starting on a new line at the left-hand side. When you RUN this program, all of the words appear in one line. It would have been a lot easier just to have one line of program that read:

```
1∅ PRINT "THIS IS THE MAGICAL COLOUR GENIE"
```

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at examples of that sort of thing later.

Now, as a piece of light relief, take a look at the table in Fig. 2.5. The four keys that are arranged in a line down the right-hand side of the keyboard are called *programmable keys*. This means that you can decide what you want each key to print on the screen when you press it. We'll look at programming these keys briefly in Appendix E, but when the Colour Genie is switched on, these keys are

Key action	Function
1	LIST
2	RUN
3	AUTO
4	EDIT
SHIFT 1	RENUM
SHIFT 2	DELETE
SHIFT 3	CLOAD
SHIFT 4	CSAVE

Fig. 2.5. The 'programmable' keys. These can be used to allow 'one-key' entry of selected commands. You still have to press RETURN to make the command work. Re-programming is dealt with in Appendix E.

programmed for you, giving the words on the screen that are shown in Fig. 2.5. You still need to press RETURN to carry out the action, but it's useful to have the keys there to save the effort of typing some words over and over again. Make a copy of this table and keep it by the side of the computer.

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of Fig. 2.6 should not come as too much of a surprise. Line 1∅ contains a novelty, though, in the form of two instructions in one line. The instructions are separated by a colon (:) and you can, if you like, have several instructions following one line number in this way and taking several screen lines. So long as the number of characters in the 'line' does not exceed 255 (and at 40 characters per screen line that's more than 6 screen lines), you can put instructions together in this way. In a 'multistatement' line of this type, the Genie will deal with the different instructions in a left-to-right order.

```
10 CLS:PRINT"THIS IS THE GENIE"
20 PRINT:PRINT
30 PRINT"READY TO OBEY YOU"
```

Fig. 2.6. Spacing down by two lines. Note the effect of the colon in line 2ϕ.

The other point about Fig. 2.6 is that line 2ϕ causes the lines to be spaced apart. The two PRINT instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy.

```
10 PRINT1,2,3,4
20 PRINT1,2,3,4,5
30 PRINT"ONE","TWO","THREE","FOUR"
40 PRINT"THIS IS TOO LONG!","TWO","THREE
"
```

Fig. 2.7. Printing in columns, making use of the comma.

Figure 2.7 deals with columns. Line 1ϕ is a PRINT instruction that acts on the numbers 1,2,3, and 4. When these appear on the screen, though, they appear spaced out just as if the screen had been divided into four columns. The mark which causes this effect is the comma, and the action is completely automatic. As line 2ϕ shows, you can't get five columns. Anything that you try to get into a fifth column will actually appear on the first column of the next line down. The action works for words as well as for numbers, as line 3ϕ illustrates. When words are being printed in this way, though, you have to remember that the commas must be placed outside the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into columns something that is too large to fit, the long phrases will spill over to the next column, and the next item to be printed will be at the start of the next column along. Line 4ϕ illustrates this – the first phrase spills over from column 1 into column 2, and the word TWO is printed starting at column 3.

Commas are useful when we want a simple way of creating four columns. A much more flexible method of placing words along a line exists, however. It uses the instruction word TAB, which has to be followed by a number placed within brackets. The line is imagined as divided into 4∅ portions, equally spaced, and numbered from ∅ to 39. Thus TAB(∅) means the position at the left-hand side of the line, and TAB(39) means the position at the right-hand side of the line. Figure 2.8 illustrates the appearance of words that are typed at different TAB positions. Note that we must use TAB only following a PRINT instruction, but we can type PRINT TAB or PRINTTAB as we please. Saving one space between these words means saving one place in the memory, so the use of PRINTTAB is convenient on this count.

```
10 PRINT"START"
20 PRINT TAB(5)"FIVE ACROSS"
30 PRINT TAB(10)"TEN ACROSS"
40 PRINTTAB(15)"FIFTEEN"
```

Fig. 2.8. The TAB instruction and how it is used.

The use of *tabulation*, as this is called, can make the appearance of printing on the screen much smarter, as Fig. 2.9 illustrates. In this example, the word TITLE has been centred on its line by using TAB(17). The number is found by using a formula that has been known to typists for generations – it's illustrated in Fig. 2.10. Later on, we'll look at ways of carrying out this calculation automatically, so that you can print any phrase centred in a line without having to count the letters and spaces for yourself.

```
10 CLS:PRINTTAB(17)"TITLE"
20 PRINT:PRINT
30 PRINTTAB(2)"LOOKS A LOT SMARTER!"
40 PRINT:PRINT
```

Fig. 2.9. Centring a title to improve its appearance.

(a) Count number of characters in the title, including spaces.
(b) Divide this number by two. Ignore the remainder, if any, and subtract this number from 20.
(c) Use the result as the TAB number.

Fig. 2.10. The formula that is used to centre a title.

Providing that the instruction word PRINT has been used, you can make use of TAB more than once in a line. Figure 2.11 illustrates TAB being used to print letters in five columns, something that you can't do with commas. The sections are separated from each other by semicolons, but you don't in fact have to use a semicolon between the closing bracket of the TAB number and the item that is to be printed. One point that we haven't illustrated yet is that the quantity that is enclosed within the brackets of a TAB need not be a number – it can be a letter that represents a number. We'll come back to that point in the next chapter.

```
10 PRINT"A";TAB(8)"B";TAB(16)"C";TAB(24)
"D";TAB(32)"E"
```

Fig. 2.11. Using TAB more than once in a line.

Meantime, there's another very important print modifier to look at. The @ (pronounced *at*) sign on the keyboard is used to allow text (numbers, letters, words) to be placed anywhere on the screen. One minor problem you have to be aware of is that there is no SHIFT @. If you press the @ and the SHIFT keys together, you will see the @ symbol appear on the screen, but the machine will not recognise that the code is in use. Keep your fingers well away from the SHIFT key when you use @!

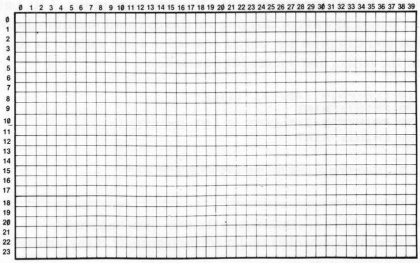

Fig. 2.12. The PRINT@ positions on the screen.

For the purpose of using @, we imagine the screen divided into a grid of 40 divisions across and 24 down, a total of 960 positions (Fig.

2.12). These are numbered, counting 0 as the top left-hand corner of the screen and 959 as the bottom right hand corner. An easy way to find the @ reference number for any position on the screen is to think of the lines as being numbered from 0 to 23 down the screen. If you multiply the line number by 40 and then add the TAB number for the position along the line that you want, you have the @ number. For example, if you want the 10th space on the number 5 line (which is the 6th one down, because we start counting at 0), then the @ number is 5*4ø+1ø = 21ø.

```
10 PRINT@10,"START"
20 PRINT@130,"SECOND"
```

Fig. 2.13. Using the PRINT@ instruction.

Figure 2.13 shows how PRINT@ is used. Keep the PRINT, the @ sign, and the number close together, and always follow the number with a comma. You can't use more than one @ following a PRINT in a line. The effect of PRINT@ is to allow you to print items wherever you want, as Fig. 2.14 shows. You don't have to print in the order of left-to-right or top-to-bottom either, because PRINT@ allows you complete freedom to print wherever you want. If your choice of PRINT@ position places a new word over an old one, then the new letters will simply replace the old ones.

```
5 CLS
10 PRINT@420,"FIRST ITEM"
20 PRINT@180,"SECOND"
30 PRINT@170,"THIRD"
```

Fig. 2.14. Using PRINT@ to place items at different parts of the screen. Items do not have to be printed in left-to-right or top-to-bottom order.

```
10 CLS:PRINT@97,"TITLE"
20 PRINT@162,"This is an example of GENI
E"
30 PRINT"UPPER and lower case letters"
```

Fig. 2.15. Upper-case (capitals) and lower-case letters.

Finally, try the program of Fig. 2.15 as an illustration of the use of lower-case letters. As we saw earlier, the Genie will print lower-case letters if you type a letter while you are holding down the SHIFT key. You can reverse this action by pressing the SHIFT LOCK key, which works just like the SHIFT LOCK of a typewriter, staying down when it has been pressed down once and springing up again next time it is pressed. When the SHIFT LOCK is down, letters will appear in lower-case unless you also press the SHIFT key. One thing that you have to be very careful about, however, is the left-arrow key. Normally when you are typing in upper-case (capital) letters, the left-arrow key acts to delete one character. It will delete more than one character only if you press the RPT (repeat) key immediately afterwards. When you have the SHIFT LOCK pressed down, though, pressing the left-arrow key will cause the whole line to be deleted. That's tough if it took you a lot of time to type it and, because the use of the left-arrow key becomes automatic after a while, you can lose a lot of text in this way. I personally never use the SHIFT LOCK on my Colour Genie in normal programming. When I want lower-case letters, I just press the SHIFT key temporarily, and try to remember to release it when I want to use the left-arrow key to delete a character.

Chapter Three
A Bit of Variation

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that go on before anything is printed. One of these is called assignment.

```
5 CLS
10 X=23
20 PRINT "2 TIMES ";X;" IS ";2*X
30 X=5
40 PRINT"X IS NOW ";X
50 PRINT"AND 2 TIMES ";X;" IS ";2*X
```

Fig. 3.1. Introducing a number variable which can take different values.

Take a look at the program in Fig. 3.1. Type it in, run it, and contrast what you see on the screen with what appears in the program. The first line that is printed is line 2∅. What appears on the screen is:

2 TIMES 23 IS 46

but the numbers 23 and 46 don't appear in line 2∅! This is because of the way we have used the letter X as a kind of code for the number 23. The official name for this type of code is a *variable name*.

Line 1∅ assigns the variable name X, giving it the value of 23. 'Assigns' means that wherever we use X, not enclosed by quotes, the computer will operate with the number 23. Since X is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X=2174.3256, for example. Line 2∅ then proves that X is taken to be 23, because wherever X appears, not between quotes, 23 is printed, and the 'expression' 2*X is printed as 46. We're not stuck with X as representing 23 for ever, though. Line 3∅ assigns X as being 5, and lines 4∅ and 5∅ prove that this change has been made.

That's why we call X a *variable* – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. Even after you have run the program of Fig. 3.1, providing you haven't added new lines or deleted any part of it, you can type PRINT X (or PRINTX), and pressing RETURN will show the value of X on the screen.

This very useful way to handle numbers in code form can use a 'name' which must start with a letter. You can add to that a second letter or a number, so that N, NA, N5 are all names that you can use for number variables, and each can be assigned to a different number. Just to make it even more useful, you can use similar 'names' to represent words and phrases also. The difference is that you have to add a dollar sign ($) to the variable name. If N is a variable name for a number, then N$ (pronounced *en-string* or *en-dollar*) is a variable name for a word or phrase. The computer treats these two, N and N$, as being entirely separate and different.

```
10 CLS:N$="NAME"
20 SN$="SINDBAD"
30 PRINT N$;" THE ";SN$;" FAMILY"
40 PRINT SN$; " THE SAILOR"
50 PRINT"- ANOTHER SATISFIED GENIE OWNER
"
```

Fig. 3.2. String variables. The name needs quotes around it when it is assigned, and the variable name must end with the dollar sign.

Figure 3.2 illustrates 'string variables', meaning the use of variable names for words and phrases. Lines 1∅ and 2∅ carry out the assignment operations, and lines 3∅ to 5∅ show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which must be surrounded by quotes.

```
10 G$="GENIE"
20 L1$="I am here to serve you"
30 L2$="What is your wish, Oh Master?"
40 CLS
50 PRINT"I AM THE ";G$:PRINT
60 PRINTL1$:PRINT
70 PRINT L2$:PRINT
```

Fig. 3.3. Longer string variables – you can use up to 255 characters!

Figure 3.3 shows another example, this time using the variable names L1$ and L2$ for longer phrases. There wouldn't be much point in printing messages in this way if you wanted the message once only, but when you continually use a phrase in a program, this is one method of programming it so that you don't have to keep typing it!

```
10 NURSE$="Lindsay"
20 NUT$="Peanuts"
30 NUMB$="Goofy"
40 NUDE$="Starkers"
50 PRINT NURSE$
60 PRINT NUT$
70 PRINT NUMB$
80 PRINT NUDE$
```

Fig. 3.4. You can use names of more than two characters, but the Colour Genie reads only the first two.

Now before you go wild on this use of variable names, a word of warning. There's nothing to stop you from using variable names of more than two characters. Nothing, except the fact that it uses up precious memory, and it can cause confusion. Take a look at Fig. 3.4. When you run this one, lines 5ϕ to 8ϕ all produce the same word – the last one that was assigned. The reason is that the computer takes notice of only the first two characters of a name. As far as the computer is concerned, all the variable names that were assigned in lines 1ϕ to 4ϕ are the same, NU$. Since the last assignation of NU$ was to 'STARKERS', that's what is printed – four times. Simple enough when you know about it, but it can cause a lot of bother if you don't.

Strings and things

Because the name of a string variable is marked by the use of the $ sign, a variable like A$ is not confused with a number variable like A. We can, in fact, use both on the same program knowing that the computer at least will not be confused. Figure 3.5 illustrates that the difference is a bit more than skin deep, though. Lines 1ϕ and 2ϕ assign number variables A and B, and string variables A$ and B$. When these variables are printed, you can't tell the difference between A and A$ or between B and B$. The difference appears, however, when the computer attempts to complete line 6ϕ. It can multiply two number variables, because numbers can be multiplied, but it can't multiply string variables. The reason is simple. A string variable can be anything. We have assigned A$ as '2', but we could just as easily have assigned it as '2 LABURNUM WAY'. You can multiply 2 by 3, but you can't multiply 2 LABURNUM WAY by 3 ACACIA AVENUE. The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other

```
10 A=2:B=3
20 A$="2":B$="3"
30 CLS
40 PRINTA;" TIMES ";B;" IS ";A*B
50 PRINT
60 PRINT A$;" TIMES ";B$;" IS ";A$*B$
70 REM IMPOSSIBLE - SO YOU GET TM ERROR.
```

Fig. 3.5. When a number is assigned as a string, it prints normally but you can't carry out arithmetic on it!

arithmetic operation on strings. Attempting to do a forbidden operation in line 6ϕ causes an error message. TM means 'type mismatch' – the operation that we called for can be done on numbers, but we have strings here. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause a TM error message. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's only a machine!

```
10 A$="SMITH"
20 B$="JONES"
30 CLS
40 PRINT"JUST CALL ME ";A$+"-"+B$;", HE
SAID."
```

Fig. 3.6. Concatenation, or joining of strings, which uses the + sign.

```
10 A$="***":B$="###"
20 G$="GENIE"
30 CLS
40 PRINTA$+B$+G$+B$+A$
50 PRINT
```

Fig. 3.7. Using concatenation to frame a title.

There is one operation, that looks rather like arithmetic being carried out on strings. It uses the + sign, but it isn't addition in the sense of adding numbers. Figure 3.6 illustrates this action of joining strings, which is often called *concatenation*. This is nothing like the action of arithmetic, and you'll see if you use numbers in place of the names. Concatenation is a very useful way of obtaining strings which otherwise would need rather a lot of typing. Take a look at Fig. 3.7. This defines strings A$ and B$ as characters which can be used as 'frames' around a title. The title is defined in line 2ϕ as

GENIE. Line 4∅ then prints a concatenated string. This has needed less typing than if you had to type all the characters between the quotes. It also allows you to rearrange the frames as you please. You can, for example, use:

B$ + A$ + G$ + A$ + B$

next time you print the title.

Getting some in

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

```
10 CLS
20 PRINT"WHAT IS YOUR NAME"
30 INPUT NM$
40 CLS:PRINT:PRINT
50 PRINTNM$;" -THIS IS YOUR LIFE!!"
```

Fig. 3.8. Putting information into a running program. No quotes are needed when you type your name this time.

Figure 3.8 illustrates this with a program that prints your name. Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

WHAT IS YOUR NAME

are printed on the screen. On the line below this you will see a question mark. The computer is now waiting for you to type something, and then press RETURN. Until the RETURN key is pressed, the program will hang up at line 3∅, waiting for you. If you're honest, you will type your own name and then press RETURN. You don't have to put quotes around your name, simply type it in the form that you want to see printed. When you press RETURN, your name is assigned to the variable NM$. The program can then continue, so that like 4∅ clears the screen and spaces down by two lines. Line 5∅ then prints the famous phrase with your name at the start.

You could, of course, have answered MICKEY MOUSE or

DONALD DUCK or anything else that you pleased. The computer has no way of knowing that neither of these is your true name. Ingenious it may be but real magic is too much to hope for!

We aren't confined to using string variables along with INPUT.

```
10 PRINT"ENTER A NUMBER"
20 INPUT N
30 PRINT
40 PRINT"TWICE ";N;" IS ";2*N
```

Fig. 3.9. Using INPUT along with a number variable.

Figure 3.9 illustrates an INPUT step which uses a number variable N. The same procedure is used. When the program hangs up with the question mark appearing, you can type a number and then press the RETURN key. The action of pressing RETURN will assign your number to N, and allow the program to continue. Line 4∅ then proves that the program is dealing with the number that you entered. When you use a number variable in an INPUT step, then what you have typed when you press RETURN must be a number. If you attempt to enter a string, the computer will refuse to accept it. Some computers stop running at this point, but the Colour Genie simply prints REDO, and this gives you another chance by typing a number and pressing RETURN again. If your INPUT step uses a string variable then anything that you type will be accepted when you press RETURN.

```
10 CLS
20 INPUT"TYPE YOUR NAME,PLEASE";NM$
30 PRINT
40 PRINT"VERY PLEASED TO MEET YOU, ";NM$
50 PRINT
```

Fig. 3.10. You can print a phrase along with an input, provided the phrase is between quotes and is followed by a semicolon.

The way in which INPUT can be placed in programs can be used to make it look as if the computer is paying some attention to what you type. Figure 3.10 shows an example – but with INPUT used in a different way. This time, there is a phrase following the INPUT instruction. The phrase is placed between quotes, and is followed by a semicolon and then the variable name NM$. This line 2∅ has the same effect as the two lines:

15 PRINT "TYPE YOUR NAME, PLEASE";
2∅ INPUT NM$

The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.11, for example, shows two variables being used after one INPUT. One of the variables is a string variable NM$, the other is the number variable N. Now when the computer comes to line 2ø, it will print the message and then wait for you to enter both of these quantities, a name and then a number. There are two ways of entering these quantities. One way is to type the name, then a comma, and then the number. Pressing the RETURN key will then assign the two variables, and the computer will continue on its way.

```
10 CLS
20 INPUT"NAME AND NUMBER,PLEASE";NM$,N
30 PRINT:PRINT
40 PRINT"NAME IS ";NM$
50 PRINT"NUMBER IS ";N
```

Fig. 3.11. More than one variable can be entered by one INPUT.

The other method consists of entering each quantity separately. If you type the name and then press RETURN, the computer will print two question marks on the next line. This is a symbol meaning 'more needed', and that's a signal for you to type the number and then press RETURN again. Whichever way you use, the name and number will be printed again in lines 4ø and 5ø.

```
10 CLS
20 INPUT"FOUR NUMBERS,PLEASE";A,B,C,D
30 PRINT
40 PRINT"THE SUM OF THESE IS ";A+B+C+D
50 PRINT
```

Fig. 3.12. Using four variables in a single INPUT.

We can extend this principle further. Figure 3.12 calls for four numbers to be entered. These can be entered one by one, pressing RETURN each time, or by typing each number, then a comma, then the next number and pressing RETURN only after typing the last number. Once again, whichever way you choose to enter the numbers (no strings allowed here), the program will print the sum in line 4ø.

Reading the data

There's yet another way of getting data into a program while it is running. This one involves reading items from a list, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list can be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the next item of the list.

```
10 CLS
20 READ NM$
30 PRINT NM$;
40 PRINT" IS VALUED AT ";
50 READ N
60 PRINT N;
70 PRINT" POUNDS"
100 DATA GOLD RING,768
```

Fig. 3.13. Using READ and DATA to put information into the program.

We'll look at this in more detail in Chapter 5, but for the moment we can introduce ourselves to the READ...DATA instructions. Figure 3.13 uses the instructions in a very simple way. Line 2∅ reads the first item on the list and assigns it to the variable NM$. This is printed in line 3∅, with the semicolon keeping printing in the same line so that the phrase in line 4∅ follows it. The semicolon at the end of line 4∅ once more keeps the printing in the same line, and line 5∅ reads the number which is the second item in the list. This is assigned to the variable name N (we could just as easily have used NM$) and printed in line 6∅. Once again, a semicolon prevents a fresh line from being taken, so that the final word of line 7∅ is printed following the number.

The READ...DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now. As before, though, we have to match the data items with the variable names that we use for them. We can read a number item and assign it to a string variable name, but we can't read a string item and assign it to a number variable name.

Number antics

The amount of computing that we have done so far should have persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for accounts or for word processing. It's time, then, to take a very brief look at the number abilities of the Colour Genie. It is a brief look because we simply don't have space to explain what all the mathematical operations do. In general, if you understand what a mathematical term like sin or tan or exp means, then you will have no problems about using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of incrementing if you are counting up and decrementing if you are counting down. Incrementing a number means adding 1 to it, decrementing means subtracting 1 from it. These actions are programmed in a rather confusing-looking way in BASIC, as Fig. 3.14 shows. Line 2∅ sets the value of variable X as 5. This is printed in line 3∅, but then line 4∅ 'increments X'. This is done using the odd-looking instruction:

$$X = X + 1$$

This means that the new value that is assigned to X is 1 more than its previous value. The rest of the program proves that this action of incrementing the value of X has been carried out.

```
10 CLS
20 X=5
30 PRINT"VALUE OF X IS ";X
40 X=X+1:PRINT
50 PRINT"NOW WE'VE USED X=X+1":PRINT
60 PRINT"VALUE IS NOW ";X
```

Fig. 3.14. Incrementing a number variable (increasing by one).

The use of the = sign to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

$$X = X-1$$

and this would have the effect of making the new value of X one less than the old value. X has been decremented this time. We could also use X = 2*X to produce a new value of X equal to double the old value, or X = X/3 to produce a new value of X equal to the old value divided by three. Figure 3.15 shows another assignment of this type,

```
10 CLS
20 X=5:PRINT"X IS ";X
30 PRINT
40 X=2*X+4
50 PRINT"IT'S CHANGED -"
60 PRINT"X IS NOW ";X
```

Fig. 3.15. Another change of variable value. Note the way that the equals sign is used.

in which both a multiplication and an addition are used to change the value of X.

Number functions

Figure 3.16 illustrates some number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 1∅ picks the value of 2.5 for X. Line 2∅ then prints the value of X squared, meaning X multiplied by X. This is programmed by typing X∧2. To get the square root of the number that has been assigned to X, we use the instruction word SQR. An alternative is X∧.5, but SQR(X) is easier to type and remember. For other roots, like the cube root you can use expressions like X∧(1/3) and so on. LOG(X) produces the natural logarithm of X.

```
10 CLS:X=2.5
20 PRINT"X SQUARED IS ";X^2
30 PRINT
40 PRINT" SQUARE ROOT IS ";SQR(X)
50 PRINT
60 PRINT" LOG OF X IS ";LOG(X)
70 PRINT
```

Fig. 3.16. Three of the number functions, demonstrating their actions.

Figure 3.17 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of interest only if you are interested in programming for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs.

How precise?

One of the problems of small computers is precision of numbers. You probably know that the fraction 1/3 cannot be expressed exactly as a decimal. How near we can get to its true value depends

ABS(X):	Gives the absolute value (meaning that the - sign will be removed) of the number or number variable X.
ATN(X):	Gives the angle, in units of radians, whose tangent is the number or number variable X.
CDBL(X):	Transforms the number or number variable into double-precision form, with 17 digits. The accuracy will only be as good as the original value of X, however.
CINT(X):	Gives the whole number just less than X. For a positive number, this is the same action as INT.
COS(X):	Gives the cosine of angle X. The value of X must be in units of radians.
CSNG(X):	Converts variable X to single-precision form, with 6 significant digits, rounded.
EXP(X):	Gives the exponential of X, e^X.
FIX(X):	Removes all figures beyond the decimal point from X.
INT(X):	Rounds a positive number down to the nearest integer. A negative number is also rounded to the next lower integer.
LOG(X):	Finds the natural (base e) logarithm of X. Divide the result by 2.303 to get the ordinary (base 10) logarithm.
RANDOM:	Ensures that a new set of random numbers will be generated by RND.
RND(X):	Gives a random number between 1 and X if X is positive and less than 32768. Using RND(∅) gives a fraction, between ∅ and 1.
SGN(X):	Gives ∅ if X is zero, −1 if X is negative, +1 if X is positive.
SIN(X):	Gives the sine of angle X. X must be in units of radians.
SQR(X):	Gives the square root of value of X. X must not be negative.
TAN(X):	Gives the tangent of the angle X. X must be in units of radians.

Fig. 3.17. Table of number functions. Some, like CINT, are very rarely used. If you don't understand them, you probably don't need them!

on the number of decimal places we are prepared to print, so that ∅.33 is closer than ∅.3, and ∅.333 is closer still. The computer converts most of the numbers it works with into the form of a fraction and a multiplier. The fraction is not a decimal fraction but a special form called a binary fraction, and this conversion is seldom exact. The conversion is particularly awkward for numbers like 1, 1∅1∅∅ and also .1, .∅1, .∅∅1; all the powers of ten, in fact. To avoid embarrassments like printing 3−2 = .9999999, the computer will round numbers of this type up or down, as need be, before displaying them. Not all computers do this well - you can be glad that you bought a Colour Genie!

Very few computers allow numbers to be stored in a much more precise way. The Colour Genie is one of this select group, because it allows numbers to be stored with three different degrees of precision. These are distinguished from each other by using marks following the variable names, and the marks that you have to learn are the integer mark % and the double-precision mark #.

An integer, as far as the Colour Genie is concerned, is a whole number whose value lies between the limits of −32768 and +32767. An integer variable name consists of the variable name followed by the % sign. If you assign a number to an integer variable, then only numbers in the correct range can be used, and any fractions will be discarded. Lines 1ϕ to 4ϕ of Fig. 3.18 illustrate this, because when the variable X, whose value is 3.7, is assigned to X% in line 3ϕ, then printing X% in line 4ϕ gives 3 only. The fraction .7 has simply been ignored.

```
10 CLS:X=3.7
20 PRINT"X% IS AN INTEGER"
30 X%=X
40 PRINT:PRINT"ITS VALUE IS ";X%
50 PRINT:PRINT" X# IS A DOUBLE-PRECISION
 NUMBER"
60 PRINT:X#=SQR(X)
70 PRINT"VALUE OF X# IS ";X#
80 PRINT"THIS IS THE ROOT OF X"
90 PRINT"IT'S SHOWN TC 15 DECIMAL PLACES
!"
100 PRINT"THAT'S DOUBLE PRECISION - CAN
YOUR"
110 PRINT"SUPER XYZ COMPUTER DO THIS?"
```

Fig. 3.18. Integer and double-precision variables illustrated.

The advantage of using integer variables is twofold. One advantage is that any arithmetic, apart from division, that we carry out on integers is exact, with no rounding up or down needed. Division is the exception because fractions are ignored. If, for example, we have A%=5 and B%=2 then A%/B% gives 2, not 2.5. The other advantage of integers is that they need less memory to store. A program that uses integers will also run much faster than one which uses any other type of variables.

The ordinary number variables, such as X, are called 'single-precision' variables. The precision is good enough for most purposes, but not all. If you are writing an accounts program which handles several hundred thousand pounds in items, then the use of single-precision numbers will cause small errors. The Colour Genie allows you to use what are called 'double-precision numbers'. These give the accuracy that you would normally associate with business computers costing more than ten times as much as the Genie. Such

```
10 DEFSTRA
20 DEFINTB
30 DEFDBLC
40 INPUT"YOUR NAME,PLEASE";A
50 INPUT"A NUMBER,PLEASE";D
60 PRINT:PRINT"USING INTEGERS, 17% OF YO
UR NUMBER IS"
70 B=17*D/100
80 PRINT B
90 PRINT
100 PRINT"IN DOUBLE-PRECISION, THAT'S-"
110 C=17+D/100
120 PRINT C
130 PRINT:PRINT"O.K. ";A;" ?"
```

Fig. 3.19. Defining variables by DEFSTR DEFDBL and DEFINT. These instructions allow you to define certain letters as being particular types of variables. When DEFSTRA is used, for example, any variable that starts with A, like A1, AB, AZ, etc., will be a string variable, and no dollar sign is needed to mark this.

numbers take more memory space to store, and can't be processed as quickly as single precision numbers. For some uses, though, they are indispensable. Figure 3.19 illustrates the use of double-precision numbers in lines 6ϕ to 11ϕ. Notice, incidentally, that the same 'name' can be used for several quite different variables. A program can use X, X$, X%, and X# for different quantities with no confusion. Programmers, being merely human, usually prefer to make a note of how each variable is being used.

Chapter Four

Repetitions and Decisions

Loops

One of the activities for which a computer is particularly well suited is repeating a set of instructions. Every computer is therefore well equipped with instructions that will cause repetition, and the Colour Genie is no exception. We'll start with the simplest of these 'repeater' actions, GOTO.

GOTO means exactly what you would expect it to mean – go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again.

```
10 PRINT"GENIE  GENIE   GENIE  GENIE   G
ENIE"
20 GOTO10
30 REM PRESS BREAK TO STOP
```

Fig. 4.1. Repeating an action in a loop. You'll have to press the BREAK key to stop this one.

Figure 4.1 shows an example of a very simple repetition or *loop*, as we call it. Line 1∅ contains a simple PRINT instruction. When line 1∅ has been carried out, the program moves on to line 2∅, which instructs it to go back to line 1∅ again. This is a never-ending loop, and it will cause the screen to fill with the word GENIE until you press the BREAK key to 'break the loop'. Any loop that appears to be running forever can normally be stopped by pressing the BREAK key, though if this does not work, you will have to press the two RST keys together.

Now try a loop in which there is slightly more noticeable activity.

Figure 4.2 shows a loop in which a different number is printed out each time the computer goes through the actions of the loop. We call this 'each pass through the loop'. Line 1∅ sets the value of the variable N at 1∅. This is printed in line 2∅, and then line 3∅ decrements the value of N. Line 4∅ forms the loop, so that the program will cause a very rapid countdown to appear on the screen. Once again, you'll have to use the BREAK key to stop it.

```
10 CLS:N=10
20 PRINTN
30 N=N-1
40 GOTO20
50 REM BREAK NEEDED AGAIN!!
```

Fig. 4.2. A countdown loop – the BREAK key will be needed again.

Now an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we prefer not to use! We don't always have an alternative, but there is one – the FOR...NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The instruction that are repeated are the instructions that are placed between FOR and NEXT. Figure 4.3 illustrates a very simple example of the FOR...NEXT loop in action. The line which contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, N is the counter variable, and its limit numbers are 1 and 1∅. The NEXT is in line 4∅, and so anything between lines 2∅ and 4∅ will be repeated.

```
10 CLS
20 FOR N=1 TO 10
30 PRINT"GENIE MAGIC"
40 NEXT
```

Fig. 4.3. Controlling a loop with a count, using FOR and TO with NEXT.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print GENIE MAGIC ten times. At the first pass through the loop, the value of N is set to 1, and the phrase is printed. When the NEXT instruction is encountered, the computer increments the value of N, from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 1∅ that has been set. If it doesn't, then line 3∅ is repeated, and this will continue until the value of N exceeds 1∅ – we'll look at that point later. The effect in this example is to cause ten repetitions.

You don't have to confine this action to single loops either. Figure

```
10 CLS
20 FOR N=1 TO 10
30 PRINT"COUNT IS ";N
40 FORJ=1 TO 500:NEXT
50 CLS:NEXT
```

Fig. 4.4. Nested loops. The loop that used J is completely enclosed (nested) inside the loop that uses N.

4.4 shows an example of what we call *nested loops*, meaning that one loop is contained completely inside another one. When loops are nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 2∅, using variable N which goes from 1 to 1∅ in value. Line 3∅ is part of this outer loop, printing the value that the counter variable N has reached. Line 4∅, however, is another loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J, and we have put nothing between the FOR part and the NEXT part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The last action of the main loop is clearing the screen in line 5∅. The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. In this example we have used NEXT to indicate the end of each loop. We could use NEXT J in line 4∅ and NEXT N in line 5∅ if we liked, but this is not essential. It also has the effect of slowing the computer down, though the effect is not important in this program. When you do use NEXT J and NEXT N, you must be absolutely sure that you have put the correct variable names following each NEXT. If you don't, the computer will stop with a NF error – meaning NEXT without FOR.

Even at this stage it's possible to see how useful this FOR...NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

FOR N=1 TO 9 STEP 2

which would cause the values of N to change in the sequence 1,3,5,7,9. When we don't type STEP, the loop will always use increments of 1.

Figure 4.5 illustrates an outer loop which has a step of −1, so that the count is downwards. N starts with a value of 1∅, and is

```
10 CLS
20 FOR N=10 TO 0 STEP -1
30 PRINT N;" SECONDS AND COUNTING"
40 FOR J=1TO500:NEXT
50 CLS:NEXT
60 PRINT"BLASTOFF!!"
```

Fig. 4.5. A countdown program, using two loops.

decremented on each pass through the loop. Line 4∅ once again forms a time delay so that the countdown takes place at a civilised speed. This is a particularly useful way of slowing the countdown. If we want to speed the rate up, the easiest way is to use an integer variable such as N% in place of N. If we do this, however, we can't use steps that contain fractions, like .1.

```
10 CLS
20 FOR N=1TO5
30 PRINT N
40 NEXT
50 PRINT "N IS NOW ";N
60 FOR N=5 TO 1 STEP -1
70 PRINT N
80 NEXT
90 PRINT"N IS NOW ";N
```

Fig. 4.6. The value of the counter variable at the end of a loop will have gone one step beyond the limit.

Every now and again, when we are using loops, we find that we need to use the value of N after the loop has finished. It's important to know what this will be, however, and Fig. 4.6 brings it home. This contains two loops, one counting up, the other counting down. At the end of each loop, the value of the counter variable is printed. This reveals that the value of N is 6 in line 5∅, after completing the FOR N = 1 TO 5 loop, and is ∅ in line 9∅ after completing the FOR N = 5 TO 1 STEP−1 loop. If you want to make use of the value of N, or whatever variable name you have selected to use, you will have to remember that it will have changed by one more step at the end of the loop.

One of the most valuable features of the FOR...NEXT loop, however, is the way in which it can be used with number variables

```
10 CLS
20 A=2:B=5:C=10
30 FOR N=A TO B STEP B/C
40 PRINT N
50 NEXT
```

Fig. 4.7. Using a FOR...NEXT loop with variables and an expression.

instead of just numbers. Figure 4.7 illustrates this in a simple way. The letters A, B and C are assigned as numbers in the usual way in line 2∅, but they are then used in a FOR...NEXT loop in line 3∅. The limits are set by A and B, and the step is obtained from an expression, B/C. The rule is that if you have anything that represents a number or can be worked out to give a number, then you can use it in a loop like this.

Loops and decisions

It's time to see loops being used rather than just demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. From what we have done so far, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set. The program of Fig. 4.8 does just this.

```
10 TT=0:CLS
20 PRINT"TOTALLING NUMBERS PROGRAM"
30 PRINT"ENTER EACH NUMBER AS REQUESTED"
40 PRINT"THE PROGRAM WILL GIVE THE TOTAL
"
50 FOR N=1 TO 10
60 PRINT"NUMBER ";N;" PLEASE ";
70 INPUT J:TT=TT+J
80 NEXT
90 PRINT:PRINT"TOTAL IS ";TT
```

Fig. 4.8. A totalling program for ten entries only.

The program starts by setting a number variable TT to zero. This is the number variable that will be used to hold the total, and it has to start at zero. As it happens, the Colour Genie arranges this automatically at the start of a program, but it's a good habit to ensure that everything that has to start with some value actually does. We can't, incidentally, use TO for this variable, because TO is a reserved word, part of the FOR...NEXT set of words.

Lines 2∅ to 4∅ issue instructions, and the action starts in line 5∅. This is the start of a FOR...NEXT loop which will repeat the actions of lines 6∅ and 7∅ ten times. Line 6∅ reminds you of how many numbers you have entered by printing the value of N each time, and line 7∅ allows you to INPUT a number which is then assigned to variable name J. This is then added to the total in the second half of line 7∅, and the loop then repeats. At the end of the program, the variable TT contains the value of the total, the sum of all the number that have been entered.

It's all good stuff, but how many times would you want to have just ten numbers? It would be a lot more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like ∅ or 999. A value like this is called a 'terminator', something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of ∅ is very convenient, because if it gets added to the total it won't make any difference.

```
10 CLS:PRINT"ANOTHER TOTAL FINDER"
20 PRINT"THE PROGRAM WILL TOTAL NUMBERS
FOR YOU"
30 PRINT"ENTER A ZERO TO STOP"
40 TT=0
50 INPUT"NUMBER,PLEASE";N
60 TT=TT+N
70 PRINT"TOTAL SO FAR IS ";TT
80 IF N<>0THEN 50
```

Fig. 4.9. A totalling program that tests in line 8∅ for the entry of a zero in line 5∅.

Figure 4.9, therefore, shows an example of this type of program in action. We can't use a FOR...NEXT loop, because we don't know in advance how many times we might want to go through the loop, so we have to go back to using GOTO. This time, however, we'll keep GOTO under closer control – the word won't even appear in the program! This time the instructions appear first, but we still have to make the total variable TT equal to zero in line 4∅. Each time you type a number, then, in response to the request in line 5∅, the number that you type is added to the total in line 6∅, and line 7∅ prints the value of the total so far. Line 8∅ is the loop controller, and the key to the control is the instructions word IF. IF is used to make a test, and the test in line 8∅ is to see if the value of N is not equal to zero. The odd-looking sign that is made by combining the 'less-than' and the 'greater than' signs, <>, is used to mean 'not equal', so the line reads: 'if N is not equal to zero, then (GOTO) line 5∅'. We can put the GOTO in, or leave it out. Since it's just a few more letters to type, I've left it out.

The effect, then, is that if the number which you have typed in line 5∅ was not a zero, line 8∅ will send the program back to repeat line 5∅. This will continue until you do enter a zero. When this happens, the test in line 8∅ fails (N is zero), and the program looks for a line

9ϕ. Since it can't find one, it stops. This kind of action is called a 'repeat...until' loop.

Now this allows you much more freedom than a FOR...NEXT loop, because you are not confined to a fixed number of repetitions. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now.

Decisions, decisions

We can make a number of types of comparisons between number variables or numbers, and these are listed in Fig. 4.10. The mathematical signs are used for convenience, and you have to remember which way round the 'greater than' and 'less than' signs have to be. It's important to note that the equals sign means

Sign	Meaning
=	Quantities are identical.
>	Quantity on left is greater than quantity on right.
<	Quantity on left is less than quantity on right.
⩾	Quantity on left is greater than or equal to quantity on right.
⩽	Quantity on left is less than or equal to quantity on right.
<>	Quantities are not equal.

Fig. 4.10. The tests that we can make on variables.

'identical to' when it is used in a test like this. If A is 3.9999999 and B is 4.ϕϕϕϕϕϕϕ then a test such as IF A = B will fail – A is not identical to B, even though it is close enough to be equal to our eyes. The important point here is that the numbers we see on the screen have been rounded, so that PRINT A in the example above might give the result 4. The test, however, is made on the numbers which have not been rounded.

```
10 CLS
20 PRINT"PRESS Y OR N KEY"
30 PRINT" -THEN RETURN"
40 INPUT A$
50 IF A$="Y"THEN PRINT "THAT'S YES"
60 IF A$="N"THEN PRINT "THAT'S NO"
```

Fig. 4.11. Testing a string for a Y or N reply.

Figure 4.11 shows another test – this time on string variables. The instructions are in lines 2ϕ to 3ϕ – you are asked to type the Y or N key. Line 4ϕ gets your answer; you have to type Y or N and then press RETURN. The key that you have pressed has its value assigned to A$, so that A$ should be Y or N. Lines 5ϕ and 6ϕ then analyse this result. If the key that you pressed was neither Y nor N, nothing is printed by line 5ϕ or line 6ϕ.

The test in this example is for identity. Only if A$ is absolutely identical to Y will the phrase 'THAT'S YES' be printed. If you typed a space ahead of Y, or a space following, or typed y in place of Y, then A$ will not be identical, and the test fails. Failing means that A$ is not identical to Y and everything that follows THEN in that line will be ignored. It's up to you to form these tests so that they behave in the way that you want!

```
10 CLS
20 PRINT"TYPE Y OR N"
30 INPUT A$
40 IF A$="Y"THEN 100ELSE IF A$="N" THEN
200
50 PRINT"YOUR ANSWER ";A$" IS NOT Y OR N
-PLEASE TRY AGAIN":GOTO30
60 END
100 PRINT"THAT WAS YES!
110 END
200 PRINT"THAT WAS NO!"
210 END
```

Fig. 4.12. Using ELSE with IF and THEN to extend a test.

The Colour Genie, fortunately for you, joins that exclusive band of computers that allows you to extend this IF...THEN test. The extension consists of the instruction word ELSE, and it offers an alternative to the test that is carried out by IF. Figure 4.12 illustrates this in action, with another Y/N program. The key line here is line 4ϕ, where we have a pair of tests that are carried out in one line. Line 4ϕ starts with IF A$ = "Y", and normally if A$ is not identical to Y the rest of the line would be ignored. The presence of the word ELSE, however, forces the computer to carry out whatever follows ELSE if the first test fails. Let's see how this works.

If A$ is Y, then the first test in line 4ϕ succeeds, and the program moves to line 1ϕϕ. This prints a message, and the program ends. If A$ is N, then the first test in line 4ϕ fails, but the presence of ELSE forces the computer to carry out the piece of program that follows ELSE. This is another test, so that if A$ is N, the program jumps to line 2ϕϕ, prints a different message, and ends.

If both tests fail, though, the program will move from line 4∅ to line 5∅. Your answer was not exactly Y or N, so that you are asked to try again, and the GOTO3∅ at the end of line 5∅ causes the program to repeat from line 30. This line constitutes a *mugtrap*, a way of trapping mistakes. Very often when you have a choice of answers, you want to be sure that only certain replies are permitted. A mugtrap is a section of program that is intended to deal with an incorrect entry. A good mugtrap should show the user the error of his/her ways, and indicate what answer or answers might be more acceptable. This is very often important, because an incorrect entry in some types of program could cause the program to stop with an error message showing. For the skilled programmer (this will be you, later!) this is just a minor annoyance, but for the inexperienced user it can cause a minor panic. A good program doesn't allow any entries that would cause the program to stop. Mugtraps are our method of ensuring this.

```
10 CLS:X=RND(10)
20 PRINT"GUESS THE NUMBER!"
30 PRINT:PRINT"IF YOU GET NEAR, I'LL TEL
L YOU"
40 INPUT N
50 IF N=X THEN PRINT"SPOT ON!":END
60 IF ABS(N-X)<3 THEN PRINT"CLOSE - IT W
AS ";X:END
70 GOTO10
```

Fig. 4.13. A number-guessing game which uses a test.

Just to emphasise the sort of power that these simple instructions give you, Fig. 4.13 illustrates a very elementary number-guessing game. Line 1∅ clears the screen, and the X = RND(1∅) step causes variable X to take a value that lies between 1 and 1∅. We can't predict what this value will be, because RND means 'select at random' – a whole number is picked, somewhere in the range of 1 to 1∅. If we had programmed RND(1∅∅), the range would have been from 1 to 1∅∅ and so on. RND picks numbers randomly enough for games purposes, but not quite randomly enough for serious statistical users. In lines 2∅ and 3∅, the instructions ask you to guess the size of the number, with the difference that you don't have to find it exactly. You enter your number at line 4∅, and the tests are made in lines 5∅ and 6∅. If the number that you picked is identical to the random number, then you get the SPOT ON message in line 5∅, and the program ends. The less obvious test is in line 6∅. The expression N−X is the difference between your guess, N, and the number X. If your guess is larger than the number, then N−X is a positive number.

If your guess is less than X, then N−X is a negative number. The effect of ABS, however, is to make any number positive, so that if X were 5 and you guessed 6 or 4, then ABS(N−X) would come to 1. If you get a difference of 1 or 2 (less than 3), the message in line 6∅ is printed. If you don't get anywhere near, the program repeats because of its GOTO1∅ in line 7∅. It's very simple, but quite effective. How about devising a scoring system?

Single key reply

So far, we have been putting in Y or N replies with the use of INPUT, which means pressing the key and then pressing RETURN. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press RETURN. For snappier replies, however, there is an alternative in the form of INKEY$. INKEY$ is an instruction that carries out a check of the keyboard to find if a key is pressed. This checking action is very fast, and normally the only way that we can make use of it is by placing the INKEY$ instruction in a loop which

```
10 CLS
20 PRINT"PRESS ANY KEY..."
30 K$=INKEY$:IF K$=""THEN30
40 PRINT"IT WAS ";K$
50 REM SOME KEYS DO NOT GIVE A CHARACTER
```

Fig. 4.14. The INKEYS loop. You can use this for one-key answers.

will repeat until a key is pressed. Figure 4.14 shows such a loop. The INKEY$ instruction will produce a string quantity when any key is pressed, so we assign INKEY$ to a string variable, K$. In this way, when any key is pressed, the quantity that it represents will be assigned to K$, and if K$ is a 'blank string', meaning that no key was pressed, the line loops back to its start again. Note how we indicate a blank string by using two quotes with no space between them. By using the program of Fig. 4.14 you can see the effect of pressing different keys. By changing line 5∅ to GOTO2∅, you can make this program repeat until you press the BREAK key. In this way, you can find which keys will have the effect. Some keys will produce no visible character on the screen in line 4∅, but will nevertheless allow the program to jump out of its loop in line 3∅. Note, by the way, that one key certainly won't work in this way – the BREAK key!

Menus and subroutines

A choice of two items, such as in Fig. 4.11, isn't exactly a consumer's dream, not in the West anyway. We can extend the choice by a program routine that is called a *menu.* A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. We could use a set of lines such as:

IF K = 1 THEN 1ØØØ
IF K = 2 THEN 2ØØØ

and so on. There is a much simpler method, however, which uses a new instruction ON N GOTO, where N is a number variable. You can use any number variable, of course, not just N.

```
10 CLS
20 PRINTTAB(18)"MENU"
30 PRINT:PRINT
40 PRINT"1. ENTER NAMES"
50 PRINT"2. ENTER PHONE NUMBERS"
60 PRINT"3. LIST ALL NAMES"
70 PRINT"4. LIST LOCAL NUMBERS"
80 PRINT"5. END PROGRAM."
90 PRINT
100 PRINT"PLEASE SELECT BY NUMBER"
110 K$=INKEY$:IF K$=""THEN 110
120 K=VAL(K$):IF K<1OR K>5 THEN PRINT"IN
CORRECT CHOICE - 1 TO 5 ONLY":PRINT"PLEA
SE TRY AGAIN":GOTO110
130 ON K GOTO150,160,170,180,190
140 END
150 PRINT"NAMES SECTION":END
160 PRINT"NUMBERS SECTION":END
170 PRINT"LIST OF NAMES":END
180 PRINT"LOCAL NUMBERS":END
190 END
```

Fig. 4.15. A typical menu choice using ON K GOTO.

Figure 4.15 shows a typical menu that uses this instruction. Lines 1Ø to 9Ø present the menu items on the screen, and line 1ØØ then invites you to pick one item by typing its number. The INKEY$ loop in line 11Ø keeps the program looking for a key until you make your choice, and then line 12Ø tests your choice with a mugtrap. There's a new instruction, VAL, in line 12Ø. VAL means 'number value', and it's used to convert a number that is in string form back into number form. This has to be done because INKEY$ produces a string variable, and you can't compare a string with a number (nor a rose with a carrot). By using K=VAL(K$) you get a number variable K which will hold a number that is in the correct form to be compared. If you had pressed a letter key then K will be zero.

The choice is then made in line 13ϕ, with the ON K GOTO instruction. Now what happens here? If K equals 1, then the first line number that follows GOTO is used. If K equals 2, then the second line number following GOTO is used, and so on. All that you have to do is to arrange the line numbers in the same order as your choices. You needn't have a list that looks neat. A line such as ONKGOTO5ϕ,216,484,714,1ϕϕϕ would be just as satisfactory so long as these numbers contained the start of routines that dealt with the menu choices. In this example, the line numbers simply lead to PRINT instructions so as to keep the example reasonably short.

This type of menu selection is useful, but an even more useful method makes use of *subroutines*. A subroutine is a section of program which can be inserted anywhere that you like in a longer program. A subroutine is inserted by typing the instruction word GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike

```
10 CLS
20 PRINT"THIS IS A ";
30 GOSUB1000
40 PRINT"SUBROUTINE":PRINT:PRINT
50 PRINT"RED LIGHT AND GREEN LIGHT MAKE
";:GOSUB1000:PRINT:PRINT"LIGHT"
60 PRINT:PRINT:END
1000 PRINT"YELLOW ";
1010 RETURN
```

Fig. 4.16. Illustrating the use of a subroutine.

GOTO, however, GOSUB offers an automatic return. The word RETURN is used at the end of the subroutine lines, and it will cause the program to return to the point immediately following the GOSUB. Figure 4.16 illustrates this. When the program runs, line 2ϕ prints a phrase, with the semicolon used to prevent a new line from being selected. The GOSUB1ϕϕϕ in line 3ϕ then causes the word YELLOW to be printed, but the RETURN in line 1ϕ1ϕ will send the program back to line 4ϕ, the instruction that immediately follows the GOSUB1ϕϕϕ. This action will also occur even when the GOSUB is part of a multistatement line, as line 5ϕ demonstrates. The GOSUB1ϕϕϕ will cause the word YELLOW to be printed, but the return is to the PRINT instructions that follow GOSUB1ϕϕϕ in

```
10 CLS:PRINT
20 PRINTTAB(10)"CHOOSE YOUR MONSTER"
30 PRINT
40 PRINTTAB(2)"1.VAMPIRE."
50 PRINTTAB(2)"2.WEREWOLF."
60 PRINTTAB(2)"3.ZOMBIE."
70 PRINTTAB(2)"4.SGT. MAJOR."
80 PRINTTAB(2)"5.SHOP STEWARD."
90 PRINT:PRINT"SELECT NUMBER,PLEASE"
100 K$=INKEY$:IF K$=""THEN 100
110 K=VAL(K$):IF K>5ORK<1THEN PRINT"FAUL
TY SELECTION - 1 TO 5 ONLY. PLEASE TRY A
GAIN.":GOTO100
120 ON K GOSUB 1000,2000,3000,4000,5000
130 PRINT"THAT'S THE END"
140 END
1000 PRINT"BLOOD,BLOOD":RETURN
2000 PRINT"HOWL,SNARL":RETURN
3000 PRINT"STILL ASLEEP":RETURN
4000 PRINT"YOU ORRIBLE LITTLE MAN":RETUR
N
5000 PRINT"EVERYBODY OUT":RETURN
```

Fig. 4.17. Using subroutines in a menu choice.

line 5ϕ; it doesn't jump to line 6ϕ. This example is, of couse, a yellow subroutine.

Now for something more serious. Figure 4.17 shows subroutines in use as part of an (imaginary) games program. Lines 1ϕ to 8ϕ offer a choice, and line 9ϕ invites you to choose. The familiar INKEY$ and mugtrap actions follow, and then line 12ϕ causes the choice to be carried out. This time, however, the program will return to whatever follows the choice. For example, if you pressed key 1, then the subroutine that starts at line 1ϕϕϕ is carried out, and the program returns to line 12ϕ to check if you might also want subroutines 2ϕϕϕ, 3ϕϕϕ 4ϕϕϕ, or 5ϕϕϕ. Since the value of K is still 1, the program then goes to line 13ϕ and ends. If line 1ϕϕϕ had altered the value of K, however, you could find that a second subroutine was selected following the first one. The use of a subroutine is extremely useful in menu choices, but it's even more useful for pieces of program that will be used several times in a program. Take a look at Fig. 4.18 by way of an example. The subroutine is simply the INKEY$ 'press-any-key' routine, and it's one that you are likely to use many times in the course of any program. Putting the INKEY$ into a subroutine means that you need to type these program lines once only. Wherever you need the action, you simply type GOSUB1ϕϕϕ (or whatever line number you have used), and the routine will be inserted when the program runs.

```
10 CLS
20 PRINT"CHOOSE 1 OR 2, PLEASE"
30 GOSUB1000
40 A=VAL(K$)
50 PRINT"CHOOSE Y OR N,PLEASE"
60 GOSUB1000
70 B$=K$
80 PRINT"YOU CHOSE ";A;" AND ";B$
90 END
1000 K$=INKEY$:IF K$=""THEN1000
1010 RETURN
```

Fig. 4.18. A subroutine used by several parts of a program.

Figure 4.19 shows an elaboration on this one. The trouble with INKEY$ is that it doesn't remind you that it's in use, there's no question mark printed as there is when you use INPUT. The subroutine in lines 1ØØØ to 1Ø4Ø remedies that by causing an asterisk to flash while you are thinking about which key to press. The asterisk is flashed by alternately printing the asterisk and the delete step. Yes, CHR$(8) is what causes the delete action, and we'll look at this instruction method later. Meantime, make friends with subroutines. They are not just a useful way of obtaining an action at several points in a program, they are an indispensable aid to program planning, of which there's much more in Chapter 6.

```
10 CLS
20 PRINT"CHOOSE 1 OR 2, PLEASE"
30 GOSUB1000
40 A=VAL(K$)
50 PRINT"CHOOSE Y OR N,PLEASE"
60 GOSUB1000
70 B$=K$
80 PRINT"YOU CHOSE ";A;" AND ";B$
90 END
1000 K$=INKEY$
1010 IF K$<>""THEN RETURN
1020 PRINT"*";
1030 PRINTCHR$(8);
1040 GOTO1000
```

Fig. 4.19. Using a flashing asterisk subroutine to improve INKEYS.

Chapter Five
Programs with Strings Attached

String functions

In Chapter 3, we took a fairly brief look at number functions. If numbers turn you on, that's fine, but *string functions* are in many ways more interesting. What makes them that way is that the really eye-catching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's look at an example. Figure 5.1 shows a program that prints GENIE as a title. What makes it more eye-catching is the fact that the word is printed with twelve hashmarks (#) on each side.

```
10 CLEAR100
20 CLS
30 A$=STRING$(12,"#")+"GENIE"+STRING$(12
,"#")
40 PRINTTAB(5)A$
```

Fig. 5.1. STRINGS used to create lines of characters.

The hash marks are produced by a string function whose instruction word is STRING$. STRING$ means 'make a string out of', and it has to be followed by two items placed within brackets and separated by a comma. The first of these items is the number of identical characters that you want in this string. The second item is the character itself. In this example, we've used the # character, and it has had to be placed between quotes.

STRING$ is a useful way of creating strings of one character, and it's particularly useful when we come to look at graphics characters. There are, however, strings attached, as it were. One is string space. When your Colour Genie is switched on, it reserves a small amount of memory for storing strings. The amount is fairly small, only enough for 50 characters, because a surprising number of programs use less than this. When you make a lot of use of the STRING$ instructions, however, you can bite deeply into this small allocation, and this will cause your program to stop with an OS message when the allocation is used up. OS means 'out of string space', and it requires you to reserve more space and try again. You can reserve more string space by the CLEAR instruction which is illustrated in line 1Ø of Fig. 5.1. By using CLEAR 1ØØ, we reserve enough memory for 100 string characters. We don't need as much as this for the program, but it's as well to be on the safe side.

```
32        33  !   34  "   35  # 36  $

37  %     38  &   39  '   40  ( 41  )

42  *     43  +   44  ,   45  - 46  .

47  /     48  0   49  1   50  2 51  3

52  4     53  5   54  6   55  7 56  8

57  9     58  :   59  ;   60  < 61  =

62  >     63  ?   64  @   65  A 66  B

67  C     68  D   69  E   70  F 71  G

72  H     73  I   74  J   75  K 76  L

77  M     78  N   79  O   80  P 81  Q

82  R     83  S   84  T   85  U 86  V

87  W     88  X   89  Y   90  Z 91  [

92  \     93  ]   94  ^   95  _ 96  `

97  a     98  b   99  c   100  d 101  e

102  f    103  g  104  h  105  i 106  j

107  k    108  l  109  m  110  n 111  o

112  p    113  q  114  r  115  s 116  t

117  u    118  v  119  w  120  x 121  y

122  z    123  {  124  |  125  } 126  ~

127  ■
```

Fig. 5.2. The ASCII codes, as produced by a printer.

The other point about STRING$ is that the second item in the brackets can be a number, with no quotes. Each character that is used by the Genie is represented by a code number, using what we call ASCII code. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced Askey) code is one that is used by most computers. Figure 5.2 shows a printout of the ASCII code numbers and the characters that they produce. In place of the hashmark that we have used between quotes in Fig. 5.1, then, we could have used the number 35, making the instruction into STRING$(12,35), which is shorter.

```
10 CLS:CHAR3
20 A$=STRING$(40,141)
30 PRINTTAB(14)"GENIE MAGIC"
40 PRINTA$
```

Fig. 5.3. Using ASCII codes of more than 127 to get graphics characters.

The number characters of ASCII code extend only from 32 to 127. The code numbers above 127 are used by the Colour Genie for other purposes, and we can select how we make use of them. Figure 5.3 gives a flavour of this, something that we'll investigate in more detail in Chapter 7. By using CHAR3 in line 1∅, we select which group of characters will appear on the screen for code numbers 128 to 255. The program then uses a string of the character whose code is 141 to make an underline for the title that is printed in line 3∅.

The logic of LEN

String variables allow us to carry out a lot of operations that can't be done with number variables. One of these operations is finding out how many characters are contained in a string. Since a string can contain up to 255 characters, a method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable, within brackets, and the result of using LEN is always a number so that we can print it or assign it to a number variable.

Figure 5.4 shows a simple example of LEN in use. Line 2∅ assigns

```
10 CLS
20 A$="GENIE"
30 PRINT"THERE ARE ";LEN(A$);" LETTERS
IN ";A$
```

Fig. 5.4. Using LEN to measure the length of a string.

a variable and line 3∅ tells you how many letters are in this variable. This is hardly earth-shattering, but we can turn it to very good use, as Fig. 5.5 illustrates. This program uses LEN as part of a subroutine which will print a string called T$ centred on a line. This is an extremely useful subroutine to use in your own programs, because its use can save you a lot of tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string T$. This number is then divided by two, and subtracted from 20, using the formula that we saw first in Chapter 2. If the number of characters in the string is an odd number, then 2∅-LEN(T$)/2 will contain a .5, but this is completely ignored by TAB when the string is printed.

```
10 CLS
20 T$="GENIE MAGIC"
30 GOSUB1000
40 PRINT
50 T$=STRING$(20,"*")
60 GOSUB1000
70 PRINT:PRINT
100 END
1000 PRINTTAB(20-LEN(T$)/2);T$
1010 RETURN
```

Fig. 5.5. A subroutine for centring titles, using LEN.

The whole process can be done in one line, in this case line 1∅∅∅ of the subroutine. Once in place, we can call this subroutine to centre anything that has the name T$. In line 2∅, T$ is assigned to the words GENIE MAGIC, and this phrase is printed centred. In line 5∅, T$ is assigned to a string of twenty asterisks, using STRING$ (some computer owners would have to type twenty asterisks here!). This is also printed centred by the subroutine.

Notice, by the way, that if we want anything printed centred by this subroutine, we have to give it the variable name of T$. This action is called 'passing a variable' to the subroutine, and it's something that we have to keep a careful eye on when we use subroutines. You can't expect a subroutine that is written to print T$ centred to have any effect on a string called A$.

By the left, slice

The next group of string operations that we're going to look at are called slicing operations. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string.

All of that might not sound terribly interesting, so take a look at Fig. 5.6. The string A$ is assigned in line 2∅, and sliced in line 3∅. What's printed in line 4∅ is the word IMAGE. Now how did this

```
10 CLS
20 A$="IMAGINATION"
30 B$=LEFT$(A$,4)+"E"
40 PRINT B$
50 PRINT:PRINT
```

Fig. 5.6. Slicing the left side of a string.

happen? The instruction LEFT$ means 'copy part of a string starting at the left-hand side'. LEFT$ has to be followed by two quantities, within brackets and separated by a comma. The first of these is the variable name for the quantity that we want to slice, A$ in this example. The second is the number of characters that you want to slice (copy , in fact) from the left-hand side. The effect of LEFT$(A$,4) is therefore to copy the first four letters from IMAGINATION, giving IMAG. The last part of line 3∅ adds an E to the four-sliced letter, so giving us the word IMAGE printed on the screen in line 4∅.

For a more serious use of this instruction, take a look at Fig. 5.7.

```
10 CLS:PRINT@80,""
20 INPUT"YOUR SURNAME,PLEASE";SN$
30 INPUT"YOUR FIRST NAME,PLEASE";FM$
40 PRINT
50 PRINT"YOU'LL BE KNOWN AS ";LEFT$(FM$,
1)+"."+LEFT$(SN$,1)+".";" AROUND HERE"
```

Fig. 5.7. Extracting initials from a name.

This has the effect of extracting your initials from your name, and it's done by using LEFT$ along with a bit of concatenation. The INPUT steps in lines 2∅ and 3∅ find your surname and forename, and assign them to variable names SN$ and FM$. We can't use the more obvious FN$ for forename, because FN is a reserved word in BASIC, though not actually in the BASIC of the Colour Genie! Line 5∅ then prints your initials by using LEFT$ to extract the first letter of each string. The letters are then assembled along with full stops, using concatenation in line 5∅. If you have two players in a game, it's often useful to show the initials and score rather than printing the full name, but the full names can be held stored for use at various stages in the game.

All right, Jack?

String slicing isn't confined to copying a selected piece of the left-hand side of a string. We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the LEFT$ one, but it's useful, none the less. Figure 5.8 illustrates the use of the instructions to avoid having to type a word

```
10 CLS
20 A$="GENIE MAGIC"
30 PRINT:PRINT
40 PRINT"IT'S ALL ";RIGHT$(A$,5)
```

Fig. 5.8. Slicing the right side of a string.

over again. There are more serious uses than this. You can, for example, extract the last four figures from a string of numbers like ∅1∅-242-7∅16. I said a string of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type N = ∅1∅-242-7∅16 then the computer assumes that you want to subtract 242 from 1∅ and 7∅16 from that result. The value for N is then −7248, which is not exactly what you had in mind! If you use N$="∅1∅−242−7∅16" then all is well.

Now we can get quite a lot of interesting effects from LEFT$ and RIGHT$. Take a look at Fig. 5.9, for example, which does odd things with the letters of your name. The program prompts you to

```
10 CLS
20 INPUT"YOUR NAME,PLEASE";A$
30 L=LEN(A$)
40 FOR N=1TOL
50 PRINTLEFT$(A$,N);TAB(20)RIGHT$(A$,N)
60 NEXT
```

Fig. 5.9. Using string slicing for an unusual print effect. Try also the effect of a delay loop between lines 4∅ and 5∅!

enter your name in line 2∅, and the name is assigned to A$. In line 3∅, we use LEN so that the number variable L contains the total number of characters in your name. This will include spaces and hyphens – nobody's likely to use asterisks and hashmarks! Line 4∅ starts a loop which used the total number of characters as its end limit. Line 5∅ is the action line. When N is 1, line 5∅ prints the first letter on the left of your name on to the left-hand side of the screen, and the first letter on the right of your name on the right-hand side.

On the next pass through the loop, a new line is selected, and two letters are printed. This continues until the entire name is printed. If you use a LEFT$ or RIGHT$ with a number that is more than the number of letters in the strong, then you simply get the whole string.

Pig in the MIDdle?

There's another string slicing instruction which is capable of much more than either LEFT$ or RIGHT$. The instruction word is MID$, and it has to be followed by three items, within brackets and using commas to separate the items. Item 1 is the name of the string that you want to slice, as you might expect by now. The second item is a number which specifies where you start slicing. This number is the number of the character counted from the left-hand side, and counting the first character as 1. The third item is another number, the number of characters that you want to slice, going from left to right and starting at the position that was specified by the first number.

It's a lot easier to see in action than to describe, so try the program in Fig. 5.10. Line 2∅ assigns A$ to COLOUR GENIE, and line 3∅

```
10 CLS
20 A$="COLOUR GENIE"
30 L=LEN(A$)
40 FOR N=1TOL
50 PRINTMID$(A$,N,1);" ";:NEXT
60 PRINT:PRINT
70 FOR N=1TOL
80 PRINTMID$(A$,N,1)+"+";:NEXT
```

Fig. 5.10. Using MIDS to slice from any part of a string.

finds L, the number of characters in COLOUR GENIE. The loop that starts in line 4∅ then prints letters taken from the words COLOUR GENIE. With the value of N equal to 1, the letter that is sliced is C, because its position in the word is 1, and we're copying one letter from this position. If we used MID$(A$,1,2), we would get CO, and if we used MIDS(A$,3,2) we would get LO. As it is, we select one letter at a time, and print a space. The semicolon in line 5∅ then ensures that the next sliced letter is printed on the same line. The net effect is that the letters are printed spaced out. The second loop in lines 7∅ and 8∅ performs the same kind of effect, but places a + sign between the letters rather than a space.

One of the features of all of these string slicing instructions is that we can use variable names or expressions in place of numbers. Figure 5.11 shows a more elaborate piece of slicing which uses expressions. It all starts innocently enough in line 2∅ with a request

```
10 CLS
20 INPUT"YOUR NAME,PLEASE";NM$
30 L=LEN(NM$):C=INT(L/2)+1
40 FOR N=1TOC
50 PRINTTAB(20-N)MID$(NM$,C-N+1,N*2-1)
60 NEXT
```

Fig. 5.11. A letter pyramid program that makes use of MID$ with number variables and expressions.

for your name. Whatever you type is assigned to variable NM$, and in line 3∅ a bit of mathematical juggling is carried out. How does it work? Suppose you type DONALD as your name. This has six letters so, in line 3∅, L is assigned to 6, and C is the whole number part of L/2 (equal to 3), plus 1, making 4. Line 4∅ then starts a loop of 4 passes. In the first pass you print at TAB(19) (because N=1), the MID$ of the name using C−N+1, which is 4−1+1=4, and N*2−1, which is also 1. What you print is therefore MID$(NM$,4,1), which is A in this example. On the next run through the loop, N is 2, C−N+1 is 3, and N*2−1 is also 3. What is printed is MID$(NM$,3,3), which is NAL. The loop goes on in this way, and the result is that you see on the screen a pyramid of letters formed from your name. It's quite impressive if you have a long name!

Some odd (and even) characters

It's time now to look at some other types of string functions. We've met VAL previously – it's used to convert a number that is in string form back into number form so that we can carry out arithmetic. There's an instruction that does the opposite conversion, STR$. When we follow STR$ by a number, number variable, or expression within brackets, we carry out a conversion to a string variable. We can then print this as a string, or assign it to a string variable name, or use string functions like LEN, MID$ and all the others. Figure 5.12 illustrates these processes – with a warning! Lines 1∅ to 3∅ show that we can do arithmetic on N$ if we use VAL with it. Line 5∅ converts the number variable V into string form. Now V has been assigned to the number 2 in line 1∅, and we would expect just one character to be present in the string. Line 6∅ reveals that there are two! The reason is that when we use STR$ to convert a number into

```
10 N$="22.5":V=2
20 CLS:PRINT
30 PRINTN$;" TIMES ";V;" IS ";V*VAL(N$)
40 PRINT
50 V$=STR$(V)
60 PRINT" THERE ARE ";LEN(V$);" CHARACTE
RS IN ";V; "!"
70 PRINT
80 PRINTN$;" ADDED TO ";V$;" GIVES ";N$+
V$
```

Fig. 5.12. Using VAL to convert a string into number form, and STRS to convert a number into string form – along with a space!

string form, a space is left at the left-hand side of the string in case we want to put in a sign, + or −. This space is, of course, an extra character, which explains why 2 appears to consist of two characters, and 42 of three characters. Line 8∅ shows the strings being concatenated, just to emphasise the difference between string variables and number variables.

If you hark back to Fig. 5.2, now, you'll remember that we introduced the idea of ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by the function ASC, which is followed, within brackets, by a string character. The result of ASC is a number, the ASCII code number for that character. If you use ASC("GENIE"), then you'll get the code for the G only because the action of ASC includes rejecting more than one character. Figure 5.13 shows this in action. String variable A$ is assigned in line 1∅ and in line 3∅ a loop starts which will run through

```
10 A$="GENIE"
20 CLS:PRINT
30 FORN=1TO LEN(A$)
40 PRINTASC(MID$(A$,N,1));" ";
50 NEXT
```

Fig. 5.13. Printing the ASCII codes of a name with the letters spaced out.

all the letters in A$. The letters are picked out one by one, using MID$, and the ASCII code for each letter is found with ASC. The space between quotes, along with the semicolons in line 4∅, make sure that the codes are all printed on one line with a space between the numbers. Simple, really!

ASC has an opposite function, CHR$. What follows CHR$, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction PRINT CHR$(65), for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for coding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.14 illustrates this use. Line 5ϕ contains an INKEY$ loop to make the program wait for you. When

```
10 CLS:PRINT
20 PRINT"WHAT'S THE MOST MAGICAL GIRL IN
 SONG?"
30 PRINT
40 PRINT"PRESS ANY KEY FOR THE ANSWER"
50 K$=INKEY$:IF K$=""THEN50
60 PRINT
70 FOR J=1TO31:READN
80 PRINTCHR$(N);
90 NEXT
100 END
110 DATA71,69,78,73,69,32,87,73,84,72,32
,84,72,69,32,76,73,71,72,84,32,66,82,79,
87,78,32,72,65,73,82
```

Fig. 5.14. Using CHRS to find the letter that corresponds to an ASCII code number.

you press a key, the loop that starts in line 7ϕ prints 31 characters on the screen. Each of these is read as an ASCII code from a list, using a READ...DATA instruction in the loop. The PRINTCHR$(N) in line 8ϕ then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 20, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent de-coders!

The law about order

We saw earlier, in Fig. 4.10, how numbers can be compared. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign, =, means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes.

The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 5.15 illustrates this use of comparison using the = and > symbols. Line 2ϕ assigns a

```
10 CLS
20 A$="QWERTY"
30 PRINT:INPUT"TYPE A WORD";B$
40 IF B$=A$ THEN PRINT "SAME AS MINE":EN
D
50 IF A$>B$ THEN C$=A$:A$=B$:B$=C$
60 PRINT"ORDER IS ";A$;" THEN ";B$
70 END
```

Fig. 5.15. Comparing words by using the ASCII codes of their letters. Words can be put into alphabetical order in this way.

nonsense word – it's just the first six letters on the top row of letter keys. Line 3ϕ than asks you to type a word. The comparisons are then carried out in lines 4ϕ and 5ϕ. If the word that you have typed, which is assigned to B$ is identical to QWERTY, then the message in line 4ϕ is printed, and the program ends. If QWERTY comes earlier in an index than your word, then line 5ϕ is carried out. If, for example, you typed TAPE, then since T comes after Q in the alphabet and has an ASCII code that is greater than the code for Q, your word A$ scores higher than B$, and line 5ϕ swaps them round. This is done by assigning a new string, C$ to A$ (so that C$ = "QWERTY"), then assigning A$ to B$ (so A$ = "TAPE"), then B$ to C$ (so that B$="TAPE"). Line 6ϕ will then print the words in the order A$ and then B$, which will be the correct alphabetical order. If the word that you typed comes earlier than QWERTY, for example, PERIPHERAL, then A$ is not 'greater than' B$, and the test in line 5ϕ fails. No swap is made, and the order A$, then B$, is still correct. Note the important point, though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts.

Put it on the list

The variable names that we have used so far are useful, but there's a limit to their usefulness. Figure 5.16 illustrates this. Lines 1ϕ to 4ϕ

```
10 CLS
20 FORN=1TO10
30 A(N)=RND(100)
40 NEXT
50 PRINT
60 PRINTTAB(15)"MARKS LIST"
70 PRINT:FOR N=1TO10
80 PRINT"ITEM ";N;" RECEIVED ";A(N)" MAR
KS"
90 NEXT
```

Fig. 5.16. A list that uses subscripted variables, otherwise known as an *array*. This way, you don't need to type a different variable name for each number on the list!

generate an (imaginary) set of examination marks. This is done simply to avoid the hard work of entering the real thing! The variable in line 3ϕ is something new, though. It's called a *subscripted variable*, and the 'subscript' is the number that is represented by N. The name that we use has nothing to do with computing, it's a name that was used long before computers were around. How often do you make a list with the items numbered 1,2,3...and so on? These numbers, 1,2,3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. A member of this group like A(2) has its name pronounced as 'A-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity numbers. Since the number can be a number variable or an expression, this allows us to work with any item of the list. Figure 5.16 shows the list being constructed from the FOR...NEXT loop in lines 2ϕ to 4ϕ. Each item is obtained by finding a random number between 1 and 1ϕϕ, and is then assigned to A(N). Ten of these 'marks' are assigned in this way, and then lines 6ϕ to 9ϕ print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

Figure 5.17 extends this another step further. This time you are invited to type a name and a mark for each of ten items. After you have pressed RETURN following the mark entry, the part of the screen where you entered the name and mark is cleared by lines 6ϕ and 7ϕ. When the list is complete, the screen is cleared and a total variable is set to zero in line 9ϕ. The list is then printed neatly, and on each pass through the loop the total is counted up (in line 13ϕ) so that the average value can be printed at the end. The important point here is that it's not just numbers that we can keep in this list form. The correct name for the list is an array, and Fig. 5.17 uses both a string array (names) and a number array (marks).

```
10 CLS:PRINT:CLEAR500
20 PRINT"PLEASE ENTER NAMES AND MARKS"
30 FOR N=1 TO 10
40 PRINT@120,"NAME";:INPUT N$(N)
50 PRINT"MARK";:INPUTA(N)
60 PRINT@120,STRING$(39,32)
70 PRINTSTRING$(39,32)
80 NEXT
90 CLS:T=0
100 PRINTTAB(8)"MARKS LIST":PRINT
110 FORN=1TO10
120 PRINTTAB(2)N$(N);TAB(20)A(N)
130 T=T+A(N)
140 NEXT
150 PRINT
160 PRINT"AVERAGE IS ";T/10
```

Fig. 5.17. Using both a string and a number array in the same program.

The programs in Figs. 5.16 and 5.17 show you how to set up arrays and use them, but one point has been omitted so far. We were careful to use only ten items in each array. That's because the Colour Genie normally provides memory space for numbers up to ten only. This allows you up to eleven items in an array, in fact, because we can use A(∅) or N$(∅) if we like, but it's still a limitation if you want to use lists of fifty or more items. All that you have to do, though, if you need longer lists, is to instruct the computer to prepare space. This is done by using the instruction word DIM.

DIM means 'dimension', and the instruction consists of naming each variable that you will use for arrays, and following the name with the maximum number, within brackets, that you expect to use. You aren't forced to use this number, but you must not exceed it. If you attempt to use a number that is higher than the one you have put into the DIM instruction, then the computer will stop with an error message – BS, meaning 'bad subscript'. You will have to change the DIM instruction and start again – which will be tough luck if you were typing in a list of 1∅∅ names! Note that you can dimension more than one variable in a DIM line, as Fig. 5.18 shows.

```
1∅ DIM A$(5∅),B(1∅∅),C(5∅),D$(5∅)
```

Fig. 5.18. Dimensioning several array variables in one line.

Rows and columns

You can imagine an array as a list of items, one after the other, but there is a variety of array which allows a different kind of list, called a *matrix*. A matrix is a list of groups or items, with all the items in a group related. We could think of a matrix as a set of rows and columns, with each group taking up a row, and the items of a group in separate columns. Take a look at Fig. 5.19 to see how this works.

```
10 CLS
20 FORN=1TO3
30 FORJ=1TO2
40 READ N$(N,J)
50 NEXTJ,N
60 FORN=1TO3
70 PRINTTAB(5);N$(N,1);TAB(25);N$(N,2)
80 NEXT
100 DATAHORSE,FOAL,COW,CALF,DOG,PUP
```

Fig. 5.19. A simple matrix of string values.

We use here a variable N$ which has two subscript numbers. The first number is the row number, the second is the column number, and we need two FOR...NEXT loops to read data into this matrix. This is carried out in lines 2∅ to 5∅. Notice the shortened NEXT J,N in line 5∅, which is a way of writing NEXT J:NEXT N. The items are then printed in columns by the loop in lines 6∅ to 8∅. In this loop, the variable N is used as the row number and we use the column numbers 1 and 2. The rows contain animal names, and the columns separate the different names that we use for adult and for young animals respectively.

Figure 5.20 shows a much more ambitious matrix program. This

```
10 CLS:DIM A$(50,2)
20 FOR N=1TO50
30 PRINT@160,"NAME";:INPUT A$(N,1)
40 PRINT"TEL. NUMBER";:INPUT A$(N,2)
50 PRINT@160,STRING$(39,32)
60 PRINTSTRING$(39,32)
70 NEXT
80 CLS:PRINT"LIST COMPLETE"
90 PRINT"PICK AN INITIAL"
100 INPUT J$
110 FORN=1 TO 50
120 IF J$=LEFT$(A$(N,1),1)THEN GOSUB 500
130 NEXT
140 END
500 PRINT"NAME- ";A$(N,1)
510 PRINT"NUMBER- ";A$(N,2)
520 RETURN
```

Fig. 5.20. A name-and-number matrix.

one uses a row number greater than 1∅, and so has to be dimensioned in line 1∅. The idea is to store sets of names and telephone numbers which are fed in by you in the course of the loop in lines 2∅ to 7∅. Once the matrix has been filled, you can pick an initial letter for a name, and ask the computer to print out the name and number that it has located. I've left out mugtraps just to keep this example reasonably short, but you would certainly need some sort of mugtrap, even if only in the form of a message like:

```
135 PRINT" SORRY, CAN'T FIND ";J$;" ENTRIES"
```

The next thing you need is to be able to record array and matrix items so that you can get all the data on tape, and play it back into the computer when you want it, without more typing. Next chapter, please!

Chapter Six
Filing and Designing

Data filing

I left you at the end of Chapter 5 with a program that will create a matrix of names and telephone numbers. Each time you switch off the Colour Genie, or type NEW and press RETURN, you will lose all the information that you painstakingly typed in. That's completely unsatisfactory and so the Colour Genie provides, as all 'serious' computers must, for storing this type of data on tape (or on disk). The big advantage of storing the data separate from the program is that it allows you to write other programs that use the same data. If you have a tape that consists of name and address data for your friends, for example, you can write several programs that use that data. One might be an envelope printer, another could print party invitations, a third put messages on Christmas cards. You need a printer for each of these actions, certainly, but the point that I'm making is that one set of data will serve for a lot of uses. The next step is to know how to record and replay such data.

To start with, let's consider how we record and replay simple variables. The important instruction here is PRINT#−1, and you have to be fussy about how you type it. The hashmark is used in the USA in the same way that we use the abbreviation *No.* to mean 'number'. This instruction can therefore be read as 'print number dash one', and it's a way of causing a variable to be 'printed' not to the screen or the paper printer but to the cassette recorder. The '1' in this instruction is used because of the possibility of expanding the system so as to use a second recorder which would be controlled by a PRINT#−2, instruction.

The PRINT#−1, must be followed by a list of the variables whose values are to be recorded, with the variable names separated by commas. It's the values, not the names that are recorded, incidentally, so we don't have to use the same names when we play the data back.

Now whenever the Colour Genie comes to the PRINT#−1, instruction, it will immediately send the data out, whether you have a cassette recorder connected or not. This is undesirable, because you need some time to gather your wits about you and get a cassette ready and have the recorder switched on and the RECORD and PLAY keys pressed before the data goes out. Each PRINT#−1, instruction, therefore, needs some previous step in which you are warned to get a cassette ready, told to prepare the machine, and given time to do all this.

Figure 6.1 illustrates this with a simple example. Lines 1ϕ to 3ϕ simply assign three variables, consisting of two strings and a

```
10 A$="COLOUR GENIE"
20 CLS:INPUT"YOUR NAME,PLEASE";B$
30 C=32
40 PRINT"PLEASE PREPARE A DATA CASSETTE
FOR"
50 PRINT"RECORDING.PRESS THE PLAY AND RE
CORD"
60 PRINT"KEYS OF THE RECORDER, THEN ANY
KEY OF"
70 PRINT" THE GENIE."
80 K$=INKEY$:IF K$=""THEN80
90 PRINT"..RECORDING NOW..."
100 PRINT#-1,A$,B$,C
110 PRINT"RECORDING COMPLETE - PRESS STOP
 KEY OF"
120 PRINT"RECORDER NOW, AND REWIND CASSE
TTE"
```

Fig. 6.1. Recording simple variable values.

number, which can be recorded. Lines 4ϕ to 7ϕ then print the message about getting a cassette ready and pressing the keys on the recorder. You should not use the same cassette for data as you use for programs. Remember that you will have to wind the cassette on past the 'leader' section.

Once you have chosen a cassette, wound it on, put it into the recorder, and pressed the PLAY and RECORD keys, you can let the Colour Genie go ahead with recording the data. Line 8ϕ has provided a 'hang-up' step while you have been doing all this, and when you press a key on the Colour Genie's keyboard, the data will be recorded. Line 9ϕ is important. You need to know what is going on, because some types of data (long lists, for example) can take a long time to record on cassette, and you'll ruin the recording if you turn off the recorder too soon. The 'recording now' message is a useful reminder that this important action is going on. The actual recording step is in line 1ϕϕ, and when it is complete, the computer will move on to lines 11ϕ and 12ϕ, printing another message. This one lets you know that recording is finished, and that you can now stop the recorder.

The messages and the INKEY$ action are as important a part of the routine as the recording itself! There's nothing worse than a program that does things without warning, doesn't let you know what is going on and doesn't tell you when it has finished. This set of program lines is one that you will probably use many times, so that it should be written as a subroutine. After reading Chapter 9, you might also want to include some other reminders in the form of sounds.

Having recorded our data, we need to be able to replay it, otherwise all the effort is wasted. The replay routine follows pretty much the same pattern as the record routine except that the instruction that is used is INPUT#–1, in place of PRINT#–1,. You don't have to use the same variable names following INPUT#–1, as you used after the PRINT#–1, that recorded the data, but you should use the correct variable types. If, for example, you recorded two strings and a number in that order, then when you replay you will need two string variables and a number variable of the same type in that order also.

```
10 CLS:PRINT
20 PRINT"PLEASE PREPARE DATA CASSETTE TO
 REPLAY."
30 PRINT"PRESS ANY KEY WHEN READY"
40 K$=INKEY$:IF K$=""THEN 40
50 PRINT"PLAYING NOW..."
60 INPUT#-1,A$,B$,C
70 K$=INKEY$:IF K$=""THEN70
80 CLS
90 PRINT@160"THIS IS ";B$+"'s";" "N;"K";
" "A$
```

Fig. 6.2. Replaying simple variable values.

Figure 6.2 illustrates this point. Once more, we need a message to remind us to get the data cassette into the recorder and rewind it to the correct position for replaying. Line 4ϕ is the familiar INKEY$ loop which keeps the computer waiting until you press a key. When you press any key on the Colour Genie keyboard, the message in line 5ϕ appears, but no data is actually replayed until you press the PLAY key on the cassette recorder and the tape starts to replay. The order of pressing keys in this case is just the opposite of the order that we use for recording. If you press the PLAY key of the recorder first, some of the signals might be replayed before you press the Colour Genie key to start the reading. The INPUT#−1, in line 6ϕ causes the data to be read from the tape and assigned to the variable names A$, B$ and C. We don't have to use the same variable names, but we do, remember, have to keep to the same order. Finally, line 7ϕ tells you that the data has been replayed, so that you can now proceed. Another 'press-any-key' step then gives you time to attend to the recorder before you start making use of the replayed data in lines 9ϕ and 1ϕϕ.

Tape-an-array time

As it happens, the ability to record or replay the odd variable or two isn't particularly important. The things that we particularly want to record are long arrays, because they represent such a large typing effort that we don't want to repeat them. On the Colour Genie,

```
10 CLS
20 PRINT
30 FORN=1TO10
40 A(N)=RND(90)+10
50 NEXT
60 PRINT"PLEASE PREPARE CASSETTE FOR REC
ORDING"
70 PRINT"PRESS ANY KEY TO START
80 GOSUB260
90 PRINT"RECORDING NOW"
100 FOR N=1TO10
110 PRINT#-1,A(N)
120 NEXT
130 PRINT"RECORDING FINISHED.."
140 PRINT"PRESS ANY KEY TO TRY PLAYBACK"
150 GOSUB260
160 PRINT"PLEASE PREPARE CASSETTE"
170 PRINT"PRESS ANY KEY...."
180 GOSUB260
190 FOR J=1TO10
200 INPUT#-1,B(J)
210 NEXT
220 PRINT"REPLAY COMPLETE-PLEASE PRESS S
TOP KEY"
230 FOR N=1TO10
240 PRINT B(N):NEXT
250 END
260 K$=INKEY$:IF K$=""THEN260 ELSE RETUR
N
```

Fig. 6.3. Recording and replaying an array. The same method is used for a matrix, but two loops are needed.

recording and replaying an array uses the same type of instructions as recording and replaying simple variables. The difference is that the PRINT#–1, and the INPUT#–1, instructions are held within a FOR...NEXT loop.

Figure 6.3 shows an example which is arranged to illustrate these actions without requiring you to do an excessive amount of typing. Lines 3ø to 5ø construct an array that consists of randomly selected numbers. We've kept to 1ø numbers so that we don't need a DIM instruction. The messages in lines 6ø and 7ø instruct you to get a data cassette ready, and the INKEY$ routine has been written in the form of a subroutine in line 26ø and called in line 8ø. The recording step which starts at line 9ø uses a loop, with the PRINT#–1, instruction contained within the loop. Lines 13ø and 14ø then inform you when recording is complete. This takes rather longer than you might expect, because the rate of putting array data on to the tape is quite slow.

The replay routine starts with the 'press-any-key' step in line 15ø. As usual, we need a message that will remind us to get the cassette ready, and another 'press-any-key' in line 18ø is used. The replay instructions are in lines 19ø to 21ø, and whenever the replay is complete, the set of numbers will be printed. Once again, the steps in lines 19ø to 21ø take quite a lot of time to complete.

The time that is needed for recording and replaying arrays is due to the fact that the PRINT#–1, and INPUT#–1, instructions deal with only one item at a time. Most computers contain instructions which will group the data into sets and then record a set at a time. When this grouped data is replayed, another instruction 'ungroups' the data again. This greatly reduces the time that is needed for recording and replaying arrays. There are ways of doing this grouping and ungrouping for yourself so that the Colour Genie array recording procedures can be almost as fast as that of other computers. These steps are rather beyond the scope of a beginner's book, but I have included them in my book *Some Useful BASIC Subroutines* (Newnes).

Do it your way

A computer, running other peoples' programs, can give you a lot of pleasure and profitable activity. Nothing, however, quite matches the challenge and enjoyment of writing your own programs and making the computer do what you want. Next to rubbing a brass lamp, it's the most magical experience there is. I could devote a whole book just to this subject, but in this section, I'll just throw out some hints and examples. As your experience grows, you can then profit from this, and for further reading, I suggest you may find helpful the advice of Mike James in *The Complete Programmer* (also published by Granada).

The main problem in writing your own program is knowing where to start. The usual answer is 'somewhere else'! The place *not* to start is at the keyboard. It is possible to write a program by sitting down at the keyboard and typing, but the program that is produced in this way is not very satisfactory. To be blunt, Dr Frankenstein's attempts would make yours look rather badly sewn together. If you want a program that will actually work before you lose interest in it, you have to start by planning it on paper.

The first step of the planning stage consists of writing down what you want the program to do. You think that's unnecessary? It's quite astonishing how easily you forget what you want a program to do when you are trying to find out what it actually does. Making a note of your intentions at the start means that your goal is in front of you, rather than a dim memory from a few days ago. Write it down! Figure 6.4 shows an example from an imaginary 'how far away do my friends live' program.

The program must:
(a) let me enter names, addresses and distances;
(b) make selection by name, address or distance;
(c) print lists in order of distance;
(d) allow deletions, additions, changes.

Fig. 6.4. A program specification for an imaginary program.

Now at this point, advice becomes a bit more tricky. I don't know what type of programs you may want to write. Some of you may want to write data processing programs, creating files of information about your interests (stamps, railways, team records, butterflies ...). Some of you may want the Genie to work for you keeping accounts, printing receipts, advice notes, balance sheets and other business purposes. Some may want to write long adventure-style games in which situations are described on the screen and you are given a choice of responses. Some may be interested in action games, arcade style, guessing games, educational games ... the list is endless.

There are, however, some common features. I'll start by considering types of programs that feature menu choices, because a lot of programs, games or otherwise, are of this type, and they are similar enough to be treated together. Action games are different, and really fast action games are just not suitable for programming in BASIC at all.

Once you have written down what the program is supposed to do, what's next? A good next step is to make a list of inputs and outputs. At various stages in the program, you will want to be able to type information on the keyboard. This might simply be Y or N replies, or it could be names, addresses and other information. At other places in the program, you will expect to see information on the screen. This might, for example, be a list of all the stamps with one particular watermark, all butterflies with blue wings, all steam locomotives with six coupled wheels, all indoor plants which need feeding once a week. Sorting out these inputs and outputs is the next step for your program design. Figure 6.5 shows what we might expect for the program whose aims were listed in Fig. 6.4. This list of inputs and outputs puts a bit more flesh on to the bones of the original scheme.

INPUTS:
Type names, addresses, distances.
Type selection of choices.
Type alterations, deletions, additions.

OUTPUTS:
Indicate what is being entered, query mistakes.
Show selections.
List in order of distance.
Prompt for next choice.

Fig. 6.5. Inputs and outputs for the program.

Now this type of designing follows a system that is called 'top-down' design. The general aim is to get the outlines correct and then to fill in the detail later, as an artist does. The programming method that helps us to the greatest extent in this is the subroutine. Wherever in the program we are uncertain of how we are going to program anything, we can put GOSUB, followed by a line number, and then a REM line to remind us what we want the subroutine to do. The golden rule is to avoid details until the last stages of planning.

What's next? If the program is 'menu-driven', meaning that it

Menu

1. Enter new list.
2. Add to existing list.
3. Select by name.
4. Select by town.
5. Print list up to given distance.
6. Alter list item.
7. End program.

Fig. 6.6. How the program menu might appear.

contained a list of choices early on, the next step is to design the menu. For our example, the menu that we want to see on the screen might look as in Fig. 6.6. Notice that the last menu choice is END PROGRAM. This is put in for two reasons. One is because it's a good idea to return to the menu after any one of the menu choices has been carried out. It's only by including a 'quit' option like this

```
10 GOSUB500
11 REM DIMENSIONING
20 GOSUB600
21 REM TITLE
30 GOSUB700
31 REM INSTRUCTIONS
40 T$="MENU":GOSUB800
41 REM 800 IS CENTRE ROUTINE
50 PRINT:PRINT
60 PRINT"1. Enter new list."
70 PRINT"2. Add to existing list."
80 PRINT"3. Select by name."
90 PRINT"4. Select by town."
100 PRINT"5. Print list up to given dist
ance."
110 PRINT"6. Alter list item."
120 PRINT"7. END PROGRAM."
130 PRINT:PRINT"PLEASE SELECT BY NUMBER."
140 GOSUB 9000
141 REM INKEY$ ROUTINE
150 ON K GOSUB1000,2000,3000,4000,5000,6
000,7000
160 CLS:PRINT
170 GOTO50
```

Fig. 6.7. A suggested 'core' program. The actions are carried out by the use of subroutines.

that you can stop the program gracefully (yes, of course you can press BREAK or even the RST keys, but that's an admission of defeat!). The other point is that if you change your mind about starting the program, an option like this allows you to stop before you go any further. The menu stage, then, is the end of the beginning for planning.

You can now think about writing the program! Not all of it, because all of the detail has been kept for later, but you can write the 'core' section. Figure 6.7 shows a suggested core program for our example. It's very short, because all of the detail has been left out. Items like printing a title centred, instructions, and the effects of menu choice are all delegated to subroutines. This might seem slightly unnecessary at first glance. The instructions, for example, would normally be looked at when the program begins and not again. By putting them into a subroutine, however, you accomplish two things. One is that it makes it easy for you to call on the instructions again at any place in the program if you decide to do so. You may find that you need this facility later, when you have been using the program for a time. The other point is that by having the instructions in a subroutine, you can change the instructions easily without having to disturb the rest of the program too much. It also makes the program easier to read. Have you ever tried to decipher a program that made you wade through twenty lines of instructions before you come to the *real* program?

It's much more obvious why some of the other subroutines exist. Items like the Y / N choice, 'press-any-key', and so on, are stock items in any menu-driven program. We expect to use each of them more than once in the program, so there's no doubt about the need to write them as subroutines.

Now at this stage you have a choice of action. You can continue planning on paper, or you can take a break to sit at the keyboard and enter your core program. The really professional way is to plan it all on paper down to the last detail, enter it in one marathon typing session, and then try to make it work. Every second blue moon, it does. For my money, entering the core program at this stage is a better course of action. Why? Because you can test it, and get the feeling that it is coming along. You need all the encouragement you can get at this stage! When you attempt to run the core program, of course, you find that it doesn't. There are no subroutines yet, so wherever the program comes across a GOSUB it will hang up with the error message UL. This means 'unlisted line' - you have asked the computer to run a line that hasn't been written.

This doesn't stop you from testing your core program, though. If you place some PRINT instruction at each of the GOSUB lines, and

```
10 GOSUB500
11 REM DIMENSIONING
20 GOSUB600
21 REM TITLE
30 GOSUB700
31 REM INSTRUCTIONS
40 T$="MENU":GOSUB800
41 REM 800 IS CENTRE ROUTINE
50 PRINT:PRINT
60 PRINT"1. Enter new list."
70 PRINT"2. Add to existing list."
80 PRINT"3. Select by name."
90 PRINT"4. Select by town."
100 PRINT"5. Print list up to given dist
ance."
110 PRINT"6. Alter list item."
120 PRINT"7. END PROGRAM."
130 PRINT:PRINT"PLEASE SELECT BY NUMBER.
"
140 GOSUB 9000
141 REM INKEY$ ROUTINE
150 ON K GOSUB1000,2000,3000,4000,5000,6
000,7000
160 PRINT"PRESS ANY KEY...":GOSUB 9000:C
LS
170 GOTO50
500 PRINT"THIS DIMENSIONS":RETURN
600 PRINT"THIS TITLES":RETURN
700 PRINT"INSTRUCTIONS":RETURN
800 PRINTT$;" (CENTRE THIS)":RETURN
1000 PRINT"ENTRY":RETURN
2000 PRINT"ADD":RETURN
3000 PRINT"SELECT NAME":RETURN
4000 PRINT"SELECT TOWN":RETURN
5000 PRINT"DISTANCE SELECTION":RETURN
6000 PRINT"ALTERATION":RETURN
7000 END
9000 PRINT"PLEASE PRESS THE APPROPRIATE
KEY"
9010 K$=INKEY$:IF K$=""THEN 9010
9020 K=VAL(K$)
9030 REM NEED MUGTRAPS LATER
9040 RETURN
```

Fig. 6.8. Placing a PRINT instruction in each subroutine line so that the core program can be tested.

follow it with a RETURN instruction in the next line number, you can then run the core section right through. All that you will get as an output will be a set of messages, but at least you will know that the core section works. This is a good time to try out the mugtrapping, for example, on the menu section, and find out what happens when you put in a silly answer. It's always much easier to sort out errors on a short piece of program than on a long program. If you have a printer, then sorting out a long program is relatively easy because you can print out all of it, and simply read through the lines. If you're working on the screen display, however, without a printer, then testing one stage at a time makes a lot more sense.

Once you have your core program running perfectly, with 'dummy' GOSUB lines, record it. I always make two recordings, both at the start of fresh C90 cassettes. This way, if anything happens as you add to the program, you can always get the program back in this form. Even a short program still represents a fair amount of typing, and a short program which has been tested and is known to work is a valuable piece of property, worth more than its weight in recording tape.

From now on, you have to write subroutines. Each subroutine is designed in just the same way as you designed the main core program – you can write down what you expect it to do, what you expect to type or see, then make a stab at the processing. Once again, you can leave the fine detail to other subroutines, and when you have completed each routine, you can test it. If you load the machine from a tape that contains everything that you have typed and tested previously, your program will grow one subroutine at a time. Only the new additions will have to be tested. It's good for your confidence, and it keeps the whole program in a manageable state. By the time that you type the last subroutine, you will have a program that should be just about what you intended.

As you develop the program, put in a few REMS to remind you of what each section is supposed to do. Put a REM in after each subroutine call, and another one after each subroutine start. I prefer to use the '1' lines (like 1ø1, 1øø1, 4øø1) for this purpose, as Fig. 6.9

```
1ø1 REM END OF MAIN SECTION
1øø1 REM FLASH ASTERISK ROUTINE
4øø1 REM RECORDING ROUTINE
```

Fig. 6.9. Using REM lines.

illustrates. Never have a REM line as the start of a subroutine because this can cause you problems later. It's useful, when you have designed a long program to keep two versions. One version should be complete with all its REMs, and detailed instructions. This will be your reference copy, the one that you will use when you want to improve the program. There should be another copy, stripped of all REMs, and with only brief instructions. This one will take up less memory and will run faster, and it will be your working copy, the one that you use. The point here is that if you have lines like:

```
100 GOSUB 1000
1000 REM SUBROUTINE FOR RECORDING
```

then when you remove REM lines you will remove line 1000, and the program will crash when it reaches line 100! In addition, you can buy programs that remove REMs and spaces, and the use of such a program will cause you problems if it strips out all the lines to which a GOSUB refers.

Off the menu

Though a lot of programs depend on the use of a menu, there are many others that don't. Among these are several types of games programs. How do we design a game which incorporates no menu steps?

The first part of design is the same – you have to write down what the game is intended to do. You will also have to write the rules for the game. This is by far the most difficult part of designing a games program, and why there are several hundred versions of 'haemorrhoids in space'. You then have to decide what the scoring system shall be, because the essence of a game is competition of some sort, so there has to be a scoring system or a win/lose decision at some point.

```
10 GOSUB500:REM DIMENSIONS
20 GOSUB600:REM TITLE
30 GOSUB700:REM INSTRUCTIONS
40 GOSUB800:REM CREATE CHARACTERS
50 GOSUB900:REM DRAW BACKGROUND
60 GOSUB1000:REM PLAY!
70 GOSUB2000:REM CHECK SCORE
80 GOSUB3000:REM KILLED?
90 IF KL=1 THEN GOSUB9000:REM DEAD
100 GOTO60:REM PLAY AGAIN
```

Fig. 6.10. A core program for a game.

Once you have decided these difficult and important points, the rest is comparatively plain sailing! A game is usually a more 'visual' type of program, so what we are about to cover in Chapters 7 and 8 will be of considerable interest to you. The greatest amount of program effort will probably go on these graphics effects – but these can be put into subroutines. The core program for your game is probably going to be as simple and straightforward as the core for a menu-driven program. Figure 6.10 shows an example. The real heart of the game will be the subroutines that carry our the PLAY action and the SCORE action – and you'll write these later!

Chapter Seven
Graphics, Plain and Fancy

The characters that we have printed on the screen up to now have been mainly 'alphanumeric'. That means that they are number digits, letters, or punctuation marks, the sort of characters that a typewriter can produce. All modern computers will also permit you to display shapes as well, shapes that can be built up into pictures. These are the characters that we call 'graphics characters'.

These graphics shapes that are provided as standard are the same overall size as the ordinary alphanumeric characters. This means that the number of these graphics characters that we can place on the screen is the same as the number of alphanumeric characters – 24 lines of 40 characters per line for the Colour Genie, which makes 960 characters. This style of graphics is called *low resolution*, meaning that only a comparatively small number of positions on the screen (960) are being controlled. The *high resolution* graphics of the Colour Genie allows you to control a total of 15360 points on the screen, as we shall see in Chapter 8.

In this Chapter, we're going to deal with the low resolution graphics characters. There are three points that we have to deal with. These are how to obtain the characters, how to print graphics and text in colour, and how to create our own character shapes to our own design.

Keyboard graphics

The graphics shapes that are illustrated in Appendix B can all be entered from the keyboard directly. While your Colour Genie is on, pressing the MODSEL (mode-select) key which is on the left of the space-bar, will select graphics. After this key has been pressed once, pressing any letter key will give the graphics character that is shown on the left-front of the key. Pressing any letter key

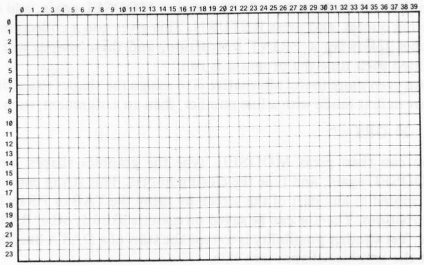

Fig. 7.1. A 40 × 24 grid for planning low resolution graphics shapes.

along with the SHIFT key will give the graphics character that is shown on the right-front of that key. By pressing the MODSEL key again, you return to normal key use.

This method of entering graphics shapes directly from the keyboard is very convenient, but it still requires some planning if you want to use the shapes to make a pattern rather than just for underlining text. The easiest scheme to follow is to draw the shape that you want on tracing paper over a 40 × 24 grid (Fig. 7.1). If the shape can't be drawn exactly with the graphics patterns that are shown on the keys, then modify the traced outline until each square of the pattern (as seen when the pattern is held over the 40 × 24 grid) corresponds to a graphics shape. Use the shapes shown in Appendix B to guide you. If you simply want to see how your pattern looks on the screen, then type the first row of shapes (remember to use MODSEL), then use the down-cursor key (and right-cursor if you need it) to get to the required place on the next line. You can then type another line of shapes and continue in this way until the whole pattern has been placed on the screen.

This is something that you have to try for yourself, because my printer won't reproduce these shapes on paper. If you want to put your pattern into program lines, start in ordinary text mode. Type a line number, then PRINT", and then press MODSEL. You can then type in a row of shapes, and use the cursor-down key to go to another row. You can type several lines in this way, depending on how many characters you put into each line, because the maximum number of characters that is permitted in a BASIC line is 255! You can then return to text mode by pressing MODSEL again, add the closing quotes, and press RETURN. This gives you a collection of shapes following a PRINT instruction, and these shapes will be printed each time the program is run.

An even more useful alternative is to assign the shapes to a string. Start in text mode with a line number, then an assignment, such as:

```
2ϕ GR$ ="
```

You can then press MODSEL, type the graphics shapes, press MODSEL again, and type the closing quotes. This string can then be printed at any part of the screen by using the TAB or PRINT@ instructions. The line can be saved on cassette like any other BASIC line. Yet another possibility is to type the start of a DATA line, then put the graphics shapes into this.

Meet the characters

We can use characters that are obtained directly from the keyboard for our own graphics programs, but this isn't the easiest method, and it's not so easy to write down what you want, or to print the program. Several of the programs in this chapter could have had their graphics characters put into place directly from the keyboard, but we've used the ASCII code numbers instead. The programs, this way, can be tested and the listing printed immediately afterwards, ensuring accuracy.

To start with, remember how the alphanumeric characters are coded. Each alphanumeric character, including space and punctuation marks, is allocated an ASCII code number. We can print the characters that correspond to the codes by using a PRINT CHR$(N) instruction, where N is the number. Figure 7.2 is a reminder of what these characters look like for each number code from 32 to 127. Code numbers lower than 32 are used for special effects, and the codes that are available are noted in Appendix C. The codes from 128 to 255 are used for graphics characters, so we need to take a look at these now.

We can't simply use the program of Fig. 7.2 with changed numbers in the loop, however. The reason is that the Colour Genie can be switched to any one of four different 'character sets'. These are detailed in Fig. 7.3 and also in the manual. The numbers that lie between 32 and 127 are the ordinary alphanumeric characters, and they remain unchanged no matter which character set we happen to

```
10 FOR N=32 TO 127
20 PRINT N;" "CHR$(N);"   ";:NEXT
```

Fig. 7.2. A program for printing the ASCII characters on the screen. RUN this and look at the Colour Genie's stock of 'alpha' characters.

ASCII codes	CHAR1	CHAR2	CHAR3	CHAR4
128 to 191	prog	prog	graph	graph
192 to 255	prog	spec	prog	spec

Note:
Codes 32 to 127 are always the alphabetical–numerical characters.
prog means programmable characters.
spec means special characters.

Fig. 7.3. The character sets of the Colour Genie. Only the character sets for numbers 128 to 255 are shown, because numbers 32 to 127 are always the 'alpha' set.

be using. The remaining numbers, however, are split into two groups, 128 to 191 and 192 to 255. These groups of numbers can be allocated to three different types of characters. The ordinary 'graphics' characters are the ones that are printed on the fronts of the keys. These characters can be allocated to codes 128 to 191. There are also 'special' characters which are not available from the keyboard so simply, and which can be allocated to codes 192 to 255. The third group consists of programmable characters, which means that the shape of each one of these characters can be controlled by numbers which you have to enter into the memory of the computer.

When you switch the Colour Genie on, it will normally use the first of its four character arrangements, in which all the code numbers 128 to 255 are allocated to the use of programmable characters. If, for example, you type PRINT CHR$(129) to see one of these characters, the result is a blank! There is no character with the code number of 129 until we enter the code numbers that create it! Of the character sets that the Colour Genie can use, four in all, the only one that will produce a character for each and every code number (128 to 255) is set 4. By typing CHAR4 (then press RETURN), or by using CHAR4 at the start of a program, we can select the graphics characters for codes 128 to 191, and the special characters for codes 192 to 255. This is illustrated in Fig. 7.4. The program prints all the characters, using sets 1 to 4, with the number

```
5 FORC=1TO4:CHAR C
7 CLS
10 FOR N=128TO255STEP4
20 PRINTN;" ";CHR$(N),N+1;" ";CHR$(N+1),
N+2;" ";CHR$(N+2),N+3;" ";CHR$(N+3)
25 PRINT
30 GOSUB1000
40 NEXT
45 GOSUB1000:NEXT
50 END
1000 K$=INKEY$:IF K$<>""THEN RETURN
1010 PRINTTAB(1)C;
1020 FORQ=1TO100:NEXT
1030 PRINTCHR$(8);CHR$(8);CHR$(8);
1035 FORQ=1TO100:NEXT
1040 GOTO1000
```

Fig. 7.4. Looking at the graphics characters. When you find an empty set, it's because these are 'prog' codes; they produce nothing until you program them for a character you want (see later).

of the character set printed on the screen as a reminder. Try it – it shows the size and shape of each character on the screen rather better than you can get from drawings of them on paper. Whatever character set was last used in a program will stay selected until it is changed, or until the computer is switched off.

Rainbow Genie

One of the many reasons that you probably had for buying a Colour Genie was that it's a colour computer, so it's time that we took a look at its colour capabilities. The names of the colours are printed on the top row of keys, the number keys, and you can print all of your text, including program lines in colour simply by pressing the CTRL key, releasing it, and then pressing a key which has a colour marked on it. After you have done this, anything that you type will be in colour. This effect lasts until you choose another colour, or until the words or shapes scroll off the screen. You can type a listing in yellow, and then select blue, type LIST, then press RETURN, and see your listing appear in blue. The colour that you select in this way remains in use as the 'foreground colour' until you cancel it by selecting another, or by switching off and on again. You will not, of course, see these colours unless you are using a colour TV, because they don't show up as noticeably different shades of grey on a black/white receiver. You can type each letter of a word in a different colour if you like, or type graphics shapes in a variety of colours. When you list again, however, the colour that you see for everything will be the colour that you most recently selected.

Generally, we want to use colour to produce special effects on the screen, eye-catching titles, and interesting graphics patterns. We need, therefore, to be able to use colour within programs. We can't produce coloured letters in programs just by having used the colour keys when we type the letter, though. For example, if you type:

```
1ø PRINT "LETTERS"
```

and you make each letter of LETTERS have a different colour, you will see these different colours as you type the line, but when you clear the screen and RUN the program line, the word LETTERS will appear in the last colour that was selected, not with a different colour for each letter. If we want to use different colours within programs, then, we have to make use of the COLOUR instruction.

Code number	Colour
1	White
2	Green
3	Red
4	Yellow
5	Orange
6	Blue
7	Cyan
8	Magenta

Fig. 7.5. The set of colours for low resolution graphics and text.

```
10 CHAR4
20 CLS
30 FOR C=1 TO 8
40 COLOUR C
50 PRINT "THIS IS COLOUR ";C
60 NEXT
```

Fig. 7.6. A program which illustrates the colours on the screen. You need that colour TV now!

COLOUR is followed by a number which takes values between 1 and 8. You can actually use numbers up to 16, but the colours that correspond to numbers between 9 and 16 are, viewed on a TV receiver, not very different from the colours in the 1 to 8 set. Figure 7.5 shows this standard colour set. The colour cyan is a mixture of blue and green lights, and magenta is a mixture of red and blue lights. To see the colours on the screen, try the program of Fig. 7.6.

The COLOUR instruction cannot, unfortunately, be used following PRINT. If you want each letter of a word to appear in a different colour, you have to use the method that is illustrated in the Colour Genie manual, reading each letter from a DATA line with a different COLOUR number selected in the loop.

```
10 CHAR4
20 CLS
30 FOR C=1 TO 8
40 COLOUR C
50 GOSUB1000
60 PRINT
65 PRINT"PRESS ANY KEY.."
70 K$=INKEY$:IF K$=""THEN70
80 NEXT
100 END
1000 FOR J=1TO4
1010 FOR N=1TO5
1020 READ A:PRINT CHR$(A);
1030 NEXT:PRINT:NEXT
1040 RESTORE:RETURN
2000 DATA128,32,202,32,144,196,196,196,1
96,196,196,196,196,196,196,230,32,230,32
,230
```

Fig. 7.7. Printing a pattern in different colours. The pattern has been produced by printing graphics shapes whose codes are in line 2ϕϕϕ.

Back to the salt mines, then. Figure 7.7 shows how a pattern can be printed in different colours. The printing of the pattern is done by the subroutine which starts in line 1ϕϕϕ. The method is to read code numbers from a DATA list, and print them by using PRINT CHR$(A). Four lines of five characters each are printed by this routine, and in all eight colours. Notice the use of RESTORE in line 1ϕ4ϕ. This ensures that the next time the subroutine is called, the data will be read again from the start, avoiding the 'out of data' error message (OD) that you would get if all the data had been read and this instruction had been omitted.

How is the shape designed? In just the way that you would expect from our earlier efforts on the keyboard. You slip a piece of tracing paper over the 40 × 24 low resolution planning grid, and trace the outline that you want over the grid. You then look at each block in the grid, and select the character number that most nearly fits what you want. These numbers are this time gathered into a DATA line, and you then have to organise the loops that print them. In lines 1ϕϕϕ to 1ϕ3ϕ of Fig. 7.7, the inner loop that uses variable N prints a row of five characters, because the semicolon at the end of line 1ϕ2ϕ keeps the printing on the same line. When this loop ends in line

1∅3∅, the extra PRINT will cause a new line to be selected, so that the next value of J will cause characters to appear on the next line. We could, of course, use PRINTTAB or PRINT@ to place the shapes at other positions on the screen.

Tying it up

A better way of printing a complicated shape is usually to place the codes into a string. This will give you the same type of string as you could obtain by typing the characters directly. The advantage of using the character numbers is that it's easier to edit and list. Another advantage that we'll come to later is that we can use the same methods for programming our own characters. Yet another advantage is that placing graphics characters into a string makes it much easier to print them wherever we want on the screen.

```
10 CHAR4:COLOUR4
20 CLEAR500
30 G$=""
40 B$=STRING$(5,32)+CHR$(26)+STRING$(5,8
)+STRING$(5,32)+CHR$(26)+STRING$(5,8)+ST
RING$(5,32)+CHR$(26)+STRING$(5,8)+STRING
$(5,32)
50 FORJ=1TO38
60 READ A:G$=G$+CHR$(A)
70 NEXT
80 DATA128,32,202,32,144,26,8,8,8,8,8
90 DATA196,196,196,196,196,26,8,8,8,8,8
100 DATA196,196,196,196,196,26,8,8,8,8,8
110 DATA230,32,230,32,230
120 CLS
130 FORK=160TO190
140 PRINT@K,G$
150 PRINT@K+1,G$
160 PRINT@K,B$
170 NEXT
180 PRINT@195,B$
```

Fig. 7.8. Placing codes into a string which can then be printed at any position on the screen.

Figure 7.8 shows an example of this type of programming. Lines 1∅ and 2∅ prepare the way by selecting the character set and the colour, and clearing enough memory space to allow for long lengths of string. Line 3∅ assigns G$ to a blank (no space between the quotes), and then line 4∅ builds up B$, which is a set of blanks and cursor-moving character codes. The purpose of B$ is to act as a 'wipe-out string'. If we print B$ at the same position as our graphics string, it will erase the graphics string without erasing the whole screen.

Lines 5Ø to 7Ø then build up the graphics string itself. The data is contained in lines 8Ø to 11Ø and, so as to keep on familiar ground, I've used the same data as before. The part of the program that really packs the characters into the string is: G$ = G$ + CHR$(A) in line 6Ø. Each time this is executed in the loop, it places another character at the end of the string until the whole string is 38 characters long.

With all of this preparation out of the way, we can now start to make some use of this graphics string. Lines 13Ø to 18Ø show how easily the string can be printed at different places on the screen. This is made particularly easy by the use of PRINT@. The effect of animation that is achieved here is done by printing the graphics string in one position, then printing it again at the next position, and then wiping out the first pattern. The wiping action is achieved by making use of the blank string B$ which is printed at the original position of the graphics string G$.

The animation of low resolution graphics patterns like this is never completely satisfactory, however, as the motion looks so jerky. This is because there are only 4Ø character positions per line, and that makes the amount of movement from one position to the next rather large. In addition, BASIC is a comparatively slow language for writing animated programs. You can speed things up to some extent by using integer variables in loops (not delay loops, though!), so that if you make line 13Ø:

```
13Ø FOR K% = 16Ø TO 19Ø
```

there will be a noticeable increase in speed at which the patterns move across the screen. Putting all the instructions that lie within the loop into one line also helps, such as:

```
14Ø PRINT@K%,G$:PRINT@K%+1,G$:PRINT@K%,B$
```

If you overdo the speed, you may find that you have to add delay loops to slow things down again! Near the end of this chapter, we'll look at an interesting method for improving the appearance of animation.

Choose for yourself!

A very useful feature of practically all modern computers is the ability to create your own characters. You might, for example, create alien figures, print Greek letters, giant letters, or even simulate handwriting. All of these effects are possible by making use of the 'user-programmable' characters of the Colour Genie. Before we can attempt this, though, we need to know what is involved in placing a pattern on the screen.

To start with, the shape of a single character on the screen is created by a moving dot, the 'beam' of the cathode ray tube. Normally, the beam is turned off, but the signals that the computer sends to the TV receiver will turn the beam on as it moves across the screen and down, lighting up parts of the screen. Each character that we place on the screen, is the result of lighting up or leaving dark several dots. There are a possible 64 dots per character. With 40 characters per line and 24 lines on the screen, this allows us to work with the equivalent of 61440 resolution. These are arranged as an 8 × 8 grid pattern, as shown in Fig. 7.9. If we leave a row of dots unused all round the grid, we will ensure that the shape which is drawn by the other dots does not touch the next character in line with it the screen.

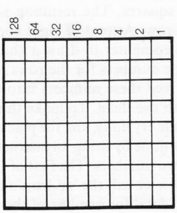

Fig. 7.9. The 8 × 8 grid for planning your own characters.

The first step to creating our own pattern, then, is to shade in dots on this grid. As usual, it's best to put a piece of tracing paper over the grid and to shade with soft pencil on the tracing paper. Remember that, when you are working with black and white, the parts that you shade will appear as white, and the unshaded parts will be black. If you are working with a blue pattern, then the parts that you are shading will eventually appear in blue and so on. Figure 7.10 shows a possible pattern – an alien face – which is obtained by shading some of the squares in the 8 × 8 block.

The next step is to convert your shaded blocks into a set of number codes. This has to be done because the computer works with number codes. The key to the conversion lies in the numbers that are printed at the top of each column in Fig. 7.9. Each number is the

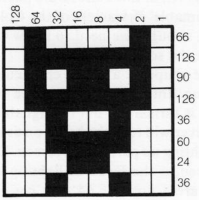

Fig. 7.10. An 'alien-face' pattern, and the codes which will produce it.

code for a shaded block, so that if you have one block shaded in a row, then that one code number is the code for the complete row. If you have two or more squares shaded in a row, then just add the codes for the shaded squares. The resulting number is the code number for the entire row. A total of eight code numbers will be enough to instruct the computer to draw a character.

Figure 7.10 shows the numbers for the rows of the example. The next thing to do is to feed these numbers into the memory of the computer, so that it can use them. To make this easier, computers always number their memory units, and the placing of a number into a position in memory is done by using an instruction POKE. POKE has to be followed by two numbers. The first of these is the memory reference number, which we call the 'address' number. The second number is the code or 'data' for one line of the character. For one character, then, we need eight POKE operations. This would be very tedious to program if we had to type out each one, but we can use a FOR...NEXT loop to read the numbers from a DATA line, and POKE each one into the correct address in the memory.

Where is the correct address in memory? As it happens, the memory of the Colour Genie appears to be rather curiously numbered (to our eyes), with some addresses being positive numbers up to 32767, and some negative numbers down to −32768. The portion of memory that we have to use bears address numbers of −3072 to −2049, no matter which size of Colour Genie we use. This is a total of 1024 memory units (1K in computer lingo), and it allows us space for up to 128 different characters of our own devising. It's a pretty large number of characters to be able to choose! As it happens, we can print all of these characters only when we have selected CHAR1. Selecting CHAR4, for example, doesn't allow us to print any of the programmable characters, though we can place the codes into memory ready to use after selecting CHAR1 (or other numbers).

The next step is to look at how this memory space us used. The number −3072 is a starting number, and that's where we have to place the first code number for the character whose ASCII code is 128. This character will use the memory locations from −3072 to −3065, a total of eight numbers (we count them inclusively). The next character, ASCII code 129, will need to have its first code number placed into the next memory address, −3064, and the next character, ASCII 130 starts at −3056 and so on. We're adding +8 to a negative number each time we do this.

```
5 CHAR1
10 ST=-3072
20 FOR AD=ST TO ST+7
30 READ D:POKE AD,D:NEXT
40 CLS:PRINT@170,CHR$(128)
100 DATA66,126,90,126,36,60,24,36
```

Fig. 7.11. The 'alien-face' program.

Enough of this theory. Let's take a look at a program which will place the codes for our 'alien face' into the memory and make this gruesome sight correspond to ASCII code 128. Figure 7.11 shows the steps that are needed. We select CHAR1 in line 5 so as to give plenty of space for programmable characters, just in case you want another 127 different faces. Line 1∅ then assigns ST (start of memory store) to −3072 so that we don't have to keep typing this number. The loop in lines 2∅ and 3∅ then reads data numbers for the character and packs them into the correct memory addresses. All that remains now is to print the character, which is done in line 4∅.

Once you have done it, it looks quite simple. The point is that once the codes have been put into the memory, they are held there until you switch off or until new codes are put there. Even if you delete the program that put the codes into memory (having run it), you don't wipe the memory, so that PRINT CHR$(128) will continue to give the alien face. If you change to CHAR4, of course, the character that appears for CHR$(128) is different, but it is stored at a different part of the memory, and the alien face will reappear when you select CHAR1 or CHAR2 and PRINT CHR$(128) again.

Now this new character can be printed in the same way as any other characters whose ASCII code you know, and we can also move the character around the screen as we would move any other

```
5 CHAR1
10 ST=-3072
20 FOR AD=ST TO ST+7
30 READ D:POKE AD,D:NEXT
40 CLS
50 FORN=940TO20STEP-40
60 PRINT@N,CHR$(128)
65 FORJ=1TO10:NEXT
70 IF N<>940THENPRINT@N+40," "
75 FORJ=1TO10:NEXT
80 NEXT
100 DATA66,126,90,126,36,60,24,36
```

Fig. 7.12. Moving a programmed character around the screen.

character. Figure 7.12 shows an illustration of this in action. The character is formed in the same way as before, and then lines 5∅ to 8∅ move the character round the screen.

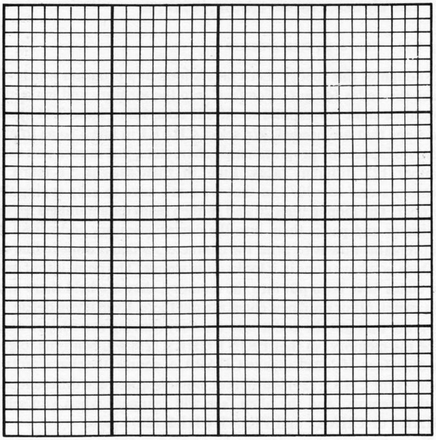

Fig. 7.13. A planning grid for combining several user-defined characters into one pattern.

Corpulent characters

A single 'user-defined character' looks fairly small on the screen, and you sometimes want to use much larger characters. This can be done by defining several characters and joining them up into a string, just as we did with the preset characters. As usual, the best way is to illustrate this. We start with a planning grid, such as the one in Fig. 7.13. This allows you to plan out objects whose size can take up to 16 characters. If you need larger objects, then it's easy to draw out your own planning grids, using graph paper. The old type of graph paper, with inch and $\frac{1}{8}''$ lines is perfect for this type of work, but not so easy to find nowadays.

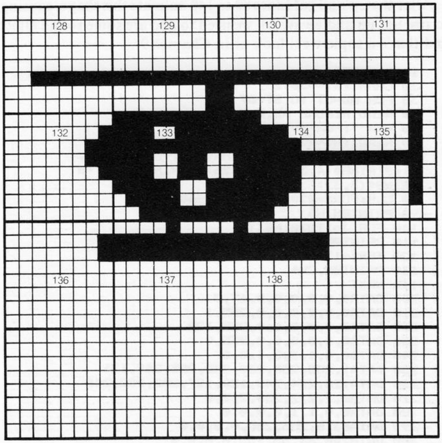

Fig. 7.14. A helicopter shape made up from eleven programmed characters.

The shape that we're going to produce is shown in Fig. 7.14 – a helicopter. It makes use of eleven different characters, from 128 to 138, so we shall have to work out the code numbers that we need for each of these characters. This will be a total of 88 numbers for the 11 characters. There's no simple lazy way to graphics! Finally, we have to combine these characters into a string that we can print.

Figure 7.15 shows the results of all this planning in the form of a program. Lines 3Ø to 4Ø carry out the character definition, using the data lines 2ØØ to 3ØØ. I have used one data line for each character, so as to make it easier to trace which one produces which character in case any alterations are needed. Once the characters have been defined, they are read into a string using lines 5Ø and 6Ø. This makes use of data lines 4ØØ to 42Ø so as to pack the character's ASCII codes, plus the cursor down-and-left codes into the string. The last piece of preparation is in line 7Ø, which defines a blank string which is exactly the right shape and size to wipe out the image of the helicopter.

After that, it's all go! Lines 8Ø to 14Ø animate this shape, causing it to rise up, up and away. All we need to add are some sound effects, and that's a topic we'll look at in Chapter 9.

```
10 CLEAR500
20 CHAR1:ST=-3072
30 FOR AD=ST TO ST+87
40 READ D:POKE AD,D:NEXT
50 H$="":FOR N=1TO21
60 READX:H$=H$+CHR$(X):NEXT
70 CH$=STRING$(4,32)+CHR$(26)+STRING$(4,
8)+STRING$(4,32)+CHR$(26)+STRING$(4,8)+S
TRING$(3,32)
80 CLS:FORN=820TO40 STEP -40
90 PRINT@N,H$
100 FOR J=1TO20:NEXT
110 PRINT@N-40,H$
120 FORJ=1TO20:NEXT
130 PRINT@N,CH$
140 NEXT
200 DATA0,0,0,0,0,0,63,0
210 DATA0,0,0,0,0,0,255,1
220 DATA0,0,0,0,0,0,255,128
230 DATA0,0,0,0,0,0,255,0
240 DATA0,0,1,3,3,1,0,0
250 DATA1,255,255,230,230,255,249,121
260 DATA128,224,248,120,127,252,252,248
270 DATA0,2,2,2,254,2,2,2
280 DATA0,0,1,1,0,0,0,0
290 DATA63,8,255,255,0,0,0,0
300 DATA240,64,55,255,0,0,0,0
400 DATA128,129,130,131,26,8,8,8,8
410 DATA132,133,134,135,26,8,8,8,8
420 DATA136,137,138
```

Fig. 7.15. The helicopter program.

One final point about animation. When we are dealing with the low resolution graphics displays, animation is never very convincing because of the comparatively large steps of movement from one line to another one, or from one character position in a line to the next. The use of user-defined characters can be harnessed to make animation look better! Take a look at Fig. 7.16, which shows three

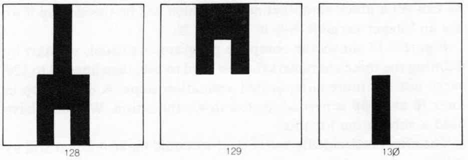

Fig. 7.16. Using 'half-shapes' to improve animation.

user-defined shapes. One of the shapes fills a block, the other two fill only half a block each, and the small units are designed so that when put together they are the same shape as the complete block, but displaced by half a block vertically. Let's imagine, stretching the imagination a bit, that the complete shape is a rocket. If we want to animate this, rising vertically (it's not the European rocket!), then the conventional method would be to print the rocket shape on the bottom line of the screen, wipe it, then print on the next line, wipe,

```
10 CLEAR200
20 CHAR1:ST=-3072
30 FOR AD=ST TO ST+23
40 READD:POKEAD,D:NEXT
50 CLS:FORN=900TO40STEP-40
60 PRINT@N,CHR$(128)
70 FORJ=1TO20:NEXT
80 PRINT@N-40,CHR$(129)
90 PRINT@N,CHR$(130)
100 FORJ=1TO20:NEXT
110 PRINT@N," "
120 NEXT
200 DATA16,16,16,16,56,56,40,40
210 DATA0,0,0,0,16,16,16,16
220 DATA56,56,40,40,0,0,0,0
```

Fig. 7.17. A 'rocket' program which uses the 'half-characters' for better animation.

and so on. By using our 'half rockets', we can make the rocket appear to move by half a line each time. We print the whole rocket shape on the bottom line. We then print the top half of the rocket on the next line, remembering that this takes up the *bottom* half of the character only. We wipe the lower line, and then print the lower half of the rocket on this line. This now looks as if the rocket had moved by half a line. We then wipe both of these characters, and print the complete rocket shape on the second line. By repeating this series of actions, we can get a much smoother motion which will be speeded up if we use an integer variable N% in place of N.

Figure 7.17 shows the complete program. As usual, we start by defining the three characters that we need to use, then lines 5Ø to 12Ø carry out the more complicated animation steps. A delay loop in lines 7Ø and 1ØØ is needed to slow down the action. We could have used a subroutine for this.

The effect is decidedly better, and we could make it better still by splitting the rocket into three parts or even four. The method becomes more awkward to use when the shapes are larger, however, because there is so much work involved in planning and shifting the sections. It can look very rewarding, however.

Chapter Eight

Smaller Pieces Make Prettier Pictures

High resolution graphics is about creating pictures and other shapes with smaller units. We refer to these units as *pixels* (picture elements). The low resolution graphics of the Colour Genie allows us the use of 40 × 24 pixels per screen, a total of 960 pixels, or 61440 pixels using programmable graphics characters which give an effective resolution of 320 × 192. The high resolution graphics allows us to use 160 × 96 pixels, a total of 15360. The difference is very noticeable.

The Colour Genie keeps the high resolution graphics separated from the low resolution. You can't, in other words, mix the two. You can't print letters in the ordinary way along with the high resolution graphics shapes, and you can't draw in high resolution graphics along with text. Changing from one to the other will always clear the screen on the way.

The reason for this is the way that the computer uses its memory. A display of high resolution graphics requires a lot more of the memory than a screen of low resolution or text. Different addresses in the memory are used for storing the codes that produce the two different types of displays, and we have to switch from one part of memory to the other rather than taking pieces from each. The switching can be done by using keys or by instructions that are placed in a program.

The use of the MODSEL and CTRL keys at the same time will switch the Genie to the high resolution graphics display. The instruction which does this in a BASIC program is FGR (full-graphics resolution). Using these keys causes the full-graphics display to remain on until you press the BREAK key (or switch off). The FGR instruction, however, will be reversed whenever the program runs out of lines and so ends. If you want to return to low resolution graphics earlier (to display a score, for example), you can use the LGR instruction. To keep a display on screen after a program has ended, you will have to make the last line of the program into an endless loop – so that it doesn't end as such!

The use of the full-graphics resolution involves quite a number of new instructions, so this Chapter will be a voyage of discovery (eat your heart out, Sindbad!). It's always difficult to predict the appearance of what you'll see on the screen from looking at the program, so considerable practice is needed, and there is no substitute for trying out all of the programs here and in the Manual. Watch out for mistakes in the Manual, however. All the programs in this chapter have been printed directly by the Colour Genie.

```
10 FCLS:FGR
20 FORN=1TO20
30 BGRD
35 FORJ=1TO100:NEXT
40 NBGRD
45 FORJ=1TO100:NEXT
50 NEXT
```

Fig. 8.1. Introducing some 'full graphics' instructions.

Let's start simply, with Fig. 8.1. Line 1∅ uses FCLS, which is the full-graphics equivalent of CLS, the clear-screen instruction. CLS clears the low resolution screen, FCLS clear the full-graphics screen, and two different instructions are needed because two different parts of the memory, have to be cleared. Most of the full-graphics instructions, incidentally, work even when the full-graphics screen is not being displayed. It's possible to have a shape drawn and ready to display on the full-graphics screen while you are reading text on the low resolution screen. By switching over, you can then make it look as if the shape on the full-graphics screen has appeared instantly.

Figure 8.1 uses FCLS to clear the full-graphics screen, then FGR to switch to this graphics screen. We then introduce the BGRD instruction. This doesn't mean that the computer feels tired, simply that it can change the 'standard' background colour. the use of BGRD causes the whole screen to turn pink. The delay in line 35 holds this for a short time, and then NBGRD causes the screen to go blank (or black) again. This can be alternately fast or slow without affecting anything that has been placed on the screen by any of the graphics instructions. BGRD and NBGRD are instructions that offer one form of control over the screen background colour, using BASIC.

The next step is to look at some foreground colours. There isn't such a wide range of colours available on the full-graphics screen as there is on the low resolution screen. There are, in fact, only four colours, which are listed along with their number codes in Fig. 8.2. If you use numbers that exceed four, you will get the FC error message and the program will halt.

Code number	Colour
1	Black
2	Blue
3	Red
4	Green

Fig. 8.2. The colour set for the full-graphics screen.

These numbers can be demonstrated along with the instruction FILL. FILL is another colour background instruction which will make the background (whole screen) colour equal to the colour whose code number follows the FILL instruction. Figure 8.3 shows

```
10 CLS:FCLS:FGR
20 FORJ=1TO4
30 FILL J
40 K$=INKEY$:IF K$=""THEN40
50 NEXT
60 PRINT"ALL DONE"
```

Fig. 8.3. Changing background colour with the FILL instruction.

this in action. The screen is filled with each colour, and an INKEY$ loop holds the screen in background colour intil a key is pressed. At the end of the loop, the automatic return to the ordinary text screen allows the message 'ALL DONE' to appear.

The Genie plot

Now that we've achieved some control over background colours, we can start to look at ways of drawing something on these backgrounds. Before we can set out, however, we need a map of the territory. This is shown in Fig. 8.4 – it's a 160 × 96 grid. The left to right direction is referred to as the X axis, and positions in this direction are indicated by reference numbers ∅ to 159. Position ∅ is at the left-hand side of the screen, and position 159 is at the right-hand side. The up-down direction is called the Y axis, and positions in this direction are indicated by numbers in the range ∅ to 95. Position ∅ is at the top of the screen, and position 95 is at the bottom. We can refer to any position on the screen, therefore by using two of these position numbers. The X number will give the position across the screen, measured from the left-hand side. The Y number will give

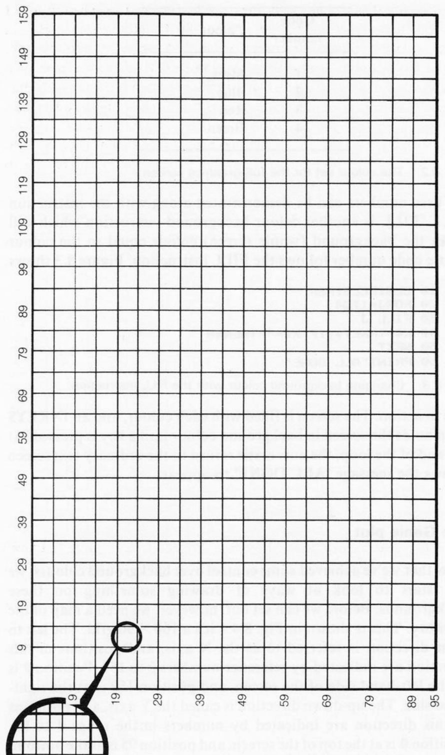

Fig. 8.4. The 160 × 96 planning grid for full-graphics patterns.

the distance down the screen, measured from the top. The middle of the screen corresponds to the position 8∅,48. Note that we always write a pair of position numbers in the X,Y order; X first, then Y.

The Colour Genie uses an instruction PLOT as a way of placing a 'graphics cursor', a small dot, at any place on the screen. PLOT has to be followed by at least one pair of numbers, which are the X,Y position numbers for the point that is being plotted. PLOT 8∅,48 will, for example, place a dot at the centre of the screen. If we want to draw a line from this point to another point at the right-hand top corner of the screen (where X=159, Y=∅), then we use:

PLOT 8∅,48 TO 159,∅

The word TO, when it is used following PLOT, carries out the action of drawing a straight line between the points. Note that we need a comma between the X and the Y numbers.

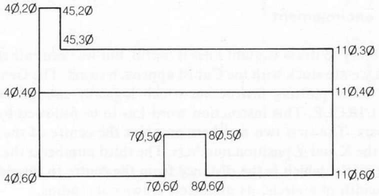

Fig. 8.5. Using the PLOT instruction.

PLOT doesn't stop there, however. We can specify more than one line in a single PLOT instruction. Figure 8.5 shows a simple shape which has been traced over the 160 × 96 planning grid. The corners are marked with the X and Y position numbers – the proper name for these is *co-ordinate numbers.* If we pick a place to start, we can follow a PLOT instruction with a whole set of position numbers,

```
10 CLS:FCLS:FGR
20 FILL4:FCOLOUR(1)
30 PLOT40,20TO40,60TO70,60TO70,50TO80,50
TO80,60TO110,60TO110,30TO45,30TO45,20TO4
0,20
40 PLOT40,40TO110,40
50 GOTO50
```

Fig. 8.6. A pattern-drawing program which uses PLOT.

using TO to indicate where a line is to be drawn. Figure 8.6 shows the resulting program. FILL4 in line 2ϕ causes the background to be in colour 4, which is green, and the new instruction FCOLOUR(1) sets the foreground colour. The foreground colour is the colour of the lines that will be drawn, and the same number codes as we used along with FILL are also used with FCOLOUR. The difference here is that the number has to be enclosed in brackets – according to the manual. It seems to work just as well if the brackets are omitted, however! FCOLOUR(1) will cause the lines to be drawn in black.

The main PLOT occurs in line 3ϕ, and it traces its way around the house shape. The gutter-line of the roof is then drawn separately in line 4ϕ, and line 5ϕ keeps the program in an endless loop so that you have time to see what has been drawn. To restore control to the keyboard, you will need to press the BREAK key. An alternative is to have an INKEY$ loop in line 5ϕ in place of the GOTO5ϕ.

Total encirclement

The ability to draw straight lines is useful, but we need rather more unless we are stuck with the Cubist approach to art. The Genie also has a circle plotting instruction which logically enough uses the word CIRCLE. This instruction word has to be followed by three numbers. The first two numbers pinpoint the centre of the circle, using the X and Y position numbers. The third number is the radius of the circle, which is the distance from the centre to the rim. The total width of a circle, its diameter, is twice its radius.

```
10 CLS:FCLS:FGR
20 FILL2:FCOLOUR(4)
30 FOR N=1TO80 STEP3
40 CIRCLE80,48,N
50 FORJ=1TO100:NEXT
60 NEXT
70 GOTO70
```

Fig. 8.7. A circle-drawing program. Variations on this can produce interesting effects.

Figure 8.7 demonstrates this with a set of circles that are drawn using green lines on a blue background. The loop that starts in line 3ϕ will cause a set of circles of different radius values to be drawn by the instruction in line 4ϕ. The STEP size has been chosen so as to allow the circles to be reasonably spaced, but rather striking effects can be obtained if smaller step sizes are used. Another interesting pattern is obtained if each circle is drawn, with the same radius, but the X value of the centre is altered on each pass through the loop.

On two TV receivers that I tried, the 'circles' were not truly circular but elliptical, with the height greater than the width. If your TV receiver has a picture height control, it is fairly easy to alter this so that the circles look truly circular. The circles in this program continue to be drawn even when the radius value is too large to allow all of the circle to be placed on the screen! This is a refreshing change from the CIRCLE instructions of some other computers which will hang up with an error message if any attempt is made to draw beyond the screen limits. Now you can put wheels on to your car shapes!

The puzzling PAINT

Working with outlines is all very well, but at times you want to be able to fill a shape with colour. Genie has an instruction for this as well, in the form of the PAINT instruction. Unlike most of the instructions that we have used so far, though, PAINT needs a lot of patience and some experience before you can use it to its best advantage. Filling a shape with colour isn't an easy problem for a computer. Where do you start, where do you stop, what happens at a boundary? These are the problems that face the computer designer, and any solution just has to be some sort of compromise.

The PAINT instruction of the Colour Genie has to be followed by four numbers, separated by commas. Like all the numbers that are used in instructions, these can be number variables or expressions. The first two numbers are the now-familiar X and Y co-ordinates. These specify where the painting has to start. The best starting position is near the bottom edge of the shape that you want to fill with colour, but not actually at the edge. The third number in the PAINT instruction is the code for the colour that you want to paint with, using the same set of numbers (1 to 4) as we have used with FILL and FCOLOUR. The fourth number in the PAINT instruction is the colour of the boundary line where you want painting to stop. You can, if you like, make the boundary the same colour as the colour you use to paint. What you can't do with any hope of success is to make this boundary colour the same as the background colour!

The PAINT action operates comparatively slowly, and in some rather curious ways. It will do particularly curious things, for example, if the shape that you are painting is not completely closed, and it will also do odd things if the direction of painting has to be reversed. Take a look at the effect of the program in Fig. 8.8. This draws a set of circles and paints the spaces between them – or so it seems. See what it actually does! Now try Fig. 8.9, which uses two PAINT instructions for each doughnut shape. This fills in much more of each shape, but it still leaves a gap at the 5 past 1 o'clock position. This final gap could probably be filled in by another PAINT instruction which started at a point in the gap – try it!

```
10 CLS:FCLS:FGR
20 FILL2:FCOLOUR(4)
30 FOR N=1TO80 STEP6
40 CIRCLE80,48,N
50 FORJ=1TO100:NEXT
60 CIRCLE80,48,N+3
70 PAINT80,48+N+2,3,4
80 NEXT
90 GOTO90
```

Fig. 8.8. Using PAINT for spaces between circles – the effects are odd, because of the limitations of the PAINT instruction.

```
10 CLS:FCLS:FGR
20 FILL2:FCOLOUR(4)
30 FOR N=0TO78 STEP6
40 CIRCLE80,48,N
50 FORJ=1TO100:NEXT
60 CIRCLE80,48,N+3
70 PAINT80+N+2,48,3,4
80 PAINT80-N-2,48,3,4
90 NEXT
100 GOTO100
```

Fig. 8.9. Using two PAINT instructions to improve the filling action.

Let's move away from circles, and try to create a chequer-board pattern. This might seem straightforward, but in fact it involves painting and repainting in rather an involved way. There's no simple rule that will help you to decide what to do for your own paintings, but the examples in this chapter should be a pretty good guide.

Figure 8.10 shows the chequer-board program. The most obvious way of creating a chequer-board would be to draw the pattern, and then paint alternate squares. It's not as simple as that, because the PAINT instruction does some perverse things. Its most awkward trick is to paint one line at the bottom of a square that you don't want to be painted, just after it has reached the top of a square at a different part of the design. It then goes on to fill in the rest of the square on which it has put one line! The program in Fig. 8.10 therefore uses an odd sequence of program instructions to get around this and other problems. We start the shape in line 4ø by

```
10 CLS:FCLS:FGR
20 FILL4
30 FCOLOUR2
40 PLOT0,0TO159,0TO159,95TO0,95TO0,0
50 PLOT0,32TO159,32
60 PLOT40,0TO40,95
70 PLOT80,0TO80,95
80 PLOT120,0TO120,95
90 PAINT100,20,3,2
100 PLOT0,64TO159,64
110 PAINT50,50,3,2
120 PAINT140,50,3,2
130 PAINT20,90,3,2
140 PAINT100,90,3,2
150 PAINT20,50,4,2
160 PAINT20,20,3,2
170 GOTO170
```

Fig. 8.10. The chequer-board program. This demonstrates the curious way in which PAINT can operate!

plotting four lines right round the chequer-board. This ensures that the board is surrounded by colour 2, with no gaps. The PAINT instruction does odd things at gaps, even more odd than it does where there are no gaps. The next lines 5ϕ to 8ϕ then start the plotting of the pattern, and line 9ϕ fills in the first of the red squares.

This is straightforward enough, but the fun starts in line 1ϕϕ. We now draw in the last line of the chequer-board, and lines 11ϕ 11ϕ to 16ϕ complete the painting. As we paint, however, we find that some of the red 'spills' on to the green squares! This makes it necessary to repaint one green square in line 15ϕ.

Plot and counterplot

Patterns that have been filled by the PAINT instruction take quite a long time to draw, and if you want to present them quickly, it's a good idea to have the drawing and painting instructions completed before the FGR instruction is carried out. You can, for example, print a set of instructions or other text on the screen, and follow this with the PLOT and PAINT instructions. The PLOT and PAINT will work, affecting the full-graphics memory, but out of sight and with no effect on the text screen. A 'press-any-key' step then allows you to move on, and the next instruction should be FGR. This instantly produces the picture. On the other hand, you may prefer the fascination of watching the pattern being drawn!

If we want to move patterns around on the full graphics screen, then the NPLOT instruction is one that we can use to good effect.

NPLOT, as its name suggests, means 'unplot', or wipe out. Any pattern that we have created using PLOT can be wiped clear by using NPLOT – but only if the background is black, and NPLOT is followed by the same set of numbers and TO instructions as the PLOT. Plotting and unplotting a complicated pattern is rather hard work, unfortunately.

```
10 CLS:FCLS:FGR
20 FCOLOUR(2)
30 GOSUB80
40 FORN=1TO1000:NEXT
50 FILL3
60 GOSUB80
70 GOTO70
80 PLOT20,20TO50,50
90 FORN=1TO1000:NEXT
100 NPLOT20,20TO50,50
110 RETURN
```

Fig. 8.11. Using PLOT and NPLOT on a diagonal line.

As usual, it's best to start in a simple way. Figure 8.11 shows a PLOT–NPLOT routine that operates on a diagonal line. The GOSUB8Ø in line 3Ø will plot the line, wait, and then unplot it, using COLOUR2, which is blue. At the end of the subroutine there is another delay, in line 4Ø, and the screen is filled with colour red. The PLOT and NPLOT is then repeated with a red background. This time, the NPLOT has quite a different effect, because you can see that NPLOT produces the shape in black lines, and black lines are invisible on a black background but visible on a red background!

An alternative, then, is to put the PLOT instructions into a subroutine, and to use FCOLOUR to specify the line colour. If this is the foreground colour, the line will appear; if this is the background colour, the line will disappear. Since all the PLOT instructions can be put into a subroutine, you don't have to type them all over again.

```
10 CLS:FCLS:FGR
20 Y=40:FCOLOUR(2)
30 FORX=10TO150
40 PLOTX,YTOX+10,YTOX+10,Y+10TOX,Y+10TOX
,Y
60 NPLOTX,YTOX+10,YTOX+10,Y+10TOX,Y+10TO
X,Y
70 NEXT
```

Fig. 8.12. PLOT and NPLOT used to animate a square.

NPLOT is useful, however, when you are working with black backgrounds, and Fig. 8.12 illustrates it in use to shift a square across the screen. X and Y are position co-ordinate values, of which X is varied in the loop that starts in line 3∅. This allows us to plot, then unplot, the square in lines 4∅ and 6∅, and move it by changing the value of X. The amount of movement can be changed by choosing a different step size in the loop.

```
10 CLS:FCLS:FGR
20 Y=40:FCOLOUR(2)
30 CIRCLE98,48,47
40 X=98:K=1
50 X=X+K
60 PLOTX,YTOX+10,YTOX+10,Y+10TOX,Y+10TOX
,Y
70 NPLOTX,YTOX+10,YTOX+10,Y+10TOX,Y+10TO
X,Y
80 IF CPOINT(X-1,Y)=1THENK=-1*K
90 IF CPOINT(X+11,Y)=1THEN K=-1*K
100 GOTO50
```

Fig. 8.13. Using CPOINT to detect a boundary in a different colour.

We can take this a stage further, as the program of Fig. 8.13 shows. The new instruction in this example is CPOINT, which 'reports' on the colour of a point. CPOINT has to be followed by the X and Y co-ordinate numbers, placed within brackets, and the result of CPOINT is a number. This number is, for some odd reason, one *less* than the colour numbers that we use for FILL and FCOLOUR, so that the number ∅ means black, and number 3 means green. We use CPOINT to test what colour we have at any position specified by values of X and Y, so that we can decide, for example, when we have reached a boundary, even before we touch it! Figure 8.13 illustrates this by drawing a circle in line 3∅. We then pick the point X=98, Y=4∅ which is at the centre of the circle, and a step size of K which is initially set to 1. Line 5∅ makes X become X+K, so increasing X by 1. Line 6∅ then plots a square, using the same instructions as we tried in Fig. 8.12 (too much novelty can damage your health!), and unplots again in line 7∅. Lines 8∅ and 9∅ then test for what is just on the right and on the left of the square. If the left-hand corner of the square is close to the left-hand edge of the circle, then CPOINT(X−1,Y) will give the circle boundary colour of 1 (one less than the 2 we used to draw it). If, on the other extreme, the right-hand corner of the square is near the right-hand edge of the circle, then CPOINT(X+11,Y) will give the value of 1. In either case, K is converted to K times −1. The effect of this is that if K was 1, it becomes −1, and if K was −1, it becomes +1 (because −1 times −1 is +1). The effect is to reverse the direction of the motion, so that the square appears to bounce to and fro across the circle.

Shapes and scales

The Colour Genie has a very unusual form of drawing instruction which allows us to make shapes like user-defined characters and to alter the size of these shapes. This uses a rather different approach to drawing, however, in which a set of 'relative movement' instructions are used. Relative movement means movement measured from one point, so that the traditional 'Treasure Island' directions of 'three paces forward and two to the right' are relative movements. Where you get to as a result of relative movements depends entirely on where you started from. Absolute movement, in contrast, is always measured from the same spot. The X,Y position numbers that we use, for example, are absolute because their starting point, X=∅, Y=∅ is a fixed place – the top left-hand corner of the screen.

The SHAPE instruction of the Colour Genie specifies a starting point in the usual form of a couple of co-ordinates, separated by a comma. Thus SHAPE 8∅,48 will specify a starting point at the centre of the screen. Unusually, though, the left, right, up, down instructions that are needed to specify the movement from this point are stored in the memory as code numbers, and you have to plot these numbers in place. A set of POKE instructions will be needed to do this before the SHAPE instruction can be used.

This type of POKE takes a form that we haven't seen before. The starting address has to be 32512 for the 16K Colour Genie, and −1664∅ for the Genie that has been expanded to 32K. The number that is placed at this starting address must be the number of movement code numbers that will follow. This ensures that the computer will read the correct number of codes.

Figure 8.14 shows the numbers that have to be used for creating shapes – the table provided in the Manual has several errors. For each of the four colours that we can use, there is one code number for each of the four possible directions. We can move invisibly from one spot to another by moving in the background colour, or we can draw a line by moveing in any other colour. It's possible, of course, to make movements in three different foreground colours. We can make the SHAPE pattern of any size that will fit on the screen, and expand it or shrink it by using SCALE!

Yes, an example would help. Figure 8.15 shows a simple pattern, a letter I. The starting point is taken as being the top left-hand corner of the letter, and the shape is planned on paper by writing down the directions of movement – RRLDDLRR – which make up the shape. I've used R,L,D,U to mean right, left, down and up, as you

Colour	Direction	Code number
Black	R	0
,,	D	17
,,	L	32
,,	U	51
Blue	R	68
,,	D	85
,,	L	1∅2
,,	U	119
Red	R	136
,,	D	153
,,	L	17∅
,,	U	187
Green	R	2∅4
,,	D	221
,,	L	238
,,	U	255

Fig. 8.14. The numbers to use along with SHAPE.

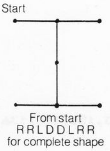

Fig. 8.15. A simple 'I' pattern, and the directions that are used.

probably guessed. Using the code numbers in Fig. 8.14, the data line 2∅∅ in Fig. 8.16 can then be written – and then the rest of the program.

Lines 1∅ to 3∅ clear the screen, switch to full graphics, and then place the numbers into the memory. They will stay there until you replace them or switch off, and they can be placed there at any time before you want to use SHAPE. Lines 4∅ onward then show off the usefulness of the SHAPE and SCALE instructions. Scale numbers from 1 to 1∅ are used and, for each scale size, the shape is drawn starting at 6∅,2∅. This means that the top left-hand corner of the letter I will be placed at this point. The final SHAPE in line 1∅5

```
10 CLS:FCLS:FGR
15 AD=32512
20 FORN=ADTOAD+8
30 READD:POKEN,D:NEXT
40 FORS=1TO10
50 SCALES
60 SHAPE60,20
70 FORJ=1TO1000:NEXT
80 NSHAPE60,20
90 FORJ=1TO200:NEXT
100 NEXT
105 SHAPE50,30
110 GOTO110
200 DATA8,204,204,238,221,221,238,204,20
4
```

Fig. 8.16. The program to create and print the shape in different sizes.

moves the I to 5∅,3∅, just to show that we're not stuck with 6∅,2∅! Line 8∅ uses NSHAPE, which blanks out the picture that was created by SHAPE on a black background. The effect of NSHAPE is very similar to the effect of NPLOT.

```
10 CLS:FCLS:FGR
20 N=32512
30 FORAD=NTON+11
40 READD:POKEAD,D:NEXT
50 FOR S=1TO5
60 SCALE S
70 SHAPE60,20
80 FORI=1TO1000:NEXT
90 XSHAPE60,20
95 FORJ=1TO1000:NEXT
100 NEXT
190 GOTO190
200 DATA11,170,170,153,153,153,136,136,1
87,170,136,136
```

Fig. 8.17. Building up a multiple image with SHAPE and XSHAPE.

Effect of SHAPE	Effect of XSHAPE
PINK or BLANK	GREEN
BLUE	RED
RED	BLUE
GREEN	PINK/BLANK

Fig. 8.18. The XSHAPE colours.

The SHAPE instruction is an interesting one, and one which needs quite a lot of experimenting with. You can, for example, create letters and figures as well as more exotic shapes, and the SCALE instruction allows you to expand or contract a pattern. Just as a final fling to this chapter, take a look at Fig. 8.17, which builds up a multiple image by using SHAPE along with XSHAPE. XSHAPE changes the colour of the picture to its opposite, as illustrated by the table in Fig. 8.18. The overall effect here is an interesting one, leaving you with lots to think about and to work into your own programs.

Chapter Nine
Sound Decisions

Most modern computers have the ability to generate signals that a loudspeaker can convert into sound. Few, however, have the range of capabilities for generating sound that you will find on the Colour Genie. The sound comes from the loudspeaker of the TV receiver! In addition, a socket on the back of the Genie allows you to connect the sound signals to a hi-fi system so that you can hear the sound at greater volume and with better bass (low notes). You can also record the sound signals that are taken out to a hi-fi system.

Before we start work on sounding out the Genie's capabilities, however, it's a good idea to make sure that the TV is correctly tuned. Turn up the volume control of the TV receiver that you are using with your Colour Genie. If you don't do this, you won't hear the Genie's sound! If what you hear when you turn up the volume control is a loud rasping buzz, then alter the tuning slightly until you find the quietest spot. You must, of course, still be able to see a picture on the screen. Very often, only a very minor adjustment is needed, perhaps none at all.

The next items concern sound itself. You can produce some interesting sound effects from the Colour Genie simply by experimenting with the programs that are contained in this Chapter. You will be very much more successful in this work, however, if you know something about sound and what it is.

Sound is the effect of a pressure wave in air (or in other materials). A pressure wave means that the pressure of the air on your eardrums rises and falls several times each second. The number of these rises and falls of pressure per second is called the *frequency* of the sound, and is measured in units that are called *hertz*, shortened to Hz. One Hz would mean one rise, fall and return to normal pressure in each second (Fig. 9.1).

You can't hear all of the possible frequencies of sound. The range of sound frequencies that we can hear, called the *audible range*,

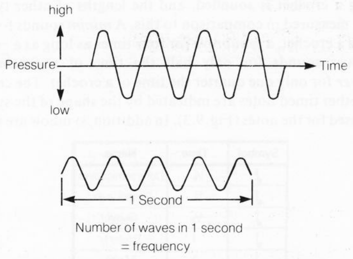

Fig. 9.1. Sound waveforms, showing how the air pressure changes with time. The number of rises and falls per second is the frequency.

extends from about 30 Hz to around 15000 Hz (called 15 kilohertz). Most of the notes that are produced by musical instruments are within the range of 100 Hz to 5 kHz (5000 Hz). We hear these differences of frequency as sounds of different pitch. A low frequency gives a note of low pitch, a *bass* note. A high frequency gives a note of high pitch, a *treble* note.

The loudness of a note is decided by how much the pressure of the air changes from normal. The quantity that measures this change is called the *amplitude*, illustrated by the graph in Fig. 9.2. The connection between amplitude and loudness is not straightforward, but the greater the amplitude is, the louder we hear a note.

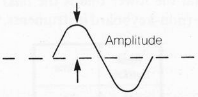

Fig. 9.2. The amplitude of a wave. This determines how loud the sound is.

For the purposes of music, we have to specify for each note how loud it will be, how long it is to be played, and what pitch it is. In written music, loudness is indicated by letters such as *f* (loud) and *p* (soft), and using more than one letter if necessary. For example, *fff* means very loud, and *ppp* means very soft. The duration of a note is indicated in two ways. One of these ways is a metronome reading. This indicates how many unit notes are sounded per minute. The unit note is called a *crochet*. The metronome reading decides for how long a crochet is sounded, and the lengths of other types of notes are measured in comparison to this. A *minim* sounds for twice as long as a crochet, a *semibreve* for four times as long as a crochet. The *quaver* sounds for only half the time of a crochet, the *semiquaver* for only one quarter the time of a crochet. The crochets and the other timed notes are indicated by the shape of the symbols that are used for the notes (Fig. 9.3). In addition, symbols are used to

Symbol	Time	Name
𝅘𝅥𝅰	⅛	Demisemiquaver
𝅘𝅥𝅯	¼	Semiquaver
𝅘𝅥𝅮	½	Quaver
♩	1	Crotchet
𝅗𝅥	2	Minim
𝅝	4	Semibreve

Adding a dot after a note lengthens it by 50%

Fig. 9.3. How the time of a note is indicated by the shape of notes in written music.

represent silences in the music (Fig. 9.4). Some music scores do not show a metronome reading, but rely on the use of Italian words to indicate the time of a crochet less precisely.

The pitch of a note is indicated in written music by placing it on a type of musical 'map' that is called the *stave*. Piano music shows two of these staves, each consisting of five lines and four spaces. The set of lines, or stave, which is printed on top is the *treble stave*, used for the higher notes, and the lower one is the *bass stave*. Instruments other than the piano (non-keyboard instruments, that is) will use only

Rest Symbol	Time
𝄿	¼
𝄾	½
𝄽	1
𝄼	2
𝄻	4

Fig. 9.4. Symbols that indicate the relative times of silences (rests).

one stave. The note that appears on a line of its own in the piano pair of staves is called *Middle C*. On a piano, this note is played by a key which is approximately in the centre of the keyboard. Figure 9.5 shows the staves, with the notes marked.

The notes that are shown in Fig. 9.5 are arranged in groups of eight (counting inclusively), called an *octave*. Music from the

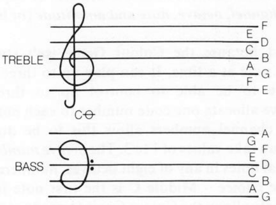

Fig. 9.5. The staves, with Middle C marked.

Western hemisphere traditionally uses a total of twelve different notes in an octave, however, and the full range of one octave is shown in Fig. 9.6, along with the corresponding positions on the piano keyboard. The half-pitch notes, or *semitones* are marked on

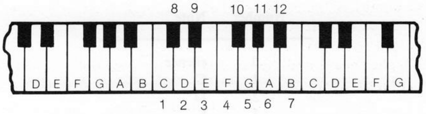

Fig. 9.6. One octave of piano keys, showing Colour Genie code numbers.

the piano by black keys, though one semitone (between D and E) is not marked in this way. On written music semitones are indicated by using the signs # (*sharp*) or ♭ (*flat*). A sharp indicates that the pitch has to be raised one semitone above the marked pitch. A flat indicates that the pitch is to be lowered one semitone below the marked pitch. When we use piano music, the semitone above one note is the same as the semitone below the next higher note, so that C# is the same as D♭.

Some general sounds

To work, then. The first Colour Genie instruction that we have to look at is one which we use for musical notes. The instruction word that we need is PLAY, and it has to be followed by four numbers that are separated by commas and all enclosed in brackets. These four numbers, which can also be number variables or expressions, are referred to as *channel*, *octave*, *note* and *amplitude* (or loudness) *numbers*.

Channel is used because the Colour Genie isn't stuck with providing just one note at a time. It can play up to three notes at once, so we have to be able to control up to three notes independently. If we allocate one code number to each note, this is possible, and the channel numbers allow this to be done. The channel numbers can take values of 1 to 3. The *octave number* alows you to select musical notes in any of eight octaves, numbered 1 to 8. This is a very wide choice – Middle C is the first note in octave number 4. This variety allows the Colour Genie to produce any pitch of sound that a musical instrument can produce, and a few more besides!

The *note number* controls which of the notes in an octave will be sounded and the relationship between the note and the number is shown in Fig. 9.7. A ∅ used in this position will give silence, a

Number	Note
0	silence
1	C
2	D
3	E
4	F
5	G
6	A
7	B
8	C#
9	D#
10	F#
11	G#
12	A#

Fig. 9.7. The Colour Genie codes for the notes in an octave.

musical 'rest'. The semitones C#, D#, F#, G# and A# are allocated numbers 8 to 12 inclusive, rather than being placed between the notes where they occur (see also, Fig. 9.6). You have to remember, for example, that C is 1, but C# is 8! You will have to consult the table quite a lot to remind yourself of these numbers when you start to write music programs, but after some experience you can write Colour Genie music without the table.

The last figure, the *amplitude number*, decides the loudness of the note. The amplitude numbers can be between ∅ and 15, with 15 producing the loudest notes. Remember that these loudness figures are relative. The actual loudness can be set with the volume control of the TV (or hi-fi) that you use to hear the sound. The loudness figure in the PLAY instruction, however, allows you to write music in which some notes are loud, some soft. If the amplitude number is zero, the result is silence.

With no more ado, then, lend an ear to our first sound program in Fig. 9.8. This sounds Middle C for a time that is decided by the delay

```
10 CLS
20 PLAY(1,4,1,10)
30 FORJ=1TO500:NEXT
40 PLAY(1,4,1,0)
```

Fig. 9.8. A Middle C program. Note that the sound has to be stopped!

loop in line 3∅. Line 4∅ is a silence instruction. This is needed, because if we omit lines 3∅ and 4∅, the note keeps playing! Pressing the BREAK or RST keys will have no effect, the note lingers on until there is a silence instruction, an error, or until you switch off. You have to be rather careful about this point later when you start writing music in harmony, because a note will sound until there is another note requested in the same channel.

How about a musical scale, the scale of C? The program of Fig. 9.9 accomplishes this. It's straightforward, with the PLAY instruction

```
10 CLS
20 T=4
30 FORN=1TO7
40 GOSUB90
50 NEXT
60 T=5:N=1
70 GOSUB90
80 END
90 PLAY(1,T,N,10)
100 FORJ=1TO500:NEXT
110 PLAY(1,T,1,0)
120 RETURN
```

Fig. 9.9. A scale-of-C program.

put into a subroutine, but with the octave number written as T. I always try to avoid using O in a program – it's too easily confused with ∅ even when the zeros are slashed. The variable T is needed because the last note in the scale of C is on the next octave above. You will find that scales that start on other notes will need the use of two different octave numbers, along with some of the sharps that use codes 8 to 12. The result is that the note numbers will not be in a simple sequence, so that you can't use a loop variable. The numbers are most easily dealt with by placing them into a DATA line, and reading in a loop.

Genie in harmony

Harmony means sounding notes together so that the result is pleasant. It went out of fashion with 'serious' composers early in this century, which is why concert halls are difficult to fill when modern music is being played! It's never been out of fashion for the rest of us, though, which is why such a huge gap has opened between 'serious' and 'pop' music. Having got that off my chest, let's look at how the Colour Genie can deal with harmony.

Figure 9.10 shows a program that produces a harmony. Lines 2∅

```
10 CLS
20 PLAY(1,4,1,10)
30 PLAY(2,4,3,10)
40 GOSUB500
50 PLAY(1,4,1,0)
60 PLAY(2,4,3,0)
100 END
500 FORJ=1TO500:NEXT:RETURN
```

Fig. 9.10. Producing harmony with two notes on different channels.

and 3∅ produce a different note on each of channels 1 and 2, so that you hear the notes sounded together. The delay subroutine keeps this sounding until lines 5∅ and 6∅ restore silence. Two 'silence' lines

```
10 CLS
20 PLAY(1,4,1,10)
30 PLAY(2,4,3,10)
40 PLAY(3,2,1,15)
50 GOSUB100
60 PLAY(1,4,1,0)
70 PLAY(2,4,3,0)
80 PLAY(3,2,5,0)
90 END
100 FORJ=1TO500:NEXT:RETURN
```

Fig. 9.11. Three channel harmony.

are needed because two channels are in use. So far, so good. Now try Fig. 9.11 which produces three channel harmony. Two of the notes are taken from octave 4, and one from octave 2 to give a good bass effect. A very small loudspeaker will not do justice to this bass note. Once more, three 'silence' lines are needed to turn the three channels off.

Just to indicate what can be achieved with a bit of blood, sweat and cursing, Fig. 9.12 plays a bit of music in three-part harmony.

```
10 CLS
20 PRINT"RED,RED ROSE"
30 FORX=1TO 23
40 READ C,T,N,V,D
50 PLAY(C,T,N,V)
60 FORJ=1TOD:NEXT
70 NEXT
75 PLAY(1,3,0,0):PLAY(2,3,0,0):PLAY(3,3,
0,0)
80 END
200 DATA1,4,4,5,500,1,4,2,5,500
210 DATA1,3,12,10,0,2,3,2,10,0,3,2,12,10
,700
220 DATA1,3,12,10,0,2,3,4,10,0,3,2,12,10
,200
230 DATA1,4,1,10,0,2,3,3,10,0,3,2,12,10,
400
240 DATA1,4,2,10,0,2,3,4,10,0,3,2,0,0,40
0
250 DATA1,4,12,10,0,2,3,2,10,200,3,2,12,
10,400
260 DATA1,4,6,10,0,2,3,2,10,0,3,2,0,0,40
0
270 DATA1,4,5,10,0,2,3,9,10,0,3,2,12,10,
400
```

Fig. 9.12. A short excerpt of music in three-part harmony.

Each note requires four PLAY data numbers plus a duration number. The PLAY numbers decide channel, octave, note and volume as usual, and the duration number is used in the delay loop of line 6∅. The effect is a real credit to the Colour Genie sound system.

Sounding-off

PLAY is the instruction that we use for musical notes, where each note is one that we can read from a music score. The Colour Genie also possesses the SOUND instruction, though, which allows you a range of effects that go well beyond the boundaries of written music and musical instruments. The instructions in the Manual are rather too brief to allow you to get the best out of this instruction, so I've dealt with it in a lot more detail here.

The instruction word SOUND has to be followed by two numbers, separated by a comma, but with no brackets. The first number is called a 'register' number. A register is a type of memory store for numbers in the range ∅ to 255, and the number that we use in the first part of the SOUND instruction will specify one of a total of sixteen registers. Each register is used for some aspect of controlling the sound system (though 14 and 15 do not affect the sound generator directly), and the second number in the SOUND instruction is the number that is placed into the chosen register. In many ways, SOUND is used rather like POKE, but with the difference that the 'address' numbers are only in the range ∅ to 15.

The effects of putting different numbers into these registers are listed in Fig. 9.13. Two registers are needed to store the numbers that

Register No.	Effect
∅	Channel 1 frequency, fine adjustment, range ∅ to 255.
1	Channel 1 coarse adjustment, range ∅ to 15.
2	Channel 2 frequency, fine adjustment, range ∅ to 255.
3	Channel 2 coarse adjustment, range ∅ to 15.
4	Channel 3 frequency, fine adjustment, range ∅ to 255.
5	Channel 3 coarse adjustment, range ∅ to 15.
6	Noise predominant frequency, range ∅ to 31.
7	Enable channels, see Fig. 9.20.
8	Channel 1 amplitude, range ∅ to 15 (16 for envelopes).
9	Channel 2 amplitude, range ∅ to 15 (16 for envelopes).
1∅	Channel 3 amplitude, range ∅ to 15 (16 for envelopes).
11	Envelope repetition time, fine adjustment (∅ to 255).
12	Envelope repetition time, coarse adjustment, (∅ to 255).
13	Envelope shape pattern (∅,1,2,4,8 only).
14	Input/output control A } Do not use!
15	Input/output control B }

Fig. 9.13. How the sound registers are used.

decide the pitch of a note, so that registers R∅ to R5 (six in all) are needed for the three channels, 1 to 3. Register 6 deals with noise, and register 7 is a selecting register. The number that is stored in register 7 is used to control how many different channels of notes and noise can be used to pass signals to the loudspeaker.

Registers 8, 9 and 1ϕ control the amplitude of each channel, and Registers 11 to 13 deal with what is called *envelope*. 'Envelope' means the pattern of a note. A musical note, or a sound effect, does not consist of just one wave of sound but many. These waves usually change amplitude while the note sounds. For example, when you strike a piano key, the note that you get starts loud and then fades away. Its envelope is therefore something like the shape in Fig. 9.14,

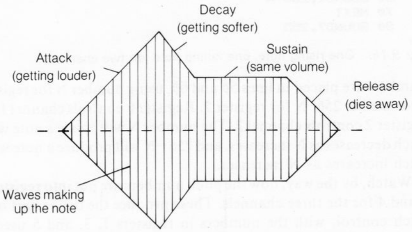

Fig. 9.14. The 'envelope' of a musical note, containing many waves.

rising very rapidly and then falling more slowly. We'll deal with the use of envelopes later.

One to some examples. Figure 9.15 produces a note whose pitch descends. Line 3ϕ puts the number 254 into register 7. This (see the

```
10 CLS
20 FORN=1TO255
30 SOUND7,254
40 SOUND8,15
50 SOUND0,N
60 NEXT
70 SOUND8,0
```

Fig. 9.15. A note of descending pitch.

Manual, page 108) has the effect of turning channel 1 on, and channels 2 and 3 off. Line 4ϕ puts the number 15 into register 8. This turns up the volume of the sound on channel 1 to maximum. The control of the note is then carried out in line 5ϕ. Register ϕ deals with the frequency of the note in channel 1, and by using N as the number that is put into this register, we can change the frequency of the note as the number value of N changes in the loop. After the loop has finished, line 7ϕ is needed to stop the sound by specifying zero volume.

Now for a bit of amusement. The program in Fig. 9.16 uses two channels. This is achieved by line 3∅, by placing the 'full volume' code number of 15 into registers 8 and 9, channels 1 and 2. The pitch

```
10 CLS
20 SOUND7,252
30 SOUND8,15:SOUND9,15
40 FORN=1TO255
50 SOUND0,N
60 SOUND2,256-N
70 NEXT
80 SOUND7,255
```

Fig. 9.16. One rising note, one falling note, on two channels.

numbers are placed in lines 5∅ and 6∅, using number N for register ∅ and number 256–N for register 2. Register ∅ controls channel 1, and register 2 controls channel 2. The number N will create a note whose pitch decreases as N increases, and 256 –N will produce a note whose pitch increases as N increases.

Watch, by the way, how the pitch numbers are put into registers ∅, 2 and 4 for the three channels. These produce the major part of the pitch control, with the numbers in registers 1, 3, and 5 used for 'coarse tuning'. You need only use registers 1, 3, and 5 if you want results that are over a very large range. The numbers are such that a high number produces a low-pitch note and a low number produces a high-pitch note.

Figure 9.17 takes us a few stages further. Line 2∅ enables all three

```
10 CLS
20 SOUND7,248
30 SOUND8,15:SOUND9,15:SOUND10,15
40 FORN=1TO255
50 SOUND0,N
60 SOUND2,256-N
70 FORJ=1TO20
80 SOUND4,J:NEXT
90 NEXT
100 SOUND7,255
110 PRINT "BLASTOFF!!"
```

Fig. 9.17. Notes on all channels to create an interesting effect.

of the channels, and line 3∅ sets maximum volume on each channel. The main loop that starts in line 4∅ then gives the same combination as is used in Fig. 9.16, but this time we have added a different sound in the third channel. This is achieved by using another loop whose SOUND command is in line 8∅. Since the other two notes keep playing while this one is altering, you hear the effect of all three notes, two steady and one changing for each value of N. Line 1ϕϕ then shuts the sound off – mercifully!

Notable notes

Before we start to look at the creation of sound effects for games programs and other purposes, it's worth noting that sound can play a useful part even in business programs. Warning notes are a useful way of drawing a keyboard operator's attention to something that is happening. This can be even more effective than messages on the screen. A warning note that sounds when a cassette has to be inserted or when a file is about to be erased, or when data has been replayed, for example, can be very useful. Let's look at some warning notes.

Figure 9.18 produces a succession of notes in which the amplitude changes. Line 2ϕ enables channel 1, and line 3ϕ sets a note value of

```
10 CLS
20 SOUND7,254
30 SOUND0,150
40 FORJ=1TO20
50 FORN=0TO15STEP.1
60 SOUND8,N
70 NEXT
80 NEXT
90 SOUND8,0
```

Fig. 9.18. Changing the amplitude of a note.

15ϕ into the channel 1 register, register ϕ. The outer loop that starts in line 4ϕ then causes the action to be repeated 20 times. The action occurs in the loop that starts in line 5ϕ, in which the volume of the channel 1 note is increased by changing the number N in the SOUND instruction. The step that is shown is .1, but this, in fact, does not cause the note to have its amplitude changed in steps of this size. The amplitude number has to be a whole number (only whole numbers can be put into registers), so the fraction is ignored. The small step is, in fact, only a way of making the loop last longer without having to include another delay loop.

The program in Fig. 9.19 generates a warbling note. This is a very effective way of getting attention, because it's somehow more difficult to ignore a note like this. The set-up conditions are the same as before, but the loop that extends from line 4ϕ to line 7ϕ sounds two different notes. The note numbers that I have used are 15ϕ and 155 – selected by trial and error! The delays in lines 55 and 65 ensure

```
10 CLS
20 SOUND7,254
30 SOUND8,15
40 FORN=1TO50
50 SOUND0,150
55 FORJ=1TO20:NEXT
60 SOUND0,155
65 FORJ=1TO20:NEXT
70 NEXT
80 SOUND8,0
```

Fig. 9.19. A warbling note which is an excellent attention-getter.

that the warble is not too fast. A very fast warble sounds like a single note with what musicians call 'tremolo'. It's another effect to note for future use, however.

A bit of noise

Some of the most impressive sound effects that the Genie can produce require the use of the noise generator, a subject which isn't covered in detail in the Manual. Noise is a mixture of frequencies, unlike a musical note in which there is one 'fundamental' note. The noise generation of the Genie depends on the use of registers 6 and 7.

We'll start with register 7, because it's the use of this register that allows noise signals to be sent to the three channels. Your choices in this matter are made by the numbers that are put in following SOUND 7, and Fig. 9.20 shows these numbers and how the values

Channels activated	Tone code	Noise code
1,2 and 3	∅	∅
2 and 3 only	1	8
1 and 3 only	2	16
3 only	3	24
1 and 2 only	4	32
2 only	5	4∅
1 only	6	48
None	7	63

Add 192 to the sum of the number(s) used.

Example: Tone on channels 1 and 2, noise on channels 2 and 3 codes are 4 and 8, which add to 12, then add 192 to get 2∅4. This, then, is the number that is placed in register 7.

Fig. 9.20. Using the register 7 to enable tones and noise.

have to be added to accomplish an effect. The number that is put into register 7, in fact, is the sum of three numbers, one of which remains constant unless you are doing some very fancy programming indeed.

Register 6, by contrast, uses numbers that range from 1 to 31. This causes the noise to have a predominant frequency, the sort of thing that makes you think you hear the sea when you hold a sea-shell to your ear. As an illustration of this sort of thing, try the program in Fig. 9.21, which produces a rather impressive 'surf on the shore' type of noise. The figure of 247 that is put into register 7 enables noise

```
10 CLS
20 SOUND7,247
30 SOUND8,15
40 FORX=1TO20
50 FORN=0TO31
60 SOUND6,N
70 FORJ=1TO50:NEXT
80 NEXT
90 NEXT
100 SOUND8,0
```

Fig. 9.21. A 'surf-on-the-shore' noise program.

only on channel 1, disabling any musical notes. The loudness of channel 1 is put to full amplitude by line 3∅. We select 20 waves in line 4∅, and then the loop in lines 5∅ to 6∅ carry out the 'wave' sound. The noise 'pitch' in register 6 is varied all the way from ∅ to 31 in each pass through the main loop. The higher values give lower pitched noise, the low values give higher pitched noise. We can now use these noises as a basis for more useful sound effects.

Sealed in an envelope?

We've mentioned the idea of a sound envelope earlier. It's time now to look at what the Colour Genie can provide in the way of such envelopes. Figure 9.22 shows the envelope shapes from which you can choose – but that doesn't tell you much until you hear their effect. You also have to know how to allow the envelope shape to take control of the amplitude of the sound.

Figure 9.23 demonstrates the effects of envelopes. The important line is 7∅ – SOUND 8,16. When 16 (or any number between 16 and 255) is used in register 8, its effect is to allow the envelope generating part of the sound generator to take control of amplitude. The sound will no longer have fixed amplitude, but values that depend on

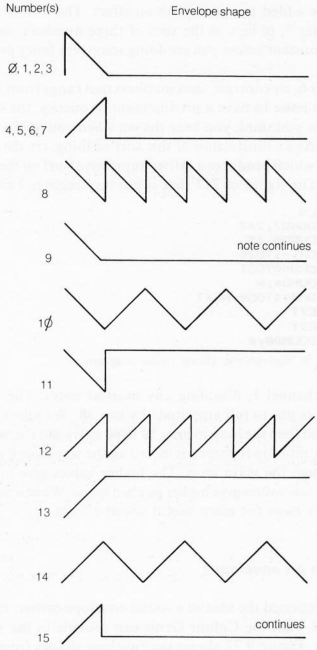

Fig. 9.22. The Colour Genie's envelope shapes, and the code numbers to select them.

whatever envelope has been chosen. The other lines are more conventional. Line 2Ø enables music on to channel 1, and line 3Ø puts a note number into the channel 1 register. Register 13 is then used to select envelopes. As Fig. 9.22 shows, several numbers can produce the same envelope shapes. You can then listen to the effects of each envelope, and compare the sounds that you hear with the

```
10 CLS
20 SOUND7,254
30 SOUND0,150
40 FORN=1TO15
50 SOUND13,N
60 PRINT"ENVELOPE NUMBER ";N
70 SOUND8,16
80 FORJ=1TO1500:NEXT
90 NEXT
```

Fig. 9.23. Demonstrating the effects of envelopes.

shapes in Fig. 9.22. The delay loop in line 8∅ gives plenty of time for one sound to be completed before the next one starts.

We can produce rather useful tinkling notes with envelope 1, as Fig. 9.24 illustrates. In this envelope, line 2∅ sets the envelope period

```
10 CLS
20 SOUND12,10
30 SOUND7,254
40 SOUND8,16
50 FORN=255TO1STEP-10
60 SOUND13,1
70 SOUND0,N
80 FORJ=1TO200:NEXT
90 NEXT
```

Fig. 9.24. An envelope for 'piano' type notes.

– short for low number, long for high numbers. This is a quantity that can be experimented with to considerable effect, because it can change the sounds considerably. Line 3∅ enables channel 1, and line 4∅ allows the envelope generator to control amplitude. The loop then selects envelope 1 – and this has to be done on each pass through the loop. Line 7∅ puts different note numbers into the music register for channel 1, and the result is – well, listen for yourself!

By way of contrast, Fig. 9.25 demonstrates what happens when we use noise along with an envelope control. The noise is selected by

```
10 CLS
20 SOUND7,247
30 SOUND8,16
40 SOUND6,8
50 SOUND13,8
60 FORN=1TO5000:NEXT
70 SOUND8,0
```

Fig. 9.25. A drumming noise program.

line 2∅, and the rest of the instructions should be reasonably familiar by now – note the use of SOUND 8,∅ to turn the sound off at the end of the program. The drumming continues for the duration of the delay loop – you don't have to have the SOUND instructions inside a loop. Finally, try Fig. 9.26, and hear what happens when we slow things down a bit. You should be able to analyse this one for yourself!

```
10 CLS
20 SOUND7,247
30 SOUND8,16
40 SOUND6,15
50 SOUND13,8
55 SOUND12,30
60 FORN=1TO5000:NEXT
70 SOUND8,0
```

Fig. 9.26. Modifying the program to produce hammering sounds.

Chapter Ten
Miscellany Corner

The Genie has such a large selection of BASIC instructions that one volume can hardly do justice to them all. What I have concentrated on in this book has been the essential instructions that allow you to write useful programs for yourself. By the time you have reached this stage you will be considerably more familiar with the capabilities of your Colour Genie, and better able to dig more advanced information from the BASIC Manual and the User's Manual.

There are, however, a few topics that deserve a mention before I leave you to your own devices. The first of them is an addition to the PRINT instruction in the form of USING. PRINTUSING forces the Colour Genie to print something in a fixed format. You might, for example, want all numbers to be printed with no more than three figures before the decimal point and no more than two following the point. This 'format' can be fixed by a string "###.##" in which each hashmark indicates the position of a figure. If we assign a string like this to a string variable, we can have, for example, USINGA$ placed anywhere after PRINT and just before the number that we want to print in this format. Figure 10.1 illustrates this in action, showing

```
10 CLS
20 PRINTTAB(14)"PRINT USING"
30 PRINT@160,""
40 N=140.2716
50 PRINT"N IS ";N
60 A$="###.##"
70 PRINT"WITH PRINT USING, IT'S ";USINGA
$;N
80 PRINT"THE V.A.T. ON N IS ";15*N/100
90 PRINT"IT LOOKS NEATER AS ";USINGA$;N*
15/100
```

Fig. 10.1. PRINTUSING being used to fix the 'format' of a number.

how a number can be printed in this fixed format. See what happens if you want to print 1234.567 using such a format. This formatting instruction is by far the most widely used application for

PRINTUSING, but you will find several other formatting strings listed in the Manual. Some of these are rather specialised (unless you deal in sums of money expressed in dollars, for example) and seldom likely to be used in your programs.

Another feature of the BASIC of the Colour Genie is what is called 'error trapping'. Normally when your computer comes across a fault which makes it impossible to proceed, it will stop and display an error message. This is useful while you are testing a program, because by this way you can get rid of any remaining syntax errors. Every now and again, though, a program will halt with an error that is not due to faulty typing or planning, but simply because a situation has arisen that the computer cannot cope with. This is the sort of problem that 'error trapping' is designed to catch. The idea is that we have, very early in the lines of a program, an instruction: ONERRORGOTO, which has to be followed by a line number. This means what it says – if an error occurs, go to the line whose number is given. This will force the computer to go to that line which will contain instructions for finding out what the problem is and dealing with it, but without stopping the program or issuing an error message (unless you want one). Error trapping is particularly valuable if the machine is to be used by relatively unskilled operators who might not make much sense of the error messages.

Figure 10.2 shows an example. Line 2∅, ONERRORGOTO1∅∅∅

```
10 CLS
20 ONERRORGOTO1000
30 FORN=1TO5
40 READX:Y$=STR$(SQR(X))
50 PRINT"NUMBER ";X;" ";"SQUARE ROOT ";Y
$
60 NEXT
70 DATA5,4,3,-2,2
80 END
1000 Y$=STR$(SQR(ABS(X)))+"J"
1010 RESUME50
```

Fig. 10.2. Error trapping with ONERRORGOTO.

ensures that if any type of error occurs, then line 1∅∅∅ will be carried out. The program then reads a set of numbers and forms a string version of the square root of each number. The catch is that one number is negative. Now the computer can't deal with the square root of a negative number, because this is an 'imaginary' number. Squaring a positive number or a negative number gives a positive number, so no real number has a square root that is negative. When −2 is read, then, the effect of line 2∅ is to make the program jump to line 1ØØØ whenever the computer attempts to find the square root. In line 1ØØØ, the absolute value of −2 is found, and the square root of this quantity taken. The letter *J* is then added. Line 1Ø1Ø then causes normal service to be resumed in line 5Ø so that the message, along with the value of Y$, is printed. The *J* is a convention used in engineering to indicate an imaginary square root of −1. If, for example, we think of −4 as being 4*(−1), then its root is $2*\sqrt{-1}$, or 2*J*. Mathematicians use *i* in place of *J* for this function.

You can do considerably more than this with ONERRORGOTO. The example I have used could have been tackled by an IF...THEN test just after the quantity was read. Not all errors are so easily trapped, however, and this is what makes ONERRORGOTO so useful. You can, for example, detect different errors in different lines. The instruction ERL means 'error line', so that you can include in your error handling routine lines such as:

```
IF ERL = 5Ø THEN GOSUB 5ØØØ
IF ERL = 1ØØ THEN GOSUB 6ØØØ
RESUME
```

This allows the computer to distinguish between errors that have happened in different lines, one in line 5Ø and one in line 1ØØ in this example, and to tackle them by different methods. The RESUME, with no line number, at the end of the error routine will cause the error line to be resumed, so that the subroutine must repair the fault. If it does not, there is a good chance that a closed loop will be formed. A variation on RESUME is RESUME NEXT, in which the program will resume action at the line following the line in which the error ocurred.

We can also separate different types of errors by using ERR. Each type of error has a code number (see page 37 of the BASIC Manual), and we can detect different types of errors by using these codes. For example:

```
IF ERR/2 + 1 = 4 THENGOSUB 4ØØØ
```

will detect error 4, which is 'out of data', and subroutine 4ØØØ could, for example, contain a RESTORE. Another code we might want to use is 11, division by zero:

```
IF ERR/2 + 1 = 11 THEN RESUME NEXT
```

will cause the program to skip the problem line if a division by zero occurs.

The error trapping routines can trap errors which you can't get rid of just by using IF...THEN tests, because you can't test for items like 'out of data' by using such tests. You must be very careful, however, how you use error trapping. If you include error trapping in a program before you have eliminated all the syntax errors, for example, you will get the most peculiar and unexpected things happening when a syntax error occurs and the error trapping takes over.

Have trouble, will shoot

The trouble-shooting commands of the Colour Genie are used to track down faults in a program, and knowing how to use them can save you a considerable amount of time when a program refuses to do what is expected of it. Inevitably, as you type in a long program, there will be syntax errors due to typing errors. These, however, will be reported by the computer when you run the program, and you can deal with each one as it appears. It should not take long before you have a program that is free of syntax faults. The errors which may still remain are the ones that will cause you more trouble, and some of them may also be reported as syntax errors. Chief among these are confusions between O and ∅ in printed programs where the ∅ is not slashed. For example, a line like:

```
IFM=10ORJ=4ORD=2
```

(don't laugh, one magazine often prints lines like this!) can cause considerable trouble, until you realise that it means:

```
IF M = 1∅ OR J=4 OR D=2
```

The other type of problem arises if you have typed @ with the SHIFT key depressed – the Manual points out this one also.

Even when all these types of errors have been dealt with, though, you can find that a program will still refuse to do as it should. It may provide incorrect answers to all or some of the items that you test it with, it may refuse to deal with more than one item of a list, it may appear to do nothing or, worst of all, go into an endless loop so that you have to press BREAK or in extreme cases the RST keys to regain control. The best weapons for investigating these problems are TRON and STOP.

TRON (where did you think the film gets its name from?) means 'trace-on'. When you type TRON, press RETURN, and then run a program, each line number will be printed on the screen as it is executed. This lets you see if the machine has stuck in a loop anywhere, because you can find that lines are being repeated. You can also find if any IF decision steps are doing strange things, because the TRON action will reveal which line is executed following the IF test line. Once you have found what is going on (make a note of it!), you can cancel the effect of TRON by typing TROFF (trace-off) and pressing RETURN.

Once you have some idea of where the program makes the program stop, as you might expect, but when this happens, all the variables in the program retain their values, and the machine keeps track of where it stopped. If, for example, you have a line 19ø that you want to investigate:

```
190 IF J=6 THEN 800 ELSE IF J=7 GOTO 900
```

then you can put a STOP in a new line 189. The computer will then stop when it comes to this new line, and you can type, using a direct command:

```
PRINT J
```

and then press RETURN to find what the value of J is at the point just before line 19ø. This value should give you a pretty good clue as to why line 19ø is causing problems.

Another even more valuable feature is that you can change the values of variables, and then continue the program! If, in the example, you type J = 6 (press RETURN) and then type CONT (then press RETURN), the program will continue, but the value of J will be 6 and you should see the effect of the test in line 19ø. You might think that this ability was so essential for fault-finding that all computers would feature it. They don't!

Having found an error, you can deal with it by using the excellent editing commands of the Colour Genie. Get to know these – in particular the use of I for Insert, and how the effect of I can be cancelled by typing the up-arrow along with SHIFT. The editing commands are well explained in the Colour Genie BASIC Manual, so I won't take up space with them here.

That's the end of the road for me, but just a beginning for you. The Genie is a box of fascinating tricks, all of which you can learn to control for yourself. This book should have unlocked some of the Colour Genie's secrets for you, and the rest is up to you. The best way to learn to make effective use of your Colour Genie is to write, correct and use your own programs, and this is something that you should have every confidence in doing now. Happy programming! You'll soon discover that your Colour Genie is certainly a more ABLE computer!

Appendix A

Cassette Head Adjustment

Cassette recorders, like open-reel tape recorders, work on the principle of pulling plastic tape, which has been coated with magnetic material, past a 'tapehead', which is a miniature electromagnet. The important part of any tapehead is the 'gap', a tiny slit in the metal, too fine to see except under a microscope. This slit should be placed so that it is at 90° to the direction of movement of the tape, but this angle, which can be adjusted by tilting the whole tapehead, is seldom precisely set, even when the recorder has been quite expensive. A poorly set-up head will make it difficult to load programs that have been recorded on correctly set-up equipment

CASSETTE RECORDER HEAD ALIGNMENT METHOD

(1) Insert a cassette, with a long program, into the recorder.
(2) Remove cable connections between the computer and the recorder.
(3) Start playing the cassette. Set the volume control to a comfortable level, and listen. Any tone control should be set to give maximum treble.
(4) Insert a thin-bladed screwdriver into the head-alignment screw-head. On some recorders this is reached with the cassette flap shut, through a hole in the casing. On other models, it will be necessary to open the flap. This may have to be done *before* playing the tape.
(5) Adjust the azimuth screw *slightly* in each direction, listening to increase in the treble (a sharper sound). If adjustment causes the note to sound more muffled, reverse the direction of turning. Adjust until the note is at its sharpest.
(6) Rewind the cassette, and make the connections between the computer and the recorder.
(7) Try to load a program. If good loading cannot be achieved, repeat the procedure, but look for *another* setting which produces maximum treble.
(8) NOTE that this procedure is needed only if a tape from a reputable source cannot be loaded. Tapes made on a recorder will be loaded by that recorder unless there is a serious fault. Once the adjustment described above has been carried out, tapes recorded *before* the adjustment may not load correctly *after* the adjustment.

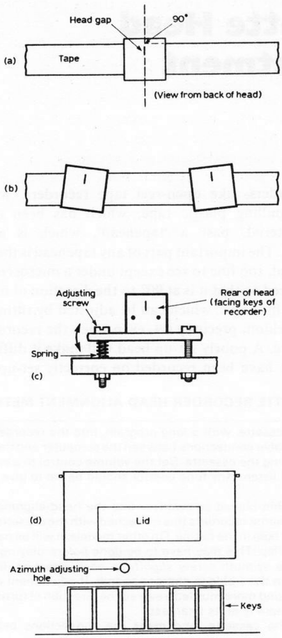

Fig. A1. Tape-head azimuth. The narrow slit in the tape-head (a) is normally at 90° to the edge of the tape. This is the correct azimuth angle, but a surprising number of recorders have this maladjusted. Any deviation from this angle (b) causes muffled sound and poor loading. The angle can be altered by turning an adjusting screw (c) which is on the head mounting. This is often reached through a hole in the casing of the recorder (d). (Courtesy of Keith Dickson Publishing.)

(bought software, for example), though you will always be able to load tapes which have been saved on the same equipment with the same head adjustment. NEVER touch the recording head with anything metal – but you can set the alignment fairly easily, following the scheme outlines here, in Fig. A1.

After a lot of use, it's important to clean the head. Use a cleaning kit, such as the BIB, and follow the instructions carefully. Alternately, Lowe Computers market a cassette head cleaning tape which should be used at least once a month. A few authorities say that the head should be demagnetised at intervals, but in five years of using the same recorder for computer loading and saving, I have never found this to be necessary.

Appendix B
Graphics Codes for the Colour Genie

ASCII CODE	GRAPHICS	ASCII CODE	GRAPHICS	ASCII CODE	GRAPHICS	ASCII CODE	GRAPHICS
128		144		16Ø		176	
129		145		161		177	
13Ø		146		162		178	
131		147		163		179	
132		148		164		18Ø	
133		149		165		181	
134		15Ø		166		182	
135		151		167		183	
136		152		168		184	
137		153		169		185	
138		154		17Ø		186	
139		155		171		187	
14Ø		156		172		188	
141		157		173		189	
142		158		174		19Ø	
143		159		175		191	

ASCII CODE	GRAPHICS	KEY	ASCII CODE	GRAPHICS	KEY (SHIFT)
192		;	211		I
193		<	212		J
194		=	213		K
195		>	214		L
196		?	215		M
197		+	216		N
198		'	217		O
199		–	218		P
2Ø0		.	219		Q
2Ø1		/	22Ø		R
2Ø2		@	221		S
2Ø3		A	222		T
2Ø4		B	223		U
2Ø5		C	224		V
2Ø6		D	225		W
2Ø7		E	226		X
2Ø8		F	227		Y
2Ø9		G	228		Z
21Ø		H	229		`

ASCII CODE	GRAPHICS	KEY	ASCII CODE	GRAPHICS	KEY (SHIFT)
23Ø		a	243		n
231		b	244		o
232		c	245		p
233		d	246		q
234		e	247		r
235		f	248		s
236		g	249		t
237		h	25Ø		u
238		i	251		v
239		j	252		w
24Ø		k	253		x
241		l	254		y
242		m	255		z

Appendix C
Non-printing Codes

The ASCII codes of numbers ∅ to 31 do not produce characters on the screen, but they do produce effects many of which you will want to use. The list below shows the codes and the effect that each code produces in a running program when PRINT CHR$(N) is used. Where numbers are omitted, no action is obtained.

Code	Action
8	Backspace cursor, erasing any character at that place.
1∅	Take a new line, left-hand side.
13	As 1∅.
14	Cursor on (in a program). } Neither of these codes has any effect on the cursor that appears at the end of a program, after the READY prompt.
15	Cursor off (in a program). }
24	Backspace cursor, no erase.
25	Cursor forward one space (no erase).
26	Cursor down one line.
27	Cursor up one line.
28	Cursor 'home' to top left-hand corner of screen.
29	Cursor to start of present line.
30	Erase to end of present line.
31	Erase to the bottom of the screen.

Appendix D
Machine Code

There are several instructions, like CALL, SYSTEM and VARPTR that have not been explained in this book, nor at length in the Manual. This is because these instructions are used to control machine code programs. A machine code program uses a set of number codes to control the action of the computer directly rather than through the BASIC instruction words. Machine code runs much faster than BASIC, and is capable of a much wider range of actions, but to learn and understand machine code you need to know very much more about how the computer works. This subject needs at least one book, preferably two, by itself. Rather than produce a short and insufficient chapter on machine code methods, I've omitted it altogether. When you are ready for machine code, there are books that deal with the machine code of the Z-80 microprocessor which operates the Colour Genie. There is also a superb program called ZEN which allows you to write machine code in 'assembly language', that is in a form which places less emphasis on number codes and more on what they do.

Appendix E

Miscellany

1. If you press the MODSEL key before you switch on the Colour Genie, and keep this key depressed as you switch on, you can get another 4K of memory for program use, but at the expense of the full-graphics use.

2. The commands:

POKE1641∅,∅ will stop the cursor blinking
POKE1641∅,32 will delete the cursor
POKE1641∅,64 will cause the cursor to blink fast
POKE1641∅,96 will cause the cursor to blink slowly

3. You can renumber the lines of your program. All of the lines are correctly renumbered, including the numbers in GOTO and GOSUB statements. The command word is RENUM. RENUM used by itself will renumber starting with line 1∅, in steps of 1∅. You can renumber with any starting line number (within reason!) and step (also!) by using two numbers following RENUM. For example, RENUM 1∅∅,5 will renumber your program so that the first line is 1∅∅, and the lines are numbered in fives.

4. AUTO (press BREAK) will cause line numbers to appear automatically when you are typing a program. The numbers start with 1∅ and step in tens. A command such as AUTO1∅∅,5 will start your numbering at 1∅∅ and step in fives.

5. LIST 1∅–1∅∅ will produce a list of the lines within these limits. It's very useful if you want to edit.

6. *Programming the user keys.* The four user keys on the right-hand side of the keyboard can be re-programmed by typing:

FKEYn="action"   (press RETURN)

where n is the number of the key, 1 to 8. The keys are numbered F1 to

F4, and the numbers F5 to F8 are obtained by using F1 to F4 along with the SHIFT key (so that F5 means F1 and SHIFT). The 'action' can be any item that you would normally have to type, but limited to seven letters maximum. This has to be placed between quotes. As examples, try:

```
FKEY1 = "PRINT"
FKEY2 = "TAB("
FKEY3 = "PRINT@"
```

Note how Key 2 will give TAB(so that you can then type the number and the closing bracket. Pressing Key 1 and the Key 2 gives you PRINTTAB(, which is a useful combination while you are writing programs. No closed quotes will produce a command and an automatic return.

7. *Playing music with SOUND.* When SOUND commands are used to obtain envelope effects, music cannot be played in the usual way. Instead, the 'channel period' registers ∅ to 5 must be used. The User's Manual (the larger booklet) for the Genie shows on page 47 what numbers must be placed in each register for each note. Use these numbers, which can be read from DATA lines, to create 'piano' music effects.

Index

RELEASE THE POWER OF THE COLOUR GENIE!

The Colour Genie takes you from your very first steps in computing right up to serious program applications. It's a micro that everyone can use and the Full Microsoft BASIC Extended included enables you to write your own programs quickly and easily. Its full size typewriter keyboard and optional joysticks with numeric pads give you powerful design tools for high resolution graphics. The sound synthesis used is the same as that in electronic music organs and gives you great sound and music facilities.

This book has been specifically written for you, the beginner, and assumes no previous knowledge of computing. It covers the BASIC (the programming language used) of the Colour Genie, including the use of the excellent colour graphics and superb sound instructions, as well as the very comprehensive set of data filing/handling instructions. A whole host of programs are illustrated for you to enjoy as you become more proficient and able! You are shown how to write your own programs so that you are soon in full command of this powerful machine.

The Author
Ian Sinclair is a well known contributor to journals such as *Personal Computing World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over forty books on electronics and computing aimed mainly at the beginner.

GRANADA PUBLISHING
Printed in Great Britain 0 246 12190 4

£5.95 net