

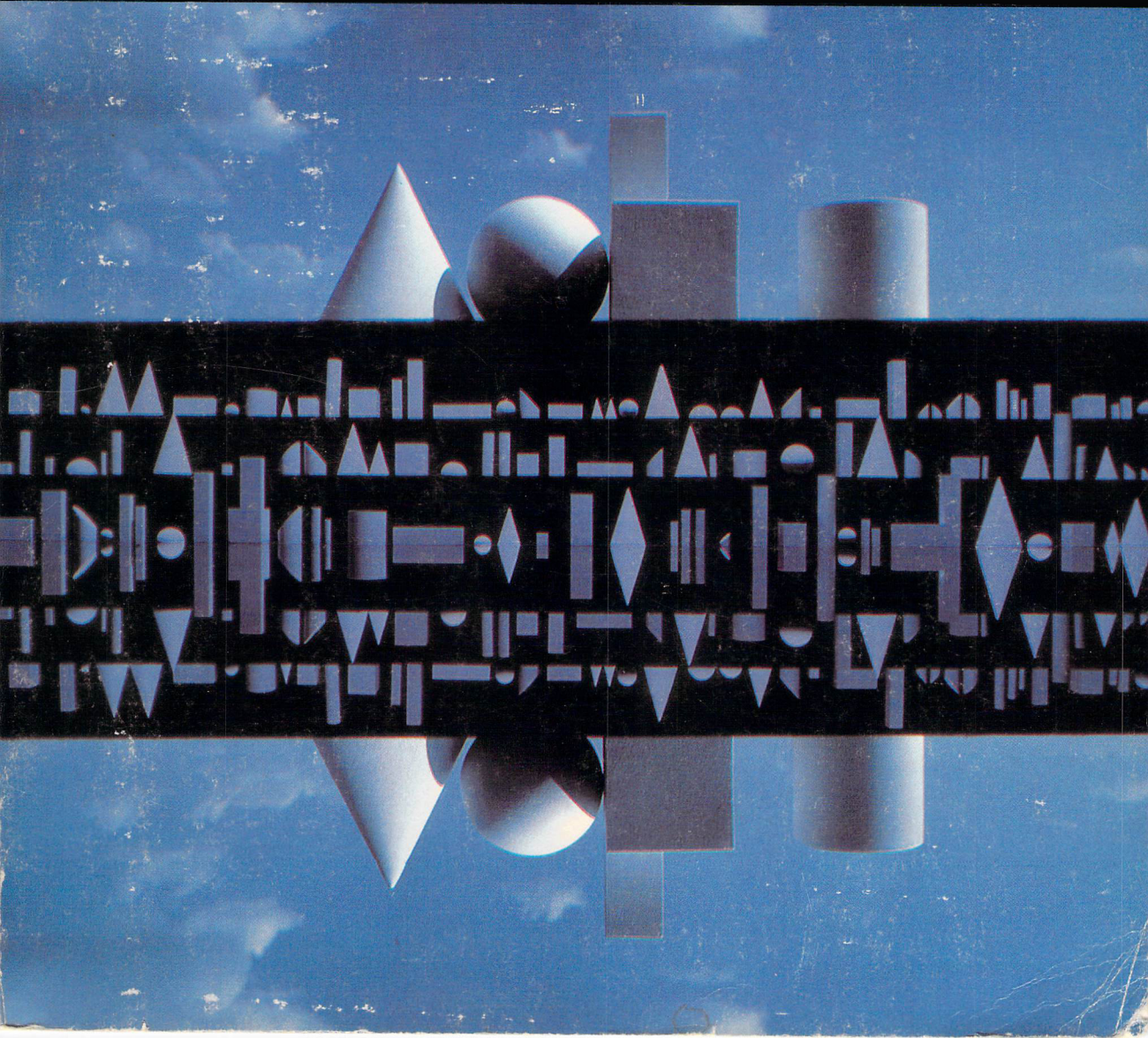
**HoneyAid™**  
A unique utility that adds 27 new  
commands to Commodore BASIC

**HAYDEN book/software**

# Commodore 64® **BASIC Programming**

LEARN TO CREATE YOUR OWN EXCITING PROGRAMS WITH SOUND, GRAPHICS,  
AND EVEN MUSIC. INCLUDES PROGRAMS TO MAKE YOUR COMPUTER EASIER TO USE.

**Peter Holmes & Derek Bush**





**Dr. Watson Computer Learning Series**

**Commodore 64<sup>®</sup>**  
**BASIC Programming**

**Peter Holmes and Derek Bush**



**HAYDEN BOOK COMPANY**

a division of Hayden Publishing Company, Inc.

Hasbrouck Heights, New Jersey

All programs in this book and the accompanying software have been written expressly to illustrate specific teaching points. They are not warranted as being suitable for any particular application. Every care has been taken in the writing and presentation of this book but no responsibility is assumed by the author or publisher for any errors or omissions contained herein.

Commodore 64 is a trademark of Commodore Business Machines, Inc., and Dr. Watson is a trademark of Glentop Publishers Ltd., both of which are not affiliated with Hayden Book Company.

*Copyright © 1983, 1984 by Glentop Publishers Ltd.* All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

*Printed in the United States of America*

---

1	2	3	4	5	6	7	8	9	PRINTING
84	85	86	87	88	89	90	91	92	YEAR

# CONTENTS

- INTRODUCTION**      The Book • The Software • Loading Programs from Diskette
- CHAPTER 1**          The Commodore 64 Keyboard • Getting Started in BASIC • Strings • INPUT • LIST • Prompts • Editing
- CHAPTER 2**          Guess the Number Game • RND • INT • GOTO • Run/Stop • IF... THEN • Stop • Program Structure—Flow Charts • FOR... NEXT • STEP • Comparisons • Multi-Statement Lines • REM • Storing a Program • SAVE • VERIFY • Comparing Numbers • OR • AND • Cursor Controls
- CHAPTER 3**          Honey.Aid • High-Resolution Graphics • HIRES • PLOT • LINE • NRM • Etcha-Sketcha Game • GET • SOUND • ENVELOPE • PLAY • TEMPO • AUTO • NUMBER • OLD • FIND • CHANGE
- CHAPTER 4**          The Hangman Game • READ... DATA • RESTORE • LEFT\$ • RIGHT\$ • MID\$ • LEN • Flags • TABS • GOSUB... RETURN • Delays • ON... GOTO
- CHAPTER 5**          Reaction Tester Game • Clock • String Concatenation • TI • POKE • PEEK • Arrays • DIM • CHR\$ • ASC
- CHAPTER 6**          User-Defined Graphics • CHAR.GEN Utility • Relocating the Character Set • Protecting the Top of Memory • Keyboard Buffer • Addressing the Keyboard • Logical Operators • AND • OR • Logical Operators and Eight Bits • VIC II Chip and the 6510 • Function Keys • Two-Dimensional Arrays • Auto Repeat on All Keys • DELETE • Self-Modifying Programs • Using CHAR.GEN
- CHAPTER 7**          Ball Games • Screen POKEing • A Moving Ball • A Randomly Moving Ball • CONT • A Moveable Bat • Building the Wall • Demolishing the Wall • Balltrap Game • Better Bouncing • Blockade Game
- CHAPTER 8**          Sprightly Sprites • Sprite Variety • Expanded Sprites • Multicolor Sprites • SPRITE.GEN Utility • Target Game • Sprite Precedence • Sprite Collisions • Game Algorithm
- CHAPTER 9**          COMPOSATUNE • Sound and the SID Chip • Playing with COMPOSATUNE • Initialization • SID POKES • Input of Musical Information • Playback of Tune • Set

Tempo • Module Testing • Transforming Note Frequencies to Peek Values • Calculating X,Y Coordinates for Note Plotting • Printing Note on Screen • The Stave • Buildasound • Pitch • Volume • Envelopes • Gate Function • Cycling through ADSR • Waveforms • Triangular Waveforms • Sawtooth Waveforms • Pulse Waveforms • Random Waveform • Filters • Resonance

## **CHAPTER 10**

Sums 'n' Things • Circles • SQR • SIN • COS • TAN • ATN • ABS • SGN • LOG • EXP • DEF FN • VAL • STR\$ • Improving the Hangman Game • Sound • 6510 Machine Language • SYS • A Screen Border Machine Code Utility

## **CHAPTER 11**

Solutions to Exercises

## **APPENDIX 1**

BIN/BCD/HEX TUTOR

## **APPENDIX 2**

Honey.Aid Syntax

## **APPENDIX 3**

Table 1 Commodore 64 Character Set  
Table 2 Hex to Decimal Conversion  
Table 3 ASCII Character Set  
Table 4 ASCII and CHR\$ Codes  
Table 5 Commodore 64 Character Symbol Representation Convention  
Table 6 VIC II (Video Interface Controller) Table  
Table 7 SID (Sound Interface Device) Table

## **Index**

# INTRODUCTION

## The Book

The Commodore 64 is by far the most exciting machine to be introduced for home/office computer use in recent years. It simply bristles with advanced features. The *Commodore 64 BASIC Programming* book/software package is carefully designed to enable you to take full advantage of this amazing machine.

What do you need to know to read and understand this book? Nothing! It is written for the absolute beginner and the early chapters have been well tried and tested—on beginners.

What will you know when you've read the book? Well, that depends on you! If you persevere, you will have used one of the most advanced home/office microcomputers on the market and will have come to grips with its most advanced features.

This book will nurse you through the early stages of BASIC and, by Chapter 2, you will be writing your first program—a number guessing game. Even though this is carefully explained step by step, your Dr. Watson package includes a safety net—a computer-aided learning program. This program complements the book and provides even more explanation of the important fundamental commands introduced in the early chapters. As you progress, the pace will quicken imperceptibly and you should soon become adept at picking up the new commands. Each of these commands is introduced with an explanation followed by a short teaching program that shows it in action. Before long it is seen playing its part in one of the many games and utilities developed in the book.

Three chapters are devoted to the development of utilities that will greatly aid your programming efforts on the Commodore 64, enabling you to build user-defined characters, sprites, and music into your own programs.

Along with the book comes “Honey.Aid”, a unique machine-code package that reaches the parts of your Commodore 64 that other packages don't! In Chapter 3 you will be given a series of programs that show Honey.Aid in use and an Appendix explains all of its features in detail.

In this book we have set out to provide what we feel the beginner will need to get started in BASIC. We hope, though, that those with some experience will not be complacent! The later chapters of the book explore the inner reaches of the Commodore 64 and will quickly elevate the beginner to a status other than ‘beginner’.

We have enjoyed the labor, though it has been long and hard and we hope that you, the reader, will enjoy the task ahead. For some of you it will be for pleasure, and for others it will be for business. Whatever your motives, may you get as much pleasure from reading it as we've had from writing it.

P. Holmes  
D.J. Bush





# THE SOFTWARE

Having bought the book and disk, you're no doubt ready to get going! That's why the disk is introduced first: you can just load a program and off you go! On the disk are three different types of programs: the games that are developed in the book, utility programs that will help you use your machine (some of which are developed in the book), and a computer-aided learning program that supplements the explanations in the book and is designed to make sure that you really do get going in BASIC.

## The Games

The games on the disk start off with a particularly simple little screen graphics game and work up to much more complex programs. This is how you will progress through the book, so that's the order of the games. No doubt you'll find the later ones of most interest and of course you're free to play them as you will.

### 1. Guess the Number

This is a number guessing game, developed in Chapter Two. The player is given six goes in which to guess a number.

### 2. Hangman

This version of the popular word game uses color and screen display techniques producing a thoroughly enjoyable, easy-to-understand game.

### 3. Reaction Tester

Developed in Chapter Five this program shows the programmer how to use the Commodore's built-in clock and how to change the screen colors, resulting in a very addictive game of speed.

### 4. Breakout

Breakout is the first of the Chapter Seven games. The reader is taken through all the steps in producing this game. The finished product will provide you not only with a great game, but also with the ability to write your own video games!

### 5. Balltrap

Developed to follow the Breakout game, Balltrap shows you how to use the game-writing skills you have learned. The object of the game is to bounce the ball from your bat and into the trap. The time limit is set to two minutes so you'll have to be quick!

### 6. Blockade

The last of the Chapter Seven games is a game for two players. They must try to block their opponent's line with their own, taking care not to be blocked themselves.

## **The Utilities and CAL Program**

There is a major utility, Honey.Aid, on the disk. Once this is loaded into your machine and run, it will make your Commodore 64 into a new machine! It adds 27 new statements to Commodore BASIC which can be typed in as any other BASIC command. These new features will help considerably when you are writing programs and will improve the appearance of your programs.

There is also a Computer-Aided Learning (CAL) program that should help you to understand the program developed in Chapter 2. That is followed by the two utilities.

### **1. Honey.Aid**

A major machine code utility program that tucks itself away in your Commodore 64 and helps you develop your own programs. In addition, it provides sound and graphics commands that are easy to use and make learning BASIC much more fun.

### **2. GUESSER**

A major Computer-Aided Learning program (CAL) that further explains the BASIC commands used in the number guessing game in Chapter 2.

### **3. CHAR.GEN**

A complete character generator program that assists in the designing of a character. It then prepares this for storage in your own program and then automatically eliminates those parts of the program that are no longer needed. The generation of this program is fully described in Chapter 6.

### **4. SPRITE.GEN**

A complete sprite generator program that enables you to design sprites of all possible configurations. As with CHAR.GEN, the finished product can be stored in your own program and the surplus parts can be eliminated.

### **5. COMPOSATUNE**

Developed in Chapter Nine, this program allows the user to input a tune and then play it back. The user is also able to change the tune, the frequency, pitch, decay, sustain, etc.

## **Loading Programs from Diskette**

First of all, you should make sure that your computer and diskette drive are connected properly. When you want to load a particular program, you should type in the full name of the program, for example,

```
LOAD"HONEY.AID", 8
```

and press RETURN. The diskette drive light will go on and the message

```
SEARCHING FOR HONEY.AID
```

```
LOADING
```

```
READY.
```

will appear on the screen. The program will then be ready to be listed or run.

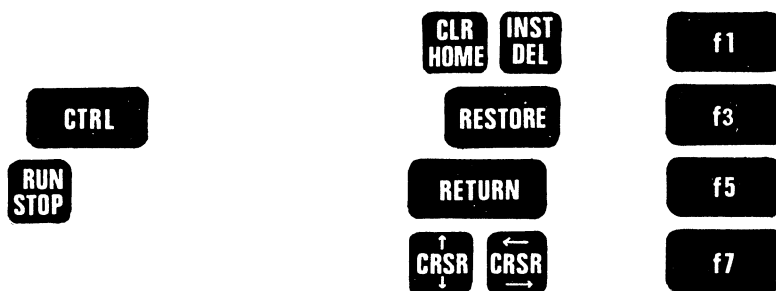
# CHAPTER

# 1

## PART ONE

### The Commodore Keyboard

**T**he Commodore 64 computer has a full keyboard which offers all the facilities of a typewriter along with many other features that only the C-64 has. It is the typewriter style keys that will occupy the first few chapters of this book. Depressing a key with a single letter or number on it will print that letter (or number) onto the screen. Some signs such as the percentage sign above the number 5 will only be printed onto the screen when the shift key is held down at the same time as the key is depressed. Other keys have very special computer functions such as keys on the right and left-hand side of the keyboard marked



Just what these do you'll see as we go along. They'll be explained when we need them.

When first turned on, the Commodore 64 will display the message:

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY
```

FIGURE 1.1

This tells you that all is okay and that there's 38911 'bytes' or character spaces for you to fill up with a program!

Below the READY sign is a flashing character or 'cursor' which is simply reporting whereabouts it is, just sitting waiting for the next entry. Try out the machine by typing a letter 'P'. If you make a mistake or just to try things out, press the <DEL> key. This should delete the 'P' or other letter and put the cursor back where it was. Armed with the power to enter and delete characters, now try to enter P,R,I,N,T and then press the RETURN key; ie:

```
PRINT <RETURN>
```

On pressing <RETURN> the READY sign will step down two lines, leaving spaces between its two positions on the screen. Try another exercise, type in:

```
START <RETURN>
```


This time, the Commodore 64 will report:


```
?SYNTAX ERROR
```

On this occasion, it's really saying

```
"I DON'T UNDERSTAND"
```

It says this because 'START' is not a part of the BASIC language that the Commodore 64 understands. Don't let this worry you when it happens, just keep trying until the C-64 accepts what you say.

One of the beauties of the Commodore 64 is the simplicity of its keyboard. Entering anything is just like typing. So if you need to enter an instruction like 'PRINT' you go ahead and type it using the "P","R","I","N" and "T" keys of the keyboard. The C-64 does however have one or two special keyboard features - these involve the use of the CONTROL, SHIFT and the COMMODORE  keys. Using a combination of these keys the programmer can produce various effects.

Let's illustrate this with some examples. The first effect is obtained by pressing the SHIFT key and the  (<Commodore key>). What this does is to change the text from upper case to lower case or vice-versa if it is already in lower case.

Assuming you have just switched your machine on and have got the message:

```
**** COMMODORE 64 BASIC V2 ****  
64K RAM SYSTEM 38911 BASIC BYTES FREE  
READY.
```

FIGURE 1.2a


Then, holding down the 'SHIFT' key and pressing the Commodore key (bottom left-hand of the keyboard) will yield the display

```
**** commodore 64 basic v2 ****  
64k ram system 38911 basic bytes free  
ready.
```

FIGURE 1.2b

Most beginners prefer to work in the CAPITALS mode so, to change back simply hold down SHIFT once more and then press the Commodore key.

Notice that the asterisks and the numbers have not changed. Asterisks and numbers come in only one type; you don't get capital numbers!

By now you have probably noticed that the keys of your Commodore 64 have more than one symbol displayed on them. For instance the 'S' key has also a small corner and a heart drawn on its front side. The character on the left (i.e the corner piece) is obtained by pressing both the Commodore key and the appropriate key at the same time. The character on the right (in this example a heart) is obtained by pressing both the 'SHIFT' key and the appropriate key (in this example being 'S'). One thing that must be noted is that if you are in small letters mode then 'SHIFT' and a letter will produce a capital letter, but if you then press 'SHIFT' and  the

display changes to capitals and the character changes. Try it and see.

As well as having other characters on them, the number keys also have colors written on them. These range from black (key 1) to yellow (key 8). The nine and zero key have other functions which will be demonstrated in a moment. The color is changed by holding down the 'CTRL' key and then pressing the number key with the required color. So for example if we wished to change the color to green then we would press down the 'CTRL' key and press '6'. Now any key we press will come out green. To change simply repeat the procedure with a different key.

As was mentioned above, the '9' and '0' keys have functions on them other than colors. The '9' key has something called 'RVS ON' and the '0' key has 'RVS OFF'. RVS ON means switch on reverse characters. These characters are exactly what they sound like, a character that is created by coloring the area around - opposed to a normal character that is created by coloring the character itself. Reverse characters are obtained by pressing 'CTRL' and '9'. From that point on all characters will be in reverse, until we press 'CTRL' and '0', which switches off the reverse characters features returning us to normality. Reverse characters can look quite impressive on the screen and their color can be varied, depending on the color of the cursor.

Two other items on the keyboard that need mentioning are 'CRSR' and 'CLR/HOME'. The 'CLR/HOME' when pressed will send the cursor to the top of the screen, where it will flicker in the left corner. But if we press 'SHIFT' and 'CLR/HOME' the screen will be cleared completely and the cursor will appear in the top left-hand corner. Thus if we have a lot of junk on the screen it can be obliterated by pressing both 'SHIFT' and 'CLR/HOME'.

The 'CRSR' (short for CuRSor) keys control the position of the cursor. There are two 'CRSR' keys, one with arrows pointing up and down and the other pointing left and right. To get the cursor to move down press the up/down key. Now hold down 'SHIFT' and press the same key and this time the cursor will move up. Left and right are controlled in the same way. Press the left/right key on its own and the cursor will move right, hold down 'SHIFT' and the cursor will move left.

It is important to note that moving the cursor over anything typed on the screen will not change the typing at all.

## PART TWO

### GETTING STARTED IN BASIC

You've probably used a calculator quite often and most of these are now pretty clever, doing 'sums' and even working out complicated routines that are actually built into the machine. Your C-64 computer is just as clever but can do much more; so much so that if you ask it to do a sum you will have to tell it what to do with the answer. Unlike the earlier calculators that could only give the answer on a screen, the C-64 could give a screen output - it could print it out on your printer, store the data on a floppy disk or even output it to some other device! However, we will concentrate on screen output.

To tell the machine to print something onto the screen the command 'PRINT' is used. Try this by typing in:

```
PRINT 4 and then press the <RETURN> key.
```

The C-64 should respond by doing what you told it and displaying a '4' on the screen. That didn't tax the computer too much, so now try:

```
PRINT 4+4 and then press the <RETURN> key.
```

The 64 should respond with the answer 8. This command 'PRINT' is part of the language that your computer speaks, known as BASIC. To those who like jargon, BASIC stands for 'Beginners All-purpose Symbolic Instruction Code'.

Unlike the calculator, however, the computer can handle not just numbers (numeric characters) but also letters and, therefore, words (alphanumeric characters). Try this with :

```
PRINT "FRED"
```

This time the 64 should print your name. In the unlikely event that your name is not FRED you can try substituting your own name between the quotation marks. One important thing to notice is that lines entered may be much longer than this one and so the machine needs to be told when you've finished typing. This you do by pressing the 'RETURN' key.

Another thing to notice here is that the alphanumeric entry, i.e. letters and numbers, MUST be included in quotation marks, as BASIC handles alphanumeric and numeric characters differently. In the example given, "FRED" (or your name) is known as a 'string' as it is really just a string of individual letters strung together.

So far we have not expected the 64 to actually 'remember' anything - now to do so! Type in:

```
LET X=4 <RETURN>
```

So far the machine appears not to have done very much; so just to check that it has done something, tell it to PRINT onto the screen by means of:

```
PRINT X <RETURN>
```

If the 64 remembered it should have told you that X=4 by printing the '4' onto the screen, as the command "PRINT" tells it to 'PRINT' on the screen!

When entering this command, the X was not typed between quotation marks. Had it been typed in so: PRINT "X", then the machine would have responded by writing what it saw between the quotes i.e. the letter X. Just try it to make sure! In the earlier case, the command PRINT X told the computer to print 'the value of the variable known as X', which was 4.

In fact, what really happened was that typing LET X=4 told the computer to create a variable called X and store the value 4 in it. Going through this in stages, the computer does the following:

(i) it reads 'LET X' and translates this into "create a space in memory and call this X" - see FIGURE 1.3.

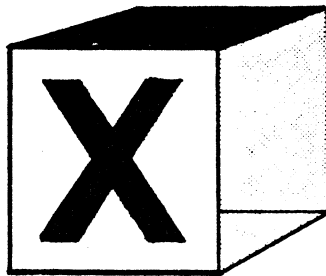


FIGURE 1.3.

(ii) next it reads the '=4' part and translates this into "and store the value 4 in X" - see FIGURE 1.4.



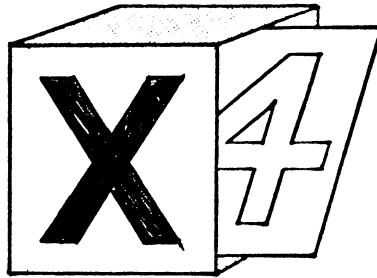


FIGURE 1.4.

Some BASICs insist that you use the keyword 'LET', as in LET X=4. 64 BASIC allows you to omit the 'LET' and normally we will do so, although use of LET is optional and if you wish to use it, please feel free.

The letter 'X' is an example of a 'numeric variable name'. 64 BASIC allows numeric variable names to be one or more letters long, but only the first two characters are actually recognized. Thus the names JOHN and JOAN would both be recognized as JO by the 64 - could be awkward! To avoid the problem of the computer potentially recognising two variable names as the same, it is best to stick to using two letters at most.

The 64 actually recognizes the following types of numeric variable:

- A single letter
- A pair of letters
- A single letter followed by a digit 0 to 9

If you insist on using 'FRED 37', then your 64 will treat it as the variable 'FR'.

If you want the fancy terms, the statement LET X=4 is referred to as 'assigning the value 4 to the numeric variable X'. The value of X is clearly very readily re-assigned; hence the term "variable", the whole 'X=4' being referred to as a mathematical or arithmetic expression.

Your 64 will remember that X=4 until something is done to make it forget. Switching it off is quite effective, as is re-assigning the variable, i.e. 'X=5'. There is however, a command, that tells the computer to 'forget' the value of any variables that we have created. The command is CLR. When entered, 'CLR' will CLear the value of 'X' ( and indeed any variable set up ). So if you type in:

CLR <RETURN>

and then attempt to PRINT X the answer would be '0' and not '4'. This 'CLR' command allows us to remove any variable data from our computer, thus - effectively - switching 'OFF and ON' the computer without the loss of a program.

## Strings

Not only can the computer store and manipulate numbers and numeric variables, it can handle alphanumeric characters and strings in variable form too.

String variables are more problematic to any computer than ordinary numbers, as they can be of different lengths: from 0 to 255 characters. Clearly, very long strings need more space in the computer's memory than numeric variables which only need 5 memory cells (or bytes). The computer must "know" whether a numeric or string variable is being used. The computer will, of course, need to store the string length as well as the variable name. Thus, string variable names must be identified by following them with a dollar sign. Commodore 64 treats the string variable 'A\$' differently from the numeric variable 'A'. To test out string variables try the exercise below:

```
LET A$="FRED" <RETURN> (or your name)
PRINT A$ <RETURN>
```

This assigns the name to a variable A\$ (see Figure 1.5) then it prints the name in A\$ onto the screen.

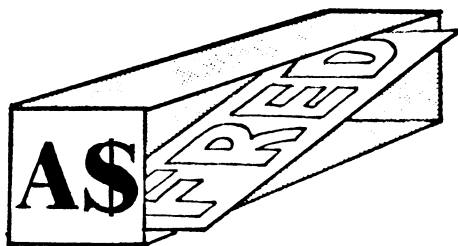


FIGURE 1.5

The name A\$ is an example of a BASIC string variable, and C-64 BASIC allows these variables to be defined the same as the numeric variables, i.e. two letters or a letter and a number are allowed. Again, string variables must end with a '\$' sign. So: A\$, B\$,

and XY\$ are all acceptable string variable names in C-64 BASIC.

So far, all the entries made have been carried out (or executed) immediately after the RETURN key was pressed. These are known as DIRECT ENTRY or IMMEDIATE mode commands. Once these have been executed, they cannot be re-activated; they're gone forever! However, when programs are used, they are stored in memory and re-activated when required. What differentiates an immediate command from one in a program is the program's line numbers. Whenever the RETURN key is pressed, the computer stores a BASIC program line in memory as a line of program. Thus, if the earlier immediate entry command is replaced by Program 1.1, this constitutes an actual program. Just what the value of the line number is doesn't really matter too much, as long as it is a whole number with a positive value between 0 and 63999. What does matter is the sequence of line numbers, as the C-64 will run the program starting at the lowest line number and then work in increasing line numbers unless told to do otherwise. Applying line numbers to the early direct entries yields:

PROGRAM 1.1

```
20 LET A$="FRED" <RETURN>
30 PRINT A$ <RETURN>
```

This time, when RETURN was pressed, the machine simply responded by moving the cursor to the next line on the screen. In order to get it to run, simply type RUN and press <RETURN>. Once you have done this, you will have run your first real program.

## INPUT

In Program 1.1, your name was stored directly in the program, which means of course that the program is only of use to you. To make it of more general use, it would be handy to be able to set the value of A\$ once the program was running. This is done by using an 'INPUT' command which causes the C-64 to stop and wait while the user enters the required information. The machine also needs to be told which variable name to assign to this information. Program 1.2 shows how an INPUT command is utilized to assign a value to A\$ at the beginning of the program.

PROGRAM 1.2

```
20 INPUT A$ <RETURN>
30 PRINT A$ <RETURN>
```

In order to erase the old line 20, it is only necessary to type in the new one and then press RETURN. The machine will then write the new line over the old.

When you type in RUN and then press <RETURN>, a question mark appears on the next line. The flashing then cursor appears after the question mark, indicating that the computer is awaiting an INPUT of information. On typing in your name and pressing <RETURN>, the computer will confirm the INPUT as it executes line 20 and prints out the value of A\$ - your name!

One thing to note about 'INPUT', is that it can't be used in immediate mode. If you try to use it without a line number the C-64 will respond with: '?ILLEGAL DIRECT ERROR'. Try it and see!

As you enter your program line by line, the computer displays it on the screen. You can only do that for so long though and eventually you'll run out of screen. The C-64 lets you enter lines until you reach the bottom of the screen. Then the whole screen is moved up so that a new line can be entered. In doing this the top line leaves the top of the screen. The whole process continues line by line and is given the fancy name 'scrolling'.

The C-64 allows you to redisplay previously entered lines in long programs by use of the LIST command.

## LIST

Just type LIST and the program lines will be sorted out by line order and displayed on the screen. The listing appears quickly and will scroll. If the listing 'scrolls' too fast for you and you want to slow it down then hold down the 'CTRL' key and the lines will appear one by one - slow enough for you to read. If you take your finger off the key the listing will continue with its normal speed. If you see the line you want going past, quickly press the RUN/STOP key and this should stop the whole proceedings.

When you want a listing of just one particular line, say line 2000, simply type 'LIST 2000'. However, should you want a listing of the whole program from line 2000 onwards, then the command 'LIST 2000-' is entered and the program listing will start from this line and continue in sequence. If you just want to list a section of your program, say from lines 2000 to 2310 then use:

```
LIST 2000-2310
```

Summarizing this:

LIST 2000	means list line 2000 only
LIST 2000-	means list line 2000 onwards
LIST 2000-2310	means list lines 2000 to 2310
LIST -2000	means list up to line 2000

So far, all the PRINT statements used have had something simple to print out. However, it is sometimes necessary to print several items at once on to the screen. This is handled in BASIC by means of features that tell the machine what "format" is required on the screen. Thus, in order to print the A\$ in Program 1.2 on the screen twice, it would be possible to write a line in a program that had a PRINT command followed by two A\$'s. Although the C-64 would understand the statement 'PRINT A\$A\$', technically the two variables should be separated. Not surprisingly, the things used to separate them are known as 'separators'!

To test this procedure, Program 1.2 will be modified to print out A\$ four times, first of all using the ',' (comma) separator. Line 30 is changed to read:

```
30 PRINT A$,A$,A$,A$
```

This yields PROGRAM 1.2(a):

PROGRAM 1.2(a)

```
20 INPUT A$ <RETURN>
30 PRINT A$,A$,A$,A$ <RETURN>
```

When this is RUN with an A\$ INPUT of "Fred" it yields a screen display of:

```
FRED△△△△FRED△△△△FRED△△△△FRED
```

FIGURE 1.6

Note: throughout this book the small delta sign '△' in listings means a space.

Program 1.2(a) will PRINT each of the strings onto the screen, followed by a group of six spaces.

The other separator is the ';' (semi-colon), which has the straight-forward effect of causing one string to be PRINTed immediately after the other. This is demonstrated in Program 1.2(b) where line 30 is further extended by means of a semi-colon separator.

PROGRAM 1.2(b)

```
20 INPUT A$ <RETURN>
30 PRINT A$,A$,A$,A$;A$ <RETURN>
```

This illustrates the effect of the semi-colon and yields the display shown in Figure 1.7.

FRED~~AAAA~~FRED~~AAAA~~FRED~~AAAA~~FREDFRED

FIGURE 1.7

In all the examples quoted so far, separators have been used with string variables; their use with numeric variables is absolutely identical. The only difference is that numeric variables are automatically printed with two trailing blank characters. So:

PROGRAM 1.2(c)

```
10 A=1 <RETURN>
20 PRINT A,A <RETURN>
```

produces:

1~~AAAAAA~~1

when it's RUN, whereas

PROGRAM 1.2(d)

```
10 A=1 <RETURN>
20 PRINT A;A <RETURN>
```

will produce

1~~AA~~1

## Asking for data...Prompts

In the course of a program the '?' is not a very informative way of asking for information and the addition of some brief message would greatly improve matters. Such a message, usually known as a 'prompt', can very readily be added using a PRINT statement. The material to be printed being generally referred to as the 'print item'. This is illustrated in line 10 of Program 1.3.

### PROGRAM 1.3

```
10 PRINT "PLEASE TYPE IN YOUR NAME" <RETURN>
20 INPUT A$ <RETURN>
30 PRINT A$ <RETURN>
```

Any string of characters such as that in line 10 of Program 1.3 is known as a 'literal string' because the line of program simply instructs the computer to print literally onto the screen what it sees in quotes. Thus a literal string is printed onto the screen exactly as it appears in the program and is affected by the two separators in exactly the same way as any other string.

When Program 1.3 is run, the machine is a little more informative and actually asks for your name. However, there's an even neater way of doing the job in BASIC! The INPUT instruction can itself be used to print a message by inserting the message between the 'INPUT' and the 'A\$'. The line required is a mixture of lines 10 and 20, and, as line 10 is rather long it would seem to be a good idea to use this to make up the new line. This can be done conveniently using the Commodore 64's EDIT function, and here's how the trick's done:

## Editing

Any editor, screen or line based, needs to perform three jobs:

- \* To correct or replace certain characters
- \* To delete or rub out unwanted characters
- \* To insert or add extra characters

Let's have a go at all three!

Firstly, LIST Program 1.3 if it's still in the computer (i.e. type 'LIST' and press <RETURN>), otherwise, type it in again! It should appear as:

```
10 PRINT"PLEASE TYPE IN YOUR NAME"  
20 INPUT A$  
30 PRINT A$  
READY
```

The first task is to replace the 'PRINT' in line 10 with 'INPUT', so the cursor needs to be moved from below the 'R' of READY to the 'P' of 'PRINT'. To do this, hold down the SHIFT key and press the CRSR (CURSOR) key with the up/down arrows, four times, i.e.

	
hold down	press four times

As you press the cursor key, the cursor will move up line by line. Once it is on the '1' of 10 (the line number), it needs to be moved to the right. This is done by pressing the horizontal cursor key (next to the vertical one) three times.

Once the cursor is over the 'P' of 'PRINT', simply type in the word: 'INPUT'.

Next, the ';A\$' has to be added to the end of line 10, and this is readily achieved by holding down the horizontal cursor key until the cursor has moved to one character past the final quote sign; ie:

```
10 INPUT "PLEASE TYPE IN YOUR NAME"
```

Now the semi-colon and A\$ can be added so the line reads:

```
10 INPUT"PLEASE TYPE IN YOUR NAME";A$
```

Once this is achieved, press <RETURN> and that line is changed.

In our example above, we performed the first and last of the three editing tasks; we corrected the characters P R I N T, changing them to I N P U T and we added the extra ';A\$'. This was done by using the cursor control keys to get to the correct line. The change from PRINT to INPUT was made character by character, and finally the extra ;A\$ was typed in.



Having modified line 10 to include an INPUT command, the program now contains two INPUTs and, thus, line 20 needs to be deleted. This is quite simply done by typing in the number 20 and then pressing RETURN.

Once line 20 has been removed, Program 1.4 should appear as below.

#### PROGRAM 1.4

```
10 INPUT "PLEASE TYPE IN YOUR NAME";A$
30 PRINT A$
```

When this is run, a prompt will appear asking for your name and, following the entry, the computer will simply print it back onto the screen with no ceremony!

When doing the previous editing, the word INPUT very conveniently fitted exactly over the word PRINT. Now let's try changing line 10 to read:

```
10 INPUT"WHAT IS YOUR NAME";A$
```

Once again, first LIST the program, by typing 'LIST'.

```
10 INPUT"PLEASE TYPE IN YOUR NAME";A$
30 PRINT A$
```

then:

- \* Move the cursor up to cover the '1' of line 10.
- \* Move the cursor across to cover the 'P' of 'PLEASE'.
- \* Type in 'WHAT IS':the text should now appear as:

```
10 INPUT"WHAT ISTYPE IN YOUR NAME";A$
```

- \* Space across with the SPACE bar, until the cursor is on the 'Y' of 'YOUR'. At this stage, the words are OK - just too many spaces.
- \* Delete the intermediate spaces on the line by means of the INST DEL key on the top-right of the keyboard. Each time you press this key, it will delete the character immediately BEFORE the cursor. Seven presses of the key should do the trick - don't forget to leave one space.
- \* Press <RETURN> and the line is edited.

The only way to really learn to edit is to try it. Nothing will go into the computer's memory until you press <RETURN>, so, if you get in a mess, just cursor away from the problem line and then try again.

One last feature to note...

Just as the INST/DEL key took out spaces, so it can insert them, when used with the SHIFT key pressed. Experiment with this and then try Exercise 1.1.

#### EXERCISE 1.1

Modify the above line 10 of Program 1.4 to read:

```
10 INPUT"PLEASE TYPE IN YOUR FULL NAME";  
A$
```

A possible solution is given on page 11.1

Fortunately, the PRINT command can also contain both a message and variables in much the same way as the INPUT command, so try to modify Program 1.4 as instructed in Exercise 1.2.

#### EXERCISE 1.2

Edit line 30 of Program 1.4 so that the program announces "YOUR NAME IS FRED". An answer is given on page 11.1

Such messages or prompts as you are now capable of putting into your programs are valuable in guiding the user through data entry. Try Exercise 1.3 using prompts.

#### EXERCISE 1.3

Modify the program developed in Exercise 1.1 so that it asks a person's name and age, and then reports back to them "YOUR NAME IS...., YOUR AGE IS .... ". A possible answer is given on page 11.1

## CHAPTER

# 2

### PART ONE

#### Guess the Number

**T**his first mini project develops a number guessing game and investigates various number manipulation techniques. In the game, the computer will think of a number between 1 and 100 and then, ask the player to guess the value of the number in less than six goes. (S)he will then be told whether this is too large, too small, or correct. After six goes, if it has not been guessed correctly, all will be revealed! At this stage, when the number is guessed correctly, the player will be asked whether or not (s)he wishes to have another go.

#### RND( )

In a game such as this, the key function is that of generating a random number for the player to guess. The 64 does this by means of the command RND( ). To try this out, tell the computer to generate a random number and then to PRINT it onto the screen. Such a combination of commands is referred to as a 'statement', which in this case says:

```
PRINT RND(0)
```

This will cause it to print a random number between 0 and 1. However, this range of numbers is not too large; the game we are writing really needs a range of about 100. To achieve such a range, we can quite simply multiply our random number by 100, i.e.

```
PRINT 100*RND(0)
```

## INT ( )

Although the range is now right, the numbers that the computer gives us have decimal points and what we really need is just the whole part of the number. BASIC contains a command that will remove the fractional part of a number and leave the whole number part or INTEGER. Not surprisingly, the command is 'INT( )', and the expression INT(6.318) would, for instance, yield the integer 6, as would INT(6.0001) or INT(6.9999).

The 'INT' expression is always followed by parentheses and expects to find the number to be operated upon enclosed in parentheses. This number is given the technical term 'argument'.

The random number function developed so far can thus be further refined to:

```
PRINT INT (100*RND(0))
```

This statement will produce integer numbers from 0 to 99. Two features of BASIC bring this about; one of these being built into the RND function itself. PRINT RND(0), would yield a random number between 0 and 1 BUT NEVER 1 ITSELF. Therefore, PRINT(100\*RND(0)) would never yield 100. Since INT always yields the whole number part, PRINT INT(100\*RND(0)) produces integers from 0 to 99 inclusive.

If we want the random number to be between 1 and 100 then it is finally fixed by adding 1 onto the integer function to yield Program 2.1(a), where RV - for Random Value, is the random number.

PROGRAM 2.1(a)

```
30 RV=1+INT(100*RND(0))
```

So far there has been no proof of the functioning of the RND function over a number of cycles. This will be investigated by modifying Program 2.1(a) so that line 30 repeats 100 times.

## GOTO

For line 30 to be activated a number of times, some command is needed which will tell the program to jump to lower or higher line numbers. The command that does this is 'GOTO' and it can be added to Program 2.1(a) as 80 GOTO 30 - to yield Program 2.1(b). Its operation is really quite clear - 'GOTO 30' tells the program to do just that - to go to line 30! Once this is done, Program 2.1(b) is said to LOOP back to line 30 from line 80.

### PROGRAM 2.1(b)

```
30 RV=1+INT(100*RND(0))
35 PRINT RV
80 GOTO 30
```

## RUN/STOP

When this is RUN, the program will enter an endless loop which PRINTs random numbers down the screen. The 64 will carry on printing these numbers and will scroll up ... well for a long time. But you can terminate the whole process by pressing the 'RUN/STOP' key. You should get a message saying 'BREAK IN ?', where ? is 30, 40 or 80, i.e. the line being executed when the 'break' into the program occurred.

So far, the program is capable of giving quantities of random numbers but in a rather uncontrolled manner. What is needed is some form of counting mechanism and some check on this count, to say when 100 numbers have been delivered.

A counting mechanism is provided by the introduction of a counting variable called 'C' - for count. This is set to zero at the beginning of the program and then increased by one (incremented) each time a random value is PRINTed onto the screen (at line 60). Just as we were able to LET C=1 (or any other number), BASIC allows us to LET C=C+1. It would appear strange in ordinary mathematics, a statement such as this but then, that's BASIC. Thus far, then, the program structure is as in Program 2.1(c)

### PROGRAM 2.1(c)

```
Set count to zero.           10 C=0
Generate a random number.    30 RV=1+INT(100*RND(0))
Print the random number
onto the screen.            35 PRINT RV
Increment the count.         60 C=C+1
Go back for another random
number.                      80 GOTO 30
```

If lines 10 and 60 are added and the program is RUN then a count will be made of the number of random numbers printed onto the screen. However, that is all that the program will do! So far it has not been told to respond in any way to this number. As an experiment, RUN the program for a few minutes. When the fun(!) has worn off press the 'RUN/STOP' key and exit from the program. Next, to check that the count routine has worked, type in: PRINT C and the machine will respond by telling you how many random numbers it has printed.

#### IF...THEN

So far so good - we can count! The next job is to modify the program so that it can carry out a check on the state of PRINTing and stop when enough lines have been displayed. This is done by the checking or CONDITIONAL statement, that is added in line 70 of Program 2.1(d).

#### PROGRAM 2.1(d)

```
70 IF C=101 THEN GOTO 90
```

This statement checks the value of Count and if - and ONLY if - it equals 101, causes the program to jump to line 90.

When putting in statements such as that in line 70, care has to be taken over the number tested against. In this case the value that brought about the loop was 101 because the incrementing was done after the random number generation and PRINTing onto the screen. Were this incrementing to have been done, say, in line 20, then the branch would have been brought about in the case where C=100.

#### STOP

Although line 70 gets the machine out of the endless loop, it sends it to a non-existent line, thus producing an undefined line error. What is required at line 90 is a line that ends or STOPS the program. For this the command STOP is used, as in Program 2.1(e).

#### PROGRAM 2.1(e)

```
90 STOP
```

## A Diversion: Program Structure.

As programs become more and more complex, they become more and more difficult to follow and some means needs to be found for representing the flow of a program in a form that can be readily understood. Such a device is known as a:

### FLOW CHART

A flow chart breaks the program down into simple elements which:

- \* START or END programs (terminators).
- \* Input and Output: commands such as PRINT, INPUT, SAVE and LOAD.
- \* Make decisions: IF...THEN.
- \* Process data: assignment statements.

There are other program statements which don't quite fit into the above pattern. GOTO, for example, changes the sequence of lines as a program is actually running.

It is often helpful to use a special diagrammatic form of FLOW CHART to understand the logic of a program. Standard symbols are used for each of the four program elements mentioned above, as their use enables the diagrams or charts to be interpreted much more readily.

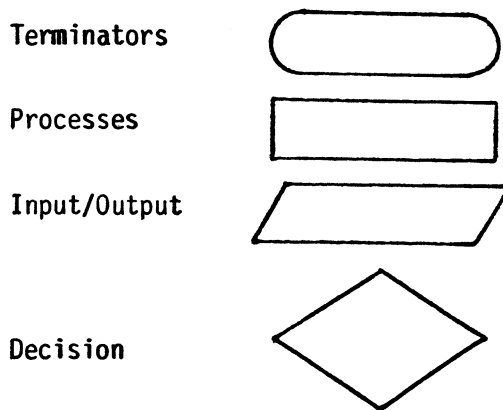


FIGURE 2.1

The rule for following a flow chart is really quite simple as it starts at the top of the chart and follows the lines connecting the boxes. The arrows on the connecting lines show the direction of flow.

Flow charts can be helpful when first designing a program. By convention, the explanations in the boxes should be written in plain English. It is a common mistake to write "BASIC in boxes" and think that is a proper flow chart. Always aim to make your flow charts language independent.

Notice that the 'GOTO' in Program 2.1(f) is represented by a flow line on the flow chart, Figure 2.2. All the other equivalents to the program statements are contained in one of the four box types given above (Figure 2.1). The combination of Programs 2.1(c), 2.1(d) and 2.1(e) gives Program 2.1(f), which when run will print out 100 random numbers.

One other thing to notice on Program 2.1(f) is the use of IF...THEN on line 70. In this line the GOTO is missed out as C-64 BASIC allows this shorthand form. Thus, when it sees "THEN 90" it interprets this as 'THEN GOTO 90'.

PROGRAM 2.1(f)

```
10 C=1
30 RV=1+INT(RND(0)*100)
35 PRINT RV
60 C=C+1
70 IF C=101 THEN 90
80 GOTO 30
90 STOP
```



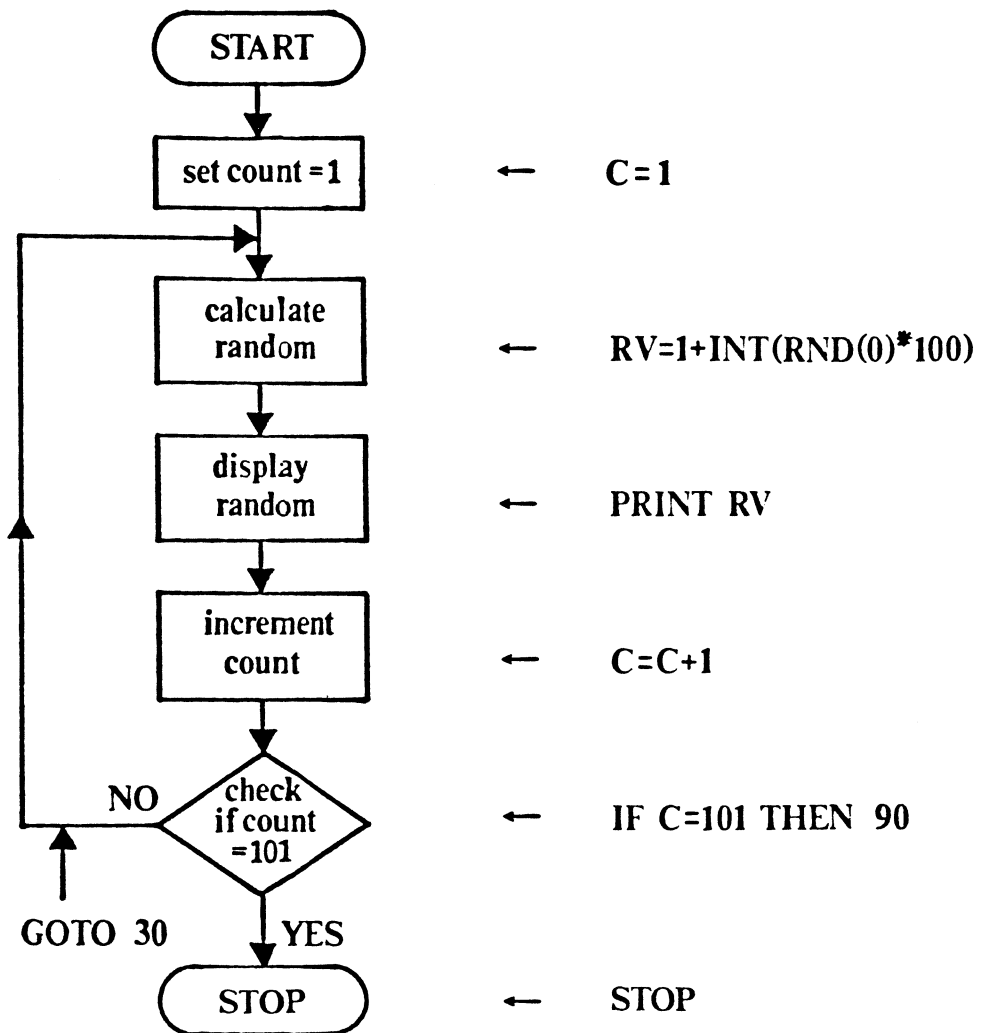


FIGURE 2.2

Other possible conditional tests are available in BASIC and all the normal mathematic functions (or operators) can be used to test values. For instance, Program 2.1(f) could be modified to use the 'greater than' sign, or '>', to bring about the loop. If you're a little unsure about the mathematical terminology, just try mentally replacing the '>' sign every time you see it with the words "is greater than". Thus line 70 of Program 2.1(f) reads in its two versions:

- (i) 70 IF C>100 THEN 90
- (ii) 70 IF C is greater than 100 THEN(GOTO)90

Line 70 (Program 2.1(g)) uses the '>' operator to replace the '=' used earlier in Program 2.1(f).

PROGRAM 2.1(g)

```
70 IF C >100 THEN 90
```

Another command available for mathematical comparisons is '<' which means 'less than' and is used in exactly the same way as the 'greater than'.

#### EXERCISE 2.1

Rewrite Program 2.1(g) to produce Program 2.1(h), which uses the line

```
70 IF C <(a number) THEN....
```

Draw a flow chart to explain the operation of your program. The program answer is given on page 11.2.

The programs using conditional tests have enabled loops to be written but BASIC contains its own built-in loop generator that makes life much easier - this is the :

FOR...NEXT loop

When using this construction it is only necessary to define the beginning and end of a loop, as shown below:

```
FOR..... Beginning of loop.  
Loop      Instructions within loop  
NEXT..... End of loop.
```

As in Program 2.1(c), the number of passes through the loop needs to be defined and this is achieved by means of a variable that is incremented on each pass of the loop. Thus the form shown above requires amendment, to become:

```
FOR C=1 TO 100
Loop
NEXT C
```

In this loop, the term 'C' is known as the loop or control variable as it controls the number of times that the loop is executed.

Incorporating this into Program 2.2 (produced from Exercise 2.1), the 'FOR' and 'NEXT' lines replace lines 10 and 60, as indicated in Program 2.2. Lines 70 and 90 have also been deleted. See if you can use the edit features to carry out these changes!

#### PROGRAM 2.2

```
10 C=1
30 RV=1+INT(100*RND(0))
35 PRINT RV
60 C=C+1
70 IF C <101 THEN 30
90 STOP
```

```
FOR C=1 TO 100
NEXT C
```

Putting this FOR...NEXT loop into practice results in much easier programming of loops. For instance, Program 2.2 can be simplified as shown in Program 2.3 below.

#### PROGRAM 2.3

```
10 FOR C=1 TO 100
30 RV=1+INT(100*RND(0))
35 PRINT RV
60 NEXT C
```

#### STEP

FOR...NEXT loops can tell the computer to count in 'steps' of more than one using the 'STEP' command. The command is added to the end of the 'FOR..' statement like so ....

```
10 FOR X=1 TO 100 STEP 'n'
```

If we do not specify a 'STEP' then a 'STEP' of one is assumed. The 'n' denotes any number. To demonstrate the use of 'FOR...NEXT...STEP' enter and run Program 2.3(a).

PROGRAM 2.3(a)

```
1 FOR X=1 TO 100 STEP 2
2 PRINT X
3 NEXT X
4 STOP
```

This particular loop starts at '1' and prints out every second number. So the display would be '1', '3', '5' up till the last value of 'X', 99.

EXERCISE 2.2.

Change line 1 of program 2.3(a) so that the loop starts at '0' and increases in 'STEPS' of three. Answer on page 11.2

The loop doesn't have to start at '0' or '1' but can begin at any value less than ( or equal to ) the 'TO' value. If the first value is larger than the second (i.e FOR X = 100 TO 50 ) then what's needed is a countdown, 100, 99, 98, 97 etc.. To do this we use a 'STEP' value of minus one ( -1 ).

So to count down the instruction would read:

PROGRAM 2.3(b)

```
1 FOR X=100 TO 50 STEP -1
```

EXERCISE 2.3

Write a short program that will count down from '10' in 'STEPS' of '-1'. When the loop has been completed then the program will PRINT 'FIRE' . A possible answer is on page 11.2

If we were unwise enough to have lines like these ...

```
        FOR X=100 TO 10 STEP 1  
    or   FOR X=10 TO 100 STEP -1  
or even FOR X=10 TO 20 STEP 30
```

then the computer will execute each loop once, and then will proceed to the next program statement after the loop. You must make sure that your starting and ending loop values are accurate.

As programs become more complex and include such features as FOR...NEXT loops, the danger of making mistakes increases. Fortunately, the 64 stays with you and when a bug creeps in the messages tell you pretty well what the error is. To demonstrate this, add line 4 to Program 2.3(a).

PROGRAM 2.3(c)

```
4 NEXT I
```

When this is RUN, the 64 will give an error message:

```
?NEXT WITHOUT FOR ERROR IN 4
```

This tells you quite clearly that you have attempted to use a NEXT without a matching FOR at line 4 as the 'FOR' line used the variable X and the 'NEXT' line, the variable I.

Errors in 64 BASIC are readily picked up in this way as the computer has been taught its own logic. For instance, if you chose to say in English "Sat the cat, the mat on", this would be incorrect in its 'syntax'. Thus, when similar errors occur in the 64's language, the computer tells you that a 'SYNTAX ERROR' has occurred. Just think of 'SYNTAX ERROR' as the computer's way of saying "I DON'T UNDERSTAND".

However well the machine knows its own logic, it cannot know what you, the programmer, are thinking. Thus, if you put logical errors into a program, the 64 will run these faithfully whatever problems they cause for it. For instance, if you enter line 4 and 6 of Program 2.3(d) and run this, the line will run even though it is logically incorrect in terms of the overall program.

PROGRAM 2.3(d)

```
4 A=1
6 IF A=A THEN 6
```

When this is RUN, line 6 checks for the comparison of 'A' with 'A' and on finding it valid sends the program back to the beginning of line 6. This process then simply carries on continuously, with the program locked into an endless loop. As this is a logical error there is no way that the machine can detect it. From time to time, a program will hang up the computer with an infinite loop, such as line 6 in Program 2.3(d). You've already met the trick that rescues you from this situation, pressing the 'RUN/STOP' key. It rescued us from Program 2.1(b) - again an infinite loop.

To test this, run Program 2.3(d): the screen will stay still as nothing appears to happen. At this stage, press RUN/STOP and the computer should report:

```
BREAK IN 6
READY.
```

At long last we can now return to the project that was started several pages ago, the number guessing game. Armed with several new commands the game can be started in earnest by putting in the basic elements of the program; these being the generation of a random number, the inputting of a guess and the response and the re-directing of the program.

Lines 30 and 50 will handle the generation of a random number and the acceptance of a guess (G) from the player.

PROGRAM 2.4(a)

```
30 RV=1+INT(100*RND(0))
50 INPUT G
```

At this stage, the guess can be compared with the number using the IF...THEN construction. In the earlier example, this was used only to redirect the program by means of a GOTO command. However, the IF...THEN can be followed by any valid BASIC command so, in this case, the statement could say: 'If the guess equals the random number then tell the player that his guess is correct.' Translating that into BASIC yields:

```
IF G=RV THEN PRINT"WELL DONE - GUESS CORRECT."
```

One small tip before adding that line, though! During the development of this game you will probably RUN it hundreds of times. Fun as this may be for the first hundred or so times, it will probably get somewhat boring - eventually. You might like to include a line 35 "PRINT RV" to PRINT out the value of the random number - it makes the game easier too! So far, then, the program reads:

PROGRAM 2.4(b)

```
30 RV=1+INT(100*RND(0))
50 INPUT G
60 IF G=RV THEN PRINT"WELL DONE - GUESS
CORRECT."
```

### A diversion : Comparisons

The mathematical operators which we are about to use in the program, i.e. '=', '<' and '>' are very precise in their operation - just as you'd expect, so this does mean that you, the programmer, must be precise too. Just to emphasize this, let's have a look at a few simple programs that use them. Firstly let's do a simple count in Program 2.4(c) up to 20 using the '<' sign.

PROGRAM 2.4(c)

```
2 LET C=0
4 PRINT C
6 LET C=C+1
8 IF C<20 THEN 4
10 PRINT"FINISHED"
12 STOP
```

Try running this and see what happens. Your 64 should PRINT on the screen for you the numbers from 0 to 19 and then announce that it has "FINISHED". It won't actually get as high as 20 because, although the value of C itself has reached 20, line 8 no longer sends it back to 4 to be printed. Thus, using the '<' operator, it would be necessary to set the loop to 21 to get a PRINT up to 20.

Now modify the program by swapping over lines 4 and 6 as in Program 2.4(d).

PROGRAM 2.4(d)

```
4 LET C=C+1
6 PRINT C
```

Now RUN this and look what the 64 gives you. This time, it will give the number series from 1 to 20 as the variable C was incremented BEFORE being PRINTed. It may appear to be a very trivial point this but it is most important to realize that the variables in a program may change from line to line and no assumptions are valid. Let's see how you get on with the '>' operator:

#### EXERCISE 2.4

Re-write Program 2.4(d) to PRINT the number series 1 to 20 using the '>' operator. Hint! It should contain a line like:  
8 IF C>...THEN...  
A possible answer is given on page 11.2

Another pair of operators in 64 BASIC facilitate control of the kind of loops we're using. These allow us to check whether a variable is greater than OR equal to and whether it is less than or equal to, i.e.

> = means greater than or equal to  
< = means less than or equal to.

To try these out let's re-write 2.4(d) with 2.4(e) so that it counts from 1 to 20. As we wish to go from 1, we must ensure that at the first PRINT statement, the variable is, in fact, 1. To achieve this, we could either set the count variable (in this case 'C') to 1 before we start and then PRINT before incrementing or set it to zero originally and increment before PRINTing. Program 2.4(e) uses the former.



PROGRAM 2.4(e)

```
2 LET C=1
4 PRINT C
6 LET C=C+1
8 IF C<=20 THEN 4
10 PRINT "FINISHED"
12 END
```

Just to see if you can really handle these operators have a go at Exercise 2.5

EXERCISE 2.5

Re-write Program 2.4(e) to count up to 30 using the '>=' operator.  
A possible answer on page 11.2

Now, back to the number guessing game as it was left in Program 2.4(b) - don't forget, though, to remove lines 2 to 12.

At this stage, the program should RUN and, when the correct answer is guessed, give a message and then end. However, if an incorrect guess is entered, the program will simply end with no message. To handle this, two further conditional statements are added at lines 70 and 80 in Program 2.4(f).

PROGRAM 2.4(f)

```
70 IF G>RV THEN PRINT"GUESS TOO LARGE -  
TRY AGAIN."  
80 IF G<RV THEN PRINT"GUESS TOO SMALL -  
TRY AGAIN."
```

## Multi-statement lines

When the current Program is run, it will handle both correct and incorrect answers but only for one input. In order to give a further chance, it clearly has to be re-routed back to the INPUT if the answer was incorrect. This re-routing needs to be done conditionally based on the IF...THEN tests performed in lines 60, 70 and 80. Once again, BASIC comes to the rescue in that a second BASIC statement can be added to the end of an existing line provided that the two parts are separated by a colon. When this is done, the line is referred to as a multi-statement line and the second statement is executed immediately after the first, just as if it were in the next line. Thus, line 60 can be modified to read as in Program 2.4(g).

PROGRAM 2.4(g)

```
60 IF G=RV THEN PRINT "WELL DONE - GUESS  
CORRECT.":STOP
```

This modification will STOP the program after a correct answer and lines 70 and 80 can then be similarly extended, in their particular case to redirect the program, i.e. as in Program 2.4(h).

PROGRAM 2.4(h)

```
70 IF G>RV THEN PRINT"GUESS TOO LARGE -  
TRY AGAIN.": GOTO 50  
80 IF G<RV THEN PRINT"GUESS TOO SMALL -  
TRY AGAIN.": GOTO 50
```

After the modifications in Program 2.4(h) the game will now allow any number of incorrect guesses but comes to a STOP when the correct guess is made. Once it has stopped in this way, you may notice that the message is now different. It should, in fact, read:

```
BREAK IN 60  
Ready
```

It is now reporting a successful RUN and is also stating that it finishes on line 60. This end, however, is rather abrupt and the program would be improved considerably were the player to be given a choice after a correct guess - either to terminate the game or to carry on further. To this end, a further routine is added at the end of the current program which offers the player the opportunity to continue. It takes the form of an INPUT with a message and a conditional test - see PROGRAM 2.4(i). In addition, the STOP will need to be removed from line 60 and the program redirected from here to the INPUT at line 110.

PROGRAM 2.4(i)

```
60 IF G=RV THEN PRINT "WELL DONE - GUESS  
CORRECT.":GOTO 110  
110 INPUT "DO YOU WANT ANOTHER GO (Y/N)";A$  
120 IF A$="Y" THEN 30
```

In line 110, the INPUT is expecting a YES/NO type of answer and the bracketed '(Y/N)' is an additional prompt that gives the player a clear indication of what is expected in the way of inputted data. The use of such prompts makes it possible to test simply following the INPUT. In line 120, it is only necessary to test for the 'Y' - meaning "Yes" - answer if this input is clearly expected. If the input is not a 'Y', then this line is ignored and the program goes on to execute the next line or, if there isn't one, to end the execution.

As the INPUT expected is a string variable, i.e. one with alphanumeric characters (letters), it was necessary to assign an appropriate string variable name - in this case A\$ is used.

As the game stands at the moment, the player can take any amount of goes to guess the number. Just to add a bit more interest the number of attempts will be restricted to six. Ways have already been explored of getting programs to loop around a given number of times and as in Program 2.3, a FOR...NEXT loop can be used. This will be required to repeat the guessing part of the program and start after the random number has been generated - say, at line 40. The loop back - the 'NEXT C(ount)' - will take place after the tests for the guess have been made and before the "another go?" question is asked - say, at 90. These are shown in Program 2.4(j) where the variable 'C' is used in the loop.

PROGRAM 2.4(j)

```
30 RV =1+INT(100*RND(0))
40 FOR C=1 TO 6
50 INPUT G
60 IF G=RV THEN PRINT "WELL DONE - GUESS
CORRECT.": GOTO 110
70 IF G>RV THEN PRINT "GUESS TOO LARGE -
TRY AGAIN.": GOTO 50
80 IF G<RV THEN PRINT "GUESS TOO SMALL -
TRY AGAIN.": GOTO 50
90 NEXT C
110 INPUT "DO YOU WANT ANOTHER GO (Y/N)
";A$
120 IF A$="Y" THEN 30
```

Just to prove this program for yourself, run it through a few times. If you count the incorrect guesses you will find that the loop is not actually activated. To help you to see why, the flow chart for this program is given in Figure 2.3. You can use this to correct Program 2.4(j). Don't worry if you get stuck; the correction is explained below. Incidentally, there are no less than three problems or 'bugs' in the program at present.

# FLOW CHART

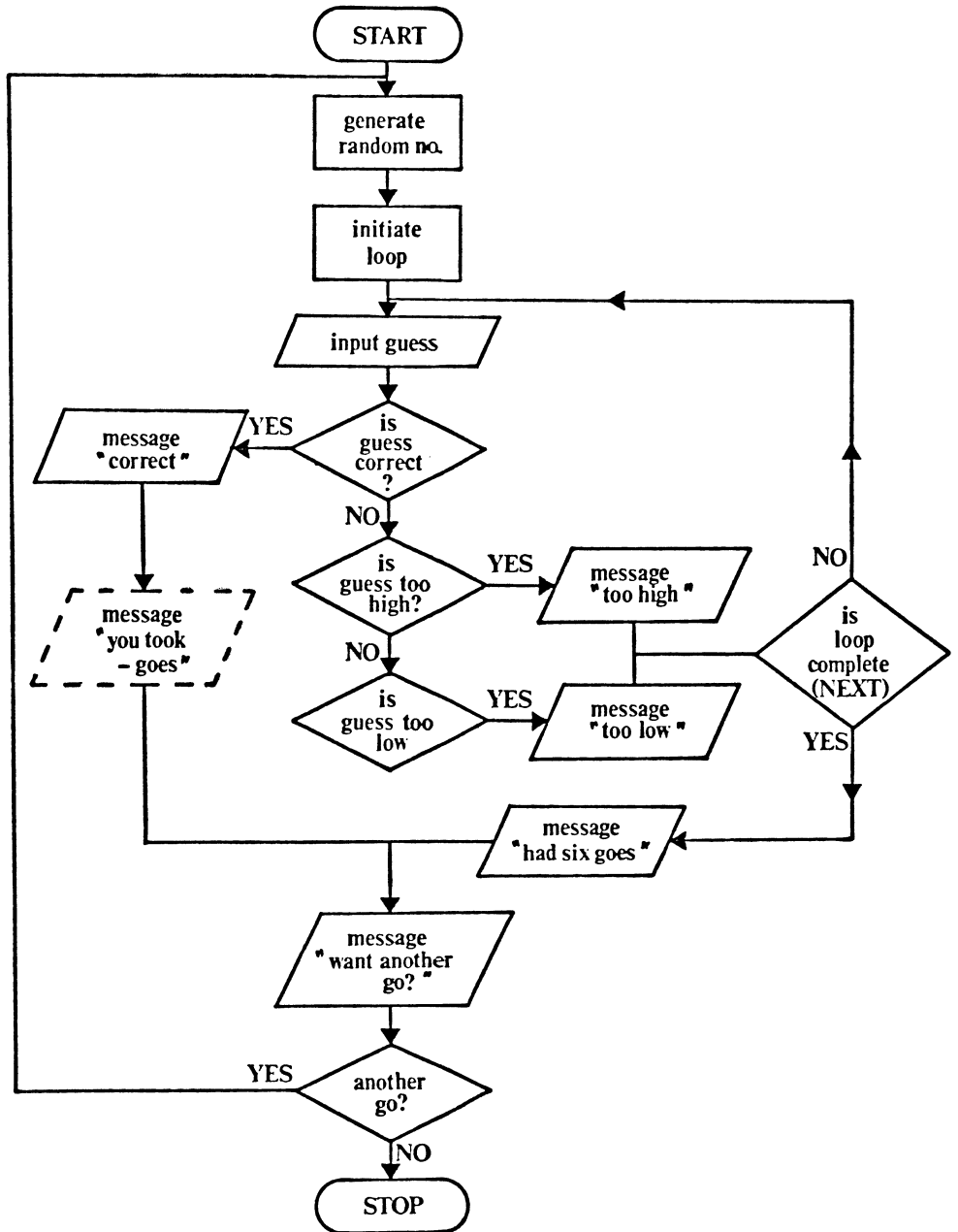


FIGURE 2.3

**EXPLANATION - don't read this until you've had a go!**

Following the program through for a correct answer yields no problems. However, the loop fails to activate after the allowed six attempts. If a 'yes' results from the check 'is guess too high?', a message is outputted and then the flow chart shows a further test for the number of attempts. However the program simply loops back from this point to allow another INPUT. Removal of the 'GOTO 50' on line 70 will allow this line to be followed by 80 and then 90 where 'NEXT C' is met. The 'NEXT' function performs the necessary incrementing of 'C' as well as checking if it has yet reached 6. Once the loop is reached, the NEXT C command allows the program to run through to the next line 110. At this point the player will be asked: "Do you want another go?" Before this he should have been told: "had your six goes matey". To accomplish this a PRINT line could be added at line 100.

To summarize, the three modifications required to Program 2.4(j) are:

- (i) Remove the GOTO 50 on line 70
- (ii) Remove the GOTO 50 on line 80
- (iii) Insert line 100:

```
100 PRINT "SORRY, YOU'VE HAD YOUR SIX GO  
ES."
```

Notice that in the flow chart, the program will only reach the 'is guess too low' decision box if the guess is in fact too low. Strictly speaking therefore, that decision box is unnecessary. A 'NO' decision from the 'is guess too high' box will automatically result in the 'too low' message being output. However, if we look at Program 2.4(k) line 80, we will discover that the IF...THEN decision is necessary after all, as we do not have the GOTO's any more on line 70. If our guess is too large, the program as written will still pass through line 80 even though the flow chart shows it jumping directly to the NEXT on line 90. The program is a little easier to follow if this GOTO is left out and line 80 left as a decision, even though the flow chart is a little easier to follow if the GOTO is left in.

Incorporating the above modifications yields Program 2.4(k).

PROGRAM 2.4(k)

```
30 RV=1+INT(100*RND(0))
40 FOR C=1 TO 6
50 INPUT G
60 IF G=RV THEN PRINT "WELL DONE - GUESS
CORRECT.": GOTO 110
70 IF G>RV THEN PRINT "GUESS TOO LARGE -
TRY AGAIN."
80 IF G<RV THEN PRINT "GUESS TOO SMALL -
TRY AGAIN."
90 NEXT C
100 PRINT "SORRY, YOU'VE HAD YOUR SIX GO
ES."
110 INPUT "DO YOU WANT ANOTHER GO (Y/N)
";A$
120 IF A$="Y" THEN 30
```

All that really remains to be done now is for the program to report back on how many goes the player took to get the right answer. This function is already shown on Figure 2.3, outlined in a dashed box.

#### EXERCISE 2.6

Add a reporting-back function to Program 2.4(k) such that it tells the player how many goes it took to get the correct answer. A possible solution is given on page 11.3

Once Exercise 2.6 is completed, the result should be a functioning number guessing game. In many ways it is a little basic (and BASIC) but from Program 2.5 onwards the rest is up to you. The major improvement that is needed is an introductory message to tell the player what the game is about and what the rules are and a polite 'goodbye' when the player signs off.

## REM

In the listing of this program, provision is made for additions at a later date in lines 10 and 20. Both of these lines start with a REM command, that identifies each line as a REMark line. Once the operating system detects a 'REM', it then ignores anything that follows on this line. By means of REMs, comments can be inserted into programs to enable either the program's author or any other user to follow its logic more readily. A generous sprinkling of REMs is to be recommended to all.

### PROGRAM 2.5

```
10 REM * *Introductory
20 REM * *message
30 RV=1+INT(100*RND(0))
40 FOR C = 1 TO 6
50 INPUT G
60 IF G=RV THEN PRINT"WELL DONE - GUESS
CORRECT.":GOTO 110
70 IF G>RV THEN PRINT "GUESS TOO LARGE -
TRY AGAIN."
80 IF G<RV THEN PRINT "GUESS TOO SMALL -
TRY AGAIN."
90 NEXT C
100 PRINT "SORRY, YOU'VE HAD YOUR SIX GO
ES."
110 INPUT "DO YOU WANT ANOTHER GO (Y/N)
";A$
120 IF A$="Y" THEN 30
```

On the software that comes with the book you will find a program called 'GUESSER'. This is a Computer Aided Learning simulation (CAL) of Program 2.5 which should help to clear up any remaining problems that you might have with the program. As you go through the 'GUESSER', you will notice that the program has been simplified somewhat so that the listing can be displayed the whole time. By now, though, you should be able to follow this quite easily!



## PART 2

### Storing a program

Once a program of any length has been developed, it becomes a chore to keep typing it into the computer. It can, of course, be saved onto a storage device and then re-loaded back into the memory when you need it. One common means of storage on the Commodore 64 allows your program to be stored on diskette. This form of storage is known as 'non-volatile' as it doesn't need any power to keep the program stored. The memory in the computer is 'volatile' as, once the machine is turned off, all the contents of its memory are lost. Commodore 64 BASIC contains two commands to facilitate this mode of storage. Such commands form part of the machine's operating system - those built-in programs that make the whole computer work.

The BASIC commands for storage are:

#### SAVE

SAVEing programs on disk should present little problem on the Commodore 64. Programs must be SAVED with a name. To SAVE a program with the name 'DR WATSON', it's only necessary to place a formatted disk into the disk drive and then type in:

```
SAVE "DR WATSON",8<RETURN>
```

The Commodore 64 will then tell you

```
SAVING DR WATSON
```

The diskette drive light will go on, then the computer will report back with

READY.

It is a good idea to check that the program is SAVED satisfactorily. Commodore's command to do this is:

#### VERIFY

The computer will compare the program in memory with the one it finds on the diskette with the same name. Use the form:

VERIFY "DR WATSON",8

Once a satisfactory check has been made, the computer will report

VERIFYING DR WATSON  
OK

READY

If the check is not satisfactory the computer will report

VERIFYING DR WATSON  
?VERIFY ERROR

READY.

If this 'VERIFY ERROR' is obtained then you must try another 'VERIFY'. If this results in a second error, then you'd be safest to try 'SAVEing' the program again and then repeat the 'VERIFY'.

## PART THREE

### Comparing Numbers

Other techniques are allowed in BASIC when comparing numbers, one very useful one allowing two comparisons to be made in one statement. Using this, Program 2.6 will be developed from Program 2.5 to produce a game that asks the player to guess two numbers. In order to simplify this, the equality check and 'larger than' and 'smaller than' lines should be removed, i.e. lines 60, 70 and 80.

Next, a second random number must be introduced so these will simply be called 'R1' and 'R2' i.e. as in Program 2.6(a), line 30. As the player is now to be asked to guess two numbers, it would also be easier if the range of possibilities of each number were to be reduced, say to 4.

PROGRAM 2.6(a)

```
30 R1 =1+INT(4*RND(1)):R2=1+INT(4*RND(1))
```

In this particular game, two guesses will be required and, as with the R's these can be called 'G1' and 'G2' as in line 50 of Program 2.6(b).

PROGRAM 2.6(b)

```
50 INPUT G1:INPUT G2
```

OR

With two guesses and the random numbers, the comparison process is obviously much more complex than in the earlier game. However, the BASIC command 'OR' eases things somewhat. It enables one, for instance, to compare a single guess with both R1 and R2. This check may say 'if G1 does not equal R1 OR if G1 does not equal R2 then PRINT 'one guess wrong!'. Translating this into BASIC yields.

```
IF G1<>R1 OR G1<>R2 THEN PRINT"ONE GUESS  
WRONG"
```

A similar line will then check for G2. Lines 70 and 80 show this in action in Program 2.6(c). (<> means 'NOT equal to').

PROGRAM 2.6(c)

```
30 R1=1+INT(4*RND(1)):R2=1+INT(4*RND(1))
50 INPUT G1:INPUT G2
70 IF G1<>R1 OR G2<>R2 THEN PRINT "ONE W
RONG"
80 IF G2<>R1 OR G2<>R2 THEN PRINT "ONE W
RONG"
```

AND

So far the program hasn't even looked for correct answers and this is done in lines 60 and 65 of Program 2.6(d). This line is able to check for both answers being correct by means of the BASIC command 'AND' which enables one to say the BASIC equivalent of:

"If this is correct AND if that is correct, then do something"

Following this line, the program can be looped to a further routine that asks if the player wants another go. The final step is to replace the loop that allows only six goes, lines 40 and 90.

PROGRAM 2.6(d)

```
30 R1=1+INT(4*RND(1)):R2=1+INT(4*RND(1))
40 FOR C=1 TO 6
50 INPUT G1:INPUT G2
60 IF G1=R1 AND G2=R2 THEN PRINT"WELL DO
NE" :GOTO 110
65 IF G1=R2 AND G2=R1 THEN PRINT"WELL DO
NE" :GOTO 110
70 IF G1<>R1 OR G1<>R2 THEN PRINT"ONE WR
ONG"
80 IF G2<>R1 OR G2<>R2 THEN PRINT"ONE WR
ONG"
90 NEXT C
100 PRINT "SORRY, YOU'VE HAD YOUR SIX GO
ES"
110 INPUT "DO YOU WANT ANOTHER GO(Y/N)";
A$
120 IF A$="Y" THEN GO TO 30
```


Using the ideas in this chapter, you can now create a whole range of logic games leading to very complex 'mastermind' - type games. Once again the door is open and you are invited to enter.

## CURSOR CONTROLS

One way of improving the number guessing game is by means of cursor controls. For instance, it would be useful if at the start of every game, the screen was cleared of any previous writing. In direct mode this is achieved by holding down 'SHIFT' and pressing the 'CLR-HOME' key - just try it to make sure! You can also get a program to do this while it's running.


Try this out. Type in :

```
PRINT "<CLR>"
```

Note that <CLR> will appear on the screen as a reverse heart i.e. 

When you press the RETURN key, the screen will clear and the cursor move to the top left hand corner of the screen. A further command 'HOME' moves the cursor to the screen's top left hand corner without first clearing the screen. Try this out with:

```
PRINT "<HOME>"
```

Note that <HOME> will appear on the screen as a reverse 'S' i.e. 

Note that the symbol is obtained by pressing the HOME key.

Thus, we have seen how to obtain the clear home character, <HOME> and clear-screen character <CLR>. These can be used in the same way as any other characters on the keyboard. They do present somewhat of a problem in listings so, in this book, they have been listed in a cleaner way as described in Appendix 3. Using this convention the HOME cursor symbol is replaced by <HOME> and the clear-screen by <CLR>.

All these control characters are used as strings and can be given string names.

Thus.....

```
A$="<CLR>" is OK
```

and .....

```
PRINT "<HOME>LOVE" combines the control character with a string.
```

Now that we have seen how to use these functions in a program, let's add them to our number guessing game!




If we add line 35 to Program 2.6(d) then every time we run the program, the screen will clear and printing will start in the top left hand corner.

PROGRAM 2.7(a)

```
35 PRINT"<CLR>NUMBER GAME"
```

Now that we can clear the screen from within a program, let's investigate cursor movements.

Like the <CLR> function, cursor keys have special print characters and these are used in the same way. To try this, type in 'PRINT"' and then press the cursor up/down key to obtain the 'down cursor' symbol and on the screen will appear '␣'. So the 'CRSR↑↓' key produces the character '␣'. Now hold down the SHIFT key and press 'CRSR↑↓' and we have the cursor up character. After performing the same operation for the 'CRSR↔' key, we will obtain the following results.

Moves Cursor Up	SHIFT		OR <UCRSR>
Moves Cursor Down			OR <DCRSR>
Moves Cursor Left	SHIFT		OR <LCRSR>
Moves Cursor Right			OR <RCRSR>
Moves Cursor To Top Of Screen			<HOME>
Clears Screen	SHIFT		OR <CLR>

Let's have a go at using these cursor controls in a program with Program 2.7(b).

PROGRAM 2.7(b)

```
110 INPUT "<2DCRSR><4RCRSR> DO YOU WANT  
ANOTHER GO (Y/N)?" ; A$
```

In those two examples (Program 2.7(a) and (b)) we have changed only two of the many PRINT commands that this program uses. On the disk that accompanies this book, a few other changes have been made to show what can be done. However, you should feel free and make any other changes in display that you want to, after all its your game now! Good luck!

## CHAPTER

# 3

### An Introduction to Honey.Aid

**I**n the diskette that comes with the book is a 'utility' program called "Honey.Aid". This is designed to take full advantage of the capabilities of your Commodore 64 by adding new commands to its BASIC. The Honey.Aid program tucks itself away at the top of computer's memory (wherever that is when Honey.Aid is loaded) and protects itself from being overwritten by normal BASIC programs. Typing in NEW will not remove Honey.Aid from your computer. To remove it you can either switch the machine off (which is rather like using a sledge-hammer) or make use of Honey.Aid's built in switch off command. It is called 'KILL' and will switch off all the Honey.Aid commands: but don't worry, when you type KILL it tells you how to restart Honey.Aid, something switching off does not do.

Find the program on your disk and load it with `LOAD"HONEY.AID",8`. When it has loaded, LIST the program. You will find a small BASIC program which is mainly concerned with displaying the Honeyfold logo and playing the introductory jingle. What you can't see is the machine code program which is Honey.Aid itself, and the other machine code program which locates Honey.Aid at the top of memory.

Now, turn the volume on your television up and type RUN to get Honey.Aid working. At this stage, you will see that Honey.Aid has set your machine into lower-case mode. If you aren't too keen on this hold down the Commodore key, (bottom left-hand corner of the keyboard) and press SHIFT. This should turn you back into capitals mode ('caps'). However, all the programs in this chapter are written in lower-case so it might be an idea to leave the machine set this way. In operation it makes no real difference one way or the other how the machine is set - all the keys work the same way. Many professional programers prefer to work in lower-case as this allows them to introduce capitals into the text more readily and it helps to make screen displays look more interesting. Whether you're in caps or lower-case, try the following:





you are using a monitor, rather than a TV, then all the 'stars' should be white. If they are not then your monitor is not working as well as it should - perhaps you should have a word with your dealer!

By the time you have finished reading the explanation above, your program should have finished plotting the star field. Good, but how do you get back to the normal screen when you have finished admiring your masterpiece? The answer is hit (not too hard) any key. <SPACE> or <RETURN> are good. Try it!

You will get quite an interesting pattern if you try this instead:

PROGRAM 3.1(b)

```
10 hires 1,0,1
20 for z=1 TO 5000
30 plot rnd(0)*319,rnd(0)*199,1
40 next z
```

This is because RND(0) generates its 'random' numbers from your computer's internal clock. If you use RND(1) instead, a more random pattern will be produced.

#### LINE and NRM

Another useful graphics command is LINE; try this:

PROGRAM 3.2

```
10 hires 1,0,1
20 line 10,100,200,50,1
```

That will draw a line. In general terms, a LINE statement is of the form:

```
starting co-ordinates of line
line [X1,Y1,X2,Y2]MODE
end co-ordinates of line
```

Where X1 and Y1 define the starting point of the line, and X2 and Y2 define the end point, i.e:

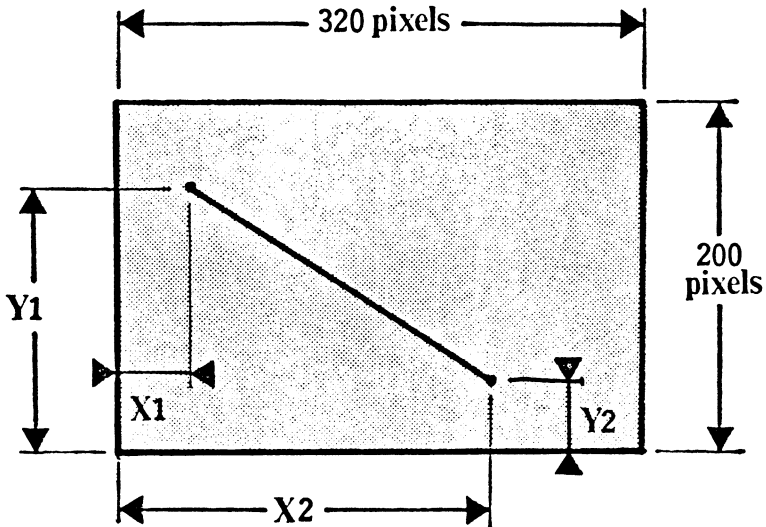


FIGURE 3.1

- \* The X's must be between 0 and 319. and the Y's between 0 and 199.
- \* Mode 0 converts all points along an existing line into the background color.
- \* Mode 1 produces a line in the foreground color, ie. the last color specified in the HIRES command.
- \* Mode 2 will cause all points along a line being plotted to be plotted in the appropriate color to that of the screen at those points- just like mode 2 of the hires command.

Try this short program.

PROGRAM 3.3

```
10 hires 6,7,1
20 for z=0 to 60
30 line rnd(0)*319,rnd(0)*199,rnd(0)*319,
rnd(0)*199,1
40 next z
```

Try something more elaborate with the following program:

#### PROGRAM 3.4

```
10 for c=0 TO 15
20 if c=1 then next c
30 hires 1,c,1
40 for y1=0 TO 199 step 199
50 for y2=199 TO 0 step -1
60 line 0,y1,319,y2,2
70 line 0,y2,319,y1,2
80 next y2:next y1
90 for w=0 TO 10000:next w
100 next c
```

When (if) you get fed up with it, press <RUN/STOP>, type: NRM <RETURN>.

The NoRMal screen will only appear after you've done all this, or on the first key pressed after the program has finished. Try changing the STEP size on line 50: values between -2 and about -10 are quite interesting. Also try changing the first number in the HIRES command on line 30.

#### An Etcha-Sketcha Game

In order to demonstrate the 'PLOT' command in action, let's develop an 'etcha-sketcha' type program. The basic idea of this is that, following an input, a character will be printed on the screen, i.e.:

#### PROGRAM 3.5(a)

```
10 print "<CLR>"
20 input a$
30 print "*"
40 goto 20
```

(To stop this program, keep RAPIDLY alternately pressing RUN/STOP and RETURN continuously until it stops!)

What Program 3.5(a) does is to print an asterisk after every INPUT. This is not the smoothest bit of programming around, as the input command itself uses two lines of the screen, prints a '?' and sits around waiting for the user to press RETURN. There is, however, a method of inputting data that does not produce a question mark or even require one to press RETURN. This is the 'GET' command.

## GET

What the 'GET' command does, as the name implies, is to 'GET' a character from the keyboard. The command takes the form of:

```
10 get a$
```

If no key is pressed, then a\$ will have the value of "", a null string. Thus, to make any sense out of the 'GET' command, we have to check whether a key has been pressed. In Program 3.5(b), a GET command is used to test the value of a\$. If a\$ is "", then no key has been pressed, so the program loops back to line 10 to accept another input. This program will loop back continuously to line 10 until a key is pressed. When this is done, the program stops.

### PROGRAM 3.5(b)

```
10 get a$
20 if a$="" then 10
```

Program 3.5(c) shows a further development where the 'GET' is used to keep a message on the screen while the user reads it. The program will then continue after a key has been pressed.

### PROGRAM 3.5(c)

```
5 print "press any key"
10 get a$
20 if a$="" then 10
```

So, using the GET command, the simple etcha-sketcha program becomes:

### PROGRAM 3.5(d)

```
10 print "<CLR>"
20 get a$
30 if a$="" then 20
40 print "*"
50 goto 20
```

Now, when run, we have an asterisk printed every time a key is pressed, but so far this character only moves down the screen.

To remedy this, we will use four control keys: 4, 5, 6 and 7. Where '4' will move the cursor left and '7' will move the cursor right, '5' will move the cursor down and '6' will move the cursor up.

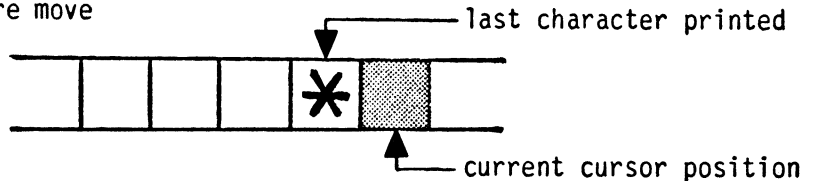
- \* 4 moves the cursor left
- \* 7 moves the cursor right
- \* 5 moves the cursor down
- \* 6 moves the cursor up

Lines 40 to 70 of program 3.5(e) carry out the processes required when moving the cursor about the screen, line 50 being the easiest case. When a '7' is detected, no great problem is presented as the program simply prints an asterisk.

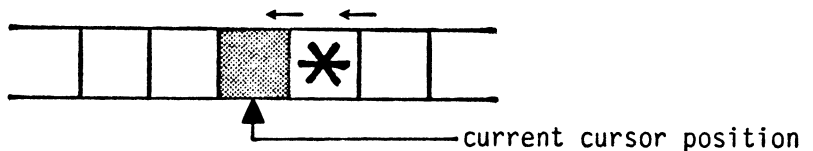
To move in other directions is more problematic, as the cursor is always positioned one place to the right of the last character printed following a 'PRINT' sequence.

Thus, to move a character one place left the process is:

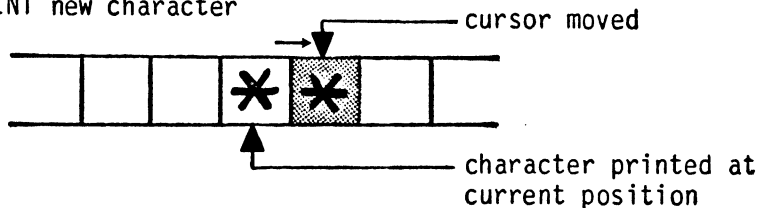
(i) before move



(ii) move cursor two places left



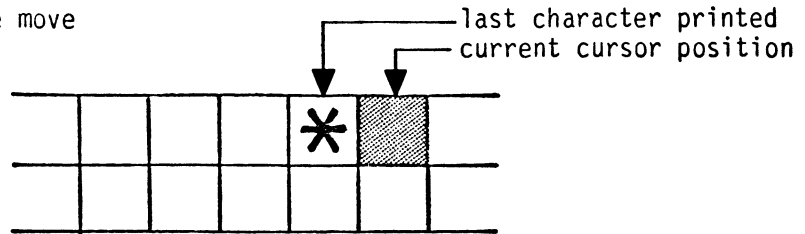
(iii) PRINT new character



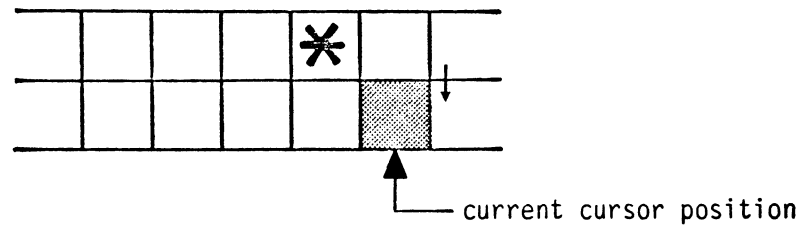
(iv) cursor now positioned for next 'PRINT' operation

In just the same way when moving the cursor up and down, the additional cursor-back control is required ie:

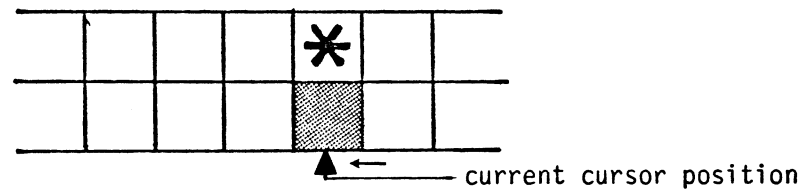
(i) before move



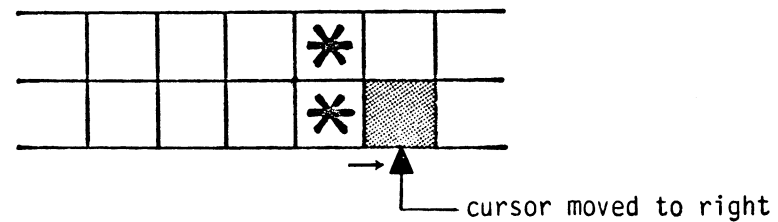
(ii) move cursor one place down



(iii) move cursor one place left



(iv) PRINT new character



(v) cursor now positioned for the next 'PRINT'

Thus, adding these control lines yields Program 3.5(e).

PROGRAM 3.5(e)

```
40 if a$ ="4" then print"<2LCRSR>*";
50 if a$ ="7" then print "*";
60 if a$ ="5" then print"<DCRSR><LCRSR>*";
70 if a$ ="6" then print"<UCRSR><LCRSR>*";
80 goto 20
```

Now, when run, the program will draw lines of asterisks in any direction we choose, giving free reign to your artistic endeavors!

#### EXERCISE 3.1

Write a small program that will print on the screen which key the user should use to move the asterisks. This piece of program should be placed in the lines 1 to 9 and end with a 'PRESS ANY KEY' type prompt serviced by a GET statement. A possible answer is given on page 11.3

Although the present version of etcha-sketcha runs satisfactorily, it still looks somewhat clumsy with its asterisks all over the screen. However, we can improve the display considerably by means of the Honey.Aid command "PLOT".

First of all, we need to set up the high resolution screen and accept a user's input, using GET.

PROGRAM 3.5(f)

```
10 hires 6,7,1
20 get a$:if a$="" then 20
```

Next, instead of using cursor controls and printed characters such as asterisks, we will merely have to increment, or decrement, the co-ordinates for the PLOT command:

#### PROGRAM 3.5(g)

```
30 if a$ ="4" then x=x+1:goto 80
40 if a$ ="5" then y=y+1:goto 100
50 if a$ ="6" then y=y-1:goto 90
60 if a$ ="7" then x=x-1:goto 70
```

Its no good trying to PLOT co-ordinates outside the screen area - indeed, Honey.Aid will detect any attempt to do so and report an error. We need to make sure therefore that we only attempt to PLOT within the screen area displayed; ie. X must be controlled to remain between 0 and 319 and Y between 0 and 199 ie:

#### PROGRAM 3.5(h)

```
70 if x<0 then x=0:goto 110
80 if x>319 then x=319:goto 110
90 if y<0 then y=0:goto 110
100 if y>199 then y=199
110 plot x,y,1
120 goto 20
```

You should now have a working instant Honey.Aid type etcha-sketcha program!

Have fun!

#### SOUND

Now to look at some of Honey.Aid's music commands (have the sound on your TV turned up). Try typing in:

```
sound 1,4,1,6 <RETURN>
```

That note is Middle C.

The first number (1) is the 'voice' number, and may be 1, 2, or 3. Any voice may be used to produce random or white noise, but more of that in chapter 9. For now voice 3 is used. Try:

```
sound 3,5,1,6
```

The second number (5) is the octave, and may be from 0 to 7.



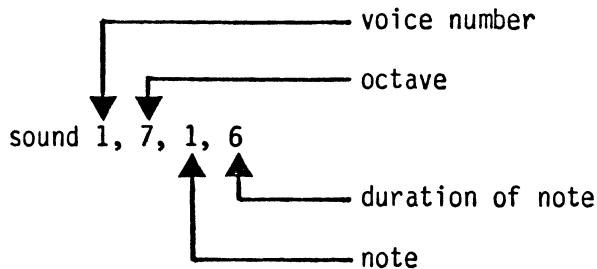
Try :

sound 1,2,1,6

and

sound 1,7,1,6

The third number (1) is the note, and may be from 0 (silent) to 12 (high note). The last number is the length or duration of the note and may be from 1 to 8. Summarising this:



Then try this short program to run through all the notes in one octave.

PROGRAM 3.6

```
10 for x=1 to 12
20 sound 1,3,x,6
30 print x;
40 next x
```

Then try this program to run through all the notes in eight octaves.

PROGRAM 3.7

```
10 for y=0 to 7
20 for x=1 to 12
30 sound 1,y,x,5
40 next x
50 next y
60 end
```

If you change line 30 to SOUND 3,Y,X,5, you will get a sound like a faulty rocket blasting off!

Try this: (First type NEW) this is a 'piano' program!

PROGRAM 3.8

```
10 n$="zxsdcvghbnjm"  
20 get x$:if x$="" then 20  
30 for i=1 to 12  
40 if mid$(n$,i,1)=x$ then nn=i:i=13  
50 next i  
60 sound 1,4,nn,5  
70 goto 20
```

Now the keys stored in N\$ are like piano keys (except that you can only play one note at a time). Any other key will give the last note pressed. Try this sequence of letters:

Z C B B N N B C Z C B B V C X B C Z V C X X Z Z C B C V C Z

To stop this program, press RUN/STOP.

ENVELOPE

Now try adding this line to the program:

```
5 envelope 1,2,9,3,0
```

and then try executing it. Honey.Aid's ENVELOPE command alters the tone quality of the note generated. Type:

```
5 envelope 1,5,5,3,10
```

and try running that.

The first number (1) is the voice number, and, as for the SOUND command above, may be 1,2 or 3 because your C-64 has three "voices". For the purpose of computer music, a musical sound is said to comprise four phases : the attack (A), decay (D), sustain (S) and release (R) phases - ADSR. This is shown on the graph below:

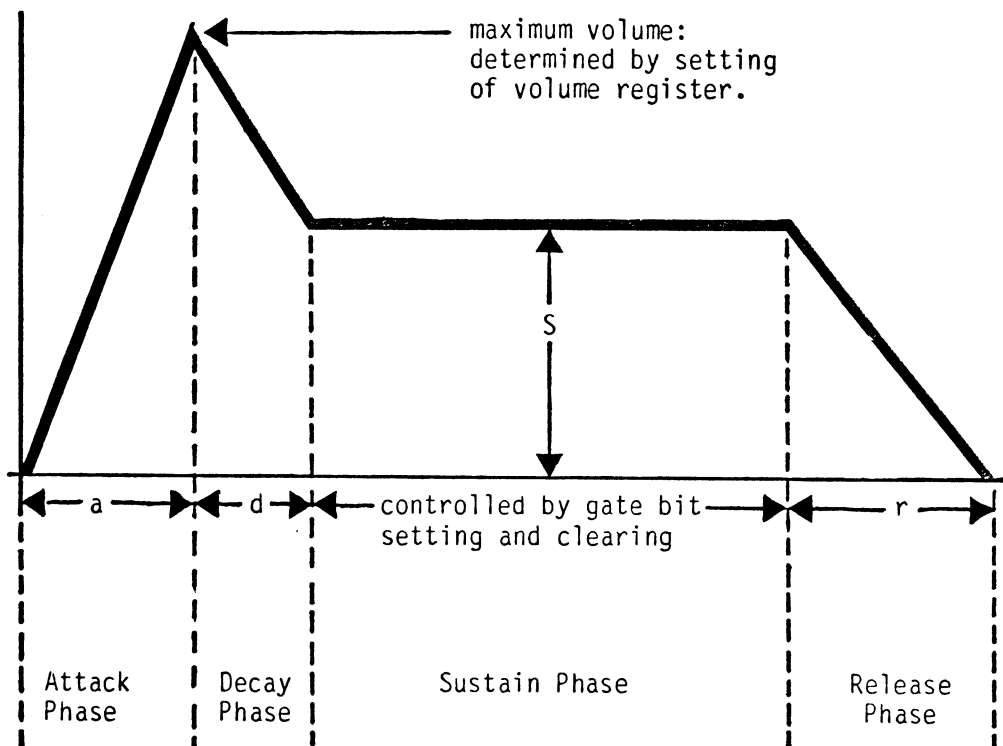


FIGURE 3.2

Different musical instruments produce notes with different "envelopes" - that is (partly) why they sound different! You will find this discussed more thoroughly in a later chapter.

The second number in the envelope defines the length of the attack phase - how long the note or sound will take to build to its peak volume. The attack number may be from 0 to 15. The smaller the number, the sharper (shorter) the attack.

Next comes the decay phase - after the note has reached its peak loudness, it decays for a period of time determined by the third number in the envelope command - again from 0 to 15.

The third number, which can be from 0 to 15 as well, but, unlike the other three numbers (which define the length of their particular phase), the sustain number defines the relative loudness of the phase, in relation to the peak volume. The length of the phase is controlled by the gate bit (see Chapter 9).

Following this is the release phase. Here the note fades out - the length of time this takes is determined by the last numbers in the envelope command, which may be from 0 to 15.

Summarizing that:

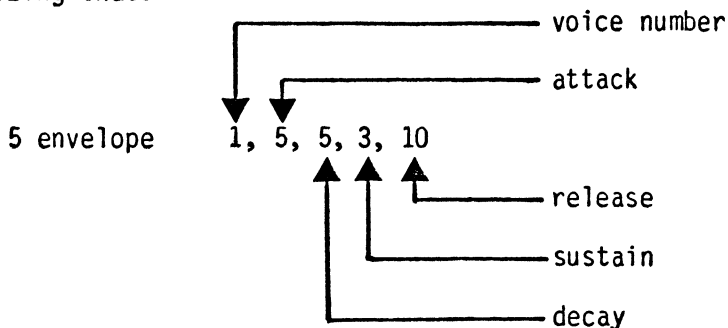


FIGURE 3.3

Try a few different envelopes in line 5 of the above program and see (or hear!) what happens.

### PLAY

This command allows your computer to remember a tune. The notes to be PLAYed are stored in a string variable.

Try running this program:

#### PROGRAM 3.9

```
10 let a$="<blk>c4<f5>C<f7>d g f g a g f g d g <f5>"
20 play a$
```

(To get the <blk>, hold down CTRL and press 1. <f5> etc. can be obtained simply by pressing one of the four function keys on the right of the C-64).

Note that the envelope from the earlier program is still operating (unless you've switched off since then) - certain Honey.Aid commands have been designed to stay as set until changed. This enables you to set new default values that stay set until you wish to change them. The ENVELOPE command is one of these, SOUND is another and so is PLAY.

The 'PLAY' command is defined as follows: each note in a\$ is defined by a string of up to four symbols:

The first of the four symbols represents the voice. The C-64 has three voices and you may choose voice 1 by using <CTRL 1>(or <BLK>), voice 2 by using <CTRL 2> (or <WHT>), and <CTRL 3> (or <RED>) for voice 3.

The second of the symbols represents the note and you can choose c d e f g a b for the naturals or C D F G A for the sharps. Flats are obtained by using the tonally similar (i.e. previous) sharp, i.e. for e flat use D#. An r gives a rest but you can use a hyphen ("-") instead.

The third of the symbols is octave number; you can specify 0 to 7.

Fourthly and finally comes the note duration for which we use the function keys <f1> breve, <f3> semi-breve, <f5> minim, <f7> crotchet, <f2> quaver, <f4> semi-quaver, <f6> demi-semi-quaver and <f8> a hemi-demi-semi-quaver. Thus, to specify a middle C with voice 1, we would place <BLK>c4<f5> in the string, which is exactly what we find in line 10 of Program 3.9.

Summarizing all that:

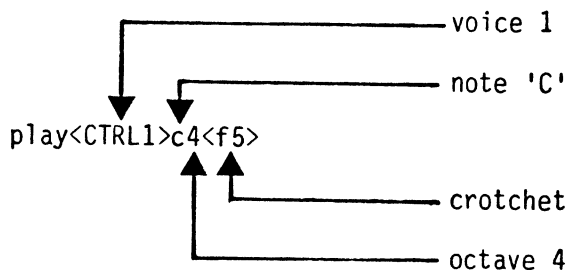


FIGURE 3.4

We did say that a note is defined by a string of up to four characters. It can be as little as one, however.

The trick is that if a note has the same voice as the previous note, then there is no need to put the voice characters in again. The same applies to the note, the octave or the duration. Looking at line 10 of Program 3.9 then you can see that the second note is another middle C using voice 1 but this time it is a quaver, the third is a D and the voice must be 1, the octave 4 and the duration a quaver and so on. In technical terms we say that Honey.Aid defaults these characters to their earlier values.

## TEMPO

If you now add this to Program 3.9:

```
2 TEMPO 300
```

the little jingle will be played more quickly.

The 'TEMPO' command controls the speed with which a sound, note, or a tune is played. A low number means play slowly, while a high number means play more quickly. The range of numbers for tempo is from 1 upwards. When Honey.Aid is first loaded, tempo is set at 120 - ie. 120 crotchets per minute.

Once Honey.Aid is loaded into your C-64 any program you write can incorporate Honey.Aid commands. For instance, the last few lines of the etcha-sketcha program can be improved further by including a beep every time an attempt is made to draw outside the screen area:

### PROGRAM 3.10

```
70 if x<0 then x=0 sound 1,4,4,6:goto 110
80 if x>319 then x=319:sound 1,5,4,6:goto 110
90 if y<0 then y=0:sound 1,2,4,6:goto 110
100 if y>199 then y=199:sound 1,7,4,6
```

Honey.Aid has many other commands, and they are listed in Appendix 2. Many will be introduced in other chapters as and when required. As well as the commands, that make it easier to incorporate the C-64's features into your programs, Honey.Aid incorporates a number of 'utility' commands that make your programming easier. One important feature of these utility commands is that any program written using these will operate on ANY C-64, not just one loaded with Honey.Aid.

## AUTO

Auto is a command that carries out line numbering automatically. When typing in a program it can get a bit tedious putting line numbers at the beginning of each line, so Honey.Aid will do it for you. If you type 'auto <RETURN>' and then type in a line number. say 10, from then on a new line number (20, 30...) will be provided every time you press <RETURN>.

Try typing:

```
auto <RETURN>
10 print "hello" <RETURN>
```

You will see a '20' appear AUTOMATICALLY. If you press RETURN instead of typing something onto line 20, AUTO will be turned off, i.e. typing in an empty line switches off the AUTO command.

The AUTO command defaults to a line spacing of 10 - i.e. if you start at line 50, the next line AUTO provides will be 60, etc. If you want a different line spacing then you just type it in:

```
auto 100 <RETURN>
```

will give spacings of 100. Note that line spacings must be positive integers.

## NUMBER

NUMBER is a command that will operate upon a program that you have written and renumber all the line numbers in equal increments, as well as adjusting all the GOTO's etc. so that they still GOTO the right place! Whereas AUTO provides each line of a program with a line number while you type it in, NUMBER is used to tidy up line numbers, for example, after you've inserted or deleted a few lines. Suppose you've been editing a program and have ended up with the following:

### PROGRAM 3.11

```
5 print "hello"
10 input a$
20 if a$<> "no" goto 90
57 print a$
90 stop
```

If you want this to start at line number 100 and increase by 10 for each line, you would type:

```
NUMBER 100,10 <RETURN>
```

and your program would end up as:

```
100 print "hello"  
110 input a$  
120 if a$<> "no" goto 140  
130 print a$  
140 stop
```

Note that the GOTO on line 120 has been corrected by the NUMBER command. NUMBER will correct all GOTO's, GOSUB's etc.

## OLD

Another useful, at times life-saving, Honey.Aid command is 'OLD'. If you have typed 'NEW' to wipe out a program and then changed your mind, you can get the program back by typing 'OLD' - provided you haven't begun typing in a new program or assigned any variables.

## FIND

Sometimes, during programing, you will want to find out where a particular command or string has been used. For instance, you may be sure that you've used GOTO 130 somewhere, but you just can't find it! Honey.Aid's 'FIND' comes to the rescue! Typing:-

```
find.goto 130.
```

will cause your computer to list all lines in which the string 'goto 130' occurs. The full stops or periods (.) are called 'delimiters' and can be almost any symbol you choose, as long as they don't appear in the string being searched for. Some examples of FIND statements are:

```
find$x=5$  
findx123.64x  
find"b6=2"
```

findaharrya ← WRONG! 'a' appears in the word harry and so cannot be used as a delimiter when searching for occurrences of harry.



Actually, the use of quotes (") as a delimiter has a special function: if you want to find something in a program that appears in quotes then you MUST put it in quotes in the 'FIND' statement. Thus:

```
find.print.
```

would find PRINT in

```
10 print "start"  
or  
40 printer ← the value of er
```

but not in

```
50 input "printer or screen";a$
```

However, if we use quotes as delimiters, i.e.

```
find "print"
```

then we will NOT find PRINT in

```
10 print "start"  
or  
40 printer ← the value of er
```

but we WILL find PRINT in

```
50 input "printer or screen";a$
```

Another thing we can do with the FIND command is tell it on which line to start FINDing and where to stop, just like the LIST command; thus:

```
find.halo.,100-200
```

will search from line 100 to line 200 inclusive for the string 'halo'.

```
find.x.,150-
```

will search from line 150 to the end of the program,

```
find.emperor.,-400
```

will search up to and including line 400.

## CHANGE

Instead of just finding a string, you may want to change it to something else. For example:

```
change.print.input.
```

will change all occurrences of the string of letters p,r,i,n and t into input.

```
change.print x.print a2.,100-
```

will change all occurrences of print x into print A2, from line 100 to the end of the program, but the comments relating to the use of quotes as delimiters in the description of FIND above apply to CHANGE as well. Some examples of CHANGE commands are

```
change$hello$goodbye$,200-400  
changea=7a=9a,-950
```

Change is a really useful command but very faithful! It will do just what you ask it to so beware - the string that you wish to change may occur in far more places than you thought! Before doing a 'change' on a long program it is a good idea to do a 'find'. That way you can see just what you will be changing.

For more information these and other Honey.Aid commands see Appendix Two.

## CHAPTER

# 4

### Putting in structure: The Hangman Game

**I**n chapter 2, a number guessing game was developed that utilized a random number generated by the Commodore 64. In this chapter a similar game will be written, but this time using words, i.e. a variation on the popular hangman game. Thus, instead of asking the player to guess a number, (s)he will be required to guess a word letter-by-letter. First of all though, this chapter will investigate ways of storing these words and then of delivering them one by one when required.

When dealing with random numbers, the C-64 can generate an endless supply to order, as it has built into its ROM (Read Only Memory) a program which can produce these as rapidly as they can be consumed. Of course, when dealing with words, the same thing is not possible. All the words to be used must be stored in the program somewhere, and thus must be defined by the programmer. A common way of storing such data is in strings, and the program could contain such statements as:

```
LET A$="COMMODORE 64"  
LET B$="KEYBOARD"  
LET C$="SCREEN"
```

### READ....DATA

This, however, would be an extremely tedious way of doing the job and BASIC provides an alternative method. It utilizes two commands, READ and DATA, the first one telling the machine to READ one piece of data and the second telling it where to find the data. The DATA statement is the one piece of program that is footloose - it can go anywhere in the program. It is, however, usual to put it right at the end so that it is out of the way. Program 4.1 illustrates this, with line 1030 reading one piece of DATA, which is PRINTed out in line 1035, the DATA originally having been stored at line 9000.

#### PROGRAM 4.1

```
1030 READ A$
1035 PRINT A$
9000 DATA "ONE", "TWO", "THREE"
```

When run, this program will retrieve only one piece of DATA, e.g. 'ONE', and then PRINT this onto the screen. String data does not have to be enclosed in quotes in a DATA statement. If the string contains a comma, as in JONES,ED then it is wise to enclose it in quotes. For example, try 9000 DATA "JONES,ED",ONE,TWO in program 4.1. Program 4.2 shows a very similar program where numbers are stored rather than words.

#### PROGRAM 4.2

```
1030 READ A
1035 PRINT A
9000 DATA 1,2,3
```

In this program, the changes from Program 4.1 are really only what one would expect: the numeric variable name 'A' replaces the 'A\$', and the data is not enclosed in quotes as it is numeric.

READ statements may be as simple or as elaborate as the program demands and a number of variables could be READ in by one line of program; e.g. READ A\$,A,B\$. However, when doing this the greatest care must be taken to ensure that when the READ statement tries to READ a number it finds a number and not a string. If a mismatch in types occurs, the machine will report an error. As this is straightforward, try a little exercise!

#### EXERCISE 4.1

Write a program to READ the numbers 1 to 4 from DATA statements in both numbers and letters, and to display them on the screen like so:

```
1 ONE
2 TWO
3 THREE
4 FOUR
```

A possible solution is given on page 11.3.

Having found a way of storing and retrieving data, some way has to be found of making a random selection from it. By using a FOR...NEXT loop to READ a particular number of items from the DATA,

one particular piece may be retrieved, as shown in Program 4.3. In this program, the loop is executed three times, and thus the third piece of data is retrieved.

#### PROGRAM 4.3

```
1020 FOR X=1 TO 3
1030 READ A$
1040 NEXT X
1050 PRINT A$
9000 DATA ONE,TWO,THREE,FOUR
```

In fact, three pieces of DATA will have been retrieved but only the string 'THREE' is printed. On the first pass through the loop, the value of A\$ would have been 'ONE' but during the second pass this would be overwritten by 'TWO' and then finally by 'THREE'. It was this string 'THREE' that was stored in A\$ at the time that line 1050 commanded it to display the string on the screen.

On each pass through the loop, line 1030 READs the next item in the DATA statement. It knows which item is next as, each time a READ is performed, a pointer is moved along one item to point to the next one to be read. This can give rise to problems if an attempt is made to read more DATA than exists. For instance, if Program 4.3 is put inside a second loop, i.e. 'nested', it will cease to run on the second pass after all four pieces of DATA have been READ. Fortunately, BASIC has a statement to use DATA more than once in a program.

#### RESTORE

This has the effect of moving the pointer back to the beginning of the DATA. This is demonstrated in Program 4.3(a) below, where the loop in Program 4.3 is nested or buried inside a second loop that runs it three times. As the nesting of loops is, in itself, a powerful feature of BASIC, PRINT statements have been inserted into lines 1000 and 1020. If you feel uncertain about the idea of nesting, just examine the screen contents after you've run Program 4.3(a). It will show you how the program handles a nested loop. Firstly it sets Z to 1 and then runs through the 'X' loop with X equal to 1, then 2 and finally 3. It then loops back to line 1000, sets Z to 2 and then repeats the 'X' loop. On completion of this, it goes back to line 1000 for the final time with Z set to 3. Generally, the number of times that the inside loop is repeated is controlled by the size of the outside loop.

PROGRAM 4.3(a)

```

1000 FOR Z=1 TO 3:PRINT"Z=";Z
1010  RESTORE
1020  FOR X=1 TO 3:PRINT"X=";X;" ";
1030    READ A$
1040    NEXT X
1050  PRINT A$
1060  NEXT Z
9000 DATA ONE,TWO,THREE,FOUR
```

Diagram illustrating the flow of Program 4.3(a):

- Line 1000: Start of Loop 1 (FOR Z=1 TO 3).
- Line 1010: RESTORE (inside Loop 1).
- Line 1020: Start of Loop 2 (FOR X=1 TO 3).
- Line 1030: READ A\$ (inside Loop 2).
- Line 1040: NEXT X (end of Loop 2).
- Line 1050: PRINT A\$ (inside Loop 1).
- Line 1060: NEXT Z (end of Loop 1).
- Line 9000: DATA ONE,TWO,THREE,FOUR.

However many times it is run, Program 4.3 or 4.3(a) will only ever deliver the third string; if a random delivery is required, then the loop variable must be replaced with a random number. In this case the loop is rewritten as FOR X=1 TO 'a random number', the random number having previously been defined, i.e.

PROGRAM 4.3(b)

```

1000 R=1+INT(4*RND(1))
1010 RESTORE
1020 FOR X=1 TO R
1030 READ A$
1040 NEXT X
1050 PRINT A$
9000 DATA ONE,TWO,THREE,FOUR
```

So far, then, a technique has been achieved for reading a word randomly from a whole list of words, and this can be used to produce a word guessing game of similar structure to that in Chapter 2.

### EXERCISE 4.2 (a long one)

Produce a flow chart and program for the following game based on Program 4.3(b).

- \* Player told rules and possible words to guess.
- \* Random word chosen.
- \* Player's guess INPUTted.
- \* Increment a count variable.
- \* INPUT checked for right/wrong.
- \* Player told "right/wrong".
- \* Player told how many goes taken.
- \* Player asked whether another go wanted.

A possible solution is given on page 11.4.

A normal hangman-style game differs from the one above in that in hangman individual letters are guessed, necessitating the dissection of strings.

### LEFT\$, RIGHT\$ and MID\$

Commodore basic provides us with a number of ways to chop up strings. The most brutal of these are LEFT\$ and RIGHT\$. These simply lop off the left and right ends of strings respectively. Let's try chopping up a few strings for practice! Firstly we'll set A\$ to "COMPUTER". To use the jargon, we will 'set the value of the variable A\$ to the literal value "COMPUTER'. The first practice will be with 'LEFT\$'. This has to be told which string it is going to chop up, so the command so far is 'LEFT\$(A\$...)', i.e. 'first find A\$'. Following the string name we put the number of characters that we want to chop off, i.e. 'LEFT\$(A\$,4)', will give us 'COMP'. Try that with

```
A$="COMPUTER"  
B$=LEFT$(A$,4)  
PRINT B$
```

The command RIGHT\$( ) works in exactly the same way except that it starts at the RIGHT hand end of the string.

Thus if A\$="COMPUTER", C\$ = RIGHT\$(A\$,5) will set the string C\$ to "PUTER", the 5 right-most characters of "COMPUTER". Try it out to make sure!

In contrast, the MID\$ function can be rather more precise in its operation. It allows one to chop away selectively small or large bits of the string under the operation. Like a versatile surgeon it can cut away from the middle or either end of the patient. Let's examine that in more detail, using the example

```
A$ = "COMMODORE"  
C$ = MID$(A$,4,3)
```

- \* When the computer sees C\$=MID\$(...) it knows that a string is to be dissected and the result stored in the string C\$.
- \* It then carries on and sees MID\$(A\$...). It translates this into 'first find A\$ and get prepared to do surgery on it'.
- \* Next it sees the '4' in MID\$(A\$,4....) and this tells it start at the fourth character of the string.
- \* Then it reads the '3' in MID\$(A\$,4,3) and, starting at the fourth character of the string strips off three characters. These it stores in C\$.

Thus, following the operation, C\$ would contain "MOD".

In general terms, the structure of the command is:

```
MID$(A$,START,LENGTH).
```

'START' and 'LENGTH' must both be whole numbers. MID\$ will cut out a part of the string A\$ starting at character number 'START' and of length 'LENGTH' characters.

As we shall see in this game, it is most often useful for our dissected string or substring to have a length of one character.



## LEN( )

Another new command that we need for the hangman game is LEN(). What this does is to report how many characters are in a string, eg:

```
A$="THIS IS VERY VERY VERY LONG"  
PRINT LEN(A$)
```

When you enter this, the computer will respond with '27', the number of characters in A\$, including spaces! Thus, if we wanted to cut a particular string in half we would first need to find the length of it.

### PROGRAM 4.4(a)

```
10 A$="CUTTING UP"  
20 L=LEN(A$)  
30 L=INT(L/2)  
40 PRINT LEFT$(A$,L)
```

To give you an idea of how LEN() will be used in the hangman game look at Program 4.4(b), below.

### PROGRAM 4.4(b)

```
10 A$="MY WORD"  
20 PRINT"YOUR WORD HAS";LEN(A$);"LETTERS"
```

## The Program's Structure : Part 1

The structure is defined in the flow chart - Figure 4.1 - the various program elements are numbered from 1 to 17. This is done to help you to match the program to the flow chart, but no direct bit-by-bit comparison can be made between flow charts and programs.

Standard symbols are used and the names of the four flags (F1 to F4) used have been inserted into the diamond shaped decision symbols. In addition, all the subroutines are numbered to cross-refer to the program - the sequence of these numbers has no significance at all, as they are purely for reference purposes. The subroutines will be studied in the numerical order shown on the chart.

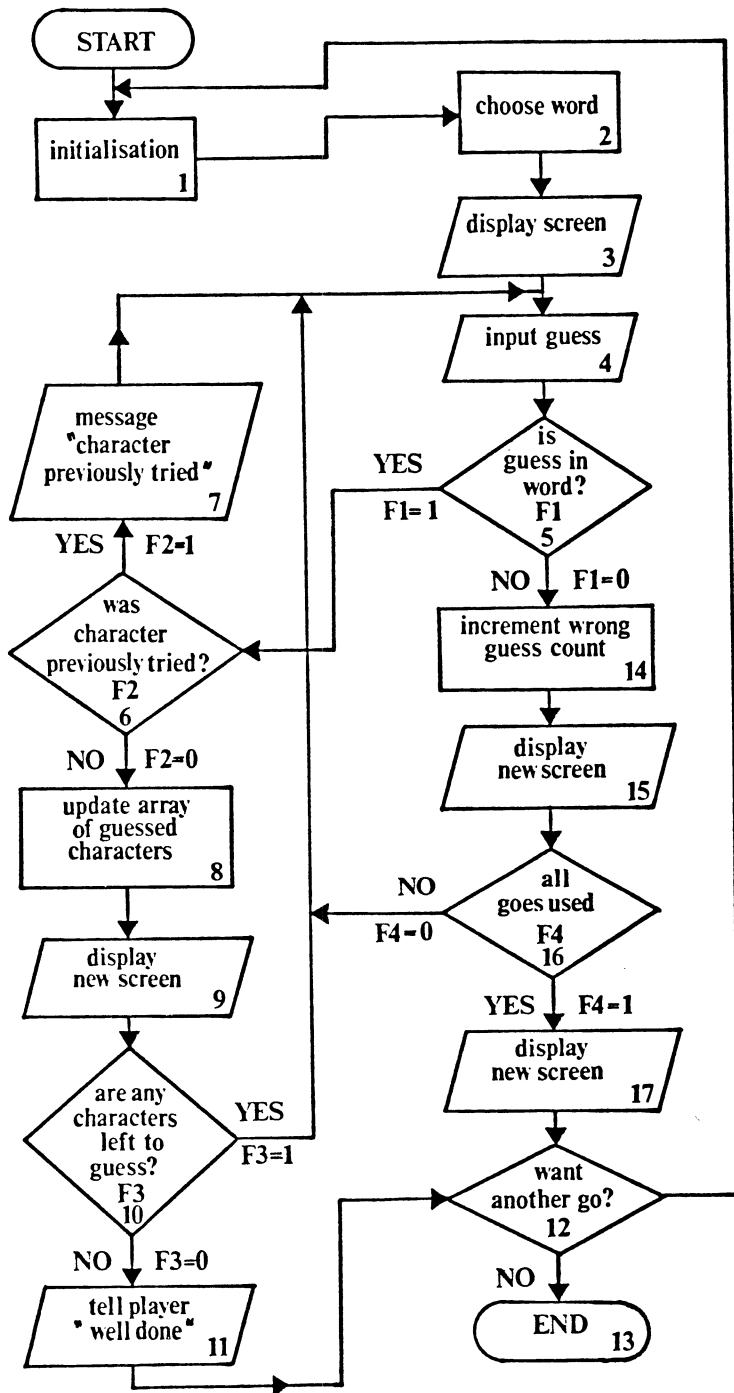


FIGURE 4.1

## Flags

In a game such as this, there are three basic elements, the structure, the detailed subroutines that carry out the various actions and the screen display. For instance, a subroutine is needed to check whether a letter guessed is in the word. Another is needed to see if it has been guessed already ... and so on. It is the structure, however, that determines the order in which the subroutines are called. Once called, a subroutine carries out its function and then returns control back to the main program. When the check is made to see whether the letter is in the word, for instance, the subroutine needs to store the answer 'yes' or 'no' in such a way that this information can be used later on. This is done in this program by means of 'flags': constants which are set to 0 (zero) or 1 (one) for a 'no' or 'yes' answer. In the case of this first subroutine, for instance, the flag which is given the name 'F1' is set to 1 for a 'yes' answer (i.e. the guess was correct) and '0' for a 'no' (i.e. the guess was incorrect). Once this process has been carried out for all the subroutines, i.e. the flags are set, the program can get on with playing the game!

We have already seen how cursor controls can be used for displays, allowing us to print at any part of the screen. The large number of cursor symbols can at times be confusing, to say the least, and this could lead to errors.

The display for the hangman requires an extensive use of PRINTs and cursor characters, so instead of having 10 right cursors, if we want to move 10 spaces to the right, we can use the TAB() command.

## TAB( )

PRINTing on the C-64 is done in terms of columns. As far as the C-64 is concerned, the video screen is divided into 40 vertical columns, numbered 0 to 39. Each character is PRINTed into one of these columns.

TAB(X) will cause the next character to be PRINTed in column X. Try the following program :

```
2 PRINT TAB(20);"A"
```

When this is run the 'A' will be PRINTed in column 20, because that was the value set for the TAB. Using TAB( ) is much easier than pressing right cursor twenty times, as demonstrated in the above example, but is not particularly advantageous if only a small number of right cursors are required. TAB( ) can only be used to move the cursor right.

## GOSUB....RETURN

So that the various subroutines of the game can be developed individually and then linked together, they are written as subroutines called by the command GOSUB. This command re-directs the program to a specified line and the C-64 continues to execute the program from that line onwards. However, when it comes to the command RETURN, it recognizes this as signifying the end of that particular subroutine and then switches back to the statement after the GOSUB. Thus, a subroutine is a small section of program that you tell the program to visit, do a particular job and then RETURN.

Program 4.5 illustrates how to do this. It incorporates three GOSUBs, one of which is repeated.

### PROGRAM 4.5

```
100 GOSUB 200
110 GOSUB 300
120 GOSUB 200
130 STOP
200 PRINT "FRED"
210 RETURN
300 PRINT "JOE"
310 RETURN
```

When this is run, line 100 is executed and this directs the program to the subroutine at line 200, which prints 'FRED'. Next, line 210 RETURNs the program to line 110 which then redirects to the subroutine at 300. This is executed, the program again RETURNed (to line 120), only to be GOSUBed once again to the subroutine at 200. On RETURN from this, the program meets the command STOP which STOPS execution and prevents the program from running once again into the subroutines.

The GOSUB command, although superficially similar to the GOTO, has an important difference - it expects to be matched by a RETURN. Thus, when the computer executes a GOSUB it automatically stores, in memory, a return line number for the program to jump back to once the subroutine is finished (i.e. at RETURN). This is stored in a special place in memory called the GOSUB stack. If a program inadvertently runs into a subroutine, i.e. if the STOP in line 130 were not there, then on finding the RETURN in line 1010 it would search for a return line number. However, as the subroutine was not called by a GOSUB, no return line number would have been stored and problems would arise. Try this out by deleting line 130 and running program 4.5 again.

## The Variables Used

In a fairly complicated program like this one, it is a good idea to make a list of the variables that will be used. This makes life much easier while developing the program and, in particular, aids debugging.

### (i) Numeric

- E the number of Erroneous (wrong) guesses.
- L the length of the word to be guessed.
- L2 the length of the array of guessed characters (X\$).
- R the random number that defines the word to be guessed.
- X a loop variable, used generally.

### (ii) Alpha-numeric (string)

- A\$ the word to be guessed.
- B\$ general variable for INPUTs.
- G\$ the currently guessed letter.
- W\$ a string made up of the letters guessed to date or dots, in the correct order. For example, this could be C.M. if the word were CBM and the letters guessed to date were 'C' and 'M'.
- X\$ a string made up randomly from all the letters that have been guessed to date.
- Y\$ a temporary variable used in GET statements.
- Z\$ a literal string, either ' FIRST ' or ' NEXT ' depending whether next entry is the the first one or not.

The various program elements are now considered, roughly in the order shown on the flow chart.

### Program Element 1: Initialisation

The initialization routine starts by clearing the screen by printing the 'clear home' character. After this, lines 450-470 are used to give the rules of the game, these having been left for you, the reader, to do your literary best. Real care must be taken over this documentation phase as, if the game is to stand on its own without you to explain it, the rules must be absolutely clear.

The initialization routine can now be written; what it will do is

- Clear the screen and set to upper case - Line 410
- Announce the game
- Give the rules - Lines 450 to 470
- Stop the program until the rules have been read and a key pressed - Line 480
- Clear the screen once the rules have been read - Line 490

Putting that into a Program:

PROGRAM 4.6(a)

```
400 REM SCREEN DISPLAY
410 PRINT "<CLR>";CHR$(142)
420 PRINT TAB(13);"<RVSON><15 SPACES>"
430 PRINT TAB(13);"<RVSON> H A N G M A N "
440 PRINT TAB(13);"<RVSON><15 SPACES>"
450 PRINT TAB(10);"THESE ARE THE RULES--"
"
460 PRINT TAB(11);"<DCRSR>THERE ARE NO R
ULES!"
470 PRINT"<2DCRSR><4RCRSR>PRESS ANY KEY
TO BEGIN"
480 GET A$:IF A$=""THEN 480
490 PRINT "<CLR>"
```

Each time the game is played, several variables need to be reset and strings cleared, for instance Z\$. On the first run through the game this is set to ' FIRST ' i.e. the screen then says what is your Z\$ (first) guess. Immediately after use, Z\$ is reset to "NEXT" so that the player is then asked 'what is your Z\$ (next) guess?' Of course, once the game has been played Z\$=" NEXT " and needs re-setting or re-initializing. Other variables need resetting too, such as E - the number of incorrect guesses so far, and X\$, a string comprising all the guesses made to date.

PROGRAM 4.6(b)

```
500 Z$=" FIRST "
510 E=0
520 X$=""
```

## Program Element 2: Choose word

This routine, shown in Program 4.6(c), is very similar to those discussed above, particularly Program 4.3. It generates a random number and then searches that number of times through the DATA statements.

### PROGRAM 4.6(c)

```
1000 R=1+INT (4*RND(1))
1001 R=4:REM development only
1010 RESTORE
1020 FOR X=1 TO R
1030 READ A$
1040 NEXT X
1050 L=LEN(A$)
```

Line 1001 sets 'R' to the value of '4', so that when testing this program we know what word to guess. This will save a lot of time when debugging the program.

Note the FOR....NEXT loop in 1020 runs from 1 to R.

In addition to choosing the word, the subroutine also calculates its length, L, as this is needed in other subroutines. The variable 'L' is then used to make up the string W\$. At the beginning of the game this simply contains the requisite number of dots - i.e. one for each letter. As correct guesses are made the appropriate letters are inserted in the correct place in the array and then displayed on the screen.

### PROGRAM 4.6(d)

```
1500 W$=""
1510 FOR X=1 TO L
1520 W$=W$+"."
1530 NEXT X
```

### Program Element 3: Display screen

At this stage of the game, all that has to be displayed is the message telling how many letters the word has, and the string W\$. One thing to note here is that every message printed on the screen is printed starting at the top left hand corner of the screen. The down-cursor character is used to move the print out into the appropriate position. This is done to prevent any message being printed in the wrong area. As the screen display in this program is very important, one cannot afford to take any chances. By starting all print events at the top left of the screen and working down, the position of a print statement no longer relies on the current cursor position.

#### PROGRAM 4.6(e)

```
1540 PRINT"<HOME><2DCRSR>";TAB(8);"YOUR  
WORD HAS ";L;" LETTERS"  
1550 PRINT"<2DCRSR>";TAB(15);W$
```

### Program Element 4 : Input a guess

All that is required here is a simple message to tell the user to input a guess. Sometimes, however, it will be the 'FIRST' guess and sometimes the 'NEXT' guess. This can be accommodated by assigning the word 'FIRST' to a string during the initialization and then, once the program has run, changing the contents of the variable to 'NEXT'. The two elements that do this are shown in Program 4.6(f). The use of a GET rather than INPUT allows a single character - presumably a letter! - to be input without the need to press <RETURN>.

#### PROGRAM 4.6(f)

```
500 Z$=" FIRST "  
1800 PRINT"<HOME><3DCRSR><8RCRSR>WHAT I  
IS YOUR";Z$;"GUESS? <LCRSR>";  
1810 GET G$:IF G$="" THEN 1810  
1820 PRINT G$  
1830 Z$=" NEXT "
```



### A Diversion : Setting the flags.

Once the guess has been keyed in, a check needs to be made to see whether the letter is a correct guess, whether it has been used before, whether there are any letters left to guess or whether all the 10 allowed goes have been used up. This is all done in a subroutine starting at 3000 and is called by line 2000. The setting of each flag is considered separately below and when each one has been set, the program is RETURNed by line 3400; i.e.

PROGRAM 4.6(g)

```
2000 GOSUB 3000
.
2840 GOTO 1800
.
3400 RETURN
```

### Program Element 5: Is guess in word?

Once the guess has been made and the word chosen, the subroutine in Program 4.6(h) simply needs to read through this word looking for a match for the inputted letter. Should it find such a match, then F1 will be set to 1. Note that, at the start of this subroutine, the flag is reset to zero and remains at zero unless the test at line 3020 proves positive.

PROGRAM 4.6(h)

```
2999 REM ****IS GUESS IN WORD?
3000 F1=0
3010 FOR X=1 TO L
3020 IF G$=MID$(A$,X,1)THEN F1=1:GOSUB
5500
3030 NEXT X
```

Notice that if flag F1 is set then a trip is made to the subroutine at 5500. Let's see what this subroutine does:

PROGRAM 4.6(i)

```
5500 IF X=L THEN 5540
5510 IF X=1 THEN 5560
5520 W$=LEFT$(W$,X-1)+G$+RIGHT$(W$,L-X)
5530 RETURN
5540 W$=LEFT$(W$,L-1)+G$
5550 RETURN
5560 W$=G$+RIGHT$(W$,L-1)
5570 RETURN
```

Remember that W\$ holds the status of the word being guessed, starting off with all dots. As correct guesses are made, the correct letters are inserted into this string at the appropriate place so that the word to be guessed is built up gradually.

The letter guessed may be one of three cases:

- \* The leftmost character of the word. In this case X will be 1 and the instruction in 5560 is followed. The L-1 rightmost characters are saved and W\$ is updated to the correct character at the left together with the rest of W\$ (line 5560).
- \* The rightmost character. Here we save the L-1 characters at the left of W\$ and replace the last character with the guessed letter (line 5540).
- \* A letter other than these - i.e. in one of the middle positions of W\$. We save all characters to the left of 'X' and all characters to the right of 'X'. We can then replace the full stop in the middle with the correct character (lines 5520).

#### Program Element 6: Was character previously tried?

When a player puts in a guess that is a repeat of a previous entry, this program treats him kindly. It would be possible to charge this guess against his number of allowed attempts but the option chosen here is to report that that particular letter has been guessed and then loop back for another input. In Program 4.6(j), F2 is initially set or reset to zero, and is only set to one if the inputted character, G\$, is found in X\$.

The value, X\$, was set to "", i.e. an empty string on line 520 of the initialization procedure, Program 4.6(b). As a guess is made, it is added to the string (program element 8) and so, at this stage it is only necessary to read through the string to check whether any of its letters equal G\$, the latest guess. One slight complication exists in that the string gets one letter longer each time around, so it is always necessary to re-calculate its length (L2), as in line 3110.

#### PROGRAM 4.6(j)

```
3100 F2=0
3110 L2=LEN(X$)
3120 FOR X=1 TO L2
3130 IF G$=MID$(X$,X,1) THEN F2=1
3140 NEXT X
3150 IF F2=0 THEN X$=X$+G$
```

#### Program Element 7: Message: "character previously tried"

The aim of this message is to inform the player clearly that the letter just guessed has been tried, and then to clear the screen back to its previous state. The message is PRINTed onto a line that is currently empty. Once on the screen, the message is held there for a time while the player takes it in and then it is cleared. What is needed here is a technique for causing the program to wait for a specific time period.

#### Delays

The common way to do this is to use a FOR...NEXT loop that does nothing but go round and round and round. The length of the delay is then set by the number of times that the program runs through the loop. Try this out with Program 4.6(i/i), which is designed to let you INPUT the loop length. Run it first with an input of 100, then 1000 and finally 10000 just to get an idea of the loop lengths.

#### PROGRAM 4.6(i/i)

```
10 PRINT"<CLR>"
20 INPUT T:REM SET UP LOOP LENGTH
30 FOR Z=1 TO T
40 NEXT Z
50 PRINT "FINISHED!"
```

A routine such as this allows you to put in delays to suit the needs of the program.

Once the message in Program Element 7 has sunk in, it needs to be removed by printing blank spaces over it. This is achieved by means of a PRINT statement that PRINTS the blank spaces (i.e. " ") over the text to be obliterated.

PROGRAM 4.6(k)

```

4000 PRINT"<20DCRSR><4RCRSR>YOU'VE ALREA
DY TRIED THAT LETTER"
4010 FOR X=1 TO 500: NEXT X
4020 PRINT"<UCRSR><40 SPACES>"
4030 RETURN

```

Program Element 8: Update array of guessed characters

In this element, as it is already known that the character guessed has not previously been guessed, it is inserted into W\$ at the appropriate place. The process is carried out by line 3020 in Program 4.6(l) which reads through the word A\$ letter by letter and, on finding a match with the guess, stores that character in the appropriate place in W\$. If one were to stick to rigid structured programming, this would be set once F1 is set, by testing for this flag later. However, once the flag is set, on line 3020, the array can be updated immediately. Indeed, if the setting is done on the same line, following a colon, you can be sure that it will only happen if the earlier 'IF' statement gives a positive (i.e. a YES) answer. (We have already seen how the subroutine at 5500 works.)

PROGRAM 4.6(l)

```

3020 IF G$=MID$(A$,X,1)THEN F1=1:GOSUB 5
500

```

Figure 4.2 demonstrates the process for the INPUT of an 'E' (i.e. G\$="E") where A\$="COMPUTER" and the 'E' has not previously been guessed.

LOOP NUMBER	A\$	W\$ before	W\$ after
1	C	.	.
2	O	.	.
3	M	.	.
4	P	.	.
5	U	.	.
6	T	.	.
7	E	.	E
8	R	.	.

FIGURE 4.2

### Program Element 9: Display new screen

It is only necessary at this stage to print out the current state of the guessed word, this being done in line 3040 of Program 4.6(m).

PROGRAM 4.6(m)

```
3040 PRINT"<DCRSR>";TAB(15);W$;"<3UCRSR>"
```

### Program Element 10: Are all characters guessed?

To check this, string W\$ needs to be read through to see if any character position remains unfilled. If this is so then F3 is set to a 1, in line 3230 of Program 4.6(n).

PROGRAM 4.6(n)

```
3199 REM ****ARE ALL CHARACTERS GUESSED?
3200 F3=0
3210 T$="."
3220 FOR X=1 TO L
3230 IF T$=MID$(W$,X,1) THEN F3=1
3240 NEXT X
```

### Program Element 11: Message "well done"

This element tells the player that (s)he has guessed the word correctly. Once the message is on the screen, the game is over and there is, therefore, no need to display it for a fixed time. As the next stage of the program is to ask the player if (s)he wants another go, the message can be left on until (s)he makes a move. Unlike the other subroutines, this one is called by a GOTO, as, on completion, the program runs into the 'do you want another go?' routine. Sticking to pure structuring, the program would be called by a GOSUB and then, having RETURNed, would be directed to the termination procedure.

PROGRAM 4.6(o)

```
4300 PRINT"<HOME><10DCRSR>";TAB(18);"WE
LL DONE"
4310 PRINT TAB(18);"YOU HAVE"
4320 PRINT TAB(18);"GUESSED"
4330 PRINT TAB(18);"THE WORD"
4340 FOR X=0 TO 500:NEXT X
4350 GOTO 5000
```

### Program Element 12: Do you want another go?

This is a simple test for the key 'Y' as shown on line 5030 of Program 4.6(p). If the answer is 'N' (or indeed, anything other than 'Y') then the closing message "BYE" terminates the whole proceedings.

#### PROGRAM 4.6(p)

```
5000 PRINT"<5DCRSR><BLU>"
5010 PRINT TAB(8);"DO YOU WANT ANOTHER G
0?"
5020 GET A$:IF A$="" THEN 5020
5030 IF A$="Y"THEN 490
5050 PRINT"<CLR><9DCRSR>";TAB(18);"BYE":
END
```

### Program Element 13: End

This part is most conveniently appended to the end of the previous element with a friendly (!) PRINT "BYE." (or whatever you wish - see line 5050 of Program 4.6(p)).

### Program Element 14: Increment wrong guess count

The variable 'E' records the number of wrong guesses and is simply incremented at the appropriate time whenever F1 is set to '0'. It is done in line 2800 following the testing of the flag in line 2300 of Program 4.6(q).

#### PROGRAM 4.6(q)

```
2300 IF F1=0 THEN 2800
.
.
2800 E=E+1
```

## Program Element 15: Display new screen

When the guess is incorrect, the player is told "SORRY THAT LETTER IS NOT IN THE WORD."

PROGRAM 4.6(r)

```
4200 PRINT"<HOME><20DCRSR><5RCRSR>SORRY
      THAT LETTER IS NOT IN THE WORD"
4210 FOR X=1 TO 500 : NEXT X
4220 PRINT"<UCRSR><40 SPACES>"
4230 RETURN
```

As in a previous subroutine, the message is maintained on the screen by the FOR.. NEXT loop on line 4210

Following this, the hangman must be drawn. The hangman we will draw will be created by using the graphics characters obtainable from the keyboard. These are characters obtained by pressing either the Commodore key or the shift keys.

## ON...GOTO

The individual pieces of the hangman will be drawn using PRINT commands, and the subroutine will be subdivided into separate program sections. One section will be for drawing the frame, one for drawing the head and so on. These individual sections will be accessed depending on the value of 'E' (the incorrect guess count), using a special version of the IF...THEN command, called ON...GOTO. This works a bit like lots of 'IF...THEN...' commands would. Taking the example:

```
ON X GOTO 100,200,300
```

The computer understands this as:

```
" ON the value of X, GOTO 100, 200, 300"
```

Thus if X = 1 then the program is directed to line 100, if X is 2 then the program goes to line 200 and so on. If X is '0', or greater than the number of line numbers listed the program will continue onto the next line. As well as using 'GOTOs', the ON... command can just as well use GOSUBs. In this case, it directs the program to the subroutine in just the same way as would any GOSUB.

For drawing the hangman graphics we will use an 'ON GOTO' command.

PROGRAM 4.6(s)

```
4400 ON E GOTO 4500, 4490, 4480, 4470, 4
460,4450, 4440, 4430, 4420, 4410
4410 PRINT "<HOME><13DCRSR><11RCRSR>┐<DC
RSR><LCRSR>┆<DCRSR><LCRSR> <9UCRSR>"
4420 PRINT"<HOME><13DCRSR><9RCRSR>┌<DC
RSR><2LCRSR>┆<DCRSR><LCRSR>┆<9UCRSR>"
4430 PRINT"<HOME><11DCRSR><11RCRSR>┌<5U
CRSR>"
4440 PRINT"<HOME><11DCRSR><8RCRSR>┌ <5UC
RSR>"
4450 PRINT"<HOME><11DCRSR><10RCRSR>└<DCR
SR><LCRSR>┆<6UCRSR>
4460 PRINT"<HOME><8DCRSR><9RCRSR>┐<DCR
SR><3LCRSR>┆┆<DCRSR><3LCRSR>┐<4UCRSR
>
4470 PRINT"<HOME><7DCRSR><10RCRSR>┐<DCRS
R><LCRSR>└<2UCRSR>"
4480 PRINT"<HOME><7DCRSR><6RCRSR>┐<DCRS
R><LCRSR>┆<3UCRSR>"
4490 PRINT"<HOME><7DCRSR><7RCRSR>└<UC
RSR>"
4500 PRINT"<HOME><7DCRSR><6RCRSR>┐<DCRSR
><LCRSR><DCRSR><LCRSR>┆<DCRSR><LCRSR>┆<D
CRSR><LCRSR>┆<DCRSR><LCRSR>┆<DCRSR><LCRS
R>┆<DCRSR><LCRSR>┆<DCRSR><LCRSR>┆<DCRSR>
<LCRSR><DCRSR><LCRSR>┆"
4520 RETURN
```



### Program Element 16: Are all goes used?

This section is a simple test of the variable 'E':if it is equal to 9, F4 is set to 1.

#### PROGRAM 4.6(t)

```
3300 F4=0
3310 IF E=9 THEN F4=1
3400 RETURN
```

### Program Element 17: Display new screen

By this point in the game, it's all over for the player and that's what the message says in lines 4900 and 4905. The player is given the correct solution to the game as some compensation!

#### PROGRAM 4.6(u)

```
4900 PRINT"<HOME><12DCRSR>";TAB(15);"SO
RRY YOU'VE HAD"
4905 PRINT"TAB(15);"YOUR TEN GOES"
4910 PRINT TAB(15);"THE WORD WAS ";A$
4920 FOR X=1 TO 500 : NEXT X
4930 RETURN
```

## The Program's Structure: Part II

All that remains to be done now is to sort out the structure of the program so that the various routines are called at the appropriate time.

There are many possible ways that this could be achieved by following through the flow chart and choosing a series of routes that covers all possibilities. However, that would soon become complicated and it would be impossible to avoid duplication. A more logical way would be to group together all the routines that set flags, once the necessary data is available to condition these. Having run through the routines, the flag structure is as shown in Figure 4.3. This figure is designed to display clearly the various flag states for a particular game state.

game status	F1	F2	F3	F4
1 guess correct, character previously tried	1	1	1	0
2 guess correct, all characters now guessed	1	0	0	0
3 guess correct, all characters not yet guessed	1	0	1	0
4 guess incorrect, all goes used	0	0	1	1
5 guess incorrect, some goes left	0	0	1	0
6 guess incorrect, character previously tried	0	1	1	0

FIGURE 4.3

The logic control section can be drawn by means of this figure and the flow chart, and should be structured to run through all the necessary subroutines as economically as possible. There's no rule for doing this but one (very logical) way would be to work through the flag structure diagram and to cover each status line-by-line with a composite flag testing statement. Thus, game state 1 could be tested by a statement:

```
IF F1=1 AND F2=1 AND F3=1 AND F4=0 THEN (Take appropriate action)
```

While this is, strictly speaking, logically correct, most programmers would divide the logical tests into sections and deal with them section by section. This is done below, although there is nothing to stop you adopting the straight logical approach.

The flag-setting routines, the logic-defining part of the game, have already been described - these being the subroutines starting at line 3000 and ending with the RETURN at line 3400.

With the approach adopted here, the first task is to look for the easiest settled case. For instance, if the character has previously been guessed, then, regardless of whether or not it is in the word, i.e. game states 1 and 6, all that remains to be done is to report the facts and then go back for another input. This is readily dealt with by the one line of Program 4.6(v) which tests the flag F2 and then directs the program to the relevant

subroutine. On completion of this, of course, the program will RETURN to line 2200 and thence be directed to line 1800 for the next input.

PROGRAM 4.6(v)

```
2200 IF F2=1 THEN GOSUB 4000:GOTO 1800
```

Once F2 has been tested and a branch taken when appropriate, it can be accepted throughout the remainder of the logic control section that F2=0.

The next test made is for a correct/incorrect answer (F1). When an incorrect one is found, the program loops to 2800, the section between 2400 and 2800 dealing exclusively with correct answers, and that from 2800 to 2830 with incorrect ones. In working with this structure it eliminates the necessity of rechecking F1 constantly. We know that between lines 2400 and 2800 F1 always equals 1 and that in the section starting at 2800, it always equals zero.

PROGRAM 4.6(w)

```
2230 IF F1=0 AND F2<>1 THEN 2800
```

Following this, it only remains to test F3 to check whether all the characters have been guessed. If they haven't, then line 2500 (Program 4.6(x)) directs the program back for another letter to be entered. If they have been guessed, then the program is directed to the subroutine at 4300 to congratulate the player and then sent to line 5000 with a GOTO. This latter choice is necessary as the routine at 5000 asks whether the player wants another go. On receipt of a 'Y' the program returns to the beginning of the game, while a 'N' produces the farewell message.

PROGRAM 4.6(x)

```
2500 IF F3=1 THEN 1800  
2700 GOSUB 4300: GOTO 5000
```

The remainder of the logic control section, from 2800 onwards, deals with game states concerned with wrong guesses (F1<>1). As the error variable 'E' is used to control the graphics at the end of the game, it must be incremented when a wrong answer is given; i.e. a simple E=E+1 (line 2800 of Program 4.6(y)). Next, the string of letter guesses must be updated using the subroutine at 4200 (line 2810). Following this, the graphics display is completed according to the value of E (GOSUB 4400) and then a test is made on the value of E to see if all 10 goes have been used up. If they have F4=1 and the message is given that it's all over and then the subroutine at 5000 is called (line 2830) and the player

asked whether another go is required.

PROGRAM 4.6(y)

```
2800 E=E+1
2810 GOSUB 4200
2820 GOSUB 4400
2830 IF F4=1 THEN GOSUB 4900:GOTO 5000
```

The final few lines of the program provide the stock of words for the player to guess. These are stored in DATA statements at line 11000. In the example given only four words are provided, but this number can be increased as desired. One other change will be needed to ensure that all these additional words are READ. The variable 'R' needs to be set to a number between 1 and the number of words in the DATA statement. Thus if there are 20 words stored in the DATA statements the value of 'R' would be set to  $R=1+INT(20*RND(1))$ . The DATA statement structure is as shown in Program 4.6(z).

PROGRAM 4.6(z)

```
11000 DATA BUG,CBM,COMPUTER,TYPE
```

Once the sections of this program are typed in, you will have a working word guessing game. What's more it's a game that you, the reader, can understand and therefore modify. The words, as you know, are stored in the data statements and these are readily modified. You can substitute your own set of words or even give the game some simple intelligence. Were the words in the DATA statements to be graded from simple at the beginning to more difficult towards the end, then one (or any other number of) successful go(es) could add a constant to the READ loop so that it reads more deeply into the DATA, thus yielding more difficult words. Other display modifications are described in Part 2 of Chapter 10.

## CHAPTER

# 5

### REACTION TESTER

**I**n this chapter a program is developed that will enable your C-64 to measure the speed of your reactions accurately. It will do this by seeing how long it takes for you to press a key once told to do so. It makes extensive use of one of the clocks that is built into the C-64 and made accessible to you, the programmer, for carrying out all sorts of programming tasks.

Computers are clearly pieces of equipment that work very rapidly and to be able to do this, they need to get their co-ordination just right! For this reason they have several very accurate time-pieces built into them. Some of these are used solely for internal control functions but the C-64 designers have brought two of these to the surface so that they can be accessed from BASIC and, therefore, built into your programs.

Very conveniently, one of these is accessed as a string (TI\$) and one as a numeric variable (TI) and, as they are stored in this way they can be very simply displayed by means of PRINT statements. Just to prove that try:

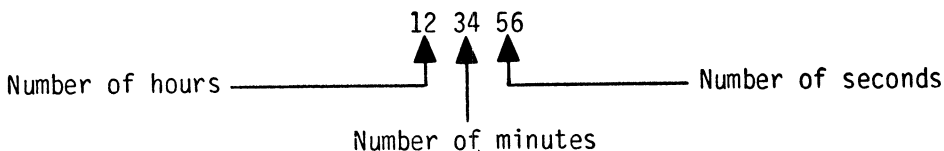
```
PRINT TI$, TI
```

This should have you a display of - well I don't really know! The reason I don't know is that both of these clocks' times are reset to zero when the machine is turned on. So, if you've just turned on the numbers will be small, whereas if you've been at it all day, the numbers will be large.

Whatever number you got, let's investigate it a little. Firstly, TI\$. We'll assume that the command 'PRINT TI\$' yielded the result:

123456

In this case, the computer is reporting back that it has been turned on for twelve hours, thirty four minutes and fifty six seconds, i.e.:



We can use TI\$ to display the time of day as it's possible to split up any string using RIGHT\$, LEFT\$ and MID\$. The hours and seconds are the easiest to get at as:

```
RIGHT$ (TI$, 2) gives the seconds and  
LEFT$ (TI$, 2) gives the hours
```

To get at the minutes we need to start at the third character and print the next two, i.e.:

```
MID$ (TI$,3,2) gives the minutes
```

With this knowledge we can now convert our C-64 into a clock. Let's have a go. Firstly we'll say what we're doing, i.e.:

```
PRINT "<CLR> THE TIME NOW IS:"
```

Next we need only print the hours and minutes, i.e.:

PROGRAM 5.1(a)

```
200 PRINT"<CLR> <3 DCRSR> THE TIME NOW I  
S:"  
210 PRINT RIGHT$(TI$,2); " SECONDS"  
220 PRINT MID$(TI$,3,2); " MINUTES"  
230 PRINT LEFT$(TI$,2); " HOURS"
```

Great eh? Running this will give you the time - well-oops my screen says that it's just after one o'clock whereas it's nearly eight. What went wrong? That's it; you've guessed it: the screen is reporting how long the computer has been turned on. What we have to do if we wish to convert the machine into a timepiece is to set it to the right time. When doing this TI\$ is treated slightly differently from any other string as each one of its six characters has to be specified when assigning a value to it. Try resetting the clock to zero with

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

Now, when you run Program 5.1 you'll get a very different time. Of course you can set TI\$ to any time you like with a straight assignment. To improve the digital clock let's add a setting feature. In order to get the time exactly right, it would be better to input the hours, minutes and seconds separately using BASIC'S feature of:

### String Concatenation

This rather fancy name simply means string addition and with the clock would enable us to input the hours as H\$, The minutes as M\$ and seconds as S\$. Once these are stored in memory, they can be added together to form TI\$ i.e.  $TI\$ = H\$ + M\$ + S\$$ . Incorporating these into Program 5.1(b) gives a time-setting feature. Add this to 5.1(a).

#### PROGRAM 5.1(b)

```
100 INPUT "<CLR><3DCRSR>ENTER HOURS"; H$
110 INPUT "<DCRSR> ENTER MINUTES"; M$
120 INPUT "<DCRSR> ENTER SECONDS"; S$
130 TI$ = H$ + M$ + S$
```

Did your version run? Well, it would have done if each INPUT was two characters long: if not, TI\$ would have ended up the wrong length! In a case like this, an error check is called for and, as it is needed three times it is best written into a subroutine: what it needs to do is:

- \* Check that the INPUT is the correct length.
- \* If correct, continue program execution.
- \* If incorrect, report to user and go back for another INPUT.

The requirement for the program to return to a line is best done by the IF ... THEN test. However, this adds a slight complication, which can be handled by means of a flag. When the INPUT is the right length, this flag, say F, can be set to 1 and when the wrong length set to zero.

A further complication arises from the fact that the test has to be performed on either H\$, M\$ or S\$, calling for a flexible subroutine. One way around this is to set the three strings to two characters length at the beginning of the program and to check that all of them remain at two characters after the INPUT. When one string is no longer the correct length, an error is flagged. To incorporate this into the overall program a few earlier lines need to be changed to test the flag on RETURN, i.e.:

#### PROGRAM 5.1 (c)

```

    90 F=1:H$="XX":M$="XX":S$="XX":PRINT"<CLR>"
    100 INPUT "<3DCRSR>ENTER HOURS";H$:
    GOSUB 300: IF F=0 THEN F=1: GOTO 100
    110 INPUT "<DCRSR>ENTER MINUTES";M$:
    GOSUB 300: IF F=0 THEN F=1: GOTO 110
    120 INPUT "<DCRSR>ENTER SECONDS";S$:
    GOSUB 300: IF F=0 THEN F=1: GOTO 120
    150 STOP
    .....
    300 IF LEN(H$)=2 AND LEN(M$)=2 AND LEN(S$)=2
    THEN RETURN
    310 PRINT "INPUTS MUST BE TWO CHARACTERS
    LONG<UCRSR>": F=0: RETURN

```

So far this clock is really a one-shot affair as it tells you the time once and then dies. The next task is to keep it running. Sorry, that should read "Your next task". Have a go at Exercise 5.1. Don't worry if you have problems, it's fully explained in the Solutions Chapter.

#### EXERCISE 5.1

Modify the digital clock program so that it gives a continuous display of the time. A possible solution is given on page 11.5.



## TI

For more accurate timing purposes the second clock is most useful. This appears in BASIC as TI. Once again this will yield a group of numbers but this time these are organized differently. The number on the screen is simply the number of one-sixtieths of a second that have passed since you turned your machine on. Next try this little exercise: type in:

```
TI$="000000":PRINT TI
```

Having done this, you should have a screen display of about 1. So, resetting TI\$, resets TI: this will be most useful to us when developing our reaction tester. Of course, if it's seconds that we're interested in and not sixtieths then we need to divide TI by 60 and then take the integer to eliminate the fractions.

To get used to using TI, lets do a few timing exercises. Firstly, lets get the C-64 to print "FRED" 50 times and see how long it takes:

### PROGRAM 5.1(d)

```
100 TI$="000000"  
110 FOR Z=1 TO 50  
120 PRINT "FRED"  
130 NEXT Z  
140 PRINT INT(100*(TI/60))/100; "SECONDS"
```

So that took approximately 1.86 seconds. Note the multiplication and division by 100 on line 140: this gives us the first decimal places for the seconds. What if we didn't ask it to print, try REMing out line 120 i.e., 120 REM PRINT "FRED". This time it took approximately 0.09 seconds. What if we remove line 120 altogether? Do this and then re-run the program. Interesting eh?

This exercise tells us quite a lot about BASIC - not just about timing!

Anyway, back to the reaction timer. What this program will do eventually is to tell the user that timing is about to begin and then, ask him/her to press a key. In the meantime we ought to provide the player with something to keep their attention. One thing that we can do is to flash though the colors on the screen but to do that on the C-64 we will need to understand the command:

## POKE

A very useful command is POKE, but one that differs somewhat from most other BASIC commands. In affect all it does is allow you to insert a particular number into a particular memory location. Thus, the command to POKE a number into memory must specify both the number and the memory location. For instance, the command:

```
POKE 828,90
```

will load a number 90 into memory location 828. Just what the effect of this is will be investigated later! Basically POKE's are used for two reasons:

- (1) to store data in specific memory locations
- (2) to control the inner workings of the C-64.

(1) There are many reasons for wishing to store data in specific memory locations and some of these will be examined later in the book.

(2) Many functions of the C-64 are controlled by the contents of particular bytes of memory. For instance, the color that you see on your screen is determined by the contents of the three memory locations, 53280, 53281 and 646. Let's investigate this with a few loops. As each color setting can range from 0 to 15, each of these memory locations can be cycled through the range 0-15, i.e. looking at all the screen colors:

### PROGRAM 5.2

```
10 FOR X = 0 TO 15
20 POKE 53281,X
25 PRINT X
28 FOR P=1 TO 500:NEXT P
30 NEXT X
```

Return the screen back to original colors by holding the RUN/STOP key and then pressing the RESTORE key. To examine border colors modify line 20 to read:

```
20 POKE 53280,X
```

and run the program. Finally, to look at character color, modify line 20 to read:

```
20 POKE 646,X
```

and run it. Really interesting eh?

## Machine crashes

This color feature will be useful when trying to occupy the mind of the user in the reaction tester. Many other features can be turned on or 'enabled' by means of POKES. However, when POKEing around, take care. Some POKES will simply turn the machine off without much ceremony. Just for fun(!), enter:

POKE 43,0

Now try and list whatever was in memory. What you have done with this command is to cause a 'crash'. However, in this case it is a recoverable crash that can be undone with a:

POKE 43,1

Not all crashes are recoverable, however, and with some of these, you will lose the contents of memory altogether. However, whatever you POKE into wherever, you won't damage the computer. The worst you will do is lose your program, so you can always just switch off and start again.

## PEEK

This command is the direct opposite of POKE, where one puts data into memory, the other retrieves it. Actually PEEK looks into memory and copies what it finds there. For instance, try:

PRINT PEEK (828)

i.e. **PRINT** what you find when you PEEK into memory location 828.

## Reaction Tester

The first task is to tell the players the rules and in programming terms, this is very straight-forward. If you wish to change this part, e.g. add your own rules, then by all means do so. The more you change things, then the more the programs become yours and the more you will learn. Type NEW and then these lines.

### PROGRAM 5.3

```
10 REM REACTION TESTER
20 PRINT"<CLR><12RCRSR>REACTION TESTER"
40 PRINT"<2DCRSR><4RCRSR>THESE ARE THE R
ULES"
50 PRINT"<DCRSR>WHEN TOLD TO DO SO YOU M
UST PRESS"
60 PRINT"ANY KEY. THE TIME IT TAKES YOU
TO"
70 PRINT"PRESS THE KEY WILL BE CALCULATE
D"
80 PRINT"AND ADDED TO YOUR TOTAL SCORE.
IF"
90 PRINT"YOU BEAT THE BEST SCORE THEN YO
U WIN."
```

Now the display is to be held on the screen until the player has read the rules and then pressed any key (but not RUN/STOP!).

### PROGRAM 5.3 (cont'd)

```
110 PRINT"<2DCRSR><4RCRSR>PRESS ANY KEY
TO BEGIN"
120 GET A$:IF A$=""THEN 120
125 PRINT "<CLR>"
```

The next part is to set up a loop controlling the number of times that the person's reaction will be tested. This is arbitrarily set at ten times, although you may change this if you wish.

### PROGRAM 5.3(a)

```
130 REM START OF REACTION LOOP
140 FOR X=1 TO 10:PRINT"<CLR>"
```

The number of times that we change the screen's color will be random, so that the delay before the tests will not always be the same. The random command in Program 5.3(b) dictates that the colors will change at least once or at the most ten times prior to the test.

PROGRAM 5.3(b)

```
150 FOR Y=1 TO INT(RND(1)*10)
```

The color to which we change the border and background colors will also be random. There will be one random value for the border color and another random value for the background color, thus ensuring that they won't always be the same.

PROGRAM 5.3(c)

```
160 B1=INT(RND(1)*10)+1  
170 B2=INT(RND(1)*10)+1
```

The random value is between '1' and '10' inclusive; this gives an acceptably wide color range and includes all the major colors.

After every POKE of color code values (B1 and B2) into memory a small delay loop is required. This enables the user to recognize the change of color before the next change. Not only does it make a good and effective display, but it makes the time between each test longer: although, hopefully not long enough to become tedious or slow. For this purpose, a delay loop of 1 to 250 will be used. If this delay is too slow or fast for you, then feel free to change lines 190 and 210.

PROGRAM 5.3(d)

```
180 POKE 53281,B1  
190 FOR DELAY=1 TO 250:NEXT DE  
200 POKE 53280,B2  
210 FOR DELAY=1 TO 250:NEXT DE  
220 NEXT Y
```

Although we have called our loop 'DELAY' lines 190 and 210 end with:

```
'NEXT DE'
```

This is because the Commodore 64 looks only at the first two characters of a variable. The final line (220) completes the loop 'Y' which controls the number of times the colors change.

If we run this program as it stands, we will have some idea of what the final game is going to look like.

When the colors have stopped flashing, it's time to tell the player to press a key. This will be done by the use of a large 'GO' drawn on the screen. Before that, however, we need to test whether the keyboard buffer is empty. If, while we were watching the screen change color, we accidentally pressed a key (we wouldn't really cheat, would we?) or were accidentally to hold down a key, then the keyboard buffer would not be empty. The way to check this is to use the GET command to say 'GET a character and if it is something other than a null string ("") then go back and GET another character'.

This is demonstrated in Program 5.3(e) which also gives us the commands to draw our large 'GO', which is constructed by means of multiple use of cursor controls on one line. This allows us to draw a large construction in a relatively small number of lines. Line 235 ensures that the screen color will not be the same as our 'GO' (because we would not be able to see it). This is done by POKEing a '1' into 53281, making the screen color white. If this color is not suitable for you, then feel free to change it, only try to avoid it clashing with the color of the GO message, or you won't be able to see it.

#### PROGRAM 5.3(e)

```
230 GET A$:IF A$<>""THEN 230
235 POKE 53281,1
240 PRINT"<CLR><6DCRSR>";TAB(7);"<RVSON
>■■■■■<6LCRSR>■<DCRSR><LCRSR>■<DCRSR>";
242 PRINT"<LCRSR>■<DCRSR><LCRSR>■<DCRSR>
<LCRSR>■<DCRSR><LCRSR>■";
245 PRINT"<DCRSR><LCRSR>■■■■■<UCRSR><LC
RSR>■<UCRSR><2LCRSR>■■■<4UCRSR>";
250 PRINT"<3RCRSR>■■■■■<DCRSR><LCRSR>■<
DCRSR><LCRSR>■<DCRSR><LCRSR>■";
252 PRINT"<DCRSR><LCRSR>■<DCRSR><LCRSR>
<DCRSR><6LCRSR>■■■■■<6LCRSR>";
254 PRINT"<UCRSR>■<UCRSR><LCRSR>■<UCRSR>
<LCRSR>■<UCRSR><LCRSR>■";
256 PRINT"<UCRSR><LCRSR>■<UCRSR><LCRSR>■
<7DCRSR>"
```

Once the message has been displayed, we can now accept a legitimate input. Prior to accepting an input, the jiffy clock, TI, must be set to zero.

### PROGRAM 5.3(f)

```
260 TI$="000000"  
270 GET A$:IF A$=""THEN 270
```

After accepting our input, another reading of the clock, 'TI', must be made. This will be stored in 'E' (for End), giving us our reaction time. This time, however, is in 60th's of a second; to convert we divide by sixty. The measurement of time for our reaction tester needs to be accurate to two decimal places. The effect is obtained, as before, by multiplying by 100, finding the integer value and dividing by 100 afterwards, giving us our reaction time to two decimal places.

### PROGRAM 5.3(g)

```
280 E=TI/60  
290 E=E*100:E=INT(E):E=E/100
```

Now we have your response time in seconds. The next step is to display the time and then memorize it. It would be very simple to add the time onto a total but, for the final stage of our reaction tester, we will need to know each individual time, so we can calculate the quickest and slowest. To be able to keep each of the ten reaction times we will use a very special type of memory location, called an array.

## Arrays

An array is a series of related items; for example, if we had ten reaction times they are all related because they are reaction times! So these times can be labelled under one heading, 'T' for time. So all our times can be stored under the collective heading of 'T', the first time could be T(1), the second T(2), etc. Program 5.4 demonstrates the array T(X).

### PROGRAM 5.4

```
1 FOR X=1 TO 10  
2 T(X)=X  
3 NEXT X  
4 FOR X=1 TO 10  
5 PRINT T(X)  
6 NEXT X
```

Lines 1 to 3 set up a loop to give the array variable 'T' ten values, T(1)=1, T(2)=2, etc, until T(10)=10. Then lines 4 to 6 print out the values of array T. Quite logically, if we wanted twenty values in array T, we simply increase the loop value. Change line 1 to

```
1 FOR X=1 TO 20
```

then run it again - you will get a 'BAD SUBSCRIPT ERROR'. This tells us we are trying to print more array values of T than there are. If we don't tell the computer how many values of T there will be, then only ten are assumed. So we can't print the eleventh value of array T because there isn't one.

The command that tells the computer how many array values we need is called ....

## DIM

The DIM command, (DIM is short for DIMension) tells the computer to allocate a given number of array locations to a certain variable.

e.g.                DIM T(20)

will reserve twenty locations for the array T. Now if we change Program 5.4 to Program 5.4(a) it will work properly:

### PROGRAM 5.4(a)

```
1 DIM T(20)
2 FOR X=1 TO 20
3 T(X)=X
4 NEXT X
5 FOR X=1 TO 20
6 PRINT T(X)
7 NEXT X
```

One thing to notice is, if we DIM an array with, say, fifty values, these will automatically be filled with zeros. Run Program 5.4(b) just to prove it.



PROGRAM 5.4(b)

```
1 DIM T(50)
2 FOR X=1 TO 50
3 PRINT T(X)
4 NEXT X
```

You can now delete lines 1 to 7.

Arrays can also be used with strings. We simply call them T\$(X) instead of T(X) for example, for our reaction game will have an array to store the player's score and also another array storing the top five player's names. These two arrays will be called MN\$ and TN figure 5.1 shows how the arrays will be arranged.

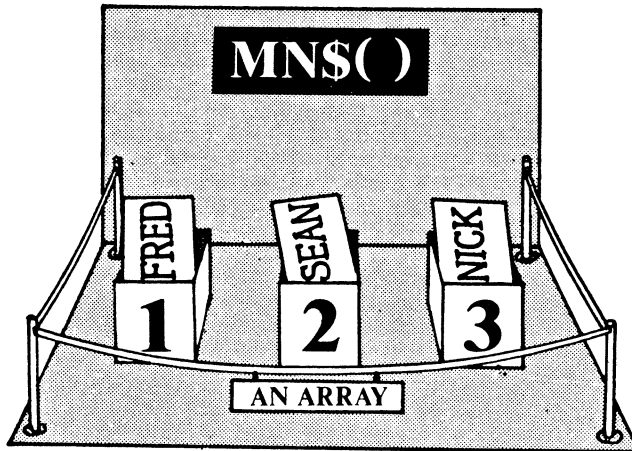


FIGURE 5.1  
The array diagram

We need to sort our scores inside the the array into numerical order, i.e. the top score will be the first value of TN( ) and the scorers name will be the first value of MN\$. Program 5.4(c) demonstrates this.

### Program 5.4(c)

```
1000 REM ARRAY SORT
1005 DIM MN$ (5), TN(5)
1010 INPUT "NAME"; N$:INPUT"SCORE";N
1020 FOR X=1 TO 5
1030 IF TN(X)>N THEN U$=MN$(X):U=TN(X):
MN$(X)=N$:TN(X)=N:N$=U$:N=U
1040 NEXT X
1050 FOR X=1 TO 5
1060 PRINT MN$(X),TN(X)
1070 NEXT X
```

Line 1030 may be a bit difficult to understand! If the value of TN(X) is greater than our inputted score we will need to replace it with 'N' and then change TN(X) into N. The same applies to MN\$(X) and N\$. To do this an intermediate variable is used. U\$ is set to MN\$(X) and U is set to TN(X). Then MN\$(X) and TN(X) become N\$ and N respectively. Last but not least, N\$ is set to U\$ and N is set to U (the old values of MN\$(X) and TN(X)). When you run the program the first time, both arrays MN\$ and TN are empty, so provided you input a positive number, your name and score will be stored in MN\$(1) and TN(1). Experiment with this program a few times.

When you've finished playing with program 5.4(c), you can delete it, as it is not the version that will be included in the reaction tester game. However, it does demonstrate the sorting technique that will be used in the game. Now back to the reaction tester....

### PROGRAM 5.5

```
300 PRINT TAB(10);"<2DCRSR>THAT TOOK";E;
" SECONDS"
310 T(X) = E
320 FOR DELAY=1 TO 1000:NEXT DE
330 NEXT X
```

There is a delay here (320) so the player can read the display before the 'X' loop which controls the number of tests is closed using a NEXT statement (330). Line 310 stores the time taken in our time array (T()).

Once the loop has been completed, i.e. all ten reaction tests have taken place, the player is to be presented with the average score for all the tests, and his shortest and fastest scores, the values of 'BB' and 'SS' are to be set at the start of each test.

PROGRAM 5.5(a)

```
115 BB=0:SS=99
340 PRINT"<CLR> THAT WAS YOUR LAST GO"
350 FOR X= 1 TO 10
360 TT=T(X)+TT
370 IF T(X)>BB THEN BB=T(X)
380 IF T(X)<SS THEN SS=T(X)
390 NEXT X
400 PRINT"YOUR TOTAL TIME WAS:";TT;"SECO
NDS"
410 PRINT"YOUR FASTEST TIME WAS:";SS;"SE
CONDS"
420 PRINT"YOUR SLOWEST TIME WAS:";BB;"SE
CONDS"
```

After a small delay a check is made to see whether we have a new best score, the lowest score being the best. Here we make use of a similar routine to that used in Program 5.4(c)

PROGRAM 5.5(b)

```
430 FOR X= 1 TO 5
440 IF TN(X)>TT THEN F=1
450 NEXT X
460 IF F<>1 THEN 560
470 PRINT TAB(13);"<2DCRSR><RVSON><RED>
NEW BEST SCORE<RVSOFF><BLUE>"
480 INPUT"<DCRSR><4RCRSR>WHAT IS YOUR NA
ME ";A$
490 A$=LEFT$(A$,8)
500 FOR X=1 TO 5
510 IF TN(X)>TT THEN U$=MN$(X):
U=TN(X):MN$(X)=A$:TN(X)=TT:A$=U$
520 NEXT X
530 FOR X = 1 TO 5
540 PRINT TAB(15);MN$(X);TN(X)
550 NEXT X
```

A point to note is that the arrays TN( ) and MN\$( ) are not DIMed at the start of the program (or anywhere else for the matter) This is because we use no more than the allowable ten array values. If we used more than 10 then the arrays would have to be DIMensioned. However, we need to set them with large values at the start of the program so that the first time will be the shortest, Program 5.5(c).

PROGRAM 5.5(c)

```
2 FOR X=1 TO 5
3 TN(X)=99:MN$(X)="X"
4 NEXT X
```

All that is left now is to ask the player if (s)he requires another go. If yes, then we need to reset TT (the score total), and F(the new best score flag), to zero and then the game will loop back to line 110, thus avoiding printing the rules, or resetting the best score variable.

PROGRAM 5.5(d)

```
560 PRINT"<4DCRSR><4RCRSR>DO YOU WANT
ANOTHER GO?(Y/N)"
570 GET A$:IF A$="" THEN 570
580 IF A$="Y" THEN F=0:TT=0:GOTO 110
590 PRINT "BYE":END
```

Now you have a fully functional computer reaction tester game with which to delight and amuse your friends!

Another reaction program could ask the user to press a particular key rather than any key. Apart from anything else, this would be good practice on learning the layout of the keys.

One way this new game could be developed is to have all the letters on DATA statements, much in the same way as we choose our words in the Hangman game.

PROGRAM 5.6

```
10 R=INT(RND(1)*26)+1
20 FOR X=1 TO R
30 READ A$
40 NEXT X
50 PRINT A$
1000 DATA "A","B","C","D",...etc..."X","
Y","Z"
```

When run, Program 5.6 will print a random letter. This small routine will be a very useful way to get a random letter, but there is a much better way and that is to use the CHR\$( ) command.

## CHR\$( )

Every key on the Commodore keyboard has a special code which the computer uses to identify that character.

Try


```
PRINT CHR$(47)
```

and you should get '/'. That's because the CHR\$( ) command tells the computer to print the character that corresponds to the CHR\$( ) value of '47'.

All the symbols have their own code number. For instance, can you see what this does? -

```
PRINT CHR$(13)
```

Well, 13 is the code number for the RETURN key. As your computer can't display 'RETURN' without being told to PRINT it, it has actually done a RETURN instead. This makes the computer start a new line.

There is a full list of the CHR\$( ) values in Appendix 3 and it is quite interesting to see what some of them are. Here are a few examples, but first make sure you are in lower case mode by pressing SHIFT and .

```
PRINT CHR$(142)
```

Now we are in upper case mode. Try this:

```
PRINT CHR$(158)
```

and the cursor has become yellow.

So the CHR\$( ) command can be used to find our random letter. Also we need to find a random number between 1 and 26, and this time we simply print the CHR\$( ) value to get our letter.

### PROGRAM 5.7

```
10 R=INT(RND(1)*26)+1  
20 PRINT CHR$(R)
```

Well! What went wrong there? Why didn't it print a letter? Although the computer printed the CHaRacter responding to the random number, it wasn't a letter. If we run Program 5.7 again, we still won't get a letter. This is because the CHR\$( ) value of the letters is greater than '26'. So what value do letters start at? An easy way of finding out it to ASC( ).

### ASC()

The ASC( ) command is the opposite of CHR\$( ). Whereas CHR\$( ) converts a number into the appropriate character, ASC() converts a character into the appropriate number. For example ...

```
PRINT ASC("/")
```

The result will be '47', and if you remember, that was the character number we printed earlier. So if we tell the computer to print the ASC( ) value of 'A', then we will know where the letter characters start. You must always remember to enclose the character in quotes when using ASC( ).

```
PRINT ASC("A")
```

This should respond with '65'. So clearly the letters start at '65'. Just to test this, type in Program 5.7(a).

### PROGRAM 5.7(a)

```
10 FOR X=65 TO 91
20 PRINT CHR$(X)
30 NEXT X
```

When run, this program will print the letters 'A' to 'Z' thus proving our theory.

Now that we understand how to use these commands, we can begin to develop our program, or rather ... you can. The program is left for you to create. The structure will be similar to that of the Reaction Tester program but, before you begin, here's a little nudge in the right direction.

### PROGRAM 5.7(b)

```
10 R=INT(RND(1)*26+65)
20 PRINT "PRESS THIS KEY";CHR$(R)
30 GET A$:IF A$="" THEN 30
40 IF ASC(A$)=R THEN PRINT "VERY WELL DO
NE": STOP
50 GOTO 30
```

# CHAPTER

# 6

## PART ONE

### User-Defined Graphics

**B**y using the Commodore 64's user-defined graphics it is possible to define a character of your own design and then call it up in programs as and when it is required. Once defined, this character can be treated as any other and it answers to the name that you give it, a name that can be any one of the characters on the keyboard or anywhere in the character set. The character you define is made up from an 8x8 matrix, of little squares known as pixels. These are set to either full or empty by reference to the patterns stored in the character data which is stored in the ROM. The computer could actually store the data as a series of 64 separate memory locations each of which is set to 'on' or 'off', thus defining the matrix as shown in Figure 6.1.

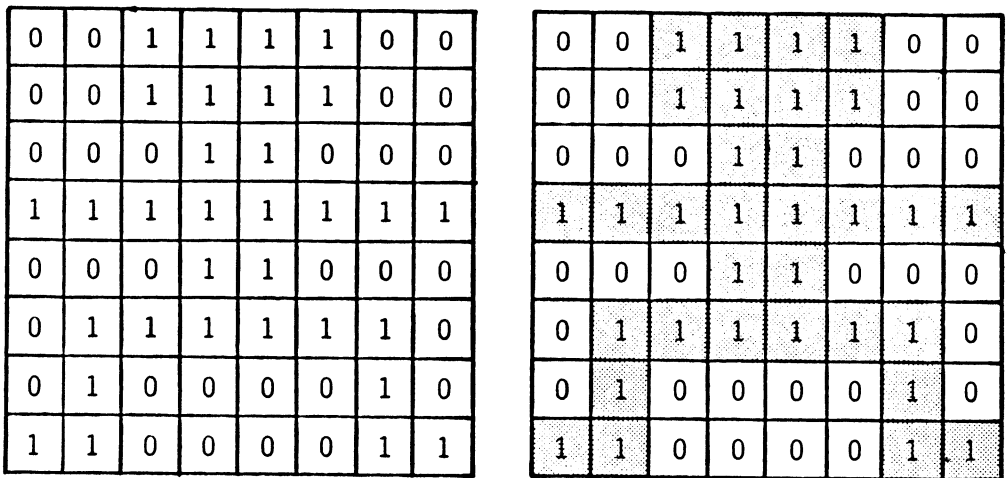


FIGURE 6.1

Defining a character by using individual bytes for each pixel would use up 64 memory locations for each character and, by the time a few characters have been stored, they start to gobble up a lot of the computer's memory. However, help is at hand in the way the computer stores data generally!

The microprocessor chip on which the C-64 is based - the 6510 - is referred to as an 8-bit chip, which means that each character is stored in memory as eight 'bits' of data. This block of eight bits is known as one 'byte'. As the chip is a 'digital' microprocessor, each of the bits of data can be set only to 'on' or 'off', i.e. a '1' or a '0'. Thus, only one byte is needed to store each of the rows for each character shapes, so a total of 8 bytes are needed to hold the entire shape of each of the C-64's characters.

The method of cramming up to eight bits of character information into the byte simply takes account of binary notation (see Appendix I). When using this technique it is necessary to add up the elements of the character bit by bit, to end up with the number to be POKEd into memory. Thus, to store the top line of the dancing man shown in Figure 6.1, the addition sum shown in Figure 6.2 is performed.

$$\begin{array}{cccccccc}
 (0 \times 128) + (0 \times 64) + (1 \times 32) + (1 \times 16) + (1 \times 8) + (1 \times 4) + (0 \times 2) + (0 \times 1) & = & 60 \\
 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & = 00111100_2
 \end{array}$$

FIGURE 6.2

From Figure 6.2 the elements of the row add up to:  $32+16+8+4=60$ . Thus the number '60' must be POKEd into memory. To obtain POKE values for the whole figure, each row must be considered separately, as shown in Figure 6.3.



BINARY								CALCULATION	DECIMAL
128	64	32	16	8	4	2	1		
0	0	1	1	1	1	0	0	$32+16+8+4$	= 60
0	0	1	1	1	1	0	0	$32+16+8+4$	= 60
0	0	0	1	1	0	0	0	$16+8$	= 24
1	1	1	1	1	1	1	1	$128+64+32+16+8+4+2+1$	= 255
0	0	0	1	1	0	0	0	$16+8$	= 24
0	1	1	1	1	1	1	0	$64+32+16+8+4+2$	= 126
0	1	0	0	0	0	1	0	$64+2$	= 66
1	1	0	0	0	0	1	1	$128+64+2+1$	= 195

FIGURE 6.3

Having calculated the values needed to define a character, these values must be stored in the area of memory where the VIC II chip looks for its characters. As this is a fairly complicated process, it's easiest to use a utility program to help. Part Two of this chapter develops such a utility and during its development, the whole process of defining characters will be explored.

Before you set out to develop a program of this complexity, it is a good idea to LOAD and RUN Honey.Aid. In addition to using Honey.Aid as a tool, we will incorporate a Honey.Aid command into the Char.Gen program, and thereby speed it up considerably.

## PART TWO

### A Graphics Utility: Specification

This utility will provide a facility for the creation of a user-defined graphics character. The character will be designed on a 'screen' or matrix which represents a single character about eight times full size, see Figure 6.4, and the actual form of the character will be displayed during the design process. By moving a block, , which represents a single pixel, around the character matrix and storing it where desired, a character will be built up, its POKE values calculated and stored in RAM. Provision will be made for storing these POKE values in DATA statements for incorporation into the user's program. To facilitate its use along with the user's own program, the graphics utility program will be numbered from line 60000 onwards.

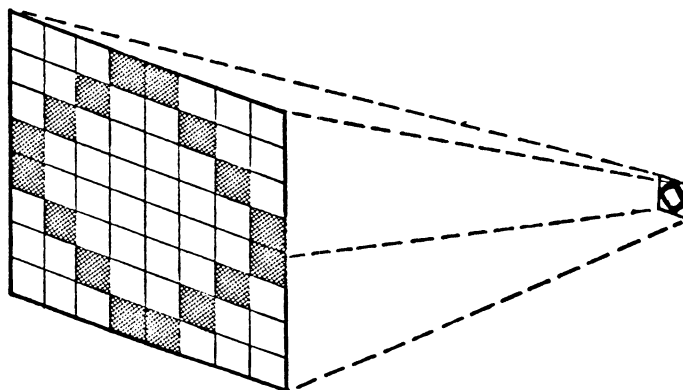


FIGURE 6.4

#### Relocating the character set.

The C-64 comes with its own ready-drawn character set - these are the characters that you see on the screen when you press the keys. These, however, are stored in ROM, or Read only Memory, and it isn't possible to change them. Thus, if you want to define a new character set, then it is first of all necessary to move the existing set into RAM where you can operate on it. As each character occupies 8 bytes and there are 256 different characters in the upper case graphics character set to be moved, 8 x 256 or 2048 bytes need to be re-located. In the ROM, the standard characters are located from 53248 (\$D000) onwards and one recommended place to move these to is 12288 (\$3000). A fairly straight-forward PEEK/POKE routine would appear to perform this operation i.e.

```
FOR I=0 TO 2047:POKE(12288+I),PEEK(53248+I):NEXT
```

However, because of the way the various chips inter-relate, a few other things need to be done before this removal process can be carried out.

### Protecting the top of memory

Once this is done, a copy of the character set exists at the top of the RAM. Of course, this is quite likely to be overwritten by any program that tries to use the same piece of RAM. This problem can be overcome by persuading the C-64 that RAM ends lower down than it really does. A switch-on the computer checks through the RAM and calculates just how much memory is available. Once the calculation is done, the location of the top of memory is recorded in memory locations 55 and 56. If these values are fixed and set to a lower figure the C-64's BASIC thinks that memory ends earlier and doesn't overwrite your character set. Before moving down the top of memory, just do a little check on what memory is available by entering:

```
PRINT FRE(0)-(FRE(0)<0)*64*1024
```

The machine will then respond with 34618 if Honey.Aid is in (otherwise 38909).

Next move down the memory by means of:

```
POKE 55,0: POKE 56,48
```

Once this is done, check memory availability again with another 'PRINT FRE(0)-(FRE(0)<0)\*64\*1024' and you should find the new value is 10237 which is some 24406 (28671) less.

Now, let's investigate the workings of the keyboard a little, in particular the:

### Keyboard Buffer

When any data is entered into the computer via the keyboard, the C-64 automatically puts it into the keyboard buffer, which is ten bytes of memory located from 631 to 640. In order to keep track of what's in this buffer, one other byte located at 198 is used. Just to investigate the operation of this buffer, type in Program 6.1.

## PROGRAM 6.1

```
2 PRINT "<CLR>";
5 T$=CHR$(34)+CHR$(20)+"<RVSON>T<RVSOFF>"
+CHR$(34)+CHR$(20)
6 M$=CHR$(34)+CHR$(20)+"<RVSON>M<RVSOFF>"
+CHR$(34)+CHR$(20)
10 PRINT:PRINT"<HOME>";CHR$(34);:FOR X=0
TO 9
20 IF PEEK(X+631)=20 AND X<10 THEN PRINT
T$;:NEXT:GOTO 10
30 IF PEEK(X+631)=13 AND X<10 THEN PRINT
M$;:NEXT:GOTO 10
40 PRINT CHR$(PEEK(X+631));:NEXT:PRINT C
HR$(34):PRINT PEEK(198);:GOTO 10
```

When you RUN this program, it will print in the top left-hand corner of the screen the current contents of the keyboard buffer. Underneath this it will print what value is stored in 198, i.e. the number of characters that the C-64 thinks are in the buffer. Why is this a zero? Well, at turn on and various other times, the buffer gets filled up with garbage that just shouldn't be there. What location 198 tells you is the official story. The C-64 will only read the number of characters that location 198 tells it are there, the rest it ignores.

To watch the buffer in action run Program 6.1 and while it is running type in FREDFREDFRED and you will see that the buffer now contains:

```
"FREDFREDFR"
```

and the line below tells you that 198 now contains the value 10.

Next, type in SID and see what happens. Anything happened? No, nothing happens because the buffer is full. It is a First In, First Out device which means that the first item put in i.e. the 'F' of the first 'FRED' is maintained ready for retrieval.

To investigate the buffer further, press RUN/STOP and then modify Program 6.1 to do the following

- \* accept the first 10 keyboard entries
- \* perform a 'GET'
- \* store the result in A\$
- \* display the first character to come out of the buffer.

## PROGRAM 6.1(a)

```
45 IF PEEK(198)=10 THEN GETA$:PRINT"<4DC
RSR>"A$
50 GOTO 10
(remove the GOTO 10 on line 40)
```

When you RUN this, repeat the above procedure by attempting to type in three 'FRED's. As you enter the 'E' of the third 'FRED', the buffer will fill i.e. 198 contains a '10'. However, immediately afterwards, the 'GET' will be performed and a character taken from the buffer to be stored in A\$ and PRINTED on the screen. Thus, the letter 'F' will be taken from the buffer and the remaining 9 characters moved along one to the first 9 positions. When the next letter 'E' is entered, it will be placed initially in location 10 of the buffer, only to be shuffled forward as an 'R' is extracted by the GET. You may now stop the program by pressing RUN/STOP, but do not erase the program yet.

## Addressing the keyboard

Before we can re-locate the character set, one other formality is needed, a necessity because of the important role that the keyboard plays. As it forms the main communications channel between the user and the operating system, the keyboard is scanned continuously. This scanning must be turned off or disabled before copying the character set. Scanning is turned off by setting bit zero of 56334 to zero and re-enabled or turned on by setting this bit to one. One problem with this process is that it is only bit zero of 56334 that needs to be changed with the remainder staying as it was. To do this without a lot of PEEKing and POKEing we may make use of :

## LOGICAL OPERATORS (a digression)

**AND** - Perform a logical AND

It may be clearer if we first examine an electronic AND gate - your C-64 is just full of them! What an AND gate does is to accept two electrical signals, compare them and produce an output based on this comparison. Figure 6.5 shows such a device with inputs 1 and 2 and output OP.



AN ELECTRONIC AND

FIGURE 6.5

In operation, this gate looks at IP1 and IP2 and, if they are both set at 5 volt, then OP is set at 5 volt. However, if either or both IP1 and IP2 are set at zero volts, the OP is set at zero volts. In computer terms, the 1 state is referred to as 'TRUE' and the zero as FALSE so the AND gate rules read: the output of a gate is TRUE if input 1 AND input 2 are TRUE. For all other cases, the output is FALSE or zero.

It is convenient to express these states in a TRUTH TABLE, see Fig. 6.6.

IP1	IP2	OP
0	0	0
0	1	0
1	0	0
1	1	1

TRUTH TABLE FOR  
ELECTRONIC AND

FIGURE 6.6

To use this, read along the line that has the requisite states of IP1 and IP2 and column 'OP' will then give the logical state of the output, 'OP'.

For instance, taking a value of IP1=0 and IP2=1, line 2 gives the state of OP as '0' (FALSE).

#### OR Perform a logical OR

As with 'AND', this operator performs a logical bit-by-bit comparison between the two pieces of data. What this means is that each bit of the data is tested and if either one OR the other is equal to 1, then a TRUE position obtains and the resultant bit is set to 1.

This is represented on the truth table, Figure 6.7. On this the two possible inputs are labelled IP1 and IP2 while the output is labelled OP.

This is represented on the truth table, Figure 6.7. On this, the two possible inputs are labelled IP1 and IP2 while the output is labelled OP.

IP1	IP2	OP
0	0	0
1	0	1
0	1	1

TRUTH TABLE FOR  
LOGICAL OR

FIGURE 6.7

### Logical operators and eight bits

So far, all the operators have been shown operating upon single bits of data. However, when logical operations are carried out on an eight bit number, each bit of that number is treated individually. Take for example the situation where  $100_{10}$  ( $01100100_2$ ) is ORed with  $50_{10}$  ( $00110010_2$ ). The process is simply one of ORing bit by bit or, as it is otherwise known, bitwise operation. Figures 6.8 and 6.9 show the stage-by-stage process where first of all the first zero of 100 is ORed with the first zero of 50 to yield a zero. Next the second character of the 100 (a zero) is ORed with the second character of the 50 (a one) to yield a 1. Figure 6.9 shows the various stages of the process along with the results.

Thus, when 100 is ORed with 50 the result is:

$$\begin{array}{r}
 \phantom{OR} 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \quad = 100_{10} \\
 OR \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \quad = 50_{10} \\
 \hline
 = 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \quad = 118_{10}
 \end{array}$$

FIGURE 6.8

Decoding this by means of the truth table, Figure 6.7, starting with bit 0 gives:

```

bit 0 : 0   ORed with 0   gives  0
    1 : 0   ORed with 1   gives  1
    2 : 1   ORed with 0   gives  1
    3 : 0   ORed with 0   gives  0
    4 : 0   ORed with 1   gives  1
    5 : 1   ORed with 1   gives  1
    6 : 1   ORed with 0   gives  1
    7 : 0   ORed with 0   gives  0

```

i.e. 0 1 1 1 0 1 1 0

FIGURE 6.9

Back to the project ...

As we were saying! The keyboard is the main means of communication between the user and the computer and it is necessary at times to break the link.

One memory location - 56334(\$DCOE) - contains one bit of data - bit 0 - that turns the keyboard ON when it's set to 1 and OFF when it's set to zero. To set it, two of the logical functions, AND and OR are used.

Firstly, to set just one bit of a byte to zero we AND it with a byte that contains all 1's except at the location to be set. Thus, when that bit is ANDed with the zero present it is automatically switched to a zero. Hence to set bit zero to zero, we AND the byte with a  $254_{10}$  ( $11111110_2$ ) i.e.:

```

      this  1  0  1  1  0  0  1  1  1
ANDed with 1  1  1  1  1  1  1  1  0
      gives 1  0  1  1  0  0  1  1  0

```

FIGURE 6.10

If you're unsure of the process, substitute any number you wish for the top number and you'll find that it will remain unchanged except for bit zero which will be switched to a zero.

Putting this into practice to turn off the keyboard is:



## PROGRAM 6.2

```
X = PEEK(56334)
Y = X AND 254
POKE 56334,Y
```

or, putting it into one line:

```
POKE 56334,(PEEK(56334)AND 254)
```

To turn the keyboard on again or to 're-enable' it, bit zero needs to be re-set to a '1'. In this case, the byte is ORed with a 1 i.e. seven zeros and a 1 in bit zero. This process will leave all the bits ORed with the zeros as they were but set bit zero to a '1'. Figure 6.11 shows this process in action:

this	1	0	1	1	0	0	1	1	0
ORed with	0	0	0	0	0	0	0	0	1
gives	1	0	1	1	0	0	1	1	1

FIGURE 6.11

Thus, the two lines to turn the keyboard off and then on again are:

```
POKE 56334,PEEK(56334)AND 254   Turn off keyboard
POKE 56334,PEEK(56334)OR 1     Turn on keyboard
```

Our short demo programs (6.1 and 6.1a) can help to illustrate that process too if we set it to turn off the keyboard when, say 3 characters have been inputted. A slight problem might arise here though, as, once the keyboard is turned off there is no easy way remaining to talk to the computer. Therefore, the turning-back-on procedure must be programmed BEFORE the keyboard is disabled. Lines 60 and 70 perform this ON/OFF function, with a suitable delay between.

PROGRAM 6.2(a) (add these lines to the program 6.1 and 6.1a)

```
47 IF PEEK(198)=3 THEN 60
60 POKE 56334,PEEK(56334)AND 254:PRINT"<
6DCRSR> KEYBOARD OFF":FOR X=1 TO 1000
70 NEXT:POKE 56334,PEEK(56334)OR 1:PRINT
"<UCRSR>     KEYBOARD ON":FOR X=1 TO 1000:NEXT
80 GOTO 10
(delete line 50)
```

Run this program and watch what occurs as you type in keyboard characters. You may then save this program and erase the memory (type in NEW).

## The VIC II CHIP and the 6510

The next task to be tackled is the relocation of the character set and to do this, the role of the VIC II chip needs to be examined. The C-64 contains many chips including the two that are very much concerned with day-to-day operations, the 6510 and the VIC II chip. The 6510 is the main microprocessor in the C-64 while the VIC II chip handles all of the video outputs to the screen, leaving the 6510 free to get on with its own tasks. Many functions call for the interaction of one of these with the other and this is particularly true during the translation of a command into an actual screen display. To understand how this works, it's necessary to look into the operation of the ROM. This is the program that is built into your machine and, in effect, makes the machine work. It has to run this program pretty quickly and so it's not written in BASIC but in the language that the chip itself uses, machine code.

Let's take a sample statement and see how the 6510 handles it. Take the statement:

```
PRINT "A"
```

This will cause the C-64 to print an 'A' at the next cursor position.

The first thing that the 6510 must do is to work out what 'PRINT' means. As this is a command word, it is not stored in memory as five separate letters but as a 'token', a one byte number that is the code for 'PRINT'. Thus, the 6510 sees 'PRINT', works out what its token is and then looks through the ROM to find that token. When it finds it, in a table, it finds an address there too, the address being the start address of the machine-code program which brings about a 'PRINT'.

It then jumps to this address and starts to run the machine-code program stored there. The first task assigned by this program is to find out where to print, i.e. to look up in memory what the current cursor position is. Next it looks at what comes after the PRINT and, on finding a quote sign ("), takes the 'A' and uses a special algorithm to convert it to screen code which turns out to be '1' in this case. The 6510 now puts this '1' into the appropriate screen memory location. It must then look at what the current character color is and insert the appropriate code for it into the color RAM area.

Having carried out all these tasks, the cursor position must be updated, i.e. moved one position to the right, and the 6510 is then free to look for the next job to do and simply carries on.

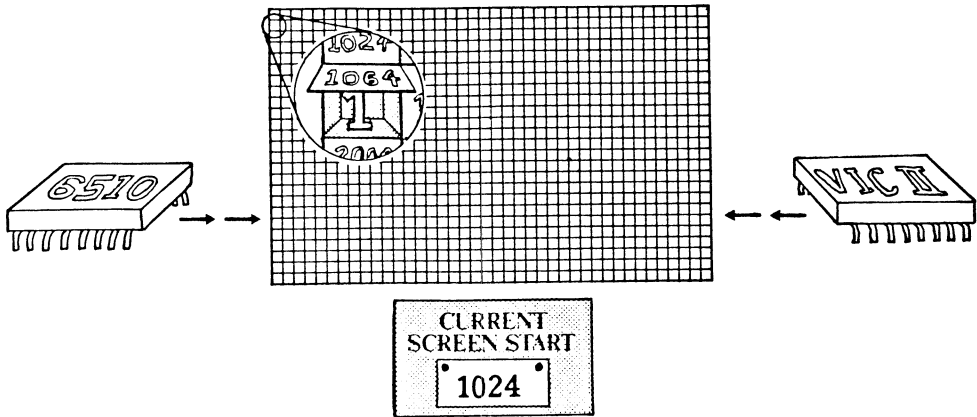
However, nothing has got printed onto the screen yet, all the 6510 has done is to update the screen's RAM area. This is where the VIC II chip comes in!

The VIC II chip carries out many screen control functions and is constantly updating the screen's contents. In fact, it is constantly scanning through the screen RAM memory and looks, in turn, at the contents of each individual screen memory location. When, in our example for instance, it looks into location 1064, it finds a '1'. This it must translate into an appropriate screen image with shape and color.

Before it can find the appropriate shape data, the VIC II chip must find out the address of the character shape data. To do this it needs to refer to two memory locations: the first of these is at 56576 (actually a register of one of the CIA chips) and the second at 53272 which is one of the VIC II chip's own registers. Once this address is found, it can look through the data stored there and pick out the appropriate pixel pattern. Next it must find the color data for that character and this it simply looks up in the color RAM, which is always stored in the same place.

Having acquired all the necessary data, the VIC II chip can go ahead and display the 'A' on the screen. Figure 6.12 illustrates this entire process.

6510/VIC II INTERACTION  
DURING EXECUTION OF A  
PRINT "A"



What 6510 does:

- \* find token
- \* find subroutine
- \* find cursor
- \* find quotes
- \* calculate screen code for 'A'
- \* store 'A's' code

What VIC II does:

- \* look in 1064
- \* find pixel data for '1'
- \* find color data for 1064
- \* produce pixel pattern
- \* display on screen

FIGURE 6.12

Normally, when the 6510 looks at the memory in 53248 to 57343 (\$D000 to \$DFFF) it sees the Color RAM (at \$D800) and the so called Input/Output devices i.e. the VIC II chip (at \$D000), the SID (Sound Interface Device) chip (at \$D400) and the two CIA (Complex Interface Adaptors) chips, (at \$DC00 and \$DD00). However, the Character ROM is there also and we need to arrange for the 6510 to 'see' the Character ROM instead of the Color RAM and I/O devices. This is quite easily achieved by changing bit 2 of memory location 1. This is normally set to 1 which allows the 6510 to see the Color RAM and I/O devices. If bit 2 is changed to a zero, however, the 6510 sees the Character ROM in this space.

Before we rush into doing this, we need to consider what would happen to the C-64 if the 6510 can no longer see the Color RAM and the I/O chips. The only real problem lies with the two Complex Interface Adaptors (CIA's). These chips handle all the Input/Output, from and to the keyboard, the tape cassette reader (if any), the disk drive (if any) and the RS232 (if any). If the CIA's cannot be seen by the 6510 then no input or output to these devices can be handled.

At first sight, this doesn't appear to be a problem. Surely, if we make sure that the Char.Gen program is not inputting or outputting to the keyboard, tape, disk or RS232 while we are copying the character ROM to RAM, then there isn't a problem, is there? Well, yes, the keyboard is the trouble. The 6510 checks the keyboard every one sixtieth of a second to see if any key has been depressed. In order to check the keyboard, the 6510 needs to be able to see the two CIA's which it can't do if we've set bit 2 of memory location 1 to a zero. It would see the character ROM, and treat the character it found as an instruction from the keyboard - disaster!

Keyboard checking normally goes on all the time, the program which the C-64 is currently running being interrupted in order to do this. We need to stop this interruption temporarily, therefore, while we use the 6510 to copy the Character ROM. Fortunately, this is quite simple. What we do is stop the system clock which measures the sixtieths of a second and this stops the keyboard scanning routine from interrupting. To stop the clock we must change bit zero of memory location 56334 (\$DC0E) to a zero. To restart the clock, following the move we change the same bit to a one. Since this memory location is a register on one of the two CIA's, we can only change the value while the 6510 can see the CIA's.

So the plan of action works out to be:

- (1) Stop the system clock controlling the keyboard scan by changing bit zero of 56334 to zero.
- (2) Let the 6510 see the Character ROM instead of the CIA's by changing bit 2 of memory location 1 to a zero.
- (3) Copy the Character ROM to RAM, using the 6510.
- (4) Let the 6510 see the I/O chips again by changing bit 2 of location 1 back to a one.
- (5) Restart the system clock by changing bit zero of 56334 back to a one.

Putting that into the program:

#### PROGRAM 6.2(b)

```
Stop system clock      60030 POKE 56334,PEEK(56334)AND 254
Let 6510 see ROM &    60040 POKE 1,PEEK(1) AND 251: FOR I=0 TO
copy character        2047:POKE I+12288,PEEK(I+53248):NEXT
ROM to RAM
Restart system clock  60050 POKE 1,PEEK(1) OR 4
                     60060 POKE 56334,PEEK(56334) OR 1
```

Finally the VIC II chip needs to be told just where you are putting your new character set. To find where it is stored, it looks in location 53272, so to redirect the VIC II chip to the new location, type in:

```
60070 POKE 53272,(PEEK(53272)AND 240)+12
```

We shall be using two special characters in the character generator program and these are redefined during the initialization. The characters that we will redefine are those which represent 254 and 255 and are thus stored in:

```
12288+(254*8)=14320
12288+(255*8)=14328
```

The character representing 255 will be a totally filled-in square. i.e. The pixels on each line coincidentally add up to 255 also - see Figure 6.13.

1	1	1	1	1	1	1	1	= 255
1	1	1	1	1	1	1	1	= 255
1	1	1	1	1	1	1	1	= 255
1	1	1	1	1	1	1	1	= 255
1	1	1	1	1	1	1	1	= 255
1	1	1	1	1	1	1	1	= 255
1	1	1	1	1	1	1	1	= 255
1	1	1	1	1	1	1	1	= 255

FIGURE 6.13

Memory locations 14328 to 14335 are, thus, filled with 255's i.e.

```
FOR X=0 TO 7:POKE(12288+255*8)+X,255
```

Figure 6.14 shows the other character, 254. This one has some empty pixels represented by a zero (0).

1	1	1	1	1	1	1	1	= 255
1	0	0	0	0	0	0	1	= 129
1	0	0	0	0	0	0	1	= 129
1	0	0	0	0	0	0	1	= 129
1	0	0	0	0	0	0	1	= 129
1	0	0	0	0	0	0	1	= 129
1	0	0	0	0	0	0	1	= 129
1	1	1	1	1	1	1	1	= 255

FIGURE 6.14

Two different numbers need to be POKEd in to define this character, 255's into the zero'th and seventh locations and 129 into the middle six locations, i.e.

```
FOR X=1 TO 6:POKE(12288+254*8)+X,129
POKE(12288+254*8)+0,255
POKE(12288+254*8)+7,255
NEXT X
```

In use, the character generator program will be suitable for redesigning any character in the character set i.e. from 0 to 253 (254 and 255 being used in the Char.Gen program). Lines 60110 and 60120 ask for the value of the character which is to be defined and store this in the variable CH.

When the program so far is run, it will move the character set and ask which character is to be re-defined. When running this program, you may be struck by the interminable time taken to move the character set. However, once this has been done, it doesn't need to be done again. Indeed, it is important that it isn't done again. For if a character has been redefined then moving the character set again will destroy the redefinition. One way to handle this would be to ask whether or not the character set has been moved, using an INPUT or GET. However, a more subtle way of checking this exists as, during the initialization procedure, byte 53272 was set by loading in a '12', i.e. setting bits 3 and 2 to 1's. Thus, a check of this byte will tell whether or not the character set has been moved. What is needed therefore is a way of looking at the byte but ignoring all the bits other than 3 and 2. This can be achieved by ANDing the contents of 53272 with '12' and then checking whether the resultant value is 12. If you're not sure, check it below.

$$\begin{array}{r}
 \phantom{\text{ANDed with}} \phantom{12 \text{ equals}} \phantom{=} \phantom{12} \\
 \phantom{\text{ANDed with}} \phantom{12 \text{ equals}} \phantom{=} \phantom{12} \\
 \text{ANDed with } 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\
 12 \text{ equals } 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \quad = \ 12
 \end{array}$$
  

$$\begin{array}{r}
 \text{and,} \\
 \text{ANDed with } 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\
 12 \text{ equals } 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \quad = \ 12
 \end{array}$$

FIGURE 6.15

Whatever the number is that is ANDed with 12, any 1's other than those in the 3rd and 2nd bit are stripped off by the 12's zeros. This is frequently known as stripping bits off by means of an AND function.



Thus, a BASIC line is required which says: If the result of ANDing the number with 12 equals 12, then jump over the character moving sub-routine i.e.

```
60000 GOSUB 62000:IF PEEK(53272)AND 12=12 THEN 60080
```

The subroutine at 62000 is left upto you: it is for a title page should you wish to include one.

Adding all the various parts of the initialization procedure, gives the routine as in Program 6.3.

### PROGRAM 6.3

```
60000 GOSUB62000
60010 IF(PEEK(53272)AND12)=12THEN60080
60020 PRINT"<CLR><BLU><11DCRSR><2RCRSR>
        PLEASE WAIT WHILE CHAR SET MOVED"
        :POKE55,0:POKE56,48
60030 POKE53280,3:POKE53281,1:POKE56334
        ,PEEK(56334)AND254
60040 POKE1,PEEK(1)AND251:FORI=0TO2047:
        POKEI+12288,PEEK(I+53248):NEXT
60050 POKE1,PEEK(1)OR4
60060 POKE56334,PEEK(56334)OR1
60070 POKE53272,(PEEK(53272)AND240)+12
60080 FORX=0TO7:POKE14328+X,255:NEXT
60090 FORX=1TO6:POKE14320+X,129:NEXT
60100 POKE14320,255:POKE14327,255
60110 PRINT"<CLR><5DCRSR><BLU><2RCRSR>W
        HICH CHARACTER WOULD YOU LIKE"
60120 INPUT"<4RCRSR>TO DEFINE(0 TO 253)
        ";CH
60130 IFCH<=0ORCH>253ORCH<>INT(CH)THEN6
        0110
```

One of the features of the character generator will be its ease of use and this is defined principally by the graphics. The main feature of the graphics is the design matrix. In this, a blown-up version of the character matrix is set up using PRINT statements. These create an eight by eight (8x8) matrix within which the new character is designed. Lines 60150 to 60170 of Program 6.4(a) contain this section. In addition to displaying the character as designed in its enlarged format, the program will actually POKE the character to the screen without any enlargement, all that is necessary being a POKE,CH:

PROGRAM 6.4(a)

```

60140 PRINT"<CLR><4DCRSR><PUR>HERE IS Y
      OUR CHARACTER SO FAR:<GRY2> ";:PO
      KE1214,CH
60150 PRINT"<HOME>"TAB(7)"<7DCRSR><RED>
      <RVSON>███":FORX=1TO8
60160 PRINTTAB(7)"<RED><RVSON>█<GRN>***
      *****<RED>█"
60170 NEXT:PRINTTAB(7)"<RED><RVSON>███
      ████"

```

When the program is being used to design a character, the filled-in square is moved about the matrix by means of four keys, the chosen ones being:

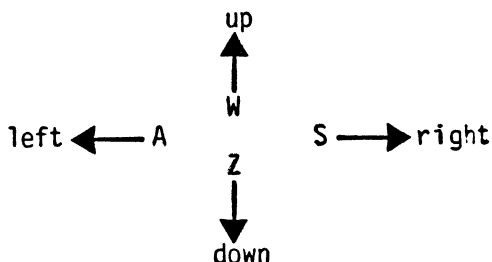


FIGURE 6.16

So that this is clear to the user, the keys and their functions are displayed on the screen, arranged in the same pattern as on the keyboard. To control the actual operation of the generator, the function keys are used and, once again, they are displayed on the screen by means of simple PRINT statements. Lines 60180 to 60240 in Program 6.4(b) complete the display:

PROGRAM 6.4(b)

```

60180 PRINT"<4DCRSR><GRY3>      W-UP"
60190 PRINT" A-LEFT S-RIGHT"
60200 PRINT"      Z-DOWN":PRINT"<RETURN>
      -END C.GEN";
60210 PRINT"<3UCRSR>"TAB(18)"<LTBLU>█<G
      RY3> F1-SET PIXEL"
60220 PRINTTAB(18)"<LTBLU>█<GRY3> F3-CL
      EAR PIXEL"
60230 PRINTTAB(18)"<LTBLU>█<GRY3> F5-DI
      SPLAY DATA"
60240 PRINTTAB(18)"<LTBLU>█<GRY3> F7-CL
      EAR BOARD<BLU>";

```

## The Function Keys

The C-64 has eight function keys, 1,3,5 and 7 being available by simply pressing the function keys alone. By means of the SHIFT and these keys, function keys 2,4,6 and 8 are activated. As with all the other keys on the keyboard, each has a characteristic ASCII code assigned. Thus, the pressing of f1 (function key 1) can be detected by looking for a CHR\$(133). The relevant codes for the function keys are:

f1	CHR\$(133)	f2	CHR\$(137)
f3	CHR\$(134)	f4	CHR\$(138)
f5	CHR\$(135)	f6	CHR\$(139)
f7	CHR\$(136)	f8	CHR\$(140)

FIGURE 6.17

## Designing the character

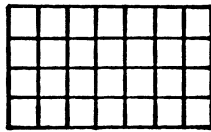
To define each individual character, 64 pixels need to be defined i.e. eight bytes of eight bits each. During processing, this data could be stored as a single stream of 64 characters but Commodore BASIC provides a much easier way of doing this by means of

## Two dimensional arrays

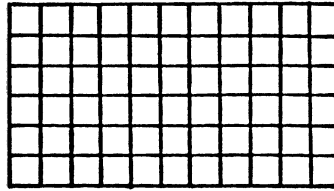
Two dimensional arrays utilize the same names as single dimensional arrays, and can in the same way be used to store both numeric and string data.

i.e.           A(X,Y)  
                AB(X,Y)  
                A9(X,Y)...etc.

In effect, a two-dimensional array is a rectangular matrix of cells that can be visualized as a series of pigeon holes with X rows in one direction and Y columns in the other. See Figure 6.18.



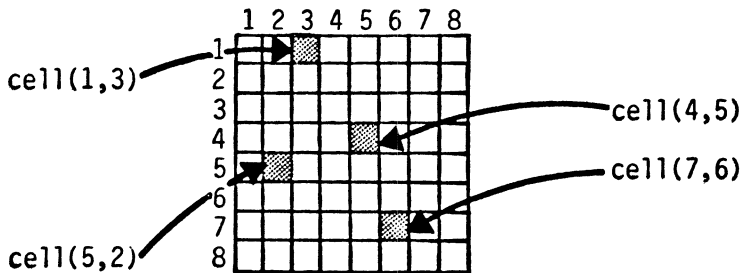
A 4 x 7 array  
eg A(4,7)



An 6 x 11 array  
eg W\$(6,11)

FIGURE 6.18

In an array, each cell can be addressed individually by defining its co-ordinates in each direction i.e.:



An 8 x 8 Array  
FIGURE 6.19

Such an array as in Figure 6.19 can be used to store the character as it's being defined with each row i.e. A(1,1) to A(1,8) being stored in one byte. As with single-dimensional arrays, the two-dimensional variety needs to be DIMensioned prior to use.

#### Auto-repeat on all keys

To move the cursor around the screen, the cursor control keys which are, very conveniently, fitted with an auto-repeat feature, are used. However, Char.Gen utilizes the W,A,S and Z keys for cursor control and these do not normally feature auto-repeat. However, this feature is very readily turned on generally by means of:

POKE 650,128 (or by REPEAT 1)

Prior to the design of the character, an initialization procedure is required to set X and Y (the co-ordinates of the design cursor) to 1 (line 60250). In addition, the character to be re-designed needs to be cleared as, prior to this process, it contains one of the C-64's standard characters.

```
60250 X=1:Y=1:POKE 55296+328,0
```

A GET A\$ is used to accept an input from the keyboard and one precaution that is well worth taking when using GET A\$ is to ensure that the keyboard buffer does not get full up with garbage. It is quite easy for the user to press two keys down or hold one down too long, especially as the repeat on all keys has been turned on. As discussed above, one location in memory, 198, records the number of bytes stored in the keyboard buffer. By setting this to zero, the C-64 is persuaded that the keyboard is empty. Thus, to empty the keyboard buffer, enter:

```
60280 POKE 198,0
```

The specially designed character that represents the current design cursor position is changed to black and this is moved around the design square in much the same way as in etcha-sketcha, by incrementing and decrementing its X and Y co-ordinates. Lines 60300 to 60330 in Program 6.4(d), simply check for the control keys being pressed and move the cursor accordingly.

PROGRAM 6.4(d)

```
60300 IFA$="A"ANDX>1THENX=X-1
60310 IFA$="S"ANDX<8THENX=X+1
60320 IFA$="W"ANDY>1THENY=Y-1
60330 IFA$="Z"ANDY<8THENY=Y+1
```

Function Key'Functions....

f1:Set Pixel

If f1 is depressed it means store current cursor position as a character element. This is detected when A\$=CHR\$(133) and the array location corresponding to the current cursor position is set to 1, i.e:

```
60350 IF A$=CHR$(133) THEN A(X,Y)=1
```

Then the current square on the character grid is set to a solid color and the color RAM is set. The position on the character matrix is found relative to the top left-hand corner in terms of X and Y i.e.  $X+Y*40$  and this is added to the co-ordinate of that point -  $1024+287$ . This is required to change the data which is in a matrix format (X,Y) into the format that the screen understands i.e. running along in rows of 40 characters. A similar calculation is carried out for the color RAM. Once this is done, the subroutine moves to line 60570 to update the current character stored in memory i.e.

```
60350 IF A$="<F1>" THEN A(X,Y)=1:
      POKE 1024+287+X+Y*40,255:
      POKE 55296+287+X+Y*40,6:GOTO 60570
```

### f3:Clear Pixel

When a pixel that has previously been set needs to be cleared, this function does the job. It simply reverses the procedure of f1 by setting the relevant array position to zero, poking a space onto the screen and finally setting the color RAM to background. Being so similar to the Set Pixel function the code is virtually identical i.e.

```
60360 IF A$="<F3>" THEN A(X,Y)=0:POKE
      1024+287+X+Y*40,254:POKE 55296+287+X+Y*
      40,5:GOTO 60570
```

### f5:Display DATA/store in program

When this function is called, the program will ask the user where the DATA statement for that particular character should be stored. If a line number of zero is indicated, then the DATA for that character will be displayed but not stored. When a line number is entered, a line will be inserted into the program which contains the requisite DATA for that character. This is a rather complicated process and will be discussed later.

```
60370 IF A$="<F5>" THEN 60400
```

### f7:Clear Board

Essentially, this function scraps the design work done to date and wipes the slate clean. In other words, it clears the design matrix and prepares it for start. The command CLR does this by setting all the variables and arrays back to zero, and, once this is done, the program is re-routed to the beginning of the input routine, i.e.

```
60380 IF A$="<F7>" THEN CLR:DIM A(8,8):GOTO 60110
```

## f5:Display DATA/store in program - the details

This is activated by pressing function key 5 and as it is required to do two different jobs, it starts with a prompt which tells us what these jobs are and asks for a line number INPUT, i.e. LN. In an attempt to avoid overwriting either the program to which user-defined graphics will be added or the Char.Gen program itself; line numbers between 10000 and 60000 are suggested. To strengthen this 'suggestion', an error-check is made in lines 60400-60420 which re-routes the program back for another INPUT when lines lie outside this range.

Once the INPUT has been accepted, the prompt is deleted by printing spaces over it, lines 60430 and 60440 in Program 6.4(e). In addition to the code for f5, this listing provides the whole program for the character design routine.

### PROGRAM 6.4(e)

```
60250 X=1:Y=1:POKE55296+328,0
60260 FORB=0T07:POKE12288+B+CH*8,0:NEXT
      :POKE650,255
60270 GETA$:IFA$="" THEN60270
60280 POKE198,0
60290 POKE55296+287+X+Y*40,5+A(X,Y)
60300 IFA$="A"ANDX>1THENX=X-1
60310 IFA$="S"ANDX<8THENX=X+1
60320 IFA$="W"ANDY>1THENY=Y-1
60330 IFA$="Z"ANDY<8THENY=Y+1
60340 POKE55296+287+X+Y*40,0:IFA$=CHR$(
      13)THEN61000
60350 IFA$="<F1>"THENA(X,Y)=1:POKE1024+
      287+X+Y*40,255:POKE55296+287+X+Y*
      40,6:GOTO60570
60360 IFA$="<F3>"THENA(X,Y)=0:POKE1024+
      287+X+Y*40,254:POKE55296+287+X+Y*
      40,5:GOTO60570
60370 IFA$="<F5>"THEN60400
60380 IFA$="<F7>"THENCLR:DIMA(8,8):GOTO
      60110
60390 GOTO60270
60400 PRINT"<HOME>ENTER LINE NO. FOR DA
      TA STATEMENT":PRINT"(10000-60000)
      ";
60410 PRINT"OR ZERO FOR DISPLAY ONLY"·I
      NPUTLN
60420 IF(LN<10000ORLN>59999ORLN<>INT(LN
      ))ANDLN<>0THEN60400
60430 PRINT"<HOME>";:FORC=1T03:FORD=1T0
      38
60440 PRINT" ";:NEXT:PRINT:NEXT:PRINT"<
      HOME><18DCRSR>";
```

Prior to PRINTing the actual DATA onto the screen, the POKE values need to be found and this can be done by PEEKing the actual locations where these are stored. Once the values have been found they could be stored on the screen or in an array. In this program, neither of these courses are chosen, the data being stored in one variable C\$, made up from the relevant PEEKs. To do this, the string is built up with alternate PEEK values and commas. One problem is created by this procedure, the presence of a final comma on the line. This is removed by adding a delete character (CHR\$(20)) to the end of the string and deleting the last comma. If you're not too sure about this, type in Program 6.5(a).

PROGRAM 6.5(a)

```
10 C$="":FOR X=1 TO 10
20 C$=C$+"A"+","
30 NEXT X
50 PRINT C$
```

When this is run, C\$ should be PRINTed as a string of ten A's separated by commas with a comma at the end. Now modify this by adding line 40 of Program 6.5(b)

PROGRAM 6.5(b)

```
40 C$=C$+CHR$(20)
```

Once the new line is added, the string C\$ will then consist of the ten A's separated from each other by commas but the final comma will have been deleted. So it does work! Once you are satisfied that the program works, delete lines 10 to 50, either by typing in the numbers or by means of the Honey.Aid command :

DELETE (A Honey.Aid command)

Its descriptive name tells you just what this command does, the syntax being just like that of LIST. Hence, to delete lines 10 to 40 of a program, three of the four different methods could be used; i.e.

- \* delete the lines individually  
DELETE 10 <RETURN>  
DELETE 20 <RETURN>  
etc.
- \* delete the block  
DELETE 10 - 40 <RETURN>
- \* delete up to line 40  
DELETE -40 <RETURN>



The other allowable method is DELETE 40 but as this would delete the bit of program that you need it's not a good idea to use it here. Go steady with DELETE though, it's useful but dangerous! It has been set up so that 'DELETE' on its own won't delete everything, (unlike LIST which will list everything) but it is still powerful!

Back to CharGen. . .

So far the data line has been created and if a line number is PRINTed onto the screen prior to this, all the elements of a DATA statement will have been created. So far this has not been entered into the program and this now needs to be done.

### Self modifying programs

In order to enter a line into a program the computer needs to be in edit mode i.e. with a 'READY' sign displayed and the cursor flashing. Of course, the computer is not in this state once a program is running and the machine would need to encounter an 'END' to turn on the 'READY' sign.

When doing an edit manually the process would be to type in the line and then press <RETURN> to add the line. So far we've worked out how to print the line and if we then follow this with an 'END' the computer will return to EDIT mode. Test this out with the following line. In this we'll attempt to modify a program that prints 'FRED' to one that prints 'FRED IS OK'. The task then is to change the program.

```
10 PRINT "FRED" into 10 PRINT "FRED"  
20 PRINT "IS OK"
```

What is needed, therefore, is a line to create line 20 and one problem that arises immediately is that of printing quotes i.e. " onto the screen. Actually, this is no problem since, ASCII code 34 is the quote sign.

```
PRINT CHR$(34)
```

should print a quote sign onto the screen. Try it and see. The next stage is to print the whole line and this time let's do it from within a program:

```
10 PRINT "FRED"  
20 PRINT"20 PRINT";CHR$(34);"IS OK";C  
HR$(34)  
999 END
```

Now when this is 'RUN', the program will yield:

```
FRED
20 PRINT"IS OK"

READY
█ <CURSOR>
```

FIGURE 6.20

From this stage, the cursor would need to be moved up four lines so that it is over line 20. This can be done in the program by adding a line to move the cursor up four lines.

```
30 PRINT "<4UCRSR>"
```

When this is added and run, the display will be as in Figure 6.20 but with the cursor flashing over line 20. At this stage, if you press RETURN, you will enter (edit) line 20 into the program. Press RETURN now, then type in LIST 1-999. You will then see the program:

```
10 PRINT"FRED"
20 PRINT"IS OK"
30 PRINT"<4UCRSR>";
999 END
```

You had to physically press the RETURN key to enter line 20 into memory. We actually want the computer to do this itself. We can force the computer to read a RETURN key press without actually pressing RETURN by placing a value of 1 into the keyboard read location (198), meaning a key has been pressed, and placing the value of a RETURN key (13) into 631 (the keyboard buffer). Modify lines 20 and 30 to now read:

```
20 PRINT"20 PRINT";CHR$(34);"IS OK";CHR$(34)
30 PRINT"<4UCRSR>";POKE 631,13:POKE 198,1
```

Now run the program and it will yield the following display:

```
READY
20 PRINT"IS OK"
█ <CURSOR>
```

FIGURE 6.21

Now type LIST 1-999 and the new program will be:

```
10 PRINT"FRED"
20 PRINT"IS OK"
30 PRINT"<4UCRSR>";POKE 631,13:POKE 198,1
999 END
```

So far so good. The program modifies itself, but what would happen were it to be incorporated into a longer program? To find out, modify the program by adding the following lines (the program is simply going to go through a PRINT loop):

```
20 PRINT"20 PRINT";CHR$(34);"IS OK";CHR$(34)
40 FOR X=1 TO 10
50 PRINT X
60 NEXT X
```

Now run the program. The display will be overlaid with the numbers 1 to 10. Type in LIST 1-999 and you will see that line 20 was not changed as it was before. The reason for this is that the computer will not check the keyboard buffer until the program calculations are complete (the FOR...NEXT loop is finished). It is necessary to have the computer perceive an END or an INPUT in order to force it to check the keyboard buffer. Modify line 30 to now read:

```
30 PRINT"<4UCRSR>";POKE 631,13:POKE 198,1:END
```

RUN the program, then LIST 1-999 and you will see that line 20, again, has been changed. However, we must now realize that the FOR...NEXT loop has not been executed. We can force the computer to execute the loop by means of typing in GOTO 40. Type this in now and you will have the numbers 1 through 10 printed on the screen. Of course, we want the computer to do this itself, without our having to type in GOTO 40. What is necessary is a repetition of the procedure used to enter line 20, i.e.

- \* PRINT the new line onto the screen
- \* move the cursor onto this line
- \* force a RETURN into the keyboard buffer
- \* tell the computer that this RETURN is in there
- \* force the computer to look into the buffer to see what to do next by means of an END

In order to integrate this into the program, the 'GOTO' is added to the program before the earlier 'keyboard buffer' line and this is modified to contain an additional CHR\$(13) i.e. Program 6.6(a).

PROGRAM 6.6(a)

```
20 PRINT"20 PRINT";CHR$(34);"IS OK";CHR$(34)
25 PRINT"GOTO 40"
30 PRINT"<5UCRSR>":POKE 631,13:POKE 632,
13:POKE198,2:END
```

Now, at long last, when this is RUN, the program will insert the new line 20 followed by the 'GOTO', cursor up over this and then

RETURN over the two lines, one a line edit and the other a direct GOTO command. One other little refinement could be added to this line and that is to print the 'GOTO' in background color so that the user is not aware of its presence. This is, in fact, done on line 60490 of Program 6.6(b). One warning though, if something of a different color occupies that space on the screen, the GOTO will be seen! You can now type in DELETE 1-999 (assuming Honey.Aid has been RUN), to remove these lines from our Char.Gen program.

When integrated into the Char.Gen program, this process will enter the DATA line and then restart the program. The routine is shown in Program 6.6(b).

#### PROGRAM 6.6(b)

```

60440 PRINT " ";:NEXT:PRINT:NEXT:PRINT"<HOME>
<18DCRSR>";
60450 IF LN>0 THEN PRINT LN;
60460 PRINT"DATA";:IF LN>0 THEN PRINT CH:",";
60470 C$="":FOR C=0 TO 7:C$=C$+STR$(PEEK(12288+
CH*8+C))+","
60480 NEXT:C$=C$+CHR$(20):PRINT C$:IF LN=0 THEN
LN=1:GOTO 60500
60490 PRINT"<WHT>GOTO 60500<5UCRSR>":POKE 631
,13:POKE 632,13:POKE 198,2:END
60500 PRINT"<HOME><2DCRSR><ORNG>      <RVSON>PRESS
ANY KEY TO CONTINUE<RVSOFF><BLU>"
60510 POKE 198,0
60520 GET A$:IF A$="" THEN 60520
60530 PRINT"<HOME><2DCRSR><30 SPACES>"
60540 IF LN=0 THEN DIM A(8,8):GOTO 61000
60550 PRINT"<HOME><18DCRSR>";:FOR C=
1 TO 70:PRINT " ";:NEXT
60560 GOTO 60140

```

This subroutine requires one final section to yield an overall workable Char.Gen program. This is the routine that loads the array A(X,Y), converts the information into a POKE value and POKE's that into memory. It is called each time a new pixel is incorporated into a character and the POKE value for the character is thus updated. As each cell of the array, in the X direction, represents a value equivalent to a two raised to its positional value, a loop is used to calculate the total value of A, i.e. for the pixel pattern:

Positional value	7	6	5	4	3	2	1	0
bit value	0	1	1	0	0	1	1	0
in decimal =	0	+ 2 <sup>6</sup>	+ 2 <sup>5</sup>	+ 0	+ 0	+ 2 <sup>2</sup>	+ 2 <sup>1</sup>	+ 0
	0	+ 64	+ 32	+ 0	+ 0	+ 4	+ 2	+ 0 = 102

FIGURE 6.22

The summation is performed by the loop:

```
FOR B = 1 TO 8:A=A(B,Y)*2 (8-B):NEXT
```

A slight warning: the array place numbers are stored from right to left, while the bit numbers are, by convention, from left to right.

i.e.

X value	1	2	3	4	5	6	7	8	array(X,Y)
bit value	7	6	5	4	3	2	1	0	

Testing for bit 3 :  $A = A(B,Y) * 2 (8-5)$

i.e.  $A = 1 * 2 (3)$

and  $A = 1 * 8 = 8$

Once the byte value is calculated, it is POKEd into the appropriate location and the program returned for the next INPUT (line 60570 of Program 6.6(c)). Actually, as you will see, we have used an AND instead of the FOR....NEXT loop:

PROGRAM 6.6(c)

```
60570 A=PEEK(12287+Y+CH*8):IFA(X,Y)=0TH
      ENA=AAND(255-(2↑(8-X)))
60580 IFA(X,Y)=1THENA=AOR(2↑(8-X))
60590 POKE12287+Y+CH*8,A:GOTO60270
```

### Read DATA/create character subroutine

When a program utilizes custom-designed characters, one necessary function is their creation from the DATA statements where all the necessary information is stored. In Char.Gen, the characters are created during the running of the program but, as this function is integrated with the rest of the program, it is not suitable for incorporation into the user's own program. To overcome this problem, a small routine is created which will read in the DATA and POKE it into the appropriate memory locations. Program 6.6(d) shows the routine along with an explanation of its function, and into this is built the facility for creating a specified number of characters (N).

PROGRAM 6.6(d)

```
60610 FOR Y = 1 TO N:READ A:FOR X=0 TO 7
      read character number
60620 READ B:POKE 12288+A*8+X,B
60630 NEXT : NEXT
      read eight lines of character Y, store as appropriate
```

Destroy unwanted program routine

Once the characters in a program have been defined, the Char.Gen program can be deleted and the largest part of this is the question and answer session. Any program that is as drastic as this needs lots of warnings! In addition, the initialization procedure sets out to find the number of characters concerned. Program 6.6(e) shows this part of the program.

PROGRAM 6.6(e)

```
61000 PRINT"<CLR><2DCRSR><4RCRSR>HAVE Y
      OU FINISHED USING"
61010 INPUT"<DCRSR><4RCRSR>CHAR.GEN (Y/
      N)";A$:IFA$="N"THEN60110
61020 IFA$<>"Y"THEN61000
61030 PRINT"<2DCRSR><4RCRSR>WARNING:THI
      S ROUTINE DELETES THE"
61040 PRINT"<DCRSR><4RCRSR>CHARACTER GE
      NERATOR."
61050 PRINT"<2DCRSR><4RCRSR>ARE YOU REA
      LLY SURE THAT YOU'VE"
61060 INPUT"<DCRSR><4RCRSR>FINISHED (Y/
      N)";A$:IFA$="N"THENEND
61070 IFA$<>"Y"THEN61060
61080 PRINT"<2DCRSR><4RCRSR>HOW MANY CH
      ARACTERS HAVE YOU":INPUT"<DCRSR><
      4RCRSR>REDESIGNED";N
```

BEFORE YOU RUN THIS SAVE IT - IT DESTROYS PROGRAMS!

Once it is established that the user really does want to eliminate the Char.Gen program, deletion can begin. In this program use is made of Honey.Aid's DELETE function although the job can be done, albeit much more slowly, using ordinary BASIC. To do this, the individual line numbers would need to be printed and then RETURNed over.

However, Honey.Aid to the rescue! In the final program, the moving down of the character set will still be required so the first DELETE starts at 60080. Also the generation of the character is required, defining the end of the DELETE as 60599. Thus, the first delete is from 60080 to 60599. It isn't possible to use DELETE in program mode, so once again we must resort to printing onto the screen and RETURNing over it.

The program from line 61000 onwards can then be deleted in one go, i.e. DELETE 61000- leaving just one final DELETE to be done, line 60000. This line, the first in the Char.Gen program, by-passes the moving of the character set once the program has been run. When this line has been removed, it exposes line 60010 which is not otherwise used, and this functions in the same way that 60000 did but directs the program over the now missing section.

Finally, the number of characters defined (N) is written permanently into the program by inserting a LET statement into the program in order to define the value of N in a permanent line. Program 6.6(f) shows these final few lines in action and completes the functional character generator program.

Program 6.6(f)

```
61090 PRINT"<CLR><WHT><2DCRSR>";
61100 PRINT"DELETE 60080-60599<2DCRSR>"
61110 PRINT"DELETE 61000-<2DCRSR>"
61120 PRINT"60600 N=";N
61130 PRINT"60000"
61140 PRINT"PRINT"CHR$(34)"<CLR><BLU>"CHR$(34)
61150 PRINT"<HOME>";
61160 FORX=631TO635:POKE X,13:NEXT:POKE 198,5:END
```

### PART 3

#### Using Char.Gen

To use Char.Gen you must first load and run Honey.Aid, then load and run Char.Gen. The program on the disk has been modified to run more quickly than the version described in the previous sections.

To see Char.Gen in action, let's define a character and incorporate this into a short program. First of all RUN Char.Gen and the machine will ask:

```
Enter:          WHICH CHARACTER WOULD YOU LIKE TO DEFINE?
                '253'
```

This character is to be replaced by that shown in Figure 6.23.

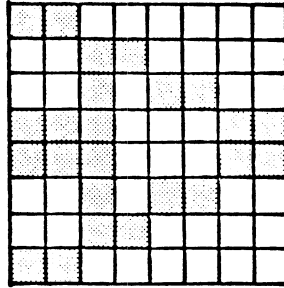


FIGURE 6.23

To create this character, the relevant blocks need to be filled in. The first block that needs filling on the top row happens to be the first block in the row. As the cursor is already at this block when the program runs it is only necessary to press f1 to set the pixel. At this stage, next to the message "HERE IS YOUR CHARACTER SO FAR" will appear the first dot of the character. The rest of the character can then be built up using the keys:

W	Move cursor up
A	Move cursor left
S	Move cursor right
Z	Move cursor down

and to set the actual characters:

f1	sets the current cursor pixel as part of the character
f3	clears the current cursor pixel and deletes it from part of the character
f5	calculates the POKE values for the re-designed characters
f7	clears the character designed to date ready to start again.



Going through the first two lines of the character the operations are, press:

```
* f1, S, f1, S
* Z, f1, S, f1, S
* Z, f1, S, f1, S
* Z, f1, S, f1
* Z, f1, A, f1, A
* Z, f1, A, f1, A
* Z, f1, A, f1, A
* Z, f1, A, f1
* W, W, S, S, f1
* W, f1, A, f1, A, f1
* W, f1, S, f1, S, f1
* W, f1, A, f1
```

Whoops! That's one too many. To correct this press f3, to clear the last pixel. Remember, if you press f1 in the wrong place, f3 will correct it. If you find that there are too many changes needed, simply press f7 to clear the whole board.

Now that you've designed the whole character and you're sure that no changes are needed, press f5. The computer will ask:

```
ENTER LINE NO. FOR DATA STATEMENT
(10000-60000):OR ZERO FOR DISPLAY ONLY
```

Just to test out the program enter a zero and the screen will then display the relevant DATA statement below the design matrix. In addition a message will be given to:

```
PRESS ANY KEY TO CONTINUE
```

Once a key is pressed, the program returns to its entry mode.

Now to transfer the DATA statement into your own program, once again press f5. The 'AT WHICH LINE....' message will re-appear and this time you should enter the relevant line number, say, 10000. When you press <RETURN>, the DATA statement will reappear, this time along with a line number. After a second or so, the "PRESS ANY KEY..." message will reappear and, on doing this you will return once again to the 'Which character' message.

If you wished to design another character you would first clear the design matrix by pressing f7 and then begin again. You can go on to define all the characters that you need and you may find some changes appearing on the screen if you redefine those characters used by Char.Gen itself! However, for now just stick to the one character.

Next, to get into the program termination phase press <RETURN> and you will be asked:

HAVE YOU FINISHED USING CHAR.GEN (Y/N)?

On entering a 'Y' you then be warned and then asked:

WARNING: THIS ROUTINE DELETES THE CHARACTER GENERATOR

ARE YOU REALLY SURE THAT YOU'VE FINISHED (Y/N)?

Once the decision has been made, you will be asked:

HOW MANY CHARACTERS HAVE YOU REDESIGNED?

Enter a '1' and press <RETURN>.

You will now be left with the remnants of Char.Gen, lines 10000-10010, 60010-60070, 60600-60640.

In order to utilize Char.Gen it needs to be called with a GOSUB 60010. Program 6.7 illustrates a simple technique for utilizing the character. It is displayed by POKEing it into the screen.

#### PROGRAM 6.7

```
10 GOSUB 60010:A=32
20 FOR X=1 TO 1000:POKE 1023+X,A:
A=PEEK 1024+X:POKE 1024+X,253:
POKE 55296+X,6:NEXT
30 END
```

When you run this its a good idea not to clear the screen first, just type RUN <RETURN>. Can you see from line 20 what it does? All it's doing is storing the character ahead of the space ship and then placing it back behind it as this moves on. Quite like a sprite eh?

One last thing about redefining characters, don't forget that you're using them as you change them. Let's have a little play with Char.Gen! Change lines 60130 so that it allows all characters to be defined, i.e. change the 253 into a 255. Then run the program and select '254' for the character to be redefined. Now, when the display is put onto the screen, you will see the '254' characters melt away before your very eyes as line 60260 gets to work. If you now create characters, you will find that the design matrix is made up of these - as you create them. By making regular grid-type characters, all kinds of patterns can be made.

## CHAPTER

# 7

### BALL GAMES

**I**n this chapter, we will develop further the ideas explored in Chapter 5. Some of the techniques explored in Chapter 5 will be extended to develop a breakout type of game which uses graphics interactively. This is really a fancy way of saying that the player will control the screen display by way of the keyboard.

One of the features of the C-64's screen is that it is 'memory mapped' which means that every character space or cell on the screen has a memory location allocated to it. Thus, if the value 90 is stored in the appropriate location for the screen's top left-hand corner, then a 90 (in CBM code, a diamond) will be displayed on the screen. However, you may not be able to see it if the screen's background color and the character color are one and the same - blue on blue is none too clear, nor is green on green. In any program, therefore, it is advisable to make sure that the colors are those that you want by setting them appropriately. Thus the stages are:

- \* Set border color
- \* Set screen background color
- \* Set character color

The order is none too important but they all need doing!

The screen colors are quite easily set by single POKEs but in order that a screen display can have more than one color on it, each character cell can be set individually. If the character is not set, you may be lucky and your characters may be visible - on the other hand you may be unlucky.

The C-64 screen is divided into 1000 cells; 40 across each line and 25 down -see Figure 7.1.

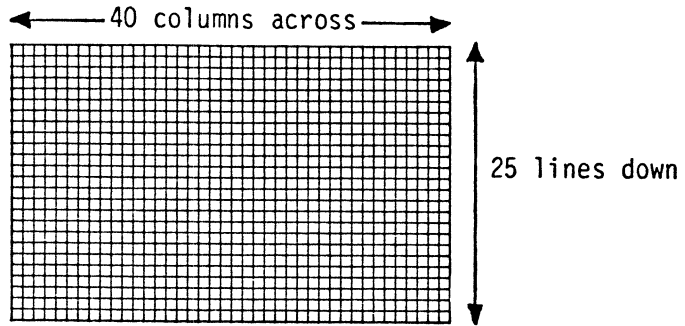


FIGURE 7.1

Its top left-hand location, the 'home' position is mapped to memory location 1024 and the other cells along the top row, to locations 1025, 1026, etc. A further memory location, 55296, contains the color information for the top left-hand screen cell. The next 39 locations hold color information for the remainder of the screen's top row, the following 40 for row two, etc. This whole area of memory is known as the 'color RAM'.

Enough talking, let's have a go! Program 7.1 POKEs onto the screen an 'A' on a white background with a green border.

#### PROGRAM 7.1

```
Set border to green      10 POKE 53280,5
Set screen background
to white                 20 POKE 53281,1
POKE a '1' onto
screen home position    30 POKE 1024,1
Set color of screen
home position           40 POKE 55296,7
```

In line 40, a '1' was POKEd into the screen memory and a letter 'A' appeared on the screen. These screen POKE codes are a Commodore form of ASCII code and the whole range of these is listed in Appendix 3. They can be investigated by modifying Program 7.1 slightly so that it cycles through them all. Try it with:

#### PROGRAM 7.2

```
10 POKE 53280,1
20 POKE 53281,2
25 FOR X=0 TO 255
30 POKE 1024+X,X
40 POKE 55296+X,X
50 NEXT X
```

When Program 7.2 is run, the whole character set is displayed in the first two and a half rows of the screen. Each character is a different color as the value stored in the color RAM is incremented on each pass through the loop. However, as there are only 16 colors and most of the values POKEd into the color RAM were over 16, the computer was clearly working on it! What was happening, in fact, was that only the least significant nybble of the color RAM was being used to define the color. The remainder was being stripped off by means of a logical operator, so....

#### EXERCISE 7.1

What logical operation is used to remove the most significant nybble? An answer is given on page 11-6.

Suppose now we wanted to fill the entire screen with the POKE character '83'. One way of doing this would be to enter '2000' lines of program - 1000 assigning the character and another 1000 giving it a color, starting:

#### PROGRAM 7.2(a)

```
POKE 1024,83
POKE 55296,2
.
.
.
and ending...
POKE 2023,83
POKE 56295,2
```

That would be a very laborious process, to say the least! As we saw in Program 7.2, we can use variables when POKEing. Thus, if we wished to fill up the whole screen with the character '83', we can use a loop. As there are 1000 locations between '1024' and '2023' (and also between '55296' and '56295'), the loop will need to run from '0' to '999', i.e.

#### PROGRAM 7.2(b)

```
20 FOR X=0 TO 999
30 POKE 1024+X,83
40 POKE 55296+X,2
50 NEXT X
```

That short program (1996 lines shorter than Program 7.2(a)) fills the screen with bright red hearts (all together now: AAH!). The next leap forward will be to have lines of different colored hearts: one line red, one blue, etc. The problem with trying to do this is that each line holds forty characters and the loop in Program 7.2(b) was just too long. What is needed, therefore, is a loop just 40 characters long i.e.,

#### PROGRAM 7.3

```
20 FOR X=0 TO 39
30 POKE 1024+X,83
40 POKE 55296+X,2
50 NEXT X
```

Now, when this is run one line of red hearts will be displayed across the screen. They are all the same color though! Let's add another loop to run through the 16 colors:

#### PROGRAM 7.3(a)

```
10 FOR Y=0 TO 15
20 FOR X=0 TO 39
30 POKE 1024+X,83
40 POKE 55296+X,Y:REM CHANGE COLOR
50 NEXT X
60 NEXT Y
```

Now, at least we've got the 16 colors but they're all on the top line. What we need now is to increase the values for the two POKES by 40 on the second loop as the second now starts at 1064. For the third line an increase of 80 is needed and 120 on the fourth etc. This can be achieved by adding 40 times the Y value to each POKE, i.e. on the first loop  $40*0=$ zero is added, on the second pass  $40*1=40$  etc. Incorporating this in a program yields:

#### PROGRAM 7.3(b)

```
10 FOR Y=0 TO 15
20 FOR X=0 TO 39
30 POKE 1024+X+Y*40,83
40 POKE 55296+X+Y*40,Y
50 NEXT X
60 NEXT Y
```

#### EXERCISE 7.2

Change Program 7.3(b) so that the hearts are POKEd column by column instead of row by row. A possible answer is given on page 11-6.

We now have the tools to start on the ball game and to POKE in the wall simply calls for a 3 x 7 loop using a POKE character of 160, an off-reverse space (i.e. a brick!).

#### Program 7.3(c)

```
10 PRINT"<CLR>"
20 FOR Y=3 TO 7
30 FOR X=0 TO 39
40 POKE 1024+X+Y*40,160
50 POKE 55296+X+Y*40,Y
60 NEXT X, Y
```

#### EXERCISE 7.3

Modify Program 7.3(c) so that the lines start at the top of the screen, but retain their original colors. A possible answer is given on page 11-6.

## A MOVING BALL

Probably the most important part of the game will be the simulation of a bouncing ball. Big oaks from little acorns grow, so we shall start by making the ball move across the top of the screen. If we choose a ball-like character say a '●'(POKE 81) we may move this across the screen by means of straight-forward POKES:

### PROGRAM 7.4

```
10 PRINT"<CLR>"
20 FOR X=0 TO 39
30 POKE 1024+X,81 : REM 81='●'
40 POKE 55296+X,2
50 NEXT X
```

When run, the program only draws a line of red balls across the screen leaving a trail behind it. On a black and white television, they may be completely invisible, so, if you can't see them, change line 40 to POKE color 10 instead of color 2. What we need to do now is to wipe out that trail. This can be done most readily by POKEing a space (a POKE of 32) behind the ball, thus ensuring that only one ball is on the screen at a time. This extra POKE can be included in the loop but care must be taken to ensure that its location is always one behind the ball i.e. the ball is at 1024+X and the space is at 1024+X-1.

### PROGRAM 7.4(a)

```
10 PRINT"<CLR>"
20 FOR X=0 TO 39
30 POKE 1024+X,81
40 POKE 55296+X,2
45 POKE 1024+X-1,32
50 NEXT X
```

#### EXERCISE 7.4

Modify Program 7.4(a) so that the ball moves across the bottom of the screen. A possible answer is given on page 11-6.



Before entering Program 7.4(b) examine it and see if you can predict the way in which the ball will move (no prizes for a correct guess).

#### PROGRAM 7.4(b)

```
10 PRINT"<CLR>"
20 FOR X=0 TO 20
30 POKE 1024+X+X*40,81
40 POKE 55296+X+X*40,2
50 POKE 1024+X-1+(X-1)*40,32
60 NEXT X
```

The ball stops short of the bottom line to avoid POKEing anything into non-screen memory locations: always a sensible precaution.

#### EXERCISE 7.5

Change Program 7.4(b) so that the ball moves diagonally from bottom right to top left. A possible answer is given page 11-6.

#### A Randomly Moving Ball

Now to extend the moving ball idea a little. Suppose we want to increase the row position and decrease the column position of the ball at the same time thus giving a diagonal movement from the top right to the bottom left. The major problem of wiping out the trail that the ball leaves is overcome by introducing variables 'XX' and 'YY', which 'remember' the position of the ball before it moves on.

#### PROGRAM 7.5

```
10 PRINT"<CLR>"
20 X=0 : Y=0
30 POKE 1024+X+Y*40,81
40 POKE 55296+X+Y*40,2
50 YY=Y : XX=X
60 Y=Y+1 : X=X-1
70 IF Y=24 THEN STOP
80 POKE 1024+XX+YY*40,32
90 GOTO 30
```

Before 'X' and 'Y' are given new values in line 60, 'XX' and 'YY' are set to the old values (line 50) and so the old position is wiped out in line 80. Program 7.5 stops when the ball reaches the bottom. Checks for the edges of the screen are very important.

By now you should have realized that the problem of the bouncing ball is not as straightforward as our first attempts might have led us to believe. Sometimes it is necessary to increment the column (X), sometimes the row (Y) and sometimes both. Then of course, sometimes it is necessary to decrement them and finally sometimes a mixture of the two is needed.

To simulate a randomly-moving ball, use can be made of the random number generator within the 64. Run the following short program:

PROGRAM 7.6

```
1 PRINT INT(RND(1)*2):GOTO 1
```

This produces a series of randomly chosen zeros and ones. When you get bored press 'RUN/STOP'. One small point about RUN/STOP! It does, in fact, stop the program as you'd expect but what about when you press it by mistake? Fortunately BASIC has a way out of this disaster; it's the command...

CONT

This is a shorthand form for 'continue' and tells the computer to continue executing the program from where it last stopped. Try it out now; type in:

```
CONT
```

The program should continue by giving you even more random numbers.

In the randomly moving ball program it will be necessary either to increase each row or column (add +1) and/or decrease each column or row (add -1) in order to generate a random bounce. The random number generator can be used to do this. Two more variables will be required to handle this random bounce - XI and YI. These will control the sign of the increments XX and YY. 'XI' controls the 'X' increment and 'YI' controls the 'Y' increment. See lines 120 and 130.

## PROGRAM 7.7

```
10 REM RANDOMLY MOVING BALL
20 PRINT "<CLR>"
30 X=10
40 Y=10
50 POKE 1024+X+Y*40,81
60 POKE 55296+X+Y*40,2
70 XX=X:YY=Y
80 XI=INT(RND(1)*2)
90 IF XI=0 THEN XI=-1
100 YI=INT(RND(1)*2)
110 IF YI=0 THEN YI=-1
120 X=XX+XI
130 Y=YY+YI
135 IF X>38 OR X<1 THEN STOP
140 IF Y>23 OR Y<1 THEN STOP
150 POKE 1024+XX+YY*40,32
160 GOTO 50
```

Lines 135 and 140 make sure that the ball doesn't stray out of the boundaries we have allowed for it, i.e. the screen. Since we want XI and YI to be either -1 or +1, and these are not consecutive numbers, we first let XI be either 0 or 1 (line 80) and then, if it's 0, we change it to -1 (line 90). If it's 1, we leave it alone. This is repeated for YI in lines 100 and 110.

When Program 7.7 is run it will, eventually, reach one of its limits and the program will stop. To remedy this situation, we must change lines 135 and 140 so that the ball 'bounces'.

### PROGRAM 7.7(a)

```
135 IF X>38 OR X<1 THEN XI=XI*-1:GOTO 120
140 IF Y>23 OR Y<1 THEN YI=YI*-1:GOTO 120
```

Multiplying a number by -1, in effect, simply changes its sign, so  $+1*-1$  results in -1 and  $-1*-1$  gives us +1. Thus lines 135 and 140 change the direction of the ball when it reaches an extreme value. By going to line 120, the new value of XI or YI is added to X, thus giving the ball a new direction away from the offending edge.

In developing the randomly bouncing ball we have investigated many important graphics techniques, not all will be called for in the first game we develop but, eventually, we will need them all. Overall, the endlessly bouncing ball program looks like Program 7.7(b).

## PROGRAM 7.7(b)

```
10 REM AN ENDLESSLY BOUNCING BALL
20 PRINT "<CLR>"
30 X=10:Y=10
40 XI=1:YI=1
50 POKE 1024+X+Y*40,81
60 POKE 55296+X+Y*40,2
70 XX=X:YY=Y
80 IF X>38 OR X<1 THEN XI=XI*-1
90 IF Y>23 OR Y<1 THEN YI=YI*-1
100 X=XX+XI
110 Y=YY+YI
120 POKE 1024+XX+YY*40,32
130 GOTO 50
```

### EXERCISE 7.6

Amend Program 7.7(b) so that the ball's initial position is random. A possible answer is given on page 11-6.

The program we have so far, bounces the ball from any edge of the screen but for the first game, the ball should only bounce from the bottom if the bat is in position; also when the ball approaches the wall, its bounce depends upon how many bricks have been demolished.

What is needed is a function which will enable one to look at a particular screen location (especially the one into which the ball is about to move) and to report back what is seen: the PEEK command!

Program 7.9 illustrates how we can use PEEK to bounce a ball from a line drawn using character code 160 (reverse space).

When this program is run, it may look very similar to the one earlier in this chapter, but the bounce is caused by a different logic. Before, we constantly kept a check on the values of 'X' and 'Y', and when they reached their extreme values we effected a bounce. In Program 7.9, however, line 115 checks the location into which we are going to move to see if anything is in the way. If there isn't anything, then we make the ball move as before, but if there is (in which case A=0), then we can't let the ball move into that position. So line 116 changes the row increment variable, YI, and the program branches back to line 100 to work out the new position.

## PROGRAM 7.9

```
10 REM ENDLESSLY BOUNCING BALL
20 PRINT "<CLR>"
21 FOR X=0 TO 39
22 POKE 1024+X,160:POKE 55296+X,0
23 NEXT X
30 X=10:Y=10
40 XI=1:YI=1
50 POKE 1024+X+Y*40,81
60 POKE 55296+X+Y*40,2
70 XX=X:YY=Y
80 IF X>38 OR X<1 THEN XI=XI*-1
90 IF Y>23 OR Y<1 THEN YI=YI*-1
100 X=XX+XI
110 Y=YY+YI
115 A=PEEK(55296+X+Y*40) AND 15
116 IF A=0 THEN YI=YI*-1:GOTO 100
120 POKE 1024+XX+YY*40,32
130 GOTO 50
```

If we hadn't used 'AND 15' on line 115, then the value of 'A' would have been '224'. The 'AND 15' part gives the lowest numeric value of the color in that location.

### EXERCISE 7.7

Rewrite Program 7.9 so that a line (character 160) is drawn vertically from the ninth location at the top (1033) and finishing at the bottom equivalent (1993). The command to draw the lines should take the place of the present lines '21' to '23'. Line 116 will also have to be changed. A possible answer is given on page 11-7.

The logic behind the bounce is controlled by looking at the color of the byte that we are about to move into. If the color is 0 (the color of our wall) then we react. However, if we POKE spaces instead of reverse space characters (Program 7.9(a)), thus removing the line from sight, the ball will still bounce as if the wall is there.

PROGRAM 7.9(a)

```
22 POKE 1024+X,32:POKE 55296+X,0
```

This is because the color memory locations are still black, so a different PEEK value is needed, that of the character in front. If it is a wall then its PEEK value is needed. If the byte in front is the wall, then the PEEK value will be '160'- the character code for a reverse space. So, for a perfect 'Bounce Game', we need to replace lines 115 and 116 with:

PROGRAM 7.9(b)

```
115 A=PEEK(1024+X+Y*40)
116 IF A=160 THEN YI=YI*-1:GOTO 100
```

Now we have two ways of making the ball bounce. Firstly by keeping a constant check on the values of 'X' and 'Y' and secondly, by means of the PEEK command.

Armed with this knowledge and experience (not to mention the PEEK and POKE commands) we can seriously start to produce a ball game. To make a fresh start, type in NEW, after saving the program if you wish, and we will begin in earnest.

### A Moveable Bat

The bat will move to and fro along the bottom line of the screen, key '1' being used to move the bat left and '9' to move it right. Obviously, an INPUT command can't be used or the program will stop after each entry and wait until the RETURN key is pressed. Instead, the 'GET' command is used which means that characters can be entered straight from the keyboard without having to press RETURN. To demonstrate this, enter the program below:

PROGRAM 7.10

```
10 PRINT"<CLR>"
20 GET A$:IF A$="" THEN 20
30 IF A$="1" THEN PRINT "LEFT"
40 IF A$="9" THEN PRINT "RIGHT"
50 GOTO 20
```

The program will not leave line 20 unless a key is pressed, and if that key is not '1' or '9', then it is ignored.

### EXERCISE 7.8

Write a program that will output "CORRECT" if the 'O' key is pressed first followed by the 'K' key and will output "INCORRECT" if any other key or combination of keys is pressed. A possible answer is given on page 11-7.

For the finished game, the bat will be made two characters long and it will move along the bottom line of the screen (locations 1984 to 2023) by means of the '1' and '9' keys. The variable 'BC' will be used as the bat column variable and 'CB' as the one that 'remembers' where the bat was (similar to 'X' and 'XX' in the moving ball program). A further refinement is added so that the program checks that the bat stays on the screen. This time we check that 'BC' lies between '0' and '38' inclusive (NOT '39' because the bat is two characters long).

### PROGRAM 7.10(a)

```
10 REM MOVING BAT
20 PRINT "<CLR>"
30 GET A$:IF A$="" THEN 30
40 IF A$="9"AND BC<38 THEN BC=BC+2:GOTO
70
50 IF A$="1" AND BC>0 THEN BC=BC-2:GOTO
70
60 GOTO 30
70 POKE 1984+CB,32 : POKE 1984+CB+1,32
80 POKE 1984+BC,160: POKE 1984+BC+1,160
90 POKE 56256+BC,0 : POKE 56256+BC+1,0
100 CB=BC
110 GOTO 30
```

### EXERCISE 7.9

Amend Program 7.10(a) so that the bat size is three characters long. A possible answer is given on page 11-7.

## Building The Wall

Using the logic demonstrated in Program 7.3(a), we will build a multi-colored wall. The top two lines of the screen will be left to display the current score and other such data.

### PROGRAM 7.10(b)

```
5 REM DRAW THE WALL
10 PRINT"<CLR>"
20 FOR X=0 TO 39
30 FOR Y=2 TO 7
40 POKE 1024+X+Y*40,160
50 POKE 55296+X+Y*40,Y
60 NEXT Y,X
```

## Demolishing The Wall

Now that the wall has been built, it is necessary to think about how the ball will demolish it! The blocks can and will be removed by POKEing a space character into that location. To experiment with this, enter the following command:

### PROGRAM 7.10(c)

```
POKE 1164,32
```

and a block will disappear from the screen. To make the game slightly easier, the blocks will be removed two at a time. It is necessary, therefore, to ensure that the POKE statements are made only with columns 2, 4, 6 and so on up to 38, i.e. the column variable must be an even number.

### EXERCISE 7.10

Extend the wall-building program so that the wall will first be built and then demolished brick by brick and row by row. A possible answer is given on page 11-7.

Suppose that the wall is still complete and the ball is approaching the tenth row. Since ten is an even number, there's no problem - we just clear a section of the wall and bounce the ball away. However, if the ball was approaching the 13th column, it's not so straight-forward. Since 13 is an odd number we need to remove two blocks starting at the 12th column. So it is necessary to check whether the column variable is even or not.



Program 7.10(d) distinguishes between even and odd numbers. Line 20 does the actual calculations (if  $A=3$  then  $\text{INT}(A/2)=1$ , whereas  $A/2=1.5$ ).

#### PROGRAM 7.10(d)

```
10 INPUT"NUMBER";A
20 IF INT(A/2)<>A/2 THEN PRINT"ODD":GOTO
10
30 PRINT"EVEN":GOTO 10
```

Now, to put two of the programs together - the one that builds the wall followed by the endlessly bouncing ball. It will be slightly different in the final version because the ball will only bounce up from the bottom if the bat is in the way, whereas it bounces automatically in the current program whenever it reaches the bottom of the screen, i.e. when  $Y=24$ .

First enter the wall building program, Program 7.10(b) and then Program 7.11.

#### PROGRAM 7.11

```
100 REM BOUNCING BALL
110 X=11:Y=11
120 XI=1:YI=1
130 POKE 1024+X+Y*40,81
140 POKE 55296+X+Y*40,2
150 XX=X:YY=Y
160 IF X<1 OR X>38 THEN XI=XI*-1
170 IF Y<1 OR Y>23 THEN YI=YI*-1
180 X=XX+XI
190 Y=YY+YI
200 A=PEEK(1024+X+Y*40)
210 IF A<>160 THEN 260
220 YI=YI*-1
230 IF INT(X/2)<>X/2 THEN X=X-1
240 POKE 1024+X+Y*40,32
250 POKE 1024+(X+1)+Y*40,32
255 IF X<1 THEN YI=1:GOTO 160
260 POKE 1024+XX+YY*40,32
270 GOTO 130
```

Line 230 makes sure that the bricks are demolished in the correct pattern (i.e. 'X' is an even number). Line 180 ensures that if the ball breaks through the wall then it bounces before it reaches the top line (remember the top line is reserved for scoring details). The rest of the program is straight-forward. If the value of 'A' is '160' (the wall), then a block is removed and a bounce is performed. If 'A' is not '160' then the ball moves continuously along its previous path.

#### EXERCISE 7.11

Modify Program 7.11 so that the ball starts off in a random position. Make sure that the start position is below the wall. A possible answer is given on page 11-7.

#### The Final Program

So, now that we've done all the hard work, it's simply a matter of putting all the pieces together and coming up with the finished product. We've used three subroutines, for moving the bat, for moving the ball and the initialisation routine (building the wall, serving the ball etc); rather than giving the list in one go, we'll first look at each subroutine separately. You may now LOAD BREAKOUT from the disk and follow the explanations of the program sections.

Program 7.12 is the subroutine for moving the bat along the bottom of the screen. This subroutine is called twice for every once that the ball moving subroutine is called, thus enabling the bat to move faster than the ball. The character for the bat has been changed from 160 (a reverse space) to 60 (a less-than sign). This now allows us to react differently every time we hit either the bat or the wall.

#### PROGRAM 7.12

```
3000 REM MOVING THE BAT
3010 GET A$:IF A$="" THEN RETURN
3020 IF A$="1" AND BC>0 THEN BC=BC-2:GOT
O 3050
3030 IF A$="9" AND BC<38 THEN BC=BC+2:GO
TO 3050
3040 GOTO 3060
3050 POKE 1984+CB,32:POKE 1984+CB+1,32
3060 POKE 1984+BC,60:POKE 56256+BC,0
3070 POKE 1984+BC+1,60:POKE 56256+BC+1,0
3080 CB=BC
3090 RETURN
```

Next to be developed is the subroutine for moving the ball, Program 7.12(b). On the return from this subroutine, a check is made on the value of NB - if it is '0' then a new ball is not necessary (and the RETURN has been made via line 2210, indicating a valid move). This illustrates the important idea of passing information to and from subroutines by flags, which are set to particular values.

Notice that after we PEEK into the next location (line 2060), before anything else is done, a check is made to see if the ball is approaching the bat. If it is, then the value of A is 60 and the ball is bounced away - line 2065.

The other addition to this subroutine is the score element in line 2090, the 'Y' value being used to calculate the score. 'Y' is subtracted from ten and this value is multiplied by '5' so the bricks nearer the top are worth more points than the bricks lower down, e.g. if Y=3 then  $10-3=7$  and  $7*5=35$  so 35 points are scored for removing a top brick.

#### PROGRAM 7.12(b)

```
2000 REM MOVING BALL
2010 YY=Y:XX=X
2020 IF X<1 OR X>38 THEN XI=XI*-1
2025 IF X<0 OR X=0 THEN XI=+1
2030 IF Y<2 OR Y=2 THEN YI=+1
2035 IF Y>23 THEN NB=1:RETURN
2040 X=XX+XI
2050 Y=YY+YI
2060 A=PEEK(1024+X+Y*40)
2065 IF A=60 THEN YI=YI*-1:GOTO 2040
2070 IF A<>160 THEN 2140
2080 YI=YI*-1
2090 SC=SC+5*(10-Y)
2100 IF INT(X/2)<>X/2 THEN X=X-1
2110 POKE 1024+XX+YY*40,32
2120 POKE 1024+(X+1)+Y*40,32
2130 GOTO 2010
2140 POKE 1024+XX+YY*40,32
2150 IF Y=1 THEN YI=1:GOTO 2010
2160 POKE 1024+X+Y*40,81
2170 POKE 55296+X+Y*40,2
2180 RETURN
```

Now for the final subroutine - Program 7.12(c) - or, rather, the final two subroutines in one. Lines 1000 to 1050 are concerned with building the wall, displaying score information, setting SCORE to '0' and Ball to '1' (the ball number currently in play). Three goes have been allowed and a check has been made towards the end of the subroutine on the value of the current ball number. This part is needed only at the beginning of each new game.

The second part of the subroutine is concerned with serving a new ball, and so 'NB' is reset to zero (remember it would have been set to '1' in the previous subroutine to indicate the last ball had been lost) and lines 1240 to 1270 ensure a random starting position (from row 10) and initial random direction. Best score will be explained in the next section.

#### PROGRAM 7.12(c)

```

1000 REM INITIALIZATION ROUTINE
1010 FOR X=0 TO 39
1020 FOR Y=2 TO 7
1030 POKE 1024+X+Y*40,160
1040 POKE 55296+X+Y*40,Y
1050 NEXT Y,X
1060 BC=0
1070 POKE 1984+BC,60:POKE 56256+BC,0
1080 POKE 1984+BC+1,60:POKE 56256+BC+1,0
1090 PRINT"<HOME>SCORE:";SC;TAB(12);"BALL:";B,"BEST:";BEST
1100 B=1
1110 SC=0
1120 REM NEW BALL SERVED
1130 NB=0
1140 Y=11
1150 X=INT(RND(1)*36)+2
1160 YI=1
1170 XI=INT(RND(1)*2)
1180 IF XI=0 THEN XI=-1
1190 RETURN

```

The final part of the listing - Program 7.12(d) - is the main program that controls the calling of the subroutines.

The program loops around lines 60 to 160, continually checking to see whether a new ball is necessary (lines 80 to 130), or if all the bricks have been demolished. The maximum possible score is set to 3300 (line 70), and a bonus of 1000 is also made for each ball left over after the wall has been demolished.

When the current game is over - either all 3 balls have been used or the entire wall has been demolished - lines 180 to 220 display the score and prompt for the next game. The best score ('BEST') is initially set to 0 and a check is made in line 190 to see whether the current score is better, with appropriate action being taken.

#### PROGRAM 7.12(d)

```
5 PRINT "<CLR>"
10 REM FINAL PROGRAM
20 POKE 53281,1
30 POKE 53280,1
40 BEST=0
50 GOSUB 1000
60 GOSUB 2000
70 IF SC=3300 THEN 170
80 IF NB=0 THEN 140
90 IF B=3 THEN 180
100 POKE 1024+X+Y*40,32
110 B=B+1
120 GOSUB 1120
130 GOSUB 3000
140 GOSUB 3000
150 GOTO 60
160 PRINT"<HOME>SCORE:";SC;TAB(12);"BALL
   :";B,"BEST:";BEST
170 REM GAME OVER
180 PRINT"GAME ","OVER"
190 IF SC>BEST THEN PRINT"NEW BEST SCORE
   ":BEST=SC
199 PRINT"SCORE",SCORE
200 PRINT" PRESS ANY KEY FOR ANOTHER GO"
210 GET A$:IF A$="" THEN 210
220 PRINT"<CLR>":SC=0:B=0:NB=0:GOTO 50
```

So that's the game. Finito!! You can improve the game by incorporating some of the exercises that you've done, e.g. introducing a skill factor via a variable bat size of '1' or '2' or even '3' characters long and by improving the bounce features of the ball as it comes off the bat.

This program has been explained in a fair amount of detail, and hopefully has stimulated you along the way such that you have evolved some ideas of your own for video games. There really is an almost infinite variety of games that could be devised. We shall look at a couple of spin-offs from this game in the rest of the chapter, but by now you should be busy thinking up your own.

## PART TWO

### Balltrap

This is the first of the two games that are a direct follow-on from the work done during the development of the breakout game. All the techniques used in this game will already be familiar to you. So it's merely the logic (or algorithms) that has to be explored.

The aim of this game is to trap a bouncing ball in a 'basket' at the top of the screen - not by moving a bat, but by drawing colored lines from which the ball can bounce and hence be guided towards the desired destination.

Programs 7.13 and 7.14 will illustrate the concepts discussed. The actual BALLTRAP game can be loaded later in this chapter.

First of all, Program 7.13 looks at the drawing of the lines. As before, the 'I' and '9' keys control sideways movement but in addition here, the 'Z' and 'M' keys control movement up and down the screen. Since this time the aim is to leave a trail, the coding is a lot easier. Once again the top row is being kept for scoring details.

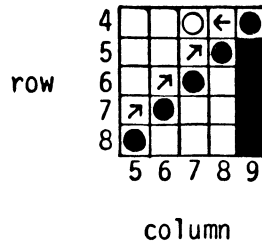
#### PROGRAM 7.13

```
10 REM COLORED LINE
20 PRINT "<CLR>"
30 BX=10: BY=10
40 POKE 1024+BX+BY*40,160
50 POKE 55296+BX+BY*40,2
2000 GET A$:IF A$="" THEN 2000
2010 IF A$="I" AND BX<>0 THEN BX=BX-1:GOTO 2060
2020 IF A$="9" AND BX<>39 THEN BX=BX+1:GOTO 2060
2030 IF A$="Z" AND BY<>24 THEN BY=BY+1:GOTO 2060
2040 IF A$="M" AND BY<>1 THEN BY=BY-1:GOTO 2060
2050 GOTO 2000
2060 GOTO 40
```

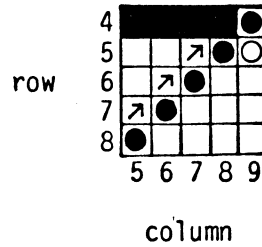
### Better Bouncing

Now for the bouncing ball! To create this it would be possible to use exactly the same code as before, but this time the aim is to obtain a more realistic bounce. The following diagrams illustrate some of the problems:

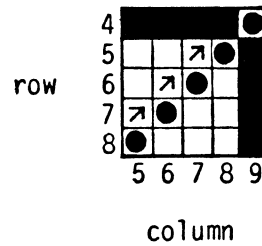
Example 1



Example 2



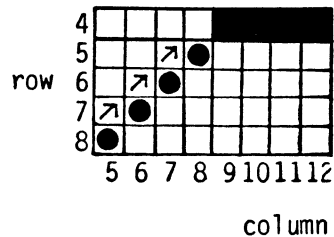
Example 3



In the previous program, position (9,4) would have been examined (using PEEK) and, if found unoccupied, the ball would have moved there. However, in the first two examples, the dotted ball indicates a more realistic move and, in Example 3, the ball really ought simply to retrace its steps.

The other case that must be considered is illustrated in Example 4.

Example 4



This time  $(X, Y+YI)$ , i.e.  $(8,4)$ , and  $(X+XI, Y)$ , i.e.  $(9,5)$ , will have been checked, and being found to be empty, the ball will be moved to  $(X+XI, Y-YI)$ , i.e.  $(9,4)$ , which is occupied by the wall!

What is needed in this case is for the ball to retrace its steps, so that after  $(X, Y+YI)$  and  $(X+XI, Y)$  have both been checked,  $(X+XI, Y+YI)$  is checked and if found to be occupied, both  $XI$  and  $YI$  are altered.

This is done in Program 7.14 (lines 250 to 340) which draws some random lines to test this ball-bouncing logic. In this program, the variables that were used last time to 'remember' the old position of the ball are not necessary now because in lines 250, 280 and 310, the relevant shape code is found before  $XI$  and  $YI$  are altered.

#### PROGRAM 7.14

```
10 REM DRAWING ASSORTED LINES
20 FOR X=10 TO 35
30 POKE 1104+X, 160:POKE 55376+X,2
40 NEXT X
50 FOR Y=1 TO 15
60 POKE 1114+Y*40, 160:POKE 55386+Y*40,
  2
70 NEXT Y
80 BX=10: BY=20
200 REM BETTER BOUNCING BALL
210 X=5: Y=20
220 XI=1: YI=1
230 IF X<1 OR X>38 THEN XI=XI*-1
240 IF Y<1 OR Y>23 THEN YI=YI*-1
250 A=PEEK(1024+X+Y*40)
260 IF A<> 160 THEN 280
270 XI=XI*-1
280 A=PEEK(1024+X+(Y*40)+YI*40)
290 IF A<>160 THEN 310
300 YI=YI*-1
310 A=PEEK(1024+(X+XI)+(Y*40)+YI*40)
320 IF A<> 160 THEN 340
330 XI=XI*-1:YI=YI*-1
340 POKE 1024+X+Y*40,32
350 X=X+XI:Y=Y+YI
360 POKE 1024+X+Y*40,81
370 POKE 55296+X+Y*40,2
380 GOTO 230
```



Now for the trap. This has to be made in a different color and also has to be surrounded by a border of the same color that the lines will be drawn in. One suggested color scheme is white for the background, a black ball and red for the trap. (The line numbers used are as they will appear in the final program.) You may now LOAD BALLTRAP from disk and follow these sections.

#### PROGRAM 7.14(a)

```
4000 REM DRAWING THE TRAP
4010 FOR X=1 TO 4
4020 POKE 1148+X,160:POKE 55420+X,2
4030 POKE 1188+X*40,160:POKE 55460+X*40,2
4040 POKE 1228+X,160:POKE 55500+X,2
4042 POKE 1109+X,83:POKE 55381,5
4044 POKE 1249+X,83:POKE 55781,5
4050 NEXT X
4060 RETURN
```

The next Program (7.14(b)) handles the scoring - again a 'best score' is used to add interest. This time, of course, since the ball has to be trapped in the shortest possible time, 'B3' is initially set to 1000, and this also provides the time limit so each time the program loops a check is made to make sure that this limit has not been exceeded.

The final problem arises when attempting to determine exactly when the ball has been trapped - and this is quite easily done by filling the trap with green hearts. Thus when the PEEK value of the next co-ordinate is 83 the ball has hit a heart and is now in the trap.

So here's the final listing, subroutine by subroutine (remember to include the one above for drawing the trap!).

### PROGRAM 7.14(b)

```
1000 REM INITIALIZATION ROUTINE
1010 T=0:SC=0
1020 TF=0
1030 X=INT(RND(1)*38)
1040 Y=INT(RND(1)*23)
1050 IF X<11 AND X>1 THEN 1030
1060 XI=1:YI=1
1070 BX=38:BY=23
1080 POKE 1024+BX+BY*40,160
1090 POKE 55296+BX+BY*40,2
1100 RETURN
```

Program 7.14(c) should present no great difficulty as the techniques have already been well explored. Lines 1100 to 1130 give a random starting position for the ball (ensuring that it isn't in the trap to start with!). Lines 1140 and 1150 make sure that the ball is initially moving away from the trap, and lines 1160 to 1190 set the starting position of the line to the bottom right-hand corner.

The flag TF, "trapflag", is used in a similar way to the way NB, "newball", was used last time and is set to 1 only when the ball enters the trap.

### PROGRAM 7.14(c)

```
2000 REM DRAWING THE LINES
2010 GET A$:IF A$="" THEN RETURN
2020 IF A$="1" AND BX<>0 THEN BX=BX-1:G
OTO 2070
2030 IF A$="9" AND BX<>38 THEN BX=BX+1:
GOTO 2070
2040 IF A$="Z" AND BY<>2 THEN BY=BY-1:G
OTO 2070
2050 IF A$="M" AND BY<>23 THEN BY=BY+1:
GOTO 2070
2070 POKE 1024+BX+BY*40,160
2080 POKE 55296+BX+BY*40,2
2090 RETURN
```

So far, this program is very similar to the earlier ball game except that, this time, if no key is pressed or the wrong key is pressed the program RETURNS.

PROGRAM 7.14(c) contd.

```
3000 REM MOVING THE BALL
3010 IF X<1 OR X>38 THEN XI=XI*-1
3020 IF Y<1 OR Y>23 THEN YI=YI*-1
3060 A=PEEK(1024+X+(Y+YI)*40)
3070 IF A<>160 THEN 3090
3080 YI=YI*-1
3090 A=PEEK(1024+(X+XI)+(Y+YI)*40)
3100 IF A<>160 THEN 3120
3110 XI=XI*-1
3120 POKE 1024+X+Y*40,32
3130 X=X+XI
3132 IF X>38 OR X<1 THEN XI=-XI:GOTO 3130
3140 Y=Y+YI
3142 IF Y>22 OR Y<2 THEN YI=-YI:GOTO 3140
3150 A=PEEK(1024+X+Y*40)
3160 IF A=83 THEN TF=1
3170 POKE 1024+X+Y*40,81
3180 POKE 55296+X+Y*40,2
3190 RETURN
```

The only addition to this subroutine has been lines 3150 and 3160 which check (before printing the ball!) to see whether or not the trap has been entered, and if so the "trapflag" is set.

After the ball enters the trap, it would look more effective were the ball to continue to move until it hits the back of the trap, and this is accomplished by the following subroutine, Program 7.14(d) (and is an example of one subroutine calling another!).

PROGRAM 7.14(d)

```
5000 REM BALL IN TRAP
5010 GOSUB 3000
5020 IF X<>8 THEN 5010
5030 RETURN
```

Now for the final part of the program, which loops around lines 55 to 130, constantly checking that time is still available (line 100) and that the ball hasn't been trapped (line 80, via the flag 'TF'). Again, the line-drawing subroutine is called twice for every once that the ball-moving subroutine is called.

PROGRAM 7.14(e)

```
5 PRINT"<CLR>"
10 REM BALL TRAP
20 POKE 53281,1:POKE 53280,1
30 T1$="000000":S=T1
40 GOSUB 4000:REM THE TRAP
50 GOSUB 1000:REM INITIALIZATION
55 S=TI
60 T=INT(((S/60)*100)/100)
70 GOSUB 3000:REM MOVING THE BALL
80 IF TF=1 THEN 140
90 PRINT"<HOME>TIME:";T
100 IF T=120 THEN 240
110 GOSUB 2000:REM DRAWING THE LINES
120 GOSUB 2000:REM DRAWING THE LINES
130 GOTO 55
140 GOSUB 5000:REM CHECK TRAP
150 IF T>BS THEN 180
160 BS=T
170 PRINT"NEW BEST TIME",T
180 FOR X=1 TO 500
190 NEXT X
195 GET A$:IF A$<>"" THEN 195
200 PRINT"PRESS ANY KEY FOR ANOTHER GO"
210 GET A$:IF A$="" THEN 210
220 PRINT"<CLR>":GOTO 30
230 REM OUT OF TIME
240 PRINT"OUT OF TIME !!!!"
250 GOTO 180
```

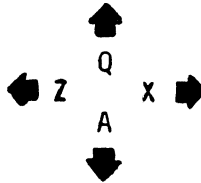
## PART THREE

### Blockade

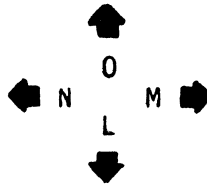
Now for the final part of this project. PART 2 improved upon the ball-movement from the breakout game and we will now improve upon the user control aspect of the game. In the breakout game, the user controls the (limited) movement of the bat and in balltrap this was extended to draw lines anywhere on the screen. This blockade game is a game for two people, who simultaneously draw lines on the screen, trying to avoid going over any lines already drawn. The first person to cross a line loses the game! Of course, the movements are not simultaneous - they just happen so quickly that it appears that way! Programs 7.15(a) to 7.15(c) will be used to illustrate the concepts.

The keys to be used for the different directions are:

Player 1



Player 2



Player 1 will be assigned the coordinates (AX,AY) with increments AR and AC, while Player 2 will have (BX,BY) with BR and BC as the increments. Diagonal movements are not permitted, so this introduces a further constraint to the problem.

First of all, the coding for Player 1 is developed in Program 8.15(a). This subroutine evaluates the new direction if one of the four controlling keys is pressed; otherwise, the direction remains the same as it was before and the RETURN is made from line 1010. However, if one of the controlling keys has been pressed, then both AC and AR are made zero before proceeding (diagonal movement not being permitted means that one of these increment variables must always be zero). By the time line 1060 is reached, there is only one possibility left - "A" has been pressed. So, of

course, it is not necessary to test for this.

PROGRAM 7.15(a)

```
1000 GET A$
1010 IF A$<>"Z" AND A$<>"X" AND A$<>"Q"
AND A$<>"A" THEN RETURN
1020 AR=0:AC=0
1030 IF A$="Z" THEN AC=-1:RETURN
1040 IF A$="X" THEN AC=+1:RETURN
1050 IF A$="Q" THEN AR=-1:RETURN
1060 AR=+1:RETURN
```

This subroutine must of course be called, and the initialization and calling procedure is developed in Program 7.15(b).

PROGRAM 7.15(b)

```
10 PRINT"<CLR>"
20 AX=0:AY=0
30 AR=1:AC=0
50 POKE 1024+AX+AY*40,160:POKE 55296+AX
+AY*40,2
60 GOSUB 1000
70 AX=AX+AC
80 IF AX=39 THEN AX=0
90 IF AX=-1 THEN AX=38
100 AY=AY+AR
110 IF AY=24 THEN AY=1
120 IF AY=0 THEN AY=23
130 GOTO 50
```

Once this section is inserted, the program will run but does not yet have the characteristics required in the game. However, if you insert the following lines - Program 7.15(c) - then you'll begin to get the idea of the game.

PROGRAM 7.15(c)

```
25 SC=0
35 SC=SC+1
125 A=PEEK(1024+X+Y*40)
127 IF A=160 THEN 140
130 GOTO 35
140 PRINT "SCORE=";SC
150 PRINT "PRESS 'P' TO PLAY AGAIN"
160 GET A$:IF A$ <> "P" THEN 160
170 GOTO 10
```

From this stage, it's simply a question of getting similar subroutines for Player 2 and deciding on the strategy and, hence, scoring of the game. There are two choices, either the game is lost when a player moves over a location which his opponent has already occupied, or the game is lost when a player moves over any square which has been occupied, and the first player to win, say, 10 games is the overall winner.

These two alternatives will be incorporated in the same program, INPUT statements being used to determine the player's colors. First, the two subroutines corresponding to the above listings, one for Player 1 and one for Player 2, are given in Program 7.16(d). Obviously, these are basically identical, the major difference in their use being how the program calls each one. First type in NEW, then load in the BLOCKADE program from disk by LOAD"BLOCKADE",8.

#### PROGRAM 7.16(a)

For Player 1:

```

1000 REM PLAYER ONE
1010 GET A$ :IF A$="" THEN 1090
1020 IF A$<>"Z" AND A$<>"X" AND A$<>"Q"
AND A$<>"A" THEN 1090
1030 AC=0: AR=0
1040 IF A$="Z" THEN AC=-1:GOTO 1090
1050 IF A$="X" THEN AC=+1:GOTO 1090
1060 IF A$="Q" THEN AR=-1:GOTO 1090
1070 AR=+1
1090 AX=AX+AC
1100 IF AX=40 THEN AX=0
1110 IF AX=-1 THEN AX=39
1120 AY=AY+AR
1130 IF AY=24 THEN AY=1
1140 IF AY=0 THEN AY=23
1150 RETURN

```

For Player 2:

```

2000 REM PLAYER TWO
2010 GET A$ :IF A$="" THEN 2080
2020 IF A$<>"N" AND A$<>"O" AND A$<>"M"
AND A$<>"L" THEN 2080
2030 BC=0:BR=0
2040 IF A$="N" THEN BC=-1:GOTO 2080
2050 IF A$="M" THEN BC=+1:GOTO 2080
2060 IF A$="O" THEN BR=-1:GOTO 2080
2070 BR=+1
2080 BX=BX+BC
2090 IF BX=40 THEN BX=0

```

```

2100 IF BX=-1 THEN BX=39
2110 BY=BY+BR
2120 IF BY=24 THEN BY=1
2130 IF BY=0 THEN BY=23
2140 RETURN

```

As this is a two-player game, it would look much more professional were players' names to be input, rather than using the terms 'Player 1' and 'Player 2'. The next subroutine, Program 7.16(b), prompts for the players' names and colors (displaying an appropriate message depending on whether both colors are the same or not). It initializes the scores and sets up the board (again using row 0 for scoring details). Notice that a check is made on the length of the players' names (lines 3020 and 3040) and if either is greater than 11 characters long it has to be input again in an abbreviated form. Also, a check is made on the color number input, to ensure that it is in the correct range.

#### PROGRAM 7.16(b)

```

3000 REM START POINT
3005 Z=PEEK(53281) AND 15
3010 INPUT"NAME OF PLAYER ONE";P$
3020 IF LEN(P$)>11 THEN 3010
3030 INPUT"NAME OF PLAYER TWO";Q$
3040 IF LEN(Q$)>11 THEN 3030
3050 PRINT"COLOR CODE (0-15) FOR ";P$;
3060 INPUT C1
3070 IF C1=Z THEN 3050
3080 PRINT"COLOR CODE (0-15) FOR ";Q$;
3090 INPUT C2
3100 IF C2= Z THEN 3080
3110 IF C2=C1 THEN PRINT"AVOID ALL LINE
S": GOTO 3130
3120 PRINT"AVOID ALL YOUR OPPONENT'S LI
NES"
3130 PRINT"PRESS ANY KEY TO BEGIN"
3140 GET A$ : IF A$="" THEN 3140
3150 S1=0:S2=0
3160 PRINT "<CLR>" P$;S1,P2;S2
3170 RETURN

```

The next subroutine, Program 7.16(c), initializes the players' starting positions.



### PROGRAM 7.16(c)

```
4000 REM START POSITIONS
4010 REM PLAYER ONE
4020 AX=0 : AY=1
4030 AR=1 : AC=0
4040 POKE 1024+AX+AY*40,160
4050 POKE 55296+AX+AY*40,C1
4060 REM PLAYER TWO
4070 BX=39 : BY=23
4080 BR=-1 : BC=0
4090 POKE 1024+BX+BY*40,160
4100 POKE 55296+BX+BY*40,C2
4110 RETURN
```

Finally, to the central part of the game - Program 8.16(d) - which serves to call the initializing subroutines and then repeatedly calls 1000 and 2000. As it returns, it checks that the line isn't drawing over an opponent's line already there (lines 60 and 100), or any line when both players are using the same color. When this check is positive, the appropriate player's score is incremented and then a further check is made to see whether it has yet reached 10 (lines 190 and 230).

### PROGRAM 7.16(d)

```
10 REM BLOCKADE
20 REM
30 GOSUB 3000 : REM START POINT
40 GOSUB 4000 : REM START POSITIONS
50 GOSUB 1000 : REM PLAYER ONE
60 A=PEEK(55296+AX+AY*40) AND 15
70 IF A=C2 THEN 160
80 POKE 1024+AX+AY*40,160
90 POKE 55296+AX+AY*40,C1
100 GOSUB 2000 : REM PLAYER TWO
110 A=PEEK(55296+BX+BY*40) AND 15
120 IF A=C1 THEN 250
130 POKE 1024+BX+BY*40,160
140 POKE 55296+BX+BY*40,C2
150 GOTO 50
160 POKE 1024+AX+AY*40,102
170 POKE 55296+AX+AY*40,C1
180 FOR X=0 TO 300 : NEXT X
190 S2=S2+1
200 GOSUB 3160 : REM PRINT SCORE
210 IF S2=10 THEN A$=Q$: GOTO 350
220 PRINT" PRESS SPACE BAR TO CONTINUE"
230 GET A$ : IF A$<>" " THEN 230
```

```

240 GOTO 40
250 POKE 1024+BX+BY*40,102
255 PRINT "<CLR>"
260 POKE 55296+BX+BY*40,C2
270 FOR X=0 TO 300 : NEXT X
280 S1=S1+1
290 GOSUB 3160:REM PRINT SCORE
300 IF S1=10 THEN A$=P$:GOTO 350
310 GOTO 220
320 GET A$:IF A$<>" " THEN 230
330 GOTO 40
340 REM THE END
350 PRINT"CONGRATULATIONS !!!! ";A$
360 PRINT" PRESS ANY KEY FOR A NEW GAME
"
370 GET A$ : IF A$="" THEN 370
380 PRINT "<CLR>"
390 GOSUB 3050:REM TO COLOR CHOICE
400 GOTO 30

```

At this stage you have your third 'bouncing-ball' game and, once again, further developments are up to you. Using the ideas in parts 1, 2 and 3 of this chapter, there are many possible variations on the game and the more you modify them, the more you can identify with them as your own creations! One thing you could try is changing the logic behind moving the ball in Breakout. For instance, you could try poking the space BEFORE you actually poke the ball into the next screen location. There are lots of possibilities.

## CHAPTER

# 8

### PART ONE

#### Sprightly Sprites

**A** further advanced feature of the C-64 is its ability to use sprites or, in the Commodore jargon, Moveable Object Blocks (MOBs). Eight different sprites, numbered from zero to 7, can be used at any one time. These are really rather well described by the dry name 'Moveable Object Block', as they are just that: a large block approximately  $7\frac{1}{2}$  characters in size, which is very readily moved about the screen. In addition, facilities are available for checking when sprites collide either with other sprites or with objects in the background. In its simplest form, a sprite is a character object ( $3 \times 2\frac{1}{2}$ ) of a single color. They can, however, be enhanced by being designed in three colors or expanded to twice the size in either the horizontal or vertical directions or both. Thus, the available features of sprites are:

- \* Normal size - 3 characters by  $2\frac{1}{2}$  characters.
- \* Expanded horizontal size - 6 characters by  $2\frac{1}{2}$  characters.
- \* Expanded vertical size - 3 characters by 5 characters.
- \* Expanded both axes - 6 characters by 5 characters.
- \* Three colors, any of above sizes (multicolor mode).

In fact, the sprite's size is more accurately defined in terms of pixels. As Figure 1 shows, each normal-sized sprite occupies a rectangle 24 pixels wide in the horizontal direction and 21 pixels in the vertical.

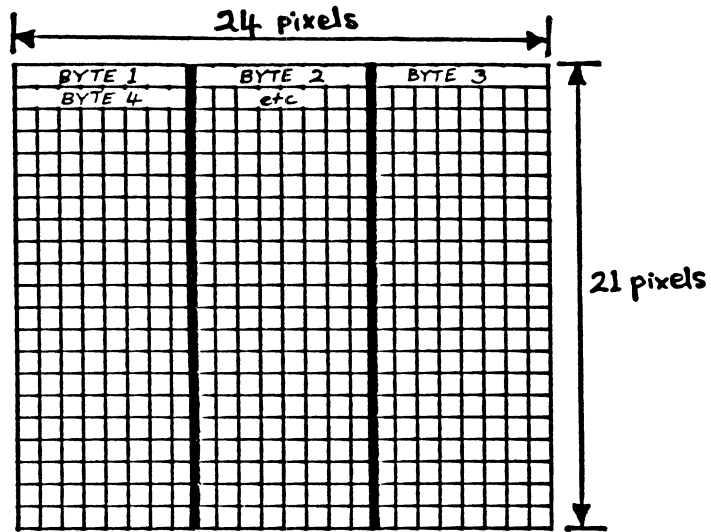


FIGURE 8.1

As a single-color sprite is made up of pixels that are either on or off, each pixel requires one bit of information to define its state. To store all the information necessary to define a sprite, therefore,  $24 \times 21 = 504$  bits of memory are needed. This is arranged in bytes as shown in Figure 8.1, with the eight top left-hand pixels stored in the first byte, the top middle eight bits being stored in byte 2 etc. Calculations to work out the byte value for each group of eight pixels are done in exactly the same way as for the graphics characters discussed in Chapter 6. Thus, the sprite shown in Figure 8.2 is defined by the POKE values:

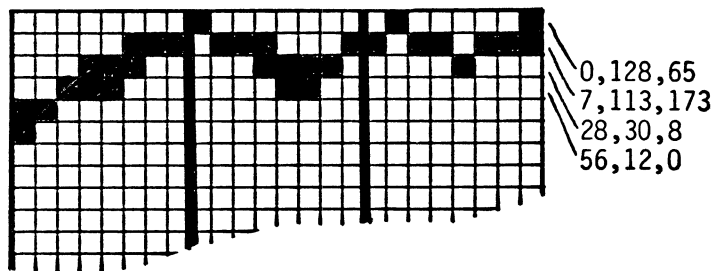


FIGURE 8.2

## EXERCISE 8.1

Calculate the POKE values for the first 12 bytes of the sprite shown in Figure 8.3.

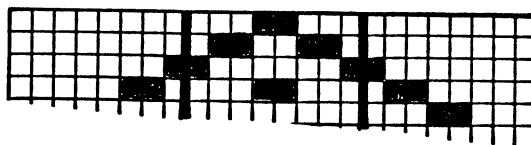


FIGURE 8.3

For the control of sprites, an area of memory is reserved, starting at 53248. Thus, the addresses of the various sprite functions are located in the 46 bytes following this base address. As it's far easier to remember numbers of two digits than the five-digit memory locations, all sprite function addresses are given as  $R+N$  where  $R$ =sprite base register address of 53248 and  $N$ =sprite number.

### Sprite Variety

The discussion so far has considered only the simplest of sprites but other varieties exist, as described above. The nature of each sprite other than the normal sized, single-colored variety is described below, i.e.

#### \* Expanded horizontal sprites

One byte, 53277 ( $R+29$ ) stores information about the horizontal size of sprites, each bit defining the scale of one particular sprite. For instance, to expand sprite number 0 horizontally, bit 0 of 53277 ( $R+29$ ) is set to a 1. When expanded horizontally, each horizontal pixel of the sprite is doubled and thus the resolution remains at 2 even though the sprite occupies 48 pixels on the screen.

#### \* Expanded vertical sprites

One byte, 53271 ( $R+23$ ) stores information about the vertical size of sprites. As with the horizontal size, each bit defines the scale of one particular sprite. When expanded vertically, each vertical pixel of the sprite is repeated and thus the resolution remains at 21.

**\* Expanded both axes**

When both the relevant bits are set to 1 then both axes are expanded. Thus, if both bit 2's of R+23 and R+29 are set to 1 then sprite 2 appears as a double sized sprite.

**\* Review of sprite size definition**

Figure 8.4 shows the two size bytes set and reviews the effect this has on the relevant sprites.

Address	7	6	5	4	3	2	1	0	Register
53271	1 0 0 1 0 1 1 0								R + 23 (vertical)
53277	1 0 1 1 1 0 0 1								R + 29 (horizontal)

Sprite number	Horizontal scale	Vertical scale
0	double	single
1	single	double
2	single	double
3	double	single
4	double	double
5	double	single
6	single	single
7	double	double

FIGURE 8.4

**\* Multicolor sprites**

All eight sprites can be defined as multicolored by setting the relevant bit of R+28 (53276) to a 1, i.e.

	7	6	5	4	3	2	1	0	
53276	0 1 1 1 0 0 0 1								R+28

means that sprites 1,2,3, and 7 are single color and sprites 0,4,5 and 6 are multicolored.

Multi-colored sprites can have up to three different colors defined in the usual C-64 way, but two of these must be common to all sprites. Thus some sample allowable combinations are:

```

Sprite 0  red  yellow  blue
Sprite 1  red  yellow  white
Sprite 3  red  yellow  purple
    
```

It is the method of storing the color data that gives rise to this limitation as the least-significant nybbles (LSN - remember, a nybble is half a byte - i.e. four bits) of two bytes are made available for storing the two common sprite colors and one LSN for each of the individually defined colors i.e.

Register number (R+...)	Effect
37	set overall color 1
38	set overall color 2
39	set sprite 0 to sprite color
40	set sprite 1 to sprite color
41	set sprite 2 to sprite color
42	set sprite 3 to sprite color
43	set sprite 4 to sprite color
44	set sprite 5 to sprite color
45	set sprite 6 to sprite color
46	set sprite 7 to sprite color

FIGURE 8.5

As multicolor sprites are more complex than the single colored variety, more information is needed to define them fully. In multicolor mode, each pixel can be off, color 1, color 2 or sprite color. Two bits of information are needed to store the four possible states. These are allocated as below:

bit 1 state	bit 2 state	resultant color
0	0	background (transparent)
0	1	color 1 (general)
1	1	color 2 (general)
1	0	sprite color (individual)

FIGURE 8.6

Thus, multicolored sprites have only half the resolution of single colored ones in the horizontal direction (12 pixels) as the bits are paired, each bit pair containing the information necessary to define one two-pixel element.

Putting this into practice, Figure 8.7 shows the top left-hand corner of a sprite, both as it is stored and as it appears on the screen.

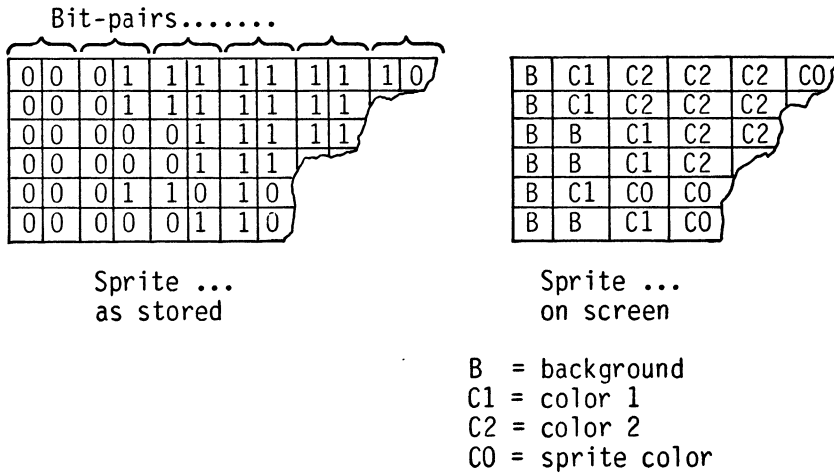


FIGURE 8.7

### The Sprite Generator: SPRITE.GEN

As with user-defined graphics, sprites will be examined during the development of a sprite generator, a utility that facilitates the design of a sprite and its incorporation into a sprite-using program. In structure, this is identical to the Char.Gen program developed in Chapter 6. As Char.Gen was described in detail in Chapter 6, only the differences between Sprite.Gen and Char.Gen are covered in detail in this chapter.

During operation of the program, the sprite will be built up pixel by pixel and continuously displayed on the screen. The sprite chosen for this honor will be sprite zero but at the end of the process, the design can be transferred to any other sprite.

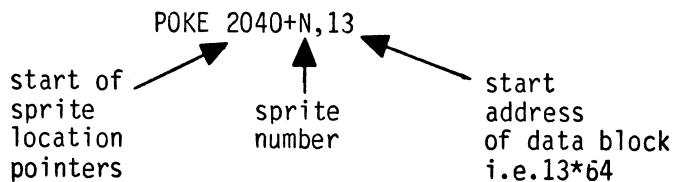


## Initialization

You can now load in the SPRITE.GEN program and follow the explanation of segments. Once designed, sprite data can be stored anywhere within the 16K block of memory that is currently visible to the VIC II chip. In our case this means the first 16K of memory. There is just one restriction however - the first byte of the 63 making up the sprite must be stored at the start of a 64 byte block 0, 64, 128 etc.. One such block lies within the cassette buffer located from 828 to 1023. Its start address is  $13 \times 64$  or 832, and the pixels of an empty sprite can be loaded in with a:

```
FOR X=0 TO 62:POKE 832+X,0:NEXT X
```

Once a sprite is defined, it needs to be assigned a number and the C-64 needs to be told that sprite N is located at block X. This is done using a block of memory from 2040 to 2047 as:



Thus, the address of sprite zero is POKEd in location 2040, that for sprite one into 2041 etc. Therefore, the first few lines of Sprite.Gen read:

### PROGRAM 8.1

```
Clear screen          60000 GOSUB 62000:PRINT "<CLR>"
Set screen color      60010 POKE 53280,3:POKE 53281,1:
Report sprite zero    POKE 2040,13
at block 13
Set sprite zero to   60020 FOR X=0 TO 62:POKE 832+X,0:NEXT
```

The subroutine called from line 60000 puts a title page onto the screen, should you wish to include it. Once defined, the sprite needs to be switched on or, in the jargon, 'enabled'. This is done by setting individual bits of byte R+21 to one. Bit zero controls sprite zero, bit one sprite 1 etc. i.e.

bit number	7	6	5	4	3	2	1	0
R+21	0	0	0	0	0	1	0	0

turns sprite 2 on, i.e.  
POKE R+21,4

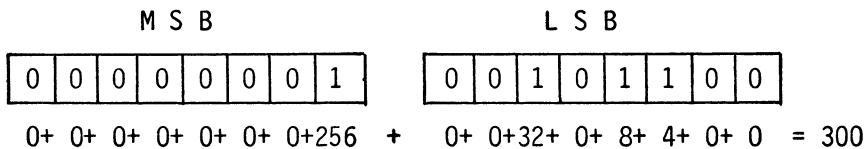
Finally, the sprite needs to be given a location on the screen, there being 200 addressable positions in the vertical (Y) direction and 320 in the horizontal (X).

To define a vertical (Y) position on the screen, one byte is used for each sprite, the actual bytes being: R+1 for sprite 0, R+3 for sprite 1, R+5 for sprite 2 etc. So, to set a Y position of 100 for sprite 0, the requisite command is:

POKE R+1,100

In the case of the X direction, life's a bit more complicated as any one of the 320 horizontal points needs to be addressed. As some of the numbers 0-319 are greater than the maximum value that can be stored in a single byte, extra capacity needs to be found. Eight-bit arithmetic is clearly insufficient and sixteen bit arithmetic is used. Actually, that's not quite true - indeed, two bytes are used but only 9 bits of these are actually needed. The extra one bit required per X address can be provided for all eight available sprites in one byte, R+16 being used for this. As with other bit-wise storage, bit zero is used for the Most Significant Bit of sprite zero's X address, bit 1 for sprite 1 etc. Let's investigate this by setting an address of 300, for sprite 3, and calculating the POKEs required.

The least significant byte cannot exceed 255, so bit zero for the most significant byte needs to be set to 1. As this has a place value of 256, the content of the least significant byte needs to be 44 (300-256) i.e.



Using this knowledge, the next line of Sprite.Gen can be written:

PROGRAM 8.2

```
Switch on sprite zero      60030 POKE 53269,1:
Set Y co-ord for display sprite POKE 53249,74:
Set X co-ord for display sprite POKE 53264,1:POKE 53248,35
```

Once the basic sprite is set up, the other parameters, size and color need to be defined, this routine simply setting the variables for:

```
* horizontally enlarged sprite: EX=1
* vertically enlarged sprite  : EY=1
* multicolor sprite           : MC=1
    color 1                   : C1
    color 2                   : C2
    sprite color              : CO
```

Lines 60040 to 60130 of Program 8.3 carry out this function and set up the sprite by means of the relevant POKEs.

### PROGRAM 8.3

```
60040 CO=6
60050 PRINT"<GRY1><CLR><4DCRSR>"TAB(12)
      ;:INPUT"ENLARGED X  <3LCRSR>";A$
      :IFA$="Y"THENEX=1:GOTO60060
60055 IFA$<>"N"THEN60050
60060 PRINTTAB(12);:INPUT"<3DCRSR>ENLAR
      GED Y  <3LCRSR>";A$:IFA$="Y"THEN
      EY=1:GOTO60070
60065 IFA$<>"N"THENPRINT"<5UCRSR>":GOTO
      60060
60070 MC=0:PRINTTAB(11);:INPUT"<3DCRSR>
      <RED>M<BLU>U<PUR>L<ORNG>T<BRWN>I<
      GRY1>C<GRN>O<LTRED>L<GRY2>O<LTBLU
      >U<RED>R<GRY1>  <3LCRSR>";A$:IFA
      $="N"THEN60120
60075 IFA$<>"Y"THENPRINT"<5UCRSR>":GOTO
      60070
60080 MC=1
60090 INPUT"<RED><2DCRSR><3RCRSR>MULTIC
      OLOUR #1 (<RVSON> <RVSOFF>)<GRY1>
      <5LCRSR>";C1
60095 IF C1<0ORC1>15THENPRINT"<4UCRSR>"
      :GOTO60090
60100 INPUT"<BRWN><DCRSR><3RCRSR>MULTIC
      OLOUR #2 (<RVSON> <RVSOFF>)<GRY1>
      <5LCRSR>";C2
60105 IF C2<0ORC2>15THENPRINT"<3UCRSR>"
      :GOTO60100
60110 POKE53285,C1:POKE53286,C2
60120 IFMC=0THENPRINTTAB(4);
60122 PRINT"<BLU><2DCRSR><4RCRSR>SPRITE
      COLOUR ";:IFMC=1THENPRINT"(<RVSO
      N> <RVSOFF>.)";
```

```

60124 INPUT"<GRY1>      <5LCRSR>";CO$:CO
      =VAL(CO$):IFCO=0ANDCO$<>"0"THENCO
      =99
60126 IF CO<0ORCO>15THENPRINT"<4UCRSR>"
      :GOTO60120
60130 POKE53287,CO:POKE53277,EX:POKE532
      71,EY:POKE53276,MC

```

### Screen display

With the larger screen display required by sprites, no room remains for the clear cursor display that was provided on the Char.Gen program. Also, the initial grid must be simplified to a grid of dots. A much simpler program results as lines 60140 to 60170 testify.

### PROGRAM 8.4

```

60140 PRINT"<CLR><3DCRSR>";TAB(27);"<BL
      U>YOUR":PRINTTAB(27);"SPRITE"
60150 PRINT"<HOME><DCRSR><RED><RVSON>
      ":FORX=1T
      021
60160 PRINT"<RED><RVSON> <RVSOFF><GRN>..
      .....<RED><RVSON>
      > <RVSOFF>"
60170 NEXT:PRINT"<RED><RVSON>
      <BLU>"

```

### Set sprite and display

As with Char.Gen, the array for data storage is DIMed, although this time it is somewhat larger at 24x21. The initial cursor position is then set to 1 (X=1,Y=1). Once this is done, the sprite is cleared to zero in memory and auto-repeat on all keys is turned on.

Once an input has been made, the current cursor position color is reset and then the input is decoded—lines 60300 to 60340 of Program 8.5. The cursor color is set back to zero (black) and a check made for a RETURN (CHR\$(13)) input. The current sprite form is then displayed to the right of the screen, lines 60400 to 60450.

## PROGRAM 8.5

```

60250 X=1:Y=1:POKE55296+81,0:IFZX=0THEN
      ZX=1:DIMA(24,21)
60260 FORB=0TO62:POKE832+B,0:NEXT:POKE6
      50,128:POKE53269,1
60270 GETA$:IFA$=""THEN60270
60280 POKE198,0
60290 POKE55296+40+X+Y*40,5+A(X,Y)
60300 IFA$="A"ANDX>1THENX=X-1
60310 IFA$="S"ANDX<24THENX=X+1
60320 IFA$="W"ANDY>1THENY=Y-1
60330 IFA$="Z"ANDY<21THENY=Y+1
60340 POKE55296+40+X+Y*40,0:IFA$=CHR$(1
      3)THEN61000

```

Tests are then made for the pressing of the function keys, again as was done with Char.Gen.

### f1: set current pixel in sprite

This function is virtually identical to the corresponding function in Char.Gen: it first sets the relevant cell in the array A(X,Y) to 1, turns the screen location ON and sets the screen color. It then sends the program to the routine at 60570, where the pixel is set in memory i.e.

## PROGRAM 8.6

```

      set array           60350 IF A$="<f1>"THEN A(X,Y)=1:
      set screen pixel   POKE 1024+40+X+Y*40,160:
      set screen color   POKE 55296+40+X+Y*40,6:
      GOTO update sprite GOTO 60570

```

The first task is to calculate the byte number (C) for a given array position X. This is defined by  $C = \text{INT}((X-1)/8)$ .

Next, the POKE values for the individual bytes need to be calculated by means of a loop which adds up the individual bit values, i.e.:

## PROGRAM 8.7

```

Calculate byte position 60570 C=INT((X-1)/8)
Add up bit values      { 60575 A=PEEK(829+Y*3+C):IFA(X,Y)=0THENA
                        =AAND(255-(2^(8-(X-(C*8))))))
                        60580 IFA(X,Y)=1THENA=AOR(2^(8-(X-(C*8)
                        )))
Store value in memory  60590 POKE829+Y*3+C,A:GOTO60270

```

### f3: turn current pixel off in sprite

This function is very similar to f1 but in reverse. Little explanation seems necessary.

#### PROGRAM 8.8

```
unset array          60360 IF A$="<f3>"THEN A(X,Y)=0:
clear screen pixel   POKE 1024+40+X+Y*40,46:
set screen color to  POKE 55296+40+X+Y*40,5:
background
GOTO update sprite   GOTO 60570
```

### f5: Print DATA on screen

As sprites are so much larger than individual characters, it is not possible to display both the sprite DATA and the design matrix on the screen at the same time. It is necessary, therefore, to split the two processes such that f5 simply covers the 'display on screen' routine.

To prepare the screen, the message "YOUR" and "SPRITE" are blanked off and the sprite is turned off (POKE 53296,0). Next, the first three bytes are recovered (FOR D=0 TO 2) by a PEEK into the storage location and these are prepared for printing onto the screen. Line 60410 does this by stripping off the extra space inserted by the C-64's BASIC and then prints this and the following comma. Once the 0 TO 2 loop has been executed, the 0 TO 20 loop increments to read the next three bytes of the sprite.

Once the user has finished with the DATA and presses any key, the screen is again set up as before.

#### PROGRAM 8.9

```
60370 IFA$="<F5>"THEN60400
```

### f6: Write DATA into program

As this is the same as for Char.Gen, no explanation is needed.

#### PROGRAM 8.10

```
60375 IFA$="<F6>"THEN60460
```

## f7: Clear design matrix

To carry out the clearing of the matrix, it is only necessary to clear the various variables and the screen. The command CLR does the first job and a GOTO 60000 the second: line 60390 completes the input loop by jumping back to the GET.

### PROGRAM 8.11

```
60380 IF A$="<f7>"THEN CLR:GOTO 60000
60390 GOTO 60270
```

### Store sprite in DATA statements.

First the work sprite is turned off (POKE 53269,0) and then comes the interrogation to find out where the sprite is to be stored, at what line interval and, finally, an error check in an attempt to prevent over-writing (line 60475).

### PROGRAM 8.12

```
60460 POKE53269,0
60465 PRINT"<RED><CLR><3DCRSR>AT WHICH
        LINE DO YOU WISH TO STORE YOUR"
60470 INPUT"  SPRITE DATA(10000-60000)
        ";LN
60475 IFLN<10000ORLN>59999THEN60465
60480 INPUT"<3DCRSR>LINE NUMBER INCREME
        NTS OF";SP
```

Next the sprite number and its storage block are inputted.

### PROGRAM 8.13

```
60485 INPUT"<3DCRSR>WHICH SPRITE WILL T
        HIS BE(0-7)  <4LCRSR>";SN
60487 IFSN<0ORSN>7THENPRINT"<5UCRSR>":G
        OTO60485
60490 PRINT"<3DCRSR>AT WHICH BLOCK DO Y
        OU WANT THE DATA TO"
60492 INPUT"BE STORED (0-255)  <5LCRS
        R>";BN
60494 IFBN<0ORBN>255THENPRINT"<6UCRSR>":
        GOTO60490
```

Once the sprite's general data is collected, it has to be stored prior to being loaded into DATA statements. As the rest of the data for these is POKEd into memory, it would seem logical to store the sprite's general data in the same way. The process is further assisted by the presence below the cassette buffer of enough empty bytes to store the sprite general data. Lines 60495 and 60500 store these eight bytes.

#### PROGRAM 8.14

```
60495 POKE824,SN:POKE825,BN:POKE826,CO:
      POKE827,EX
60500 POKE828,EY:POKE829,MC:POKE830,C1:
      POKE831,C2
```

Once these bytes are in place, a single re-load routine will store both the general and specific data in DATA statements. One of the important features of such a procedure is that the data is organized in a systematic way. When this is done, the DATA can be used to reconstruct sprites on demand. Recovery of the data and its insertion into the program follows much the same pattern as that used for the DATA only module.

#### PROGRAM 8.15

```
60505 PRINT"<BLU><CLR>":FORX=0TO6:PRINT
      :PRINTLN+X*SP;"DATA";:FORY=0TO9
60510 A$=STR$(PEEK(824+X*10+Y)):PRINTRI
      GHT$(A$,LEN(A$)-1);:PRINT",";
60515 NEXT:PRINTCHR$(20);:NEXT:A$=STR$(
      PEEK(824+X*10+Y))
60520 PRINT","RIGHT$(A$,LEN(A$)+1):PRIN
      T"<WHT>GOTO60530<HOME>";
60525 FORX=0TO7:POKE631+X,13:NEXT:POKE1
      98,8:END
60530 PRINTTAB(6)"<RED><2DCRSR><RVSON>
      PRESS ANY KEY..TO CONTINUE <RVSOF
      >";
60535 GETA$:IFA$=""THEN60535
60540 GOTO61000
```

Once Sprite.Gen has been used to create a sprite and store it in memory, the program that will be using this will need to be able to decode it satisfactorily: the implication of this is that the order and location of storage must be defined in such a way that the reading-back routine performs its task in the right order. It is this routine which remains once the termination procedure has been called, the stages it goes through being:



- \* reset all sprites to unexpanded single color.
- \* READ in sprite general data to define block, number, size, color etc.
- \* READ pixel data and POKE into appropriate location: assign sprite general data.

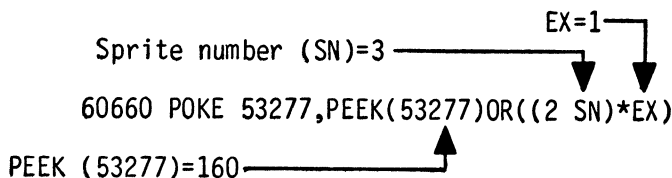
Only the penultimate stage gives rise to a slight complication, as single bytes are used to store data for all eight sprites. Taking line 60660 as an example, this stores the X expansion data. Say, then, that sprite 3 is to be expanded in the X direction and that sprites 5 and 7 have already been set to expanded. In this case, EX is set to 1. Thus working through line 60660:

Sprites 5 and 7 are already set to expanded, i.e. location 53277 appears as:

7	6	5	4	3	2	1	0	
1	0	1	0	0	0	0	0	

$$128 + 0 + 32 + 0 + 0 + 0 + 0 + 0 = 160$$

i.e. a PEEK (53277) would yield 160.



the line simplifies to:

$$60660 \text{ POKE } 53277, (160 \text{ OR } 2 \uparrow 3 * 1)$$

i.e. 60660 POKE 53277, (160 OR 8\*1)

i.e. 60660 POKE 53277, (160 OR 8)

i.e.

1	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0
1	0	1	0	0	1	0	0

$$128 + 0 + 32 + 0 + 8 + 0 + 0 = 168$$

line 60660 then becomes:

```
60660 POKE 53277,168
```

A similar process is used to set the Y expansion and multicolor modes in lines 60670 and 60680.

PROGRAM 8.16

```
60605 POKE53277,0:POKE53271,0:POKE53276
      ,0
60610 FORY=1TON:READSN,BN,CO,EX,EY,MC,C
      1,C2:FORX=0TO62
60620 READC:POKE(BN*64)+X,C
60630 NEXT
60640 POKE2040+SN,BN
60650 POKE53287+SN,CO
60660 POKE53277,PEEK(53277)OR((2↑SN)*EX
      )
60670 POKE53271,PEEK(53271)OR((2↑SN)*EY
      )
60680 POKE53276,PEEK(53276)OR((2↑SN)*MC
      )
60690 IFMC=1THENPOKE53285,C1:POKE53236,
      C2
60700 NEXT:RETURN
```

Sprite.Gen removal routine

Once the necessary sprites have been developed, the bulk of Sprite.Gen can be removed. This is done in exactly the same way as was done in Char.Gen. Therefore, no explanation is necessary.

PROGRAM 8.17

```
61000 PRINT"<BLU><CLR><2DCRSR><4RCRSR>H
      AVE YOU FINISHED USING"
61010 INPUT"<DCRSR><4RCRSR>SPRITE.GEN (
      Y/N)";A$:IFA$="N"THEN60050
61020 IFA$<>"Y"THEN61000
61030 PRINTTAB(4)"<2DCRSR>DO YOU WISH T
      O CREATE THE SPRITE"
61040 PRINTTAB(4)"<DCRSR>SUBROUTINE FOR
      YOUR PROGRAM?"
61045 PRINTTAB(4)"<RED><DCRSR><RVSON>WA
      RNING<RVSOFF>:<BLU> IF YOUR REPLY
      IS 'Y'"
```

```

61050 PRINTTAB(4)"THEN SPRITE.GEN WILL
      BE DELETED"
61060 INPUT"<DCRSR><4RCRSR>CREATE SUBRO
      UTINE (Y/N) <3LCRSR>";A$: IFA$="
      N"THENEND
61070 IFA$<>"Y"THENPRINT"<2UCRSR>":GOTO
      61060
61080 PRINTTAB(4)"<2DCRSR><BRWN>HOW MAN
      Y SPRITES HAVE YOU":INPUT"<DCRSR>
      <6RCRSR>DESIGNED <4LCRSR>";N
61090 PRINT"<CLR><WHT><2DCRSR>";
61100 PRINT"DELETE60000-60599<2DCRSR>"
61110 PRINT"DELETE61000-<2DCRSR>"
61120 PRINT"60600 N=";N
61140 PRINT?"CHR$(34)"<CLR><BLU>"CHR$(
      34)
61150 PRINT"<HOME>";
61160 FORX=631TO634:POKEX,13:NEXT:POKE1
      98,5:END

```



With this book came a disk containing Sprite.Gen. You may now experiment with this program. Make sure that you LOAD and RUN Honey.Aid first, as the Sprite.Gen program uses the DELETE function of Honey.Aid.

## PART 2

### Using Sprite.Gen

In Part 2 of this chapter, we will develop a game called Target that uses Sprite.Gen to generate a sprite and then explore its various features.

### Target: The Game

The aim of 'Target' is to hit a target in the center of the screen with a circular missile, which can be controlled in three axes by the pressing of certain keys. Two keys control its X and Y direction (Z and X respectively) while the two unshifted cursor keys control the Z direction, the cursor  key moving the missile into the screen while the cursor  key moves it towards the player.

As the target is suspended in space, the missile can pass either below or above the target and, in order to score a hit, the missile must be at the correct height. While attempting to hit the target, the player must avoid obstacles on the screen and a collision with one of these terminates the game as does the hitting of the target. On termination the player's progress is reported, along with the running time of the game.

### How it works.....

An imaginary Z co-ordinate for the target is generated by means of a random number function. As the Z-axis of the missile is adjusted it may yield a location coincident with the target or one either in front of it or behind it. When the X, Y and Z co-ordinates of the missile and target coincide then a hit is scored and that round of the game is concluded with a congratulatory message. When the Z co-ordinate of the missile is in front of the target, the missile must pass in front of it and when its co-ordinate is behind the target, the target must obscure the missile as they pass.

## Sprite precedence

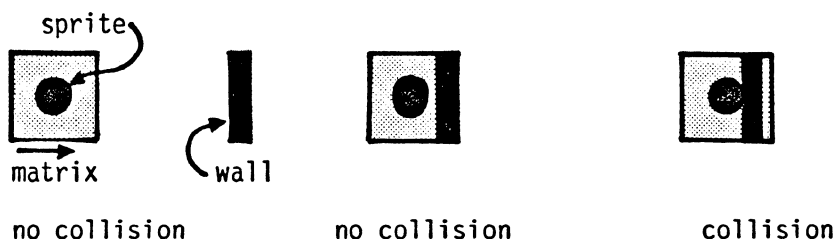
The three dimensional effect is most readily achieved by utilizing the sprite precedence feature which defines the order in which sprites appear to pass in front of each other. In the Commodore pecking order, sprite 0 always passes in front of all other sprites and 7 always hides away at the back. What is actually required in this game is one sprite - the missile - which passes, sometimes in front of and sometimes behind a second sprite. Not too easy to achieve that, so three sprites are used, 0 and 2 for the target and 1 for the missile. The two targets can then be switched on and off to allow the ball to pass in front of or behind these as appropriate.

## Sprite collisions

First though: what is a collision?

A sprite consists of a block with its 24x21 pixel matrix and collision only occurs when this block comes in contact with a similar block of another sprite or the substance of a screen object. Figure 8.8 illustrates this.

### Sprite to background



### Sprite to sprite



FIGURE 8.8

Note: a collision occurs ONLY when the body or substance of a sprite is in contact with that of another sprite.

(i) Collision with other sprites

One byte in the sprite register, 53278 (R+30), is used to detect sprite/sprite collisions. Thus, when sprite zero collides with another sprite, bit zero of R+30 is set to 1. Also, of course, the relevant bit of the other sprite is also set. A PEEK at R+30, therefore, will tell if any sprite collisions have occurred, but beware! PEEKing into this location resets the byte back to zero-so you must get it right first time.

When only two sprites are being used, this collision checking is easy as the PEEK of 53278 will read something other than zero when a collision has occurred. If a specific collision is checked for, then the contents of the register must be checked more thoroughly. For instance, following a sprite 5 to sprite 1 collision, bits 1 and 5 will be set to 1. Thus, the register will contain a  $2^1$  plus a  $2^5$  i.e.

7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	0

$0 + 0 + 32 + 0 + 0 + 0 + 2 + 0 = 34$

The PEEK of 53278 (R+30) will be 34. A more sophisticated way of checking this would be to OR the contents of the PEEK with the relevant amount - in this case 34. On doing this the 1's in the 1st and 5th bit would remain behind along with any 1's in the original byte. Thus if the result of the byte ORed with 34 is 34, then that's what the original byte contained i.e.

	0	0	1	0	0	0	1	0
ORed with	0	0	1	0	0	0	1	0
gives	0	0	1	0	0	0	1	0
=	0	+ 0	+ 32	+ 0	+ 0	+ 0	+ 2	+ 0 = 34

but....

	0	1	0	0	1	1	1	0
ORed with	0	0	1	0	0	0	1	0
gives	0	1	1	0	1	1	1	0
=	0	+ 64	+ 32	+ 0	+ 8	+ 4	+ 2	+ 0 = 110

If more than one collision is to be detected then the PEEK of the register must be stored prior to, or at the time of the first testing otherwise the register's contents will be destroyed.

(ii) Collisions with other objects

When a collision occurs between a sprite and anything in other than background color, then register R+31 (53279) is set appropriately. This function is virtually identical to that for sprite to sprite collisions and is tested for in the same way. Thus, to test for a sprite 4 collision, R+31, (53279) would be PEEKed to see if bit 4 is set, i.e.

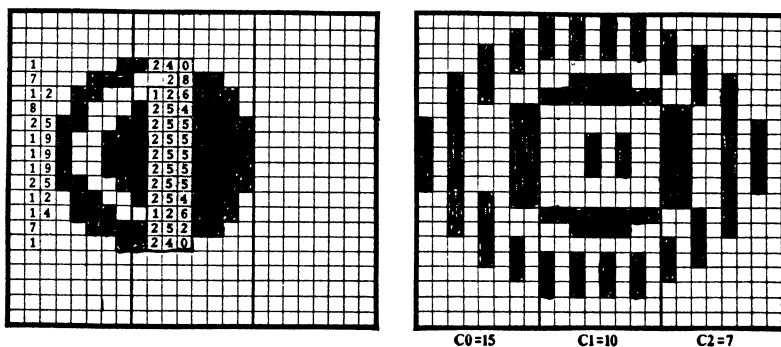
IF PEEK(53279 AND 2↑4)=2↑4 THEN (a collision)

Note that this will only test for sprite 4 collisions and destroy the evidence of any other collisions having occurred.

Developing the program

Your disk contains the entire TARGET program; however, you may want to follow the instructions on designing and entering the sprites. There will be instructions in a few pages to load the TARGET program and to follow the game design.

As the program is all about sprites, the first task to be undertaken will be to design the missile and the target using Sprite.Gen. Figure 8.9 gives the designs used on the disk but as you will probably find these not to your taste, you're free to design your own. That's one of the great advantages of sprites; once the algorithm for manipulating them is developed, it's operation is entirely independent of their form.



Missile

Target

FIGURE 8.9

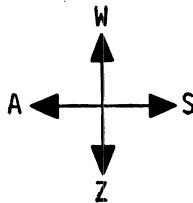
To create these, first of all LOAD and RUN Honey.Aid (if it is not already there) and then LOAD and RUN Sprite.Gen (if it is not already there).

### First the missile...

The 64 asks

ENLARGED X?	Reply N
ENLARGED Y?	Reply N
MULTICOLOR?	Reply N

Next the computer will display the sprite design grid and await your pleasure. At this stage, the design cursor is located in the top left-hand corner of the grid and it is moved by the keys:



The function keys operate in the same way as those on Char.Gen with the addition of f6.

- f1 turn current pixel in sprite ON.
- f3 turn current pixel in sprite OFF.
- f5 print DATA on screen.
- f6 add DATA to program.
- f7 clear design matrix.

Entering of the actual design data is done exactly as with Char.Gen, so no further description is necessary.



Once the design is ready, press function key 6, obtained by holding down shift and pressing the function key marked f5/f6. You will then be asked:

AT WHICH LINE DO YOU WANT TO STORE  
YOUR SPRITE DATA (10000-60000) ?                      Reply 10000

Next question....

IN STEPS OF ?    Reply 10

Next....

WHICH SPRITE WILL THIS BE (0-7) ?                      Reply 1

And....

AT WHICH BLOCK DO YOU WANT THE DATA  
TO BE STORED ?    Reply 14

Then the program will display the added lines on the screen and enter them into the program giving a

PRESS ANY KEY TO CONTINUE

On pressing a key the SPRITE.GEN program will ask you

HAVE YOU FINISHED USING  
SPRITE.GEN (Y/N)?

Reply with an N and the SPRITE.GEN program will resume, allowing the target to be designed next.

Repeat the above process for the target but for:


ENLARGED X ?	Reply N
ENLARGED Y ?	Reply N
MULTICOLOR ?	Reply Y
MULTI-COLOR#1	Reply 10
MULTICOLOR#2	Reply 7
SPRITE COLOR	Reply 15

Then design the sprite; press function 6 and answer the questions as follows:

STORE AT LINE?	Reply 10070
IN STEPS OF?	Reply 10
WHICH SPRITE?	Reply 0
BLOCK STORED?	Reply 13

And then.....

As the two targets are the same, the design for the first can be used for the second and so it is only necessary to tell the computer to look at the same memory for each sprite. This is done at the beginning of the program with a direct POKE. Of course, if you wish to go through the proceedings exactly you can repeat the above procedure.

POKE 2042, 13  
i.e. set sprite 2  to block 13

At this stage, if you're happy with your sprites, you can select RETURN while in edit mode and, should you manage to persuade the computer that you're serious, you may remove Sprite.Gen, leaving the DATA and sprite creation procedure.

At last: the game....

Basically the program breaks down into five sections:

- 1 Initialisation
- 2 Game algorithm
- 3 Reporting on progress
- 4 Printing obstacles on the screen
- 5 DATA and sprite creation

#### Module 1: Initialisation

You may now want to load the TARGET program and follow the game design. As the segments are discussed, you can LIST the segments (i.e. LIST 100-160).

This section commences with the usual sprite initialization routines, Program 8.18 lists these along with notes on their functions.

#### PROGRAM 8.18

Switch off all sprites	100 POKE 53269,0
Clear screen	PRINT"<CLR><NOCBM><UCASE>";:
Read data and create	GOSUB 60600:
Sprites:MODULE 5	
Set sprite 2 to same	POKE 2042,13
as sprite 0	
Set screen color	110 POKE 53280,3:POKE 53281,1

```

Set sprite 2's          130 POKE 53289,15:
individual color
Set multicolor mode    POKE 53276,5
for sprite 0 and 2
Set X co-ordinates    140 POKE 53248,170:POKE 53252,170
for sprites 0 and 2
Set Y co-ordinates    150 POKE 53249,100:POKE 53253,100
for sprites 0 and 2
print obstacles       160 GOSUB 820

```

Once the sprite initialization is done, the start co-ordinates for the missile can be set up. Line 240 does this with the simple assignment X=24:Y=60. It's necessary in this game to be able to reverse the direction of the missile when controlled by a key, or when it hits the electro-magnetic shield around the field of play. This is done by means of a pair of increments XD and YD which are set up to take either positive or negative values. During initialization, these are both set to 5 which gives the number of pixels of increment for each cycle. Changing this value will change the speed of the missile across the screen - a value of 1 giving slower progress and a 10 giving faster.

In addition to X and Y co-ordinates, both the missile and target need to be given a Z value or depth. This is worked out by means of two random number functions which calculate L, Z co-ordinate of the missile and RL, Z co-ordinate of the target. A simple check on line 250, Program 8.19 sends the program back to try again when both Z co-ordinates are the same i.e.

#### PROGRAM 8.19

```

240 X=24:Y=60:XD=5:YD=5:X1=0:X2=0:L=INT(
RND(0)*10):RL=INT(RND(1)*5)+3
250 IF L=RL THEN 240

```

Finally, the two collision registers are cleared by PEEKing them, the clock (TI\$) is set to zero and auto-repeat is switched off i.e.

#### PROGRAM 8.20

```

260 CS=PEEK(53278):CD=PEEK(53279):
TI$="000000":POKE 650,64

```

## Module 2 : The Game Algorithm

At the beginning of the game the X and Y co-ordinates are both incremented by 5 (XD and YD), producing movement across the screen at 45 degrees. Before these are actually POKEd in to re-locate the missile sprite, it is turned off with a POKE. Once this is done, the new sprite co-ordinates can be POKEd into memory prior to the sprite being turned on once more. The X co-ordinate is slightly awkward as the possibility of it being over 255 must be catered for. Thus, the actual co-ordinate must be split into its least and most significant parts. Firstly, to find its least significant byte, AND it with 255. Try it out with, say 500:

ANDed	0	0	0	0	0	0	0	0	1	=	1	1	1	1	0	1	0	1	= 500
with	0	0	0	0	0	0	0	0	0	=	1	1	1	1	1	1	1	1	= 255
gives	0	0	0	0	0	0	0	0	0	=	1	1	1	1	0	1	0	1	= 244

i.e. the POKE becomes:

POKE 53250, X AND 255

To obtain the most significant byte, the X co-ordinate is ANDed with 256 to test for a co-ordinate over 255. As this has bit eight set to 1 and the remainder of its bits set to zero, this strips off any bits that are set in bit 0 to 7 and allows any 1's through that are set in bit 8. Test this out with 500.

ANDed	0	0	0	0	0	0	0	0	1	=	1	1	1	1	0	1	0	1	= 500
with	0	0	0	0	0	0	0	0	1	=	0	0	0	0	0	0	0	0	= 256
gives	0	0	0	0	0	0	0	0	1	=	0	0	0	0	0	0	0	0	= 256

When a 1, i.e. a 256, is present, it must be converted into a 1 in the appropriate bit for that particular sprite, as location 53264 contains MSB information for all seven sprites. For sprite 0, the appropriate POKE value would be 1 or  $2^0$ , while for sprite 4 it would be 16 or  $2^4$ . Thus the 256 needs to be divided by 256 to obtain the sprite zero POKE or 2 to obtain the sprite 7 value; i.e. in general terms, the dividend is  $2^{(8-\text{sprite number})}$ . In the particular case of sprite 1, therefore, the 256 needs to be divided by 128 to yield a 2. Doing this yields a line:

POKE 53264, (X AND 256)/128

## PROGRAM 8.21

```
Increment X and Y          270 X=X+XD:Y=Y+YD
Set new X and             280 POKE 53250,X AND 255:POKE 53251,
Y co-ordinates           Y:POKE 53264,(X AND 256)/128
turn missile sprite      POKE 53269,N+2
on again
```

As the missile traverses the screen, checks must be made to see if it has yet reached the limits of its travel. Fairly standard sorts of things these, as when the edges of the screen are reached, the missile is deflected back to the center of the screen. (We'd call it a 'bounce' in another game!) As with the ball-game, it's simply done by reversing the sign of the increment, on lines 290 and 300.

## PROGRAM 8.22

```
290 IF X>324 OR X<24 THEN XD=-XD
300 IF Y>230 OR Y<60 THEN YD=-YD
310 IF L<5 THEN POKE 53269,6:N=4
320 IF L>=5 THEN POKE 53269,3:N=1
```

Following the GET A\$ which checks the keyboard buffer, the buffer is cleared with a POKE 198,0 and the two sprite collision bytes are saved as CS(sprite collision) and CD (background collision).

One of the problems with a game that uses three axes, is that of how to display the third axis. Movement in the Z axis, being in effect into and out of the screen, is not really visible to the player. To overcome this, two different features are built into the game to indicate that:

- a) movement along the Z-axis has taken place
- b) the limit of movement along this axis has been reached.

The mode of indication chosen is a flash of the background color for movement and a very quick flash through the first ten colors to show that the limit of movement has been reached. Two of these lines have the basic syntax:

"If the GET A\$ was a cursor down (<DCRSR>) and the Z co-ordinate is greater than zero (Z>0) and no sprite-to-sprite collision has occurred (CS=0) then decrement the Z co-ordinate (L=L-1) and flash the background (POKE 53281;0:POKE 53281,1)" i.e.:

### PROGRAM 8.23

```
330 GETA$
340 POKE 198,0:CS=PEEK(53278):CD=PEEK(53
    279)
350 IFA$="<DCRSR>"ANDL>0ANDCS=0THENL=L-
    1:POKE53281,0:POKE53281,1
```

Similarly, for cursor acrosses:

### PROGRAM 8.23(a)

```
360 IFA$="<RCRSR>"ANDL<10ANDCS=0THENL=L
    +1:POKE53281,0:POKE53281,1
370 IF(A$="<DCRSR>"ANDL=0)OR(A$="<RCRSR
    >"ANDL=10)THENGOSUB810
```

Next comes a check for a collision, at which point the Z co-ordinates of the target and missile will coincide (Z=RL) and the sprite collision register will have been set (CS>0). When this condition is detected, the program is routed to a routine at 580.

### PROGRAM 8.23(b)

```
380 IF L=RL AND CS>0 THEN A$=TI$:GOTO 580
```

While the sprite is moving about the screen, its direction can be changed by means of the 'Z' and 'X' keys which, in effect create an invisible wall from which the missile bounces. The syntax of the line is fairly straight-forward; the only precaution necessary being a check that the missile has not already reached one of the edges of the field. When this has happened in the current cycle the direction of movement has already been reversed and another reversal would be slightly disastrous!

### PROGRAM 8.24

```
390 IF A$="Z"AND X<324 AND X>24 THEN XD=-XD
400 IF A$="X"AND Y<230 AND Y>60 THEN YD=-YD
```

At last the game algorithm gets as far as checking for a sprite to background collision. Although a check for a non-zero value would be sufficient, an AND is used to demonstrate how to check for a specific sprite's data. When a sprite to background collision is detected, the program is sent to the sub-routine at 450 that reports back.

## PROGRAM 8.25

```
420 IF (CD AND 2)=2 THEN 450
```

It's worth noting again that when more than one sprite is involved, the storage of the PEEK in a variable is absolutely necessary. PEEKing the sprite's collision registers destroys the data. This, of course, is NOT so for any other memory location, except the last few SID registers.

Finally, at the conclusion of an impactless cycle, the time TI\$ is reported and the program loops back for another go.

## PROGRAM 8.26

```
430 PRINT"<HOME><RVSON>";TI$  
440 GOTO 270
```

## Module 3: Reporting on progress

### Report miss/blow missile

When a collision has occurred, the first task is to store the time, TI\$, to see how long the game lasted; i.e. A\$=TI\$.

Once the missile has hit an object, it needs to be blown up. Such an effect can be simulated by defining a series of disintegrating missiles or, more easily, by just drawing a series of randomly fragmented objects in the sprite's block of screen. As the VIC chip knows where a sprite's data is by means of the block numbers stored in locations 2040 to 2047, the particular location for the disintegrating sprite could be varied rapidly, sending the VIC chip looking all over the place for the data; i.e.

```
FOR X=16 TO 0 STEP -1:POKE 2041,X:NEXT
```

This tells the VIC chip that sprite 1's data is in block 16, then block 15 etc. down to zero. If the routine is nested within another loop, say a four times one, then the effect will last for a reasonable time. One danger associated with this technique is that it leaves sprite 1 in block zero - that's zero page. However, as the program is re-started with a 'RUN' (line 730), the sprite is re-located by the program.

Prior to displaying the end of game message, the missile is switched off (POKE 53269,N) and finally the target is turned off and the screen cleared; i.e. Program 8.27.

#### PROGRAM 8.27

```
remember time          450 A$=TI$:
draw random sprites   FOR Y=1 TO 4:FOR X=16 TO 0 STEP -1:
4x16 times            POKE 2041,X:NEXT:NEXT
delay                 460 FOR X=1 TO 1000:NEXT:
switch off missile    POKE 53269,N
delay                 470 FOR X=1 TO 1000:NEXT:
switch off target     POKE 53269,0:PRINT"<CLR>";
```

Next the program reports back on how long the player survived by means of the subroutine at 510. Also it tells the player that (s)he failed to hit the target. After this it jumps to the "Do you want another go?" subroutine; i.e.

#### PROGRAM 8.28

```
480 PRINT "<CLR><7DCRSR><9RCRSR> <2CRS
R><ORNG>YOU SURVIVED FOR :":GOSUB 510
490 PRINT "<2DCRSR><5RCRSR><GRN>AND YOU
DID NOT HIT THE TARGET!"
500 GOTO 690
```

#### Report time

Time in this game is recorded in TI\$ which was initialized at the beginning of the game cycle (line 260) and was stored in A\$ when a game is terminated. Prior to reporting a time, the string TI\$ must be dissected to yield the hours, minutes and seconds; i.e.

```
left-most two digits - hours LEFT$(A$,2)
middle two digits - minutes MID$(A$,3,2)
right-most two digits - seconds RIGHT$(A$,2)
```

The decoding of the string can be done straight-forwardly in the print statement as in Program 8.29, lines 510 to 530.



### PROGRAM 8.29

```
510 PRINT "<DCRSR><17RCRSR><BLU>;LEFT$(A
$,2);" HOURS<DCRSR><8LCRSR>";
520 PRINT MID$(A$,3,2)" MINUTES
<DCRSR><14LCRSR> AND ";
530 PRINT RIGHT$(A$,2);" SECONDS"
```

Once the reporting back is done, the game title is printed onto the screen and the subroutine returned.

### PROGRAM 8.30

```
540 PRINT "<HOME><2DCRSR>      ";
TAB(16);"<RVSON><RED>      "
550 PRINT TAB(16);"<RVSON> TARGET
<RVSOFF>"
560 PRINT TAB(16);"<RVSON>
<RVSOFF>"
570 PRINT "<10DCRSR>";:RETURN
```

### Report a hit

The remainder of this module is concerned with printing messages onto the screen after TI\$ has been stored in A\$. A call at the subroutine at 510 reports the time. A special message is then chosen from the selection stored in lines 630 to 660, the particular choice being based on the actual time taken to hit the target. To do this, the program reads down through the IF...THEN checks and, when the test succeeds, prints out the appropriate message and is then directed to the "Do you want another go?" subroutine, i.e.

### PROGRAM 8.31

```
580 FORX=1TO50:POKE53269,3:POKE53269,1:
NEXT
590 FORX=1TO1000:NEXT:POKE53269,0
600 PRINT"<CLR><7DCRSR><LTBLU>"TAB(16);
"WELL DONE!":PRINTTAB(10);" YOU HIT
THE TARGET."
610 PRINT"<DCRSR><14RCRSR><BLK>YOUR TIM
E WAS:":GOSUB510
620 PRINT"<DCRSR><5RCRSR><BRWN>WHICH WA
S ";
630 IFA$<"000010"THENPRINT"BRILLIANT! F
ANTASTIC! AMAZING!":GOTO690
640 IFA$<"000030"THENPRINT"BRILLIANT":G
OTO690
```

```

650 IFA$<"000100"THENPRINT"VERY GOOD":G
    OT0690
660 IFA$<"001000"THENPRINT"NOT AT ALL G
    OOD":GOTO690
670 PRINT"NOT VERY GOOD BUT AT LEAST YO
    U HIT IT. JUST KEEP ON"
680 PRINT"YOU'LL GET A BETTER SCORE SOO
    N."

```

Do you want another go?

This is a fairly standard routine. It prompts for a 'Y' or 'N' response and on receipt of the 'Y's, re-RUNs the program by means of the RUN command in line 730. One of the advantages of this over a GOTO 100 (for this program) is that when the operating system sees the command RUN, it immediately re-sets all the variables ensuring that the game starts once more with a clean sheet. Program 8.32 shows the whole subroutine:

PROGRAM 8.32

```

690 PRINT "<4DCRSR><8RCRSR><ORNG>
    DO YOU WNAT ANOTHER GO?"
700 PRINT "<DCRSR><15RCRSR><ORNG>
    <LTRED>Y<ORNG>ES OR <LTRED>N<ORNG>0"
710 GETC$:IFC$=""THEN 760
720 IF C$="N" THEN 750
730 IF C$="Y" THEN RUN
740 GOTO 710

```

Sign off

When the player indicates that (s)he has finished the game, the 'Target' screen is displayed and a friendly message given. This is held on screen for a while with provision being made for the player to terminate the proceedings if so desired.

PROGRAM 8.33

```

760 PRINT"<CLR>":GOSUB540:PRINTTAB(12)"
    <2DCRSR><PUR>GOODBYE FOR NOW."
770 FORX=1TO1000:GETA$:IFA$=""THENNEXT
780 IFX<1000THENX=1010:NEXT
790 PRINT"<CLR>";
800 POKE650,0:PRINT"<OKCBM>":END

```

## Color cycle

This is the sub-routine that produces a cycle through colors 0 to 10 when an attempt is made to move the Z cursor beyond its limit.

### PROGRAM 8.34

```
810 FOR Z=0 TO 10:POKE 53281,Z:
NEXT:POKE 53281,1:RETURN
```

## Module 4: Present obstacles

No attempt will be made to explain this module, as it simply prints characters onto the screen. It is clearly a major advantage of sprites that collisions are so readily detected and this enables one to print out a pattern of obstacles that suit one's personal taste.

### PROGRAM 8.35

```
820 PRINT"<CLR><16DCRSR><29RCRSR><YEL>V
<LTRED>*4<YEL>V";
830 PRINT"<DCRSR><4LCRSR><LTRED>/<CYN>U
I<LTRED>/";:PRINT"<DCRSR><4LCRSR><L
TRED>7<CYN>JK<LTRED>7";
840 PRINT"<DCRSR><4LCRSR><YEL>V<LTRED>*
4<YEL>V";
850 PRINT"<4DCRSR><LCRSR><GRY2>V<UCRSR>
V<UCRSR>V<UCRSR>V<UCRSR>V<UCRSR>V<U
CRSR>V<UCRSR>";
860 PRINT"<BLU><HOME><RVSON>
<RVSOFF
>";
870 PRINT"<HOME><16DCRSR><7RCRSR><YEL>V
<LTRED>*4<YEL>V";
880 PRINT"<DCRSR><4LCRSR><LTRED>/<CYN>U
I<LTRED>/";:PRINT"<DCRSR><4LCRSR><L
TRED>7<CYN>JK<LTRED>7";
890 PRINT"<DCRSR><4LCRSR><YEL>V<LTRED>*
4<YEL>V";
900 PRINT"<2UCRSR><10LCRSR><GRY2>V<DCRS
R>V<DCRSR>V<DCRSR>V<DCRSR>V<DCRSR>V
<DCRSR>V<DCRSR><BLU>";
910 RETURN
```

If you find this a boring old set of obstacles, simply change the print statements to suit yourself. The sprite collision detection feature will handle the problem of collisions automatically.

This facility gives you a free reign to design your own screen. You could design this as a maze or what you will. It's up to you!

### Module 5: Sprite DATA and sprite creation

Sprite.Gen, as its last act before it was ruthlessly destroyed, printed out the sprites' DATA statements. All that is left after you pushed the button is enough to set up the sprites and this was explained with Sprite.Gen. Again, therefore, no explanation.

#### PROGRAM 8.36

```
10000 DATA1,14,6,0,0,0,0,0,0,0
10010 DATA0,0,0,0,0,0,0,1,240,0
10020 DATA7,28,0,12,126,0,8,254,0,25
10030 DATA255,0,19,255,0,19,255,0,19,25
5
10040 DATA0,25,255,0,12,254,0,14,126,0
10050 DATA7,252,0,1,240,0,0,0,0,0
10060 DATA0,0,0,0,0,0,0,0,0,0
10070 DATA0,13,15,0,0,1,10,7,0,170
10080 DATA0,2,170,128,10,170,160,10,0,1
60
10090 DATA40,60,40,40,255,40,35,0,200,1
63
10100 DATA0,202,163,20,202,163,20,202,1
63,20
10110 DATA202,163,0,202,35,0,200,40,255
,40
10120 DATA40,60,40,10,0,160,10,170,160,
2
10130 DATA170,128,0,170,0,0,0,0,0,0
60600 CN= 2
60605 POKE53277,0:POKE53271,0:POKE53276
,0
60610 FORY=1TOCN:READSN,BN,CO,EX,EY,MC,
C1,C2:FORX=0TO62
60620 READC:POKE(BN*64)+X,C
60630 NEXT
60640 POKE2040+SN,BN
60650 POKE53287+SN,CO
60660 POKE53277,PEEK(53277)OR((2↑SN)*EX
)
60670 POKE53271,PEEK(53271)OR((2↑SN)*EY
)
60680 POKE53276,PEEK(53276)OR((2↑SN)*MC
)
60690 IFMC=1THENPOKE53285,C1:POKE53286,
C2
60700 NEXT:RETURN
```

Now the game is loaded, just a recap of the rules:

Z and X move the missile, effectively bouncing it back in X and Y directions.

The cursor keys move the missile in and out of the screen i.e. in the Z direction.

    moves the missile out of the screen.

    moves the missile into the screen.

Right then, it's all yours. Bit of a frustrating game, isn't it?



## CHAPTER

# 9

### PART 1: SOUND

**O**ne of the very advanced features of the Commodore 64 computer is its SID or Sound Interface Device chip. By means of this, the programmer can control all the usual musical parameters, and much more. The much more refers to the ADSR and tone-color controls which reproduce the sound of musical instruments such as the trumpet or piano. However, controlling such a wide range of parameters requires a considerable amount of programming and in this chapter, modules will be developed that should help you in your task of controlling SID.

First a summary ....

SID gives you the following:

3 voices or separate channels of  
sound each controllable for:

- \* Pitch
- \* Note duration
- \* ADSR
- \* Waveform
- \* Filtering
- \* Modulation

The SID chip itself is persuaded to yield all these features by changing the values stored in certain memory locations. There are 25 such locations and it's only necessary to remember their addresses - from 54272 to 54296 - to be able to totally control SID.

Let's first of all look at one of the 3 voices and see if we can control it. Then perhaps we'll get more ambitious.

According to SID's memory map, the pitch controls for voice 1 are located at 54272 and 54273. As far as SID is concerned however, he has 25 registers and they just happen to be wired in from 54272 to 54296 on the Commodore 64. He'd be just as happy were these to be numbered from 0 to 24 and these lower numbers are much easier to understand too. In this chapter, then, in the early stages of a program, the variable R will be defined as 54272 and, thereafter, the chip's registers will be allocated the register numbers 0 to 24.

One of the most important characteristics of a sound is its pitch and to set this, a number has to be POKEd into SID's registers. However, as each of these registers is only 8 bits - the 6510 being an 8 bit chip - a single register can only accommodate a number up to 255-i.e. POKE 54272,255 would load a '255' into address 54272. An attempt to load anything greater than 255 into a memory location would yield an error of the type:

?ILLEGAL QUANTITY ERROR

To be able to POKE numbers only up to 255 becomes something of a restriction when this is to define the frequency of SID's note. In order to extend this range, SID uses 16 bit arithmetic and treats the two 8-bit bytes 54272 and 54273 as one 16-bit byte. This it does by allocating place values of  $2^0, 2^1, \dots, 2^7$  to the low byte, 54272 and place values of  $2^8, 2^9, \dots, 2^{15}$  to the high byte. Thus, the place-value of the right-most bit of the high byte is  $2^8$  or 256. Hence, a 1 in the high byte and some ones in the low byte could yield a total of  $256+205=461$ :

Place value

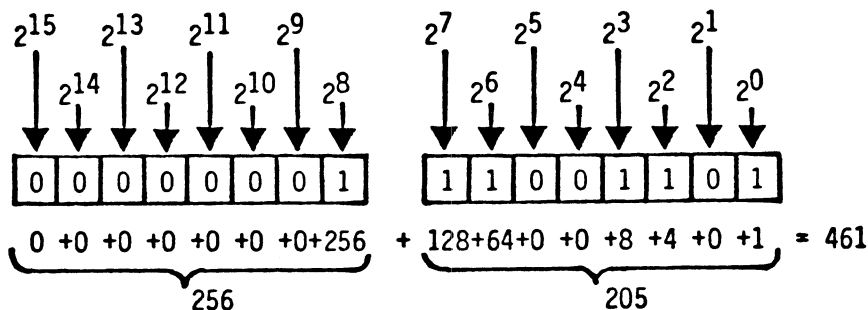


FIGURE 9.1



Thus the line

```
R=54272:POKE R+1,1:POKE R,205
```

will POKE in the 16 bit number  $256+205=461$ . This, the manual tells us, will yield the note A0 with a frequency of 28 Hz on an American Commodore 64 or 27 Hz on a European CBM64. (The generally accepted standard frequency for this note is actually in between these two values at 27.5 Hz.) However, if you POKE those two numbers into the machine in direct mode, then most probably, no sound will emerge. This is because SID needs a certain amount of preparation before you can actually get him to work. However, if you have previously used a program with sound built in, then the POKES may, on their own, give a sound output (a very low note and quite quiet, especially if played on a smallish loud speaker).

To prepare SID for sounds...

The first task is to clear out whatever's left in from the last program - there's nothing worse than other people's noises! Such a clear-out is achieved by setting all the sound registers to zero i.e. POKEing a zero into all addresses from R to R+24. It can be done with a single loop:

```
R=54272:FOR Z=R TO R+24:POKE Z,0:NEXT Z
```

Setting up the note

Once the chip is clear, we're free to set up the note which, on SID, has four basic parameters:

- 1 Pitch
- 2 Volume
- 3 Duration/tone color
- 4 Envelope

\* Set Pitch is discussed above, it's the characteristic which makes a note sound high or low.

\* Set Volume is simply the loudness of the note which can be varied between 0 and 15 by POKEing the appropriate value into Register 24. To get maximum volume therefore, the command is

```
R=54272:POKE R+24,15
```

It should be noted that the volume register controls the volume for the whole of the SID chip, not just the volume of a single voice. There is a way of controlling the relative volume of two or more voices which will be discussed later.

- \* Set Duration/tone color is obtained by switching the note on and off as appropriate. With SID, once a note is turned on, it stays on until switched off so a delay loop is used to hold the note on as long as is needed. Each note is made up of several stages, and 'set duration' is only one of these. Detailed discussion of this is left, therefore, until the structure of the sound is discussed below. For now, we will turn on both the sound and its waveform at the same time by POKEing a 33 into R+4 and holding it there for as long as we wish the note to stay. We will then turn the note off by POKEing a 32 into this address, i.e.

```
R=54272:POKE R+4,33:FOR X=1 TO 1000:NEXTX:POKE R+4,32
```

- \* Envelope is complicated, so for now let's dodge this one and simply set a tone color. It deserves a whole section on its own. To set a particular tone color (or timbre) we'll use:

```
R=54272:POKE R+5,9:POKE R+6,0
```

Now try putting that all together! The order will be:

- 1 Set R = 54272
- 2 Clear Chip
- 3 Set ADSR (tone color)
- 4 Set volume
- 5 Set high and low frequency (NB 2144 used)
- 6 Define nature of sustain sequence
- 7 Set up delay then turn off sustain
- 8 Turn volume to zero

i.e.

#### PROGRAM 9.1

```
100 R = 54272
1020 FOR Z = R TO R+24:POKE Z,0:NEXT Z
1030 POKE R+5,9:POKE R+6,0
6000 POKE R+24,15
6020 POKE R+4,33
6050 POKE R+1,6:POKE R,96
6060 FOR Z=1 TO 1000:NEXT
6070 POKE R+4,32
6090 POKE R+24,0
```

So far so good, that's a sound on voice 1 and the other channels operate in a similar way. In fact the register structure for voices 2 and 3 is identical to that of 1 and as each voice utilizes seven bytes of memory, each of three seven-byte blocks from 54272 onwards contains the registers for one voice. Thus, Program 9.1 can be modified to operate with voice 2 by changing line 1 to read  $R=54272+7$  or, to use voice 3, to  $R=54272+14$ .

Single notes are OK as far as they go but still not too exciting. The frequency can be varied quite readily by setting a variable that is then POKEd into the relevant location. As you'd expect with such a complicated chip, it's very important that this process is done in the correct order to the pattern of Program 9.1. As it is the intention in Part 1 of this chapter to write a program that accepts inputs in conventional musical notation, the problem arises of changing the note names - A, B, C, C# etc. into the relevant frequencies. Conversions are given in the Commodore literature in terms of high and low frequency POKEs but it will be far more convenient to use the frequencies. What is required are two arrays, one with the note names and one with the frequency so that a match can be found for the note name and then the frequency looked up. These arrays, A\$(Z) and A(Z), will be set initially by READING in the data from DATA statements as in Program 9.2 which should be added to Program 9.1. Once this program has been run, A\$(Z) will contain the note names, A, B, C etc. and A(Z) their respective frequencies.

#### PROGRAM 9.2

```
1000 DIM A$(8),A(8)
1010 FOR Z=1 TO 8:READ A$(Z),A(Z):NEXT
30000 DATA C,2146,D,2409,E,2704,F,2864
30010 DATA G,2215,A,3609,B,4051,C,4292
```

The DATA statements in lines 30000 and 30010 are correct for American or Japanese CBM 64s (or strictly speaking, C-64s intended for use with NTSC standard TVs). European C-64s (intended for use with PAL standard TV's) need a different set of values if the C-64 is to be tuned to international 'concert pitch' (A4=440 Hz). These values are:

#### PROGRAM 9.2(a)

```
30000 DATA C,2149,D,2500,E,2806,F,2973
30010 DATA G,3337,A,3746,B,4205,C,4297
```

This DATA is in CBM oscillator frequencies and needs to be converted into 8 bit hi and lo bytes so that it can be POKEd into the relevant register. The high byte (FH for 'frequency hi') is the number of 256's in the original frequency. It can be found by taking the INT of the original frequency divided by 256. Test this out with a few values for A().

- (i) The frequency 256 should give a hi byte of 1  
i.e.  $FH = \text{INT}(256/256) = 1$
- (ii) The frequency 300 should give a hi byte of 1  
i.e.  $FH = \text{INT}(300/256) = 1$
- (iii) The frequency 1025 should give a hi byte of 4  
i.e.  $FH = \text{INT}(1025/256) = 4$

The low byte (FL for frequency lo) is slightly more complicated, as it is what remains when all the hi bytes have been stripped off.

$$\text{i.e. } FL = A() - FH * 256$$

Again, testing this:

- (i) For a frequency of 256  
 $FL = 256 - 1 * 256 = 0$   
Testing this: frequency = hi byte x 256 + lo byte  
 $= 1 \times 256 + 0 = 256$
- (ii) For a frequency of 300  
 $FL = 300 - 1 * 256 = 44$   
Testing this: frequency = hi byte x 256 + lo byte  
 $= 1 \times 256 + 44 = 300$
- (iii) For a frequency of 1025  
 $FL = 1025 - 4 * 256 = 1$   
Testing this: frequency = hi byte x 256 + lo byte  
 $= 4 \times 256 + 1 = 1025$

Two lines, 6030 and 6040 in Program 9.2(b), are added to incorporate this in the program. Line 6050 POKEs these values into the register.

#### PROGRAM 9.2(b)

```
6030 FH = INT(A(X)/256)
6040 FL = A(X)-256*FH
6050 POKE R,FL:POKE R+1,FH
```

So far, however, the program will still only play one note and the FOR...NEXT loop inserted at lines 6010 and 6080 in Program 9.2(c) will ensure that the program now plays through the scale of C major. On completion of this, the volume is turned off, line 6090.

PROGRAM 9.2(c)

```

Set register constant 100 R=54272
DIM                  1000 DIM A$(8),A(8)
READ note, freq.    1010 FOR Z=1 TO 8:READ A$(Z),A(Z)
                    :NEXT
Clear chip          1020 FOR Z=R TO R+24:POKE Z,0:NEXT
Set ADSR           1030 POKE R+5,9:POKE R+6,0
    set volume     6000 POKE R+24,15
P → set sawtooth  6010 FOR X=1 TO 8
L   set sawtooth  6020 POKE R+4,33
A   calc hi/lo   6030 FH=INT(A(X)/256)
Y   calc hi/lo   6040 FL=A(X)-256*FH
L   POKE note    6050 POKE R,FL:POKE R+1,FH
O   hold note    6060 FOR Z=1 TO 1000:NEXT Z
O   release sawtooth 6070 POKE R+4,32
P → set volume to 6080 NEXT X
    zero         6090 POKE R+24,0

                    30000 DATA C,2149,D,2500,E,2806,F,2973
                    30010 DATA G,3337,A,3746,B,4205,C,4297

```

So far, so good! The program works and plays a scale but needs much elaboration if it's to be of any real use. As the final program is going to end up as a pretty complex animal, we have no choice but to adopt a pretty rigid structured approach. There's a bonus too in this approach, as the modules that we develop will be mobile and capable of being incorporated into any other music program, game or whatever that you care to write.

## PART 2 - COMPOSATUNE

### The Structure

Comosatune is to be a utility that will help you to add sound to your programs but will also demonstrate the many features of the SID chip and, hence, your C-64. As the chip is so versatile and full of features, Comosatune will be a long and fairly complex thing. It's important, therefore, to give some thought to its shape before we get too far. One way to explore it is to RUN the version on disk and test out its major features.

### Playing with Comosatune

First LOAD and RUN Comosatune and this will present you with MENU 1.

\* Select 'I' for input.

enter NOTE 'F'

OCTAVE 4

DURATION 3

then NOTE 'F'

OCTAVE 4

DURATION 2

check progress by means of

NOTE 'P' <RETURN> for playback

then carry on through the sequence:

F,4,3;A,4,3;A,4,2;C,5,1;

C,5,1;F,5,5

Repeat the playback check by means of:

NOTE 'P' <RETURN>

Now to change the tempo, enter:

NOTE 'T' <RETURN>

This will produce the prompt "TEMPO". Enter:

TEMPO '300'

and then NOTE 'P' <RETURN> to see the effect. Try this out with a few more tempi.

To return to the menu, enter:

NOTE 'R' <RETURN>

\* Select 'P' for 'playback', this will playback your tune.

\* Select 'B' for Buildasound this will produce:

an introduction telling you something about Buildasound. On the second screen you will be asked which note mode. Select 'T' for tune then:

### **Buildasound Stage 1: pitch**

Select 'R' for 'raise pitch'. The screen will then report that the current frequency is H and you will hear the pitch raised and lowered as 'R' and 'L' are selected. When you have finished experimenting with this, leave stage 1 by entering an 'N'. This will direct the program into:

### **Buildasound Stage 2: volume**

Select to set the volume and enter a value of 1. On pressing <RETURN>, the volume will decrease considerably. A suggestion is that you leave the stage with the volume set at 15. Leave by means of an 'N' and enter:

### **Buildasound Stage 3: envelope**

Select 'C' to cycle ADSR, and then choose 'A' to cycle the attack phase. You will see that the attack starts at zero, while the others are at their current default values. Cycle the attack using the space bar. After the attack reaches 15 you will be returned to the menu - select 'A' again, and set it to a low attack number - say 4, using 'S' to set it. Next, try cycling the decay, sustain and release phases in turn, listening to the effects. Once this is done, try out the 'Set ADSR' and 'Pre-defined' options just to investigate the wide range of envelopes possible.

### **Buildasound Stage 4: waveform**

Select 'L' and listen to some of the waveforms. Note that when you listen to the Pulse waveforms you will be asked to set the 'pulse width'. Well, don't just sit there! Have a go! Use 'X' to exit when you're satisfied with the pulse width. Then try:

### **Buildasound Stage 5: filters**

Try the various filters to see what they do. You can hear the sounds when you use 'I' or 'D' to alter the frequencies. Next, have a go at:

### **Buildasound Stage 6: resonance**

Start off at zero, and 'I' your way through to 15. After this, comes:

## Buildasound Stage 7: revision

At this stage you can go back and change any of the earlier settings, if you wish.

As you saw in Program 9.1, many registers have to be set before SID produces a sound and this leads to the necessity of a very complex playback structure. However, most of these registers can quite safely be left set and then only one of those registers needs to be set or reset to make SID speak. Because of this, the process of specifying register values and setting these can be separated out into two discrete subroutines which are simply called when needed.

During the initialization phase of the program in lines 11060 to 11120, values are defined for all the variables that will subsequently be POKEd into the registers. These 'default' values will serve the purpose of enabling some sound to be produced easily and will be replaced when other values need to be set.

Once these, or any other variables have been set, a separate subroutine, located at 30500 onwards POKEs these into memory. This is written as a general-purpose subroutine that loads all the SID registers used in this program. Its general-purpose nature means that it can be called whenever registers need to be set. Some redundancy is involved in this as, in setting all the registers, the subroutine inevitably sets some that have not been changed. In the following pages, the Compositure program will be detailed in stages. You may follow along with the discussion by listing the appropriate sections as they are described.

### Initialisation

This routine prepares the way for the main program by setting the screen colors, variables etc., prior to the running of the main program.

Lines 10-30 set up the screen/border colors, turns the machine into lower-case and print an introductory message. This is supplemented on the disk version by a border routine that is set up by a GOSUB 60000 (line 11000) and then called by a SYSH, H being the start address of the machine-code routine.

The SID chip is cleared, line 11000 and a value of 47.5 is set for tempo (T). Next, the variables are DIMed in line 11010 and the first block of DATA read in line 11020. As DATA statements are used for various different purposes, a precaution is taken by reading in the first DATA element. If this is not 'XYZ' then further DATA is READ until this is found. This procedure ensures that the correct block of DATA is READ.



The next procedure is to set default values for envelope, pitch, volume and waveform. Following this, the POKE routine is called (see below) and the program sent to the first MENU.

In addition, the initialization procedure contains two 'GET' routines, one to return A\$ (line 11500) and one to return B\$ (line 11600).

PROGRAM 9.3(a)

```
2 POKE53280,3:POKE53281,1:PRINT"<BLU>"
10 PRINTCHR$(14)CHR$(8)"<CLR><7DCRSR>"T
   AB(13)"COMPOSATUNE
20 PRINT"<6DCRSR>"TAB(6)"(C)COPYRIGHT P
   HOLMES 1983
10999 REM INITIALISATION ROUTINE
11000 GOSUB60000:CLR:H=PEEK(55)+256*PEE
      K(56):R=54272:T=47.5:FORX=RTOR+24
      :POKEX,0:NEXT
11010 DIMA$(12),B$(12),A(12),FL(50),FH(
      50),DU(50):CN=0:X=0
11020 READA$:IFA$<>"XYZ"THEN11020
11030 FORZ=1TO12:READA$(Z),B$(Z),A(Z):N
      EXT
11040 DATAXYZ,C,C,268,C#,D@,284,D,D,301
      ,D#,D@,318,E,E,337,F,F,358
11050 DATAF#,G@,379,G,G,401,G#,A@,425,A
      ,A,451,A#,B@,477,B,B,506
11060 REM SET DEFAULT VALUES
11070 AT=5:DE=8:AD=88
11080 SU=5:RE=9:SR=89:FR=4454
11090 FH(1)=INT(FR/256)
11100 FL(1)=FR-256*FH(1)
11110 VO=15
11120 WF=32
11130 GOSUB30500:GOTO12000
11500 POKE198,0
11510 GETA$:IFA$=""THEN11510
11520 RETURN
11600 GETB$:IFB$<>""THEN11600
11610 GETB$:IFB$=""THEN11610
11620 RETURN
```

This program won't do much other than set the various sound variables. To complete the partnership we can put in the SID POKE ROUTINE.

### SID POKES

Many of SID's registers can be handled in a fairly straight-forward way, registers 5, 12 and 19, for instance, setting the attack/decay parameters for voices 1, 2 and 3. Others, however, set several disparate functions and one of the major advantages of a routine such as this is that by using it as a standard POKE section, one can be sure that all the registers have been set. In the case of register 4, this sets the parameters shown on Figure 9.2(a). In the case of registers 11 and 18, their structure is similar but they operate on different voices.

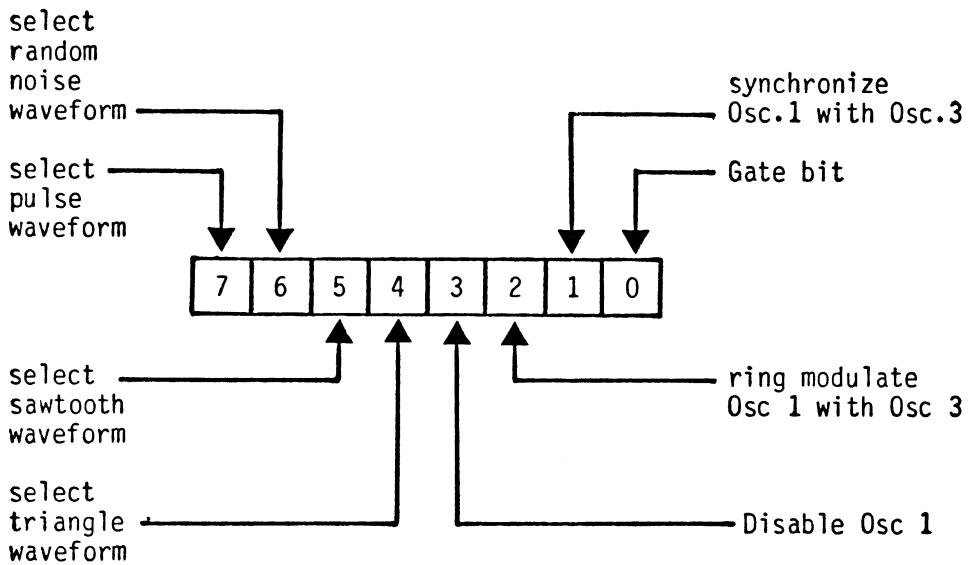


FIGURE 9.2(a)

The problem caused by this register structure is that, if one selects a sawtooth waveform by POKEing a 1 into bit 5 (i.e. a  $2^5$  into the register) then all the other bits are reset to zero and as bits 0 to 3 do other things, these would be turned off. To tackle the problems this routine POKEs everything appropriate that is set into register 4 each time; i.e.

```
30620 POKE R+4,WF+G+RM
```

The 'logical' way to tackle the problem would be to use logical operators but - beware

ALL THE INTERNAL SOUND CONTROL REGISTERS OF  
THE SID CHIP ARE WRITE-ONLY

Because of this the only way to know what's in a chip is to 'remember'. Routines such as this SID POKE help to ensure that nothing is 'unset' that should remain set. The various variables used in the SID chip POKEs are tabulated below (figure 9.2(b)) as a reference although their uses have not yet been explained. This will be done when required.

REGISTER FOR VOICE (V)	VARIABLE	FUNCTION
1 2 3		
0 7 14	FL(V)	Lo byte frequency
1 8 15	FH(V)	Hi byte frequency
2 9 16	LP	Lo byte pulse width
3 10 17	HP	Hi byte pulse width
4 11 18	WF+G	Register 4 POKE
4 11 18	WF	Waveform
4 11 18	G	Gate - enable
5 12 19	AD	Attack/decay
6 13 20	SR	Sustain/release
ALL VOICES		
21	LO	Lo nybble filter cut-off frequency
22	HI	Hi byte filter cut-off frequency
23	RP+FV	Register 23 POKE
23	RP	Resonance
23	FV	Voice filter enable
24	VO+FT	
24	VO	Volume
24	FT	Filter type

FIGURE 9.2(b)

Program 9.3(b) shows the simple sub-routine that executes the POKES. As this is a straight POKE sequence it in no way modifies the variables.

PROGRAM 9.3(b)

```
30500 POKER,FL(1)
30510 POKER+1,FH(1)
30520 POKER+2,LP
30530 POKER+3,HP
30540 POKER+4,WF+G+RM
30550 POKER+5,AD
30560 POKER+6,SR
30570 POKER+15,VO
30580 POKER+21,LQ
30590 POKER+22,HI
30600 POKER+23,RP+FV
30610 POKER+24,VO+FT
30620 POKER+4,WF+G+RM
30630 RETURN
```

One advantage of the SID POKES routine is that, at any stage in the program a sound can be turned on simply by setting the gate 'G' to 1 and going into the subroutine at 30500.

It can be just as easily turned off by setting G to zero and entering this subroutine. This provides for a controlled way of producing a sound by the process:

```
* set G=1
* GOSUB 30500
* set up delay loop
* set G=0
* GOSUB 30500
```

Many routines in Compositatune call for a single 'beep' sound and the two subroutines added at 30100 and 30300 call the POKE routine, execute a delay and then recall the POKE routine after setting the gate to zero (off) i.e. turning the sound off. A further routine provides for the play back tune choice when this option is chosen. The operation of the chip is fully discussed below, the aim of this introduction being to get a working sound routine.

To facilitate the use of this technique, two subroutines are provided to give delays of different lengths:

#### PROGRAM 9.3(c)

```
30000 GOSUB30500:FORZ1=1TO100:NEXT:G=0:
      GOSUB30100:RETURN
30100 GOSUB30500:FORZ1=1TO500:NEXT:G=0:
      GOSUB30500:RETURN
30200 REM G=1:FORZ1=1TOCN:GOSUB30700
30210 REM FORZ2=1TOT*2^CJ(Z1):NEXTZ2
30220 REM NEXTZ1:G=0:POKER+4,WF+G+RM:
      RETURN
30300 IFPB$="T"THENGOSUB17020:RETURN
30310 GOSUB30100:RETURN
```

#### MENU 1

This is the major control menu that calls the control sub-elements shown in Figure 9.3.

#### MENU 1

1.1	INPUT TUNE
1.2	PLAY-BACK TUNE
1.3	STORE IN DATA
1.4	RECOVER FROM DATA
1.5	BUILDASOUND
1.6	EXIT FROM COMPOSATUNE

FIGURE 9.3

At this stage of development, the program routing will be dummied in and the overall structure defined. Initially the options are printed onto the screen as in lines 12000 to 12070 of Program 9.3(d) along with the required input.

PROGRAM 9.3(d)

```

12000 PRINTCHR$(14)"<CLR><3DCRSR>"::SY
      H:PRINTTAB(15);"MENU1<DCRSR><5LCR
      SR>—————"
12010 PRINTTAB(8)"<2DCRSR>INPUT TUNE...
      .....I<DCRSR>"
12020 PRINTTAB(8)"PLAY BACK TUNE....P<D
      CRSR>
12030 PRINTTAB(8)"STORE DATA.....S<D
      CRSR>
12040 PRINTTAB(8)"READ DATA.....R<D
      CRSR>
12050 PRINTTAB(8)"BUILDSOUND.....B<D
      CRSR>
12060 PRINTTAB(8)"EXIT COMPOSATUNE..X<D
      CRSR>
12070 GOSUB11500

```

A\$, the inputted character, is checked and, on an erroneous entry, the program loops back to present another menu. A small point here! The loop is made to the <CLR> rather than to the GET, the flicker on the screen showing that something has happened i.e. the input has been accepted.

PROGRAM 9.3(e)

GET a character	12070 GOSUB 11500
prepare for 'INPUT'	12080 IF A\$<>"I" THEN 12140
INPUT routine	
prepare for 'PLAYBACK'	12140 IF A\$<>"P" THEN 12180
PLAYBACK routine	
prepare for 'STORE'	12180 IF A\$<>"S" THEN 12200
STORE routine	
prepare for 'READ'	12200 IF A\$<>"R" THEN 12220
READ routine	
prepare for 'BUILDSOUND'	12220 IF A\$<>"B" THEN 12240
BUILDSOUND routine	12240 IF A\$<>"X" THEN GOSUB 11500:
	GOTO 12070
EXIT routine	12250 GOTO 29000

Following the MENU, a character needs to be inputted from the keyboard and the standard 'GET' routines at 11500 and 11600 are to be used for this.

### Option 1: Input musical information

This routine, called by selecting an 'I' in MENU 1, inputs all the data necessary to define a musical note, displays this on the screen and plays the note. Other functions must be attended to as well by this routine and provision is made for the current tune to be played back at any stage, for the tempo to be set, for different voices to be selected, and finally for a return to MENU 1.

The section that controls this lies in lines 12080 to 12120, each routine being called by a GOSUB. Thus, the structure for the 'Input' option is:

- |                          |             |
|--------------------------|-------------|
| (1) Print stave          | GOSUB 15000 |
| (2) Print input prompts  | GOSUB 13000 |
| (3) Decode inputted data | GOSUB 14800 |
| (4) Print note on screen | GOSUB 15500 |

### PROGRAM 9.3(f)

```
12080 IF A$ <> "I" THEN 12140
12090 GOSUB 15000
12100 GOSUB 13000: IF N$ = "R" THEN 12000
12110 GOSUB 14800: GOSUB 15500
12120 GOSUB 17100: GOTO 12100
```

#### (i) Print Stave

This display remains on the screen for all of the time that the input routine is running and is most simply achieved by means of PRINT statements which utilize the C-64's built-in graphics characters:



PROGRAM 9.3(g)

```

15000 PRINTCHR$(142)"<CLR><DCRSR>  ^
      ^ | | |":PRINT" ^ |
      | | |"
15010 PRINT" ● =1 ● =2 ● =3 ○ =4 ○
      =5"
15020 PRINT"<2DCRSR>  ^"
15030 PRINT"—— H ——
      | |"
15040 PRINT"—— H ——
      | /"
15050 PRINT"—— V ——
      |"
15060 PRINT"—— / V ——
      | | |"
15070 PRINT"—— \ V ——
      |"
15080 PRINT" / |"
15090 PRINT" V"
15100 RETURN

```

(ii) Print input prompts

Each input of data is prompted and requires error checking routines. This error checking is provided by a separate 'GET' routine integrated into this subroutine and called each time a new input is made. On return from the 'GET' routine, the inputted data, returned as B\$, is re-assigned a variable name as appropriate.

Overall the input data routine must input:

- \* note data : pitch  
: duration
- \* control data : playback tune  
: set tempo  
: return from input subroutine

Where the inputted data is recognized as control data, the necessary action is taken immediately. For instance, a 'T' input for 'set tempo' sends the program to the subroutine at 14700 while a request for 'playback' directs the program to the playback routine at 17000.

Each time a prompt is displayed, the program is re-directed to the GET routine:

## GET a character ....

In order to facilitate the inputting of the various data elements, this subroutine is provided which has built-in error checking to eliminate some of the obvious errors. As some entries are to be one character long and others two, a variable (E) is set to 1 or 2 prior to entering the subroutine. The inputted characters are returned in the variable B\$. Program 9.3(h) shows the main features of this program.

### PROGRAM 9.3(h)

```
initialize loop counter      13200 Z$=" ":K=1:
set input variable to
null string                  B$ = "":
Set up loop                  FOR Z5=0 TO E:
Make up space                Z$=Z$+"<LCRSR>":
for inputted
character
move cursor                  PRINT"<RCRSR> ";;
end loop                      NEXT
print spaces                 13210 PRINT"<RETURN><8LCRSR>";Z$
print a cursor               13220 PRINT"█<LCRSR>";:
perform a GET                GOSUB 11500
check for a delete           13230 IF A$ = CHR$(20) AND K>1 THEN
decrement loop counter      K=K-1:
print inputted character     PRINT "<LCRSR> <2LCRSR>";:
remove last character        B$ = LEFT$(B$,LEN(B$)-1):
go for another character     GOTO 13210
check for RETURN pressed    13240 IF A$=CHR$(13)THEN PRINT " ":
if so perform a RETURN      RETURN
enough characters input,    13250 IF K>E THEN 13220
if not GET more
check for control           13260 IF ASC(A$)<32 OR
characters
check for off reverse       ASC(A$)>127 THEN 13220
characters
GET another character
Build up B$                 13270 B$=B$+A$:
increment loop counter      K=K+1:
print inputted character    PRINT A$;;
GET another character        GO TO 13220
```

When this has been added, the program, when run, will carry out the initialisation, print the menu and when 'I' is selected run into the GET a character routine. Once this has been done, the next stage is to decode the character that has been entered and act accordingly.

### The input routine

The GET a character subroutine returns the inputted data as the variable B\$ and the first task on returning from this, is to check whether one of the control functions has been selected ie:

The control functions provided allow for the tune inputted to data to be played back, for the tempo to be set and for the option to be quit.

### Playback of tune to date

If a 'P' is inputted at the "NOTE" prompt, then a jump to the playback routine can be implemented. As this is a fairly trivial programming task it is readily achieved, although not in too structured a way by putting in a check and GOSUB during the subroutine itself, ie:

```
13020 IF NS="P" THEN GOSUB 17000:GOTO 13000
```

check for  
"Play"

go to  
playback  
routine

return to  
input  
routine

### Set tempo

A further subroutine that can be incorporated here is the tempo change facility. Again an input check and GOSUB handles the process, ie:

```
13030 IF N$="T" THEN 14700
```

check for  
tempo

go to tempo  
setting routine

When the input detects that a setting of tempo is required, the tempo routine first of all prints the word 'TEMPO' over the earlier prompt 'NOTE'. It then jumps to the input a character routine to collect the tempo value. The tempo value 'T' is then evaluated by means of:

$$T = \text{INT}(5000/\text{VAL}(B\$))$$

This factor 5000 may be modified to suit your own needs but it is calculated so that the inputting of a tempo T will result in the tune being played at T crotchets per minute:

i.e. if T=120 tune plays at 120 crotchets per minute.

```
14700 E=3:PRINT"<UCRSR>TEMPO=";;GOSUB 13
200:T=INT(5000/VAL(B$)):PRINTT:GOTO 13000
```

### Return to MENU 1

The final function accessed from this input subroutine is the 'return to MENU 1'. An 'R' inputted achieves this by means of a simple RETURN.

Together these three functions appear in Program 9.3(i) as:

### PROGRAM 9.3(i)

```
13020 IF N$="P" THEN GOSUB 17020:GOTO 13000
13030 IF N$="T" THEN 14700
13040 IF N$="R" THEN RETURN
```

Once the musical note name is inputted, decoding it into a frequency requires only that the note name be matched with a position in the array A\$( ) and that the corresponding element in array A( ) be found. As this can be done in one line, it is included in this subroutine although this decoding is, strictly-speaking, a routine of its own.

At the 13090 stage where the octave is inputted, an error check is utilized to sift out inputs which are:

- below zero
- above seven
- not integers
- zero when the note input was not zero

i.e. IF OC<0 OR OC> 7 OR OC<>INT(OC) OR OC=0 AND B\$<>"0" THEN ...

Error checks are implemented after the duration input to sift out durations which are:

below one  
above five  
not integers

i.e. IF DU<1 OR DU>5 OR DU<>INT(DU) THEN

When such errors are found, the cursor is moved back over the previous input to await the next one.

Program 9.3(j) shows all this in action!

### PROGRAM 9.3(j)

position cursor	13000 PRINT"<HOME><19DCRSR>"
display prompt	13010 PRINT "NOTE= <4LCRSR>";:
set variable:number of	E=2:
characters for input	
go to input character	GOSUB 13200:
routine	
store character as N\$	N\$=B\$
check for playback	13020 IF N\$ = "P" THEN GOSUB 1700:
	GOTO 1300
check for tempo	13030 IF N\$ = "T" THEN 14700
check for stop	13040 IF N\$ = "R" THEN RETURN
increment character number	13050 CN=CN+1
clean & entry from screen	13070 NEXT:IF N=0 THEN PRINT "
	<2UCRSR>":
go back for another entry	GO TO 13000
display prompt	13080 PRINT"OCTAVE=<RCRSR><LCRSR>";:
set loop control variable	E=1:
input a character	GOSUB 13200:
store character in OC	OC=VAL(B\$)
check for errors	13090 IF OC<0 OR OC>7 OR OC<>
	INT(OC) OR (OC=0 AND B\$<>"0")
	THEN PRINT "<2UCRSR>":
go back for another entry	GOTO 13080
	13100 PRINT"DURATION= <LCRSR>";:
set loop counter variable	E=1:
input a character	GOSUB 13200:DU(CN)=VAL(B\$)
check for errors	13110 IF DU(CN)<1 OR DU(CN)>5
	OR DU(CN)<>INT(DU(CN))THEN
	PRINT"<2UCRSR>":
go back for another entry	GO TO 13100
end subroutine	13120 RETURN

As it stands, the program runs as far as inputting the characters required and can be tested by running the program, once the next routine has been written. However, a safer way is to test the module individually under more controlled conditions.

Generally speaking none of the subroutines in Composatune are so complex as to require elaborate test procedures. However, even the humblest of routines can crash a whole system. A few examples of systematic testing are given just to show how it might be done in a large complex system. Testing is illustrated below for the input of a note, its octave and duration, in subroutine 13000.

### Module Testing

When written in modular form, the subroutine can be tested prior to incorporation into the program as a whole. First of all, a short calling program should be written which calls this subroutine and then reports back on its performance. Program 9.3(j/i) does this, reporting back on the values of B\$, OC and D. This is not a section of Composatune, but can be added to test the subroutine.

#### PROGRAM 9.3(j/i)

```
0 GOTO 40
40 DIM A$(20),B$(20),A(20):GOSUB11030:GOSUB13000
42 PRINT"B$=";N$:PRINT"OC=";OC;"D=";DU(CN):END
```

And then edit line 11030 to be:

```
11030 READ A$(0):FORZ=1TO12:
      READA$(Z),B$(Z),A(Z):NEXT:RETURN
```

Type RUN and test the various functions by entering:

```
1 Note      = C
2 Octave    = 4
3 Duration  = 1
```

The report should tell:

```
B$ = C
OC = 4
D = 1
```

Next test the routine with invalid entries such as:

```
B$ >"H"
OC <1 or >6
D >5
```

Once it passes these tests, the module is ready for use.

## Decoding data

The input routine collects the data from the user and stores it in the appropriate variables. The next task is to transform this data into a form that can be used to sound the notes and display them on a musical stave.

The parameters inputted in the previous module are decoded and transformed in the following way:

### 1. Note frequency FR is transformed to hi/lo PEEK values

A musical notation input is taken and its corresponding frequency found and transformed into hi and lo POKE values i.e.

- (i) B\$ looked up in array A\$(N) to find location in array i.e. N.
- (ii) Relevant frequency found in position N in array A(N).
- (iii) If no entry found, return for another input.
- (iv) Variable FR set equal to A(N) for ease of use later.
- (v) Value of frequency hi POKE value calculated using FR.
- (vi) Value of frequency lo POKE value calculated using FR.

Putting this into a program yields:

- (i)&(ii) N=0:FOR Z=1 TO 12:IF B\$=A\$(Z)OR B\$=A\$(Z)THEN N=Z:Z=20:NEXT
- (iii) IF N=0 THEN PRINT "<2UCRSR>":GOTO 13000
- (iv) FR = A()
- (v) FH() = INT(FR/256)
- (vi) FL() = FR-256\*FH()

Note that (iv) and (v) automatically store the inputted data in the arrays FH() and FL():

```
13060 N=0:FOR Z=1 TO 12:IF N$=A$(Z)OR N
$=B$(Z)THEN N=Z:Z=12
13070 NEXT:IF N=0 THEN PRINT"<2UCRSR>":
GOTO 13000
```

## 2. Calculating (X,Y) co-ordinates for the plotting of the notes

- (i) The X co-ordinate is incremented by one on each pass through the subroutine.
- (ii) Each Y co-ordinate is calculated individually as it is plotted. C4, the lowest note, is just below the stave, 16 (17-1) lines down from the top of the screen. The next natural note, D is 15 (17-2) lines down and so on. Thus, each note's position is actually (17-N) lines down, where N is the note's position in the DATA statement. Line 14840 takes care of the processing of both X and Y and, when handling the Y co-ordinate, must take account of the sharps and flats. As these are plotted on the same line as the natural note, the value of Y is not decreased for a natural. Their presence is recognized on line 14840 as the DATA character for both a sharp and a flat is two letters long, e.g.C#(sharp) or E@(flat). When the stored character is only one letter, and only then, is the value of Y decremented.

One final adjustment is needed for the Y co-ordinate when the INPUT is octave 5. In this case a further 7 is subtracted from Y to move the character up by one octave.

The decode subroutine is thus:

### PROGRAM 9.3(k)

```
14800 N=0:FOR Z=1 TO 12:IF N$=A$(Z)OR N$
=B$(Z)THEN N=Z:Z=20
14810 NEXT Z:FR=A(N)*2↑OC
14820 FH(CN)=INT(FR/256)
14830 FL(CN)=FR-256*FH(CN)
14840 X=X+1:Y=17:IF X=6 THEN X=0:GOSUB 1
5000:READ Y
14850 FOR Z=1 TO N:IF LEN(A$(Z))=1 THEN
Y=Y-1
14860 NEXT
14870 IF OC=5 AND N<9 THEN Y=Y-7
14880 RETURN
```

### Module Testing

This module needs testing primarily to prove its transformation function, i.e. how it transforms data from one form to another. Two transformations are carried out:



- (i) B\$ is transformed into hi/lo frequency POKE values. Program 9.3(h) tests this function by inputting a note value and comparing the outputted values of FH() against the standard table.

To test this, add Program 9.3(k/i) to 9.3(j/i).

PROGRAM 9.3(k/i)

```
40 DIM A$(20),B$(20),A(20):GOSUB11030:GOSUB 14800
42 PRINT "FH(1)=";FH(1),"FL(1)=";FL(1):END
```

Clear the screen, run this and test for a 'C' INPUT with OCTAVE=6 and DUR=3, the two output values should give (European CBM 64 values in parenthesis):

FH(1) = 67(69) and FL(1)=0(155)

other tests are

```
E   FH(1) = 84(87)   FL(1) = 64(179)
G   FH(1) = 100(104) FL(1) = 64(74)
D   FH(1) = 75(78)   FL(1) = 64(33)
```

Now delete lines 0, 40, and 42 by typing in the line number and pressing RETURN. Also edit line 11030 to read:

```
11030 FORZ=1TO12:READA$(Z),B$(Z),
A(Z):NEXT
```

#### Print note on screen

Having inputted the data and decoded it, it can be displayed in its correct position on a stave on the screen. Thus, this sub-routine breaks down into two parts; the display of the stave and the display of individual notes on it.

Two elements make up the visual display, the basic stave and the notes. These two units are used differently and are, therefore, written as two separate routines.

#### 4. The Stave

As the graphics remains on the screen during most of the program, it can be put there simply by a series of PRINT routines - lines 15000 to 15100 Program 9.3(g).

## 5. Plotting the notes.

Each time a note is entered, the cursor needs to be moved to the (X,Y) co-ordinates. Line 15500 does this within the FOR...NEXT loop by cursoring down one for every unit of Y and across 7 for each unit of X. Following this, a check is made for sharps and flats (lines 15530/40) and then the notes themselves are plotted in lines 15550 to 15600.

### PROGRAM 9.3(1)

```
15500 PRINT"<HOME>"::FORZ=1TOY:PRINT"<D
CRSR>"::NEXT:FORZ=1TOX:PRINT"<7RC
RSR>"::NEXT
15520 IFN=1ANDOC<>5THENPRINT"<DCRSR><
2RCRSR>—<UCRSR><5LCRSR>";
15530 IFRIGHT$(N$,1)="@"THENPRINT"*<DCR
SR><LCRSR>*74<DCRSR><3LCRSR>*N<RC
RSR><UCRSR>";
15540 IFRIGHT$(N$,1)="#"THENPRINT":/L<D
CRSR><3LCRSR>* 4<DCRSR><3LCRSR>P7
O<UCRSR>";
15550 IFLEN(N$)=1THENPRINT"<3RCRSR><DCR
SR>";
15560 IFDU(CN)=1THENPRINT"Q<UCRSR><LCRS
R>*M<UCRSR><2LCRSR>*M<LCRSR>";
15570 IFDU(CN)=2THENPRINT"Q<UCRSR><LCRS
R>*<UCRSR><LCRSR>*M<LCRSR>";
15580 IFDU(CN)=3THENPRINT"Q<UCRSR><LCRS
R>*<UCRSR><LCRSR>*";
15590 IFDU(CN)=4THENPRINT"W<UCRSR><LCRS
R>*<UCRSR><LCRSR>*";
15600 IFDU(CN)=5THENPRINT"W";
15610 PRINT"<28UCRSR><18DCRSR><2LCRSR>"
;OC;
15620 RETURN
```

## 6. Audio Display/Playback

This subroutine has the structure:

- \* check whether any tune stored
- \* gate sound (gate function is explained more fully under 'ENVELOPE')
- \* set up loop for number of notes to be played
- \* hold program in delay loop
- \* turn gate off

### PROGRAM 9.3(m)

```
17000 IF CN<>0THEN17020
17010 PRINT"<CLR><11DCRSR>"TAB(12)"NO T
      UNE STORED":SYSH:FORXX=1TO1000:NE
      XT·RETURN
17020 G=1:FORZ1=1TOCN:GOSUB17500
17030 FORZ2=1TOT*2^DU(Z1):NEXTZ2:G=0:GO
      SUB30620:G=1
17040 NEXTZ1:G=0:POKER+4,WF+G+RM:RETURN

17100 G=1:Z1=CN:GOSUB17500
17110 FORZ2=1TOT*2^DU(Z1):NEXTZ2
17120 G=0:POKER+4,WF+G+RM:RETURN
17500 POKER,FL(Z1)
17510 POKER+1,FH(Z1)
17520 POKER+4,WF+G+RM
17530 RETURN
```

### Option 2 : Play back tune.

Having written the sub-routine that produces an audio output, the playback routine requires only that the relevant subroutine be accessed.

### PROGRAM 9.3(n)

```
12140 IFA$<>"P"THEN12180
12150 GOSUB17000:GOTO12000
```

### Option 3 : Store DATA

This option allows the user to store the inputted data as DATA statements ready for use in his/her own programs. Its basic structure is identical to that used in both Char.Gen and Sprite.Gen so no detailed explanation will be provided. It is called by a GOSUB in line 12090.

PROGRAM 9.3(o)

```

12180 IFA$<>"S"THEN12200
12190 GOTO19000
19000 PRINT"<CLR><3DCRSR>"TAB(15)"BASIC
      DATA"
19010 PRINT"<5DCRSR><RCRSR>AT WHICH LIN
      E WOULD YOU LIKE TO STORE<3RCRSR>
      THE DATA?(100-255)?"
19020 INPUT"<HOME><13DCRSR><3RCRSR>";LN
      :IFLN<100ORLN>255THEN19000
19030 INPUT"<3RCRSR>IN STEPS OF";SP
19040 PRINT"<CLR><3DCRSR>";LN;"DAABC,"C
      N"<LCRSR>,"AD"<LCRSR>,"SR"<LCRSR>
      ,"VO"<LCRSR>,"WF"<LCRSR>,"T"<LCRS
      R>,"CT
19050 A=CN:B=0:C=1:FORZ=1TOCN:POKE829+Z
      *3,FL(Z):POKE830+Z*3,FH(Z)
19060 POKE831+Z*3,DU(Z):NEXT
19070 F=0:LV=A:IFA>=24THENLV=24:A=A-24:
      F=1
19080 L1=INT((LV+2)/3):L2=LV-INT(LV/3)*
      3:IFL2=0THENL2=3
19090 FORX1=1TOL1:PRINTLN+(B*8+((X1-1))
      +1)*SP;"DATA";
19092 X3=3:IFX1=L1THENX3=L2
19095 FORX2=1TOX3
19100 PRINTPEEK(820+B*24+X1*9+X2*3);"<L
      CRSR>,";
19110 PRINTPEEK(821+B*24+X1*9+X2*3);"<L
      CRSR>,";
19120 PRINTPEEK(822+B*24+X1*9+X2*3);"<L
      CRSR>,";
19130 NEXTX2:PRINTCHR$(20):NEXTX1:PRINT
      "GOTO19170"
19140 POKE826,SP:POKE827,F:POKE828,LN:P
      OKE829,A:POKE830,B
19150 PRINT"<HOME>":FORX3=1TOL1+C+2:POK
      E630+X3,13:NEXT:POKE198,L1+C+2:C=
      0
19160 END
19170 SP=PEEK(826):LN=PEEK(828):A=PEEK(
      829):B=PEEK(830)
19180 IFPEEK(827)THENB=B+1:PRINT"<CLR><
      4DCRSR>";:GOTO19070
19190 FL=1:GOTO1

```

#### Option 4 : Read DATA

This option is also very similar to those subroutines in Chapters 6 and 8 and so, again is simply listed along with the GOSUB that calls this.

PROGRAM 9.3(p)

```
12200 IFA$<>"R"THEN12220
12210 GOSUB20000

20000 PRINT"<CLR>  READ DATA"
20010 READCN,AD,SR,VO,WF,T,CT
20020 FORZ=1TOCN:READFL(Z),FH(Z),DJ(Z):
      NEXT
20030 RETURN
```

#### Option 5 : Buildasound

The Composatune program has several menu options which we call the Buildasound routine. This option provides the facility for the user to build up a sound stage by stage, while hearing the sound. Once a sound is developed it is automatically stored as the default value used in the program generally.

Buildasound's structure is based on a series of interlinked menus, being an example of a 'menu-driven' routine. With such a structure, the user can work to the instructions displayed on the screen and thus have a path charted through the possible routines.

One MENU serves as the master menu that drives the others and redirects them on return to this. It provides the only way out of the Buildasound routine by means of its 'EXIT' option.

Each menu offers its own particular features along with the standard options to change a particular Buildasound stage or to move onto the next one. This latter feature serves to define a particular route through the program so that the user can explore its features in a systematic way. At each 'stage menu' the option 'N' will, therefore, direct the program to the next stage menu. At any menu, the option 'X' will redirect the program to the Buildasound master Menu and, ultimately, out of the Buildasound routine altogether. Figure 9.4 shows the overall menu structure of Buildasound and the various routes through it.

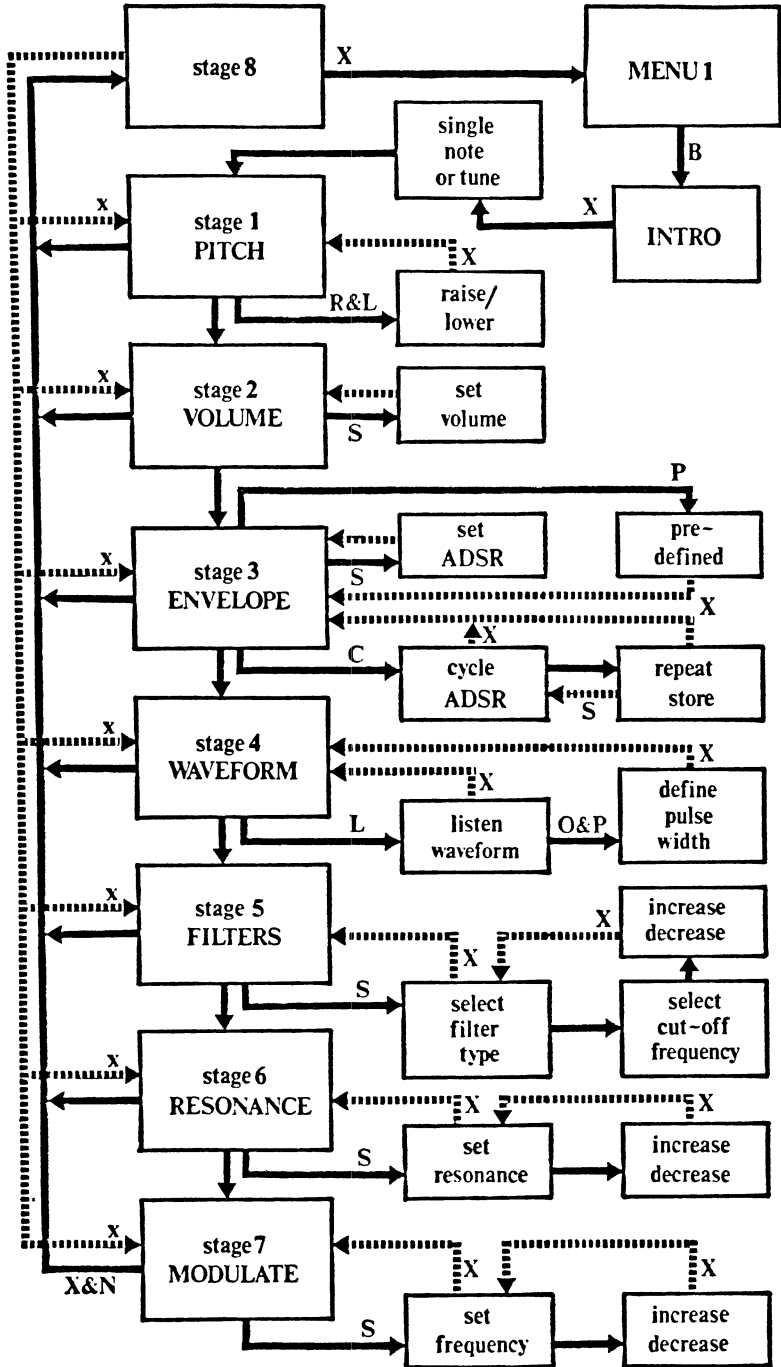


FIGURE 9.4

At each stage of Buildasound, the user can experiment with the sound parameters and, when desired, reset the default values. In this way a new sound can be built up progressively. Such is the finesse of the SID chip that a sound can, in fact, be very 'finely tuned'. However, that's the sound that's leaving the computer! If you are listening on a television with a typically very poor speaker, some of the nuances of the SID chip may be lost in its circuits!

### The Buildasound Routines

In the COMPOSATUNE program, the Buildasound routine has some preliminary material that introduces the user to the routine. This is listed below in Program 9.3(q). In addition, some variables are set to contain data strings that are used frequently in Buildasound (lines 21160-21190). The Buildasound routine is called by a GOSUB in line 12230.

#### PROGRAM 9.3(q)

```

12220 IFA$<>"B"THEN12240
12230 GOSUB21000:GOTO12000
21000 PRINT"<CLR><2DCRSR>";:SYSH:PRINTT
      AB(13);"BUILDASOUND<DCRSR><11LCRS
      R>777777777777"
21010 PRINT"<2DCRSR><3RCRSR>THIS ROUTIN
      E ENABLES YOU TO BUILD
21020 PRINT"<DCRSR><3RCRSR>A SOUND IN T
      HE FOLLOWING WAY, AND
21030 PRINT"<DCRSR><3RCRSR>TO HEAR IT A
      T ALL STAGES.
21040 PRINTTAB(7);"<2DCRSR>1  DEFINE PI
      TCH"
21050 PRINTTAB(7);"3  DESIGN ENVELOPE"
21055 PRINTTAB(7);"2  SET VOLUME
21060 PRINTTAB(7);"3  DESIGN ENVELOPE"
21070 PRINTTAB(7);"4  DEFINE WAVEFORM"
21080 PRINTTAB(7);"5  SET FILTERS"
21090 PRINTTAB(7);"6  SET RESONANCE"
21100 PRINTTAB(7)"<2DCRSR>PRESS ANY KEY
      FOR STAGE 1"
21110 GOSUB11500
21160 M$="<CLR><3DCRSR><7RCRSR>BILDASO
      UND STAGE<DCRSR><17LCRSR>77777777
      7777777777<UCRSR><LCRSR>"
21170 R$="<7RCRSR>":S$="<2DCRSR><3RCRSR
      >"+R$
21180 P$=R$+"NEXT STAGE.....N<DCRSR>
      "
21190 Q$=R$+"EXIT STAGE.....X<DCRSR>
      "

```

At each stage of Buildasound, the option is offered for the user to play a single note or repeat the tune. This choice is presented via a MENU and a variable PB\$ set to record the choice. Each time a playback is requested, this variable PB\$ is checked and when this is set to "T" the program is re-routed appropriately.  
PROGRAM 9.3(r)

```
21200 PRINT"<CLR>";:SYSH:PRINTR$"<5DCRS
R> SELECT NOTE MODE
21205 PRINTR$" 7777777777777777<3DCRSR
>
21210 PRINTR$"SINGLE NOTE.....N<DCRSR
>
21220 PRINTR$"TUNE.....T<DCRSR
>
21230 GOSUB11500
21240 PB$=A$:IFPB$="T"THEN23000
```

#### Option 5.1: Define pitch

This option is written as an example of a completely menu-driven routine which requires only that the user respond to menu prompts. It provides for four options:

- \* raise pitch
- \* lower pitch
- \* move to next stage
- \* exit from option

A short section, from 22100 to 22120 decodes the menu directing the 'raise' and 'lower' routines to a calculation section where the frequency FR (initially 4291, US or 4454 European) is broken down into FH(1) and FL(1) values. The gate variable (G) is then set to 1 and the SID POKE routine at 30500 is entered. Throughout the remainder of the option, the sound remains gated, only being turned off when either the 'exit' or 'next stage' options are selected.

When either the 'raise' or 'lower' options are selected the frequency, FR, needs to be adjusted either up or down. This could be done by starting at some arbitrary value for FR and increasing it by a percentage or a straight forward number of cycles each time. The frequency values obtained by doing this would only rarely coincide with the frequencies of 'musical notes' i.e. C, F# G etc. and so, the option chosen is to increase or decrease the notes in semitone steps. To do this, the frequency steps for semitones need to be worked out.



The frequency of a note, say C4 is exactly twice that of the note one octave below, say C3. Between these two notes, on the musical scale are twelve intervals and the debate about how to divide up this gap has raged for many centuries. We will take the rather dry and mathematical approach and adopt the equal tempered scale. It's clear that as frequencies get higher, the frequency interval between semitones gets greater; the gap between the octaves A4 and A5 is (880-440) i.e. 440Hz while that between A3 and A4 is only (440-220) i.e. 220 Hz.

Thus, the ratio between any note and the semitone above is given by the twelfth root of 2 i.e. frequency 1 and frequency 2, one semitone apart are related as:

$$\text{frequency 1} = \text{frequency 2} * (2^{(1/12)})$$

To try this out enter the following routine which will:

```
* Set frequency (FR)=200
* Set up a twelve times loop
* Multiply FR by 2^(1/12)
* Print out FR
* Do next loop
* End
```

PROGRAM 9.3(r/i)

```
i.e. 10 FR=200
      20 FOR Z=1 TO 12
      30 FR=FR*(2^(1/12))
      40 PRINT FR
      50 NEXT Z
      60 END
```

When this is run, it should give a series of numbers like:

```
211.892619
224.49241
237.841423
251.98421
266.967971
282.842713
299.661416
317.480211
336.358566
356.359487
377.549725
400
```

In other words, the octave gap has been split up by means of a geometric progression. Lines 22200 and 22210 carry out this raising and lowering of the frequency and when the value of this is reported back, it is the INT( ) of FR that is used rather than FR itself.

PROGRAM 9.3(s)

Buildasound stage 1

```

22000 PRINTM$"1":SYSH
22030 PRINTR$"<3DCRSR>RAISE PITCH.....
      .R<DCRSR>"
22040 PRINTR$"LOWER PITCH.....L<DCRSR
      >"
22060 PRINTP$:PRINTQ$
22090 GOSUB11500
22100 IFA$="N" THEN G=0:GOSUB30620:GOTO23
      000
22110 IFA$="X" THEN G=0:GOSUB30500:GOTO28
      000
22120 IFA$="R" OR A$="L" THEN 22140

22130 GOTO22090
22140 PRINTTAB(5);"<3DCRSR>CURRENT FREQ
      UENCY=      <7LCRSR>";
22180 IFA$="R" THEN FR=FR*(2^(1/12))
22190 IFA$="L" THEN FR=FR*(1/(2^(1/12)))
22200 FH(0)=INT(FR/256):IF FH(0)>255 THEN
      FH(0)=255
22210 IF FH(0)<0 THEN FH(0)=0
22220 FL(0)=FR-256*FH(0):IF FL(0)>255 THEN
      NFL(0)=255
22230 PRINTINT(FR)"<4UCRSR>":IF FL(0)<0 THEN
      HENFL(0)=0
22240 G=1:GOSUB30500
22250 GOTO22090

```

Option 5.2: Set volume.

As the whole process of setting the volume consists of one POKE into register 24 the routine used in Program 9.3(t) seems to need little explanation!

PROGRAM 9.3(t)

Buildasound stage 2

```
23000 PRINTM$"2":SYSH
23010 PRINTS$"SET VOLUME<2DCRSR>"
23020 PRINTR$"SET VOLUME.....S<DCRSR
>"
23030 PRINTP$:PRINTQ$
23040 GOSUB11500
23050 IFA$<>"S"THEN23080
23060 G=1:GOSUB30500
23070 PRINTTAB(11)"<3DCRSR>";:E=2:GOSU
B13200:VO=VAL(B$):GOSUB30500
23080 IFA$="N"THENG=0:GOSUB30620:GOTO24
000
23090 IFA$="X"THENG=0:GOSUB30620:GOTO28
000
23100 IFA$<>"N"THEN23000
```

Option 5.3: Envelopes

The envelope option of Buildasound provides facilities for the user to:

- \* cycle through the ADSR parameters
- \* set a specific envelope
- \* adopt pre-defined envelopes

The menu from lines 24000 to 24100 provides the display and basic decoding:

PROGRAM 9.3(u)

```
24000 PRINTM$"3":SYS H
24010 PRINTS$"SET ENVELOPE<2DCRSR>"
24020 PRINTR$"CYCLE ADSR.....C<DCRSR
>"
24030 PRINTR$"SET ADSR.....S<DCRSR
>"
24040 PRINTR$"PRE-DEFINED.....P<DCRSR
>"
24050 PRINTP$:PRINTQ$
24060 GOSUB11500
24070 IFA$="X"THENG=0:GOSUB30620:GOTO28
000
24080 IFA$="N"THEN25000
24090 IFA$="S"THEN24530
24100 IFA$="P"THEN24600
```

This stage allows the user to investigate a sound's envelope and to set it as required. The definition is a simple process, but the envelope is a complex thing. A sound may be analyzed in many ways but one generally accepted method is to describe the life-cycle of a sound in terms of the 'Envelope' that contains it. Within this envelope, the sound's volume rises to its maximum, decays to its steady level, holds this for a period and then decays away. On the SID chip, the process is started by a 'gate' being opened and the final decay sequence is initiated by that gate being closed. Figure 9.5 shows the envelope cycle along with the associated gating function.

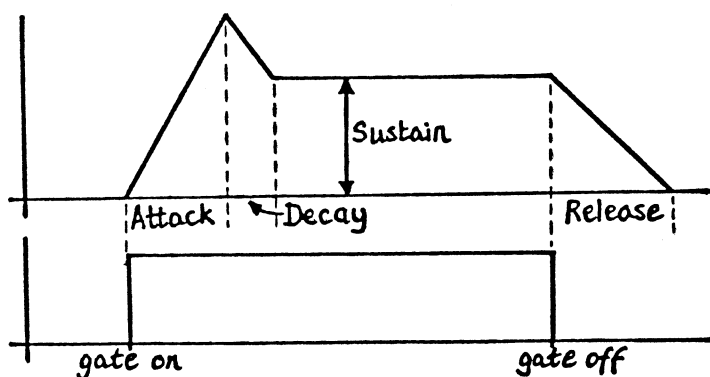


FIGURE 9.5

The gate function.....

As the process of gating the envelope is really an ON/OFF one, only one bit is needed to carry this out. Bit zero of register 4 handles the requisite information, a 1 signifying the gate ON and a zero signifying the gate OFF. This bit can only be turned on and off by setting the contents of the register appropriately, this being with a zero or one.

Following the turning on of the gate bit i.e. by setting bit 1 of R+4 to 1, the following happens automatically:

(i) SID commences the 'attack'.

The amplitude of the sound (its volume) rises to its maximum value. The time taken to do this is controlled by the value stored in the bits zero to 3 of R+5. A value of zero sets the attack time to 2 thousandths of a second (2ms or 2 milliseconds) while a value of 15 sets this to 8 seconds. Figure 9.7 tabulates the intermediate values for this.

At this completion of the 'attack' phase, SID will immediately:  
(ii) commence the 'decay'.

At this point, the amplitude of the sound begins to decay away to its steady state value. The time taken for this decay cycle is set by the least significant nybble of R+5. A value of zero in this nybble gives a cycle decay time of 6 ms, while a value of 15 gives a decay of 24 seconds. Figure 9.7 tabulates the intermediate values for this.

At the completion of the 'decay' phase, SID will automatically:  
(iii) commence the 'sustain'.

This is the steady part of the cycle where the amplitude (volume) is held at a constant value. Notice that unlike the values for 'attack', 'decay' and 'release', the values of which refer to the timing of the phases, the value of 'sustain' is the actual amplitude which is held during the 'sustain' phase and is defined by the most significant nybble of R+6. a value of zero in this nybble produces a sustain volume of zero, i.e. in effect the 'decay' phase will be the last audible phase. With the sustain nybble set to 15, there will be no fall in amplitude during the 'decay' phase and consequently there will appear to be no decay phase. A value of 8, on the other hand, will cause the amplitude to fall to a half of the peak amplitude during the 'decay' phase. Figure 9.6 illustrates the two extreme cases.

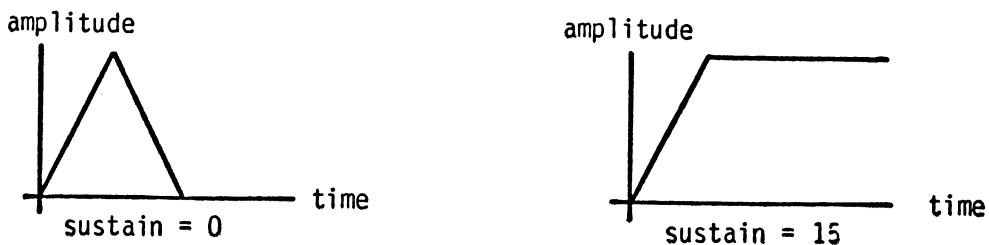


FIGURE 9.6

The 'sustain' phase will be terminated when the 'gate' bit of register 4 is switched off. This will initiate:

(iv) the 'release' phase.

At this point, the amplitude begins to decay to zero, the time taken for this phase being set by one nybble of register 6. A zero value in this nybble gives a release time of 6 ms while a value of 15 gives a 24 second cycle time. Figure 9.7 tabulates the intermediate values for this.

The description given above illustrates the normal use of the 'gate' bit to control the ADSR cycle. However, it is possible to switch the gate bit either off or on during any part of the ADSR cycle. If the gate bit is switched off during the 'attack' or 'decay' phases then the release phase will start immediately, missing out the sustain and possibly part or all of the decay and attack phases too. Similarly the gate bit may be switched back on at any time during the 'release' phase and, if this happens, the release will be abandoned and the new attack will commence starting at the volume reached by the release at the moment of 'gate on'.

VALUE DEC (HEX)	ATTACK RATE (Time/Cycle)	DECAY RELEASE RATE (Time/Cycle)
0 (0)	2 ms	6 ms
1 (1)	8 ms	24 ms
2 (2)	16 ms	48 ms
3 (3)	24 ms	72 ms
4 (4)	38 ms	114 ms
5 (5)	56 ms	168 ms
6 (6)	68 ms	204 ms
7 (7)	80 ms	240 ms
8 (8)	100 ms	300 ms
9 (9)	250 ms	750 ms
10 (A)	500 ms	1.5s
11 (B)	800 ms	2.4s
12 (C)	1s	3s
13 (D)	3s	9s
14 (E)	5s	15s
15 (F)	8s	24s

FIGURE 9.7

To control the ADSR cycle, five parameters must be set:

Attack - A  
 Decay - D  
 Sustain - S  
 Release - R  
 Gate - G

Four of these, A, D, S and R have provision on the SID chip for setting at any integer value between 0 and 15 and this can be contained within only four bits of binary, i.e.:

$$\begin{array}{cccc|c}
 1 & 1 & 1 & 1 & = 1111_2 \\
 8 & + & 4 & + & 2 & + & 1 & = 15_{10}
 \end{array}$$

In order to economize on space, one eight-bit byte can be used to store two four-bit numbers and, when one byte is used to store two 4-bit numbers, each of these four bit structures is referred to as a nybble (half a byte equals a nybble! corny eh?) Thus, when storing two nybbles each of 15 in one byte, the memory location would appear to contain eight 1's i.e.

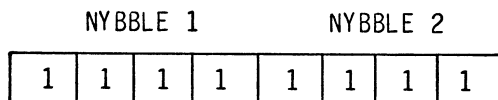


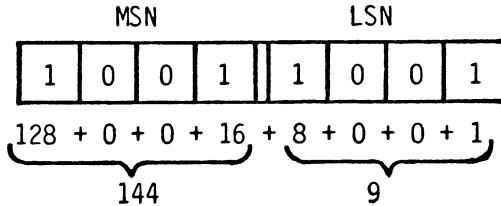
FIGURE 9.8

Just to make it possible to distinguish between the two nybbles, they are referred to as the "Most Significant Nybble" (MSN) i.e. the one on the left (Nybble 1 in Figure 9.8) and the other one as the Least Significant Nybble - the one on the right (Nybble 2 in Figure 9.8). These names derive from the fact that in an eight bit number the 1's on the left would represent much higher numerical values than the 1's on the right. The left-most 1, for instance, represents  $128_{10}$ , the humble right-most represents a mere one (a unit).

MSN				LSN					
1	1	1	1	1	1	1	1	=	$11111111_2$
$128+$	$64+$	$32+$	$16+$	$8+$	$4+$	$2+$	$1$	=	$255_{10}$

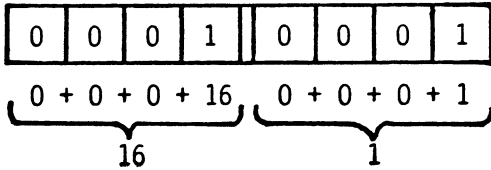
Thus, as far as the C-64 is concerned, two nybbles filled with 1's is really just one byte filled with 1's. That's an important difference for the programmer, as to load two nybbles of 15 into a memory location (s)he would have to load the byte with a 255. The question then is how to perform this operation, not only will we want to store 15's, but 1's and 6's and 13's etc. - indeed every number between 0 and 15.

Let's try an experiment, this time using 9's ( $1001_2$ ) instead of 15's, as the binary pattern of a 9 is more distinctive. The aim will be to store two nybbles, of  $9_{10}$  each, in a byte ie.:

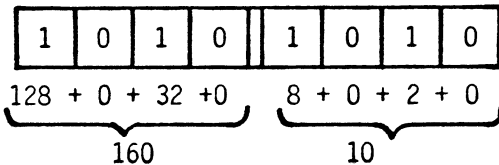


Thus when the LSN of 9 is moved up into the MSN position, its eight bit representation is 144. In fact as each bit is moved one place to the left, it increases its value two times. A four-bit move to the left, therefore, increases the value of each bit by  $2^4$  or 16. So, to recalculate the value of a nybble when it is moved from the LSN position to the MSN position it's only necessary to multiply its base 10 value by 16. Try this with a few examples:

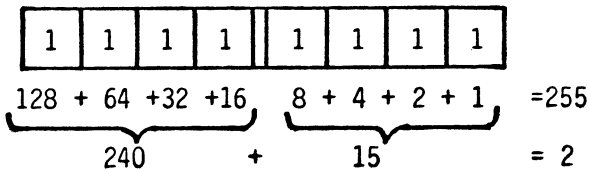
1 in LSN position = 16 in MSN



10 in LSN position = 160 in MSN



15 in LSN position = 240 in MSN





So far then, we can calculate what would be the value of a LSN when moved into the MSN position but now it's necessary to find a way of combining the two nybbles into 1 byte. Surprisingly enough it's sufficient simply to add together the two nybbles in base 10 to get the overall 8-bit number. If you're not convinced, just look at the examples above.

To carry out the storage process, therefore, the attack/decay values and the sustain/release values are combined into one byte. Taking the first case, where attack is stored in MSN of R+5 and decay in LSN of R+5, the stages are

- (i) convert LSN in MSN (attack):  $AT = AT * 16$
- (ii) combine LSN and MSN to form one byte;  
calling the sum AD (attack/decay)  
 $AD = AT + DE$
- (iii) POKE this value into R+5:  
POKE R+5, AD

For the sustain/release part, the process is identical (sustain is in MSB, the register used is R+6)

```
SU = SU * 16
SR = SU + RE
POKE R+6,SR
```

Only one bit is needed for the gate function, it is stored in bit 0 of register 4. This complicates life somewhat as the remainder of R+4 is used to store other things. Under normal circumstances, therefore, the advice would be to use a logical OR to turn the gate ON and a logical AND to turn the gate OFF but....

#### ALL THE INTERNAL SOUND CONTROL REGISTERS OF THE SID CHIP ARE WRITE ONLY

With this limitation in force, therefore, it is not possible to PEEK any of the registers of the SID chip. In order to find out what is stored in any particular register, it's necessary to hold a copy of anything that we POKE to a SID register in a variable which we set up for that purpose and interrogate this when necessary. The problem is discussed more thoroughly below when waveforms are considered as the other material stored in this register is the waveform data.

Returning to Buildasound ...

### Cycle through ADSR

With 15 possible settings for each of the ADSR parameters, at total of  $15*15*15*15= 50625$  possible different envelopes exist. While this provides an enormous variety of possible sounds, it does also lead to an embarrassment of riches - how can one choose from the great variety available. This 'cycle through ADSR' sets out to solve the problem by allowing the user to cycle through any one parameter. Once this is as desired, cycle through the next one until all four are set as required. Choice of parameter for cycling is from a menu, lines 24200 to 24290 and a 0 to 15 loop then cycles the value of that parameter. During any cycling process the values of the three other parameters are maintained at the default setting and displayed by means of lines 24400 to 24410.

PROGRAM 9.3(v) Cycle through ADSR

```
24200 PRINT"<CLR><3DCRSR>";:SYSH:PRINTT
      AB(5)"IN THIS SECTION,THREE PARAM
      ETERS
24210 PRINTTAB(5)"<DCRSR>ARE SET TO TH
      E LATEST DEFAULT":PRINTTAB(5)"<D
      CRSR>VALUE""
24220 PRINTTAB(5)"<DCRSR>THE OTHER IS C
      YCLED THROUGH 0-15
24230 PRINTTAB(5)"<3DCRSR>WHICH WOULD Y
      OU LIKE TO CYCLE?"
24240 PRINTTAB(14)"<3DCRSR>ATTACK.....A
      "
24250 PRINTTAB(14)"DECAY.....D"
24260 PRINTTAB(14)"SUSTAIN....S"
24270 PRINTTAB(14)"RELEASE....R"
24280 PRINTTAB(14)"EXIT.....X"
24290 GOSUB11500
24300 IFA$="X" THEN24000
24310 IFA$<>"A" ANDA$<>"D" ANDA$<>"S" ANDA
      $<>"R" THEN24290
24320 PRINT"<CLR><4DCRSR>":SYSH
24330 FORB3=0TO15
24340 IFA$="A" THENAT=B3
24350 IFA$="D" THENDE=B3
24360 IFA$="S" THENSU=B3
24370 IFA$="R" THENRE=B3
24380 IFB$="S" THENB$="":GOTO24100
24390 AD=AT*16+DE:SR=SU*16+RE
```

```

24400 PRINTTAB(9);"ATTACK= <3LCRSR>";
      AT;TAB(21);" DECAY= <3LCRSR>";
      DE
24410 PRINT"<2DCRSR>";TAB(8);"SUSTAIN=
      <3LCRSR>";SU;TAB(21);"RELEASE=
      <3LCRSR>";RE
24420 PRINT"<5DCRSR><5RCRSR>REPEAT PREV
      IOUS PARAMETER....R
24430 PRINT"<5RCRSR><DCRSR>JUMP BACK ON
      E STEP.....J"
24440 PRINT"<DCRSR><5RCRSR>STORE PREVIO
      US AS DEFAULT....S
24450 PRINTTAB(10);"<3DCRSR>PRESS SPACE
      TO CYCLE
24460 GOSUB11600:G=1:GOSUB30300
24470 IFB$="R"ANDB3=0THENB3=-1
24480 IFB$="R"ANDB3>0THENB3=B3-1
24490 IFB$="J"ANDB3<=1THENB3=-1
24500 IFB$="J"ANDB3>1THENB3=B3-2
24510 IFB$="S"THEN24340
24520 PRINT"<HOME><4DCRSR>":NEXTB3:GOTO
      24100

```

### Set envelope

With this option, the user is simply asked to define the ADSR parameters using straight forward INPUTS. Having stored these as variables, the POKE values then need to be calculated, as the attack and decay values need to be combined into a single POKE value, as do the sustain and release. Thus, the variables AT(attack) and DE(decay) are combined by moving AT from the LSN into the MSN by multiplying it by 16 and then adding the product to DE to produce the POKE value, AD, i.e.

$$AD = 16*AT+DE$$

Similarly for the Sustain (SU) and Release (RE) parameters:

$$SR = 16*SU+RE$$

This yields Program 9.3(w), the 'set envelope' part of the routine.

### PROGRAM 9.3(w)

```
24530 PRINT"<CLR>":SYSH:PRINT"<3DCRSR>"
      TAB(14);"PLEASE ENTER:"
24540 PRINTTAB(10)"<2DCRSR>ATTACK ";:E=
      2:GOSUB13200:AT=VAL(B$):G=1:GOSUB
      30300
24550 PRINTTAB(10)"<2DCRSR>DECAY ";:E=2
      :GOSUB13200:DE=VAL(B$)
24560 PRINTTAB(10)"<2DCRSR>SUSTAIN ";:E
      =2:GOSUB13200:SU=VAL(B$)
24570 PRINTTAB(10)"<2DCRSR>RELEASE ";:E
      =2:GOSUB13200:RE=VAL(B$)
24580 AD=AT*16+DE:SR=SU*16+RE
24590 GOTO24000
```

### Use pre-defined envelope

With this option, the user is presented with a MENU giving a choice of instrumental voices. On choosing one of these, it is simply necessary to set the ADSR appropriately. Program 9.3(x) shows this part of the routine, in addition, this routine cheats a little as it also adds in the appropriate waveform for the instrument.

### PROGRAM 9.3(x)

```
24600 PRINT"<CLR>":SYSH:PRINT"<2DCRSR>"
      ;TAB(10);"ENTER INSTRUMENT TYPE
24610 PRINTTAB(12)"<3DCRSR>VIOLIN.....V
      "
24620 PRINTTAB(12)"<DCRSR>DRUM.....D"
24630 PRINTTAB(12)"<DCRSR>PIANO.....P"
24640 PRINTTAB(12)"<DCRSR>ORGAN.....O"
24650 PRINTTAB(12)"<DCRSR>SYNTHETIC..S"

24660 GOSUB11500
24670 IFA$="V"THENAT=10:DE=8:SU=10:RE=9
      :WF=32
24680 IFA$="D"THENAT=0:DE=9:SU=0:RE=9:W
      F=128
24690 IFA$="P"THENAT=0:DE=9:SU=0:RE=0:W
      F=64:LP=255:HP=0
24700 IFA$="O"THENAT=0:DE=9:SU=0:RE=0:W
      F=16
24710 IFA$="S"THENAT=0:DE=9:SU=0:RE=0:W
      F=32
24720 GOTO24000
```

## Option 5.4: Waveform

Firstly, about waveforms and tone color.

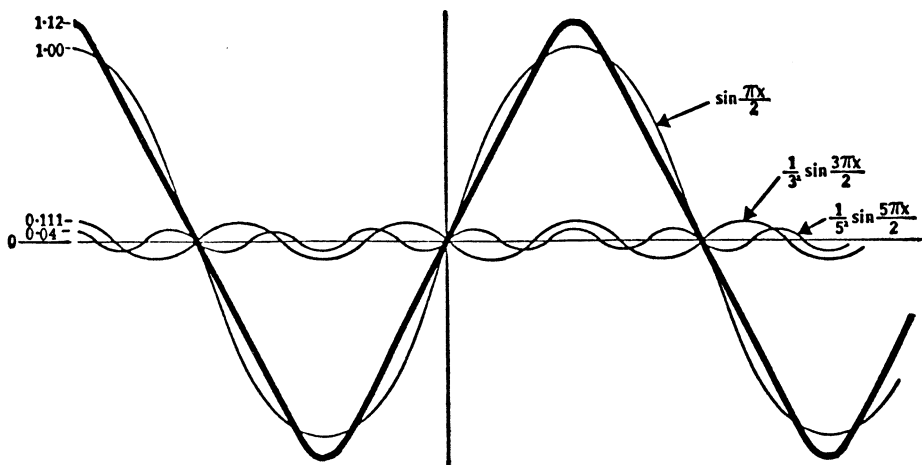
With a pure oscillating frequency, a clear 'pure' tone is produced which sounds like a clear human female/young male voice or a flute played without vibrato.

However, most musical instruments do not yield such a pure tone but have mixed in some, frequently many, harmonics which serve to create a complex wave-form. It is this complexity that allows one to listen to a steady note and identify it as being played by, say, an oboe or a viola. Thus, as well as there being present the fundamental or 1st harmonic, the 2nd, 3rd, 4th etc. will all be present as well although they will all be quieter than the fundamental. Generally, the harmonics get quieter the higher their harmonic number but this is by no means a uniform phenomenon. By means of four different waveforms, it is possible to simulate the major waveforms produced by modern musical instruments and many novel ones. These waveforms are:

### (i) Triangular waveform...

Some waves are particularly strong in the odd harmonics which means that, at every node or point where the fundamental has zero amplitude (i.e. cuts the axis) the harmonics too have zero amplitude. Because of this, the resultant wave form, which is the sum of the harmonics, rises very steeply from the axis.

Very much like the sine wave shape, the triangular wave form rises steadily to its peak and then reverses direction to fall equally steadily (Figure 9.8). Its form comes from a make-up of only odd harmonics and the fact that the 3rd harmonic is only  $1/9$  as strong (i.e. as loud) as the 1st, and the 5th,  $1/25$  as loud and so on.



A TRIANGULAR WAVEFORM  
FIGURE 9.8

(ii) Sawtooth waveform

As it contains all the harmonics, this is a non-symmetrical form. Its amplitude rises steadily and then decays almost instantaneously to its minimum value.

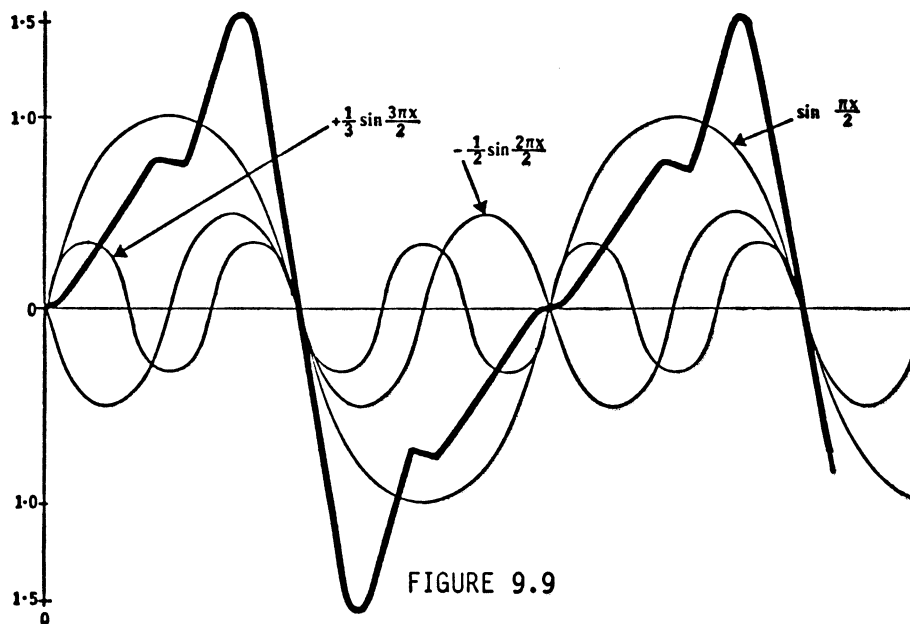


FIGURE 9.9

(iii) Pulse waveform...

This waveform is also strong in odd harmonics. However, in this case the odd harmonics do not fall off so rapidly as they do with the triangular wave form i.e. the third harmonic is  $1/3$  of the strength of the fundamental (compared with  $1/9$  for triangular), the fifth harmonic is  $1/5$  of the strength (compared with  $1/25$ ) etc. The total effect is that of a square wave rising almost vertically and falling as abruptly.

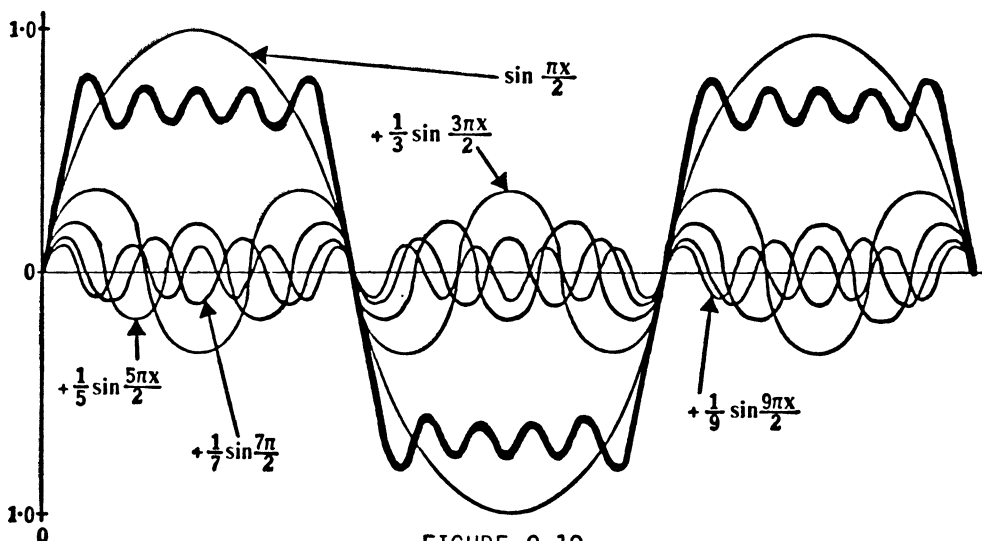


FIGURE 9.10

(iv) Random waveform (or noise)...

This waveform is quite simply (and deliberately) a mess and defies any analysis. Its main use is for sound effects: zaps, whooshes, clunks etc.

## Back to Buildasound

The first control section on waveforms offers the user the usual chance to switch on and listen to the various waveforms. This MENU leads on to the second MENU that offers a choice of five waveform options.

### PROGRAM 9.3(y)

```

25000 PRINTM$"4":SYSH
25010 PRINTS$"SELECT WAVEFORM<2DCRSR>
      <LCRSR>
25020 PRINTTAB(7)"LISTEN WAVEFORM...L<D
      CRSR>
25030 PRINTP$
25040 PRINTQ$:GOSUB11500
25050 IFA$="X"THENENG=0:GOSUB30500:GOTO28
      000
25060 IFA$="N"THEN26000
25070 PRINT"<CLR><2DCRSR>":SYSH:PRINTTA
      B(13)"ENTER WAVEFORM
25075 PRINT"<CLR><2DCRSR>":SYSH:PRINTTA
      B(13)"ENTER WAVEFORM
25080 PRINTTAB(7)"<2DCRSR>SAW TOOTH....
      .....S<DCRSR>"
25090 PRINTTAB(7)"TRIANGLE.....T<D
      CRSR>"
25100 PRINTTAB(7)"RANDOM.....R<D
      CRSR>"
25110 PRINTTAB(7)"PULSE(SQUARE).....P<D
      CRSR>"
25120 PRINTTAB(7)"PULSE(OTHER).....O<D
      CRSR>"
25130 PRINTQ$
25140 GOSUB11500

```

For voice 1, the bits of Register 4 are used to turn the wave forms on and off, a 1 in a particular bit meaning wave form ON and a zero meaning OFF. ie.

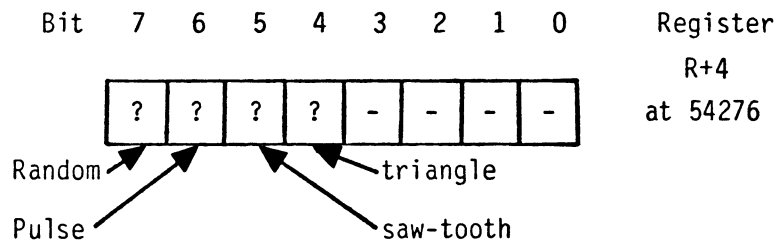


FIGURE 9.11



From Figure 9.11 it can be seen that a 1 in bit 7 of R+4 or a 128 (left-most bit) POKEd into 54276 will turn on the Random noise. Similarly, bit 6 controls the pulse wave form, bit 5 the saw-tooth and bit 4 the triangular waveform.

Thus, if Register 4 contains all zeros i.e. WF(wave form)= 0, simply adding 32 to WF i.e. WF=WF+32 will set the chips to output a saw tooth. However, once a waveform is set, i.e. one of bits 4 to 7 contains a 1, simply adding to the contents will not reset the register. For example if a 32 was already set and a 16 was then added, both bits 4 and 5 would be set. What is required is a two stage operation which first cleans the register completely and then sets it as required. The subroutine at 30500 takes care of this problem by storing the composite variable WF+G+RM in register 4 each time the register variables are modified.

Once the choice of waveform has been made, Register 4 can be set to tell SID which waveform to output on channel 1. The first stage in Program 9.3(z) in the process is to set WF to the requisite POKE value shown on Figure 9.11.

#### PROGRAM 9.3(z)

```
25150 IFA$="X"THEN25000
25160 IFA$="S"THENWF=32
25170 IFA$="T"THENWF=16
25180 IFA$="R"THENWF=128
25190 IFA$="P"ORAS="0"THEN WF=64:GOTO25
      210
25200 G=1:GOSUB30300:GOTO25140
```

For three of the waveforms, saw-tooth, triangle and random, the setting of WF is all that's required and so, once this is done the program can RETURN.

In the case of the pulse waveform, however, life is a little more complex as the waveform can vary infinitely from pulses almost 100% on to almost 100% off, ie:

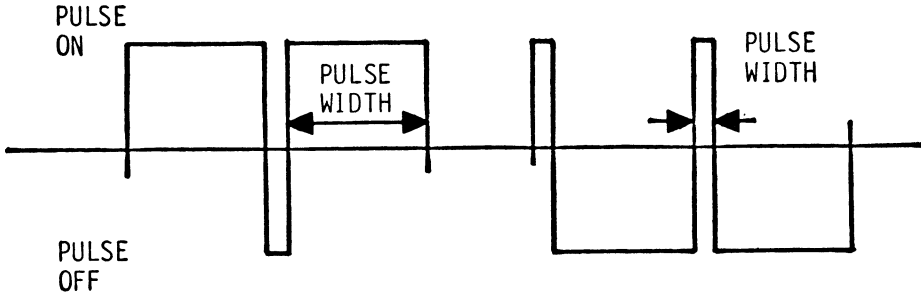


FIGURE 9.12

It's necessary, therefore, to tell SID what part of the time the pulse is to be on and what part off. This can be done to an accuracy of one part in 4095 so zero represents a zero pulse width and 4095 represents pulse on the whole time.

One particular waveform, a square wave, is obtainable, therefore, by setting the pulse length to 2048.

In order to store a number up to 4095, 12 bits of binary are needed, see Figure 9.13.

$$0 + 0 + 0 + 0 + 2048 + 1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 4095$$

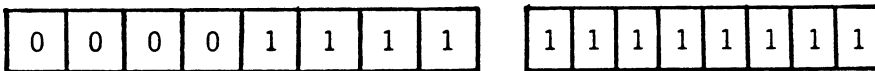


FIGURE 9.13

This means therefore, that one full byte and four extra bits are required to store numbers up to this size. Register 2 is used to store the low eight bits of the number and the four least significant register 3, the four upper bits.

So ... to set up a square wave, a 2048 needs to be stored ie:

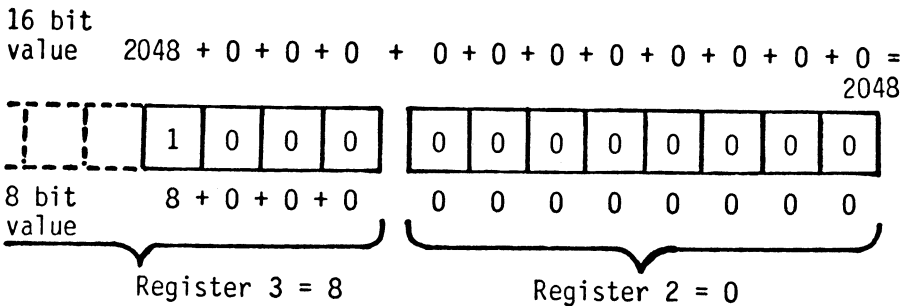


FIGURE 9.14

Thus, the statement to set up a square wave is:

```
POKE R+3,8:POKE R+2,0
```

Once a 'P' input is detected in A\$, the square state can be set up as in Program 9.3(aa):

PROGRAM 9.3(aa)

```
25210 IF A$="P" THEN LP=0:HP=8:PP=50:GO
SUB 30300:GOTO 25380
```

When an '0' is selected at MENU 4, it is first necessary to ascertain just what percentage of the wave is 'ON'. An input routine along with a simple prompt and error check collects this in Z1, as a percentage. As the range of pulse widths is from zero to 4096, the percentage needs to be converted ie:

```
PW=0.01*Z1*4095
```

The next stage is to break the number down into its High Pulse (HP) and Low Pulse (LP) parts. First the high bit. To isolate the bits of PW that are above 256, ie in the high byte, PW itself can be divided by 256. Any numbers in the high byte then become integers while those in the low byte are converted into fractions ie:

```
2048/256 = 8
1024/256 = 4
512/256 = 2
256/256 = 1
whereas....
128/256 = 0.5
64/256 = 0.25
32/256 = 0.125 etc ...
```

Thus, if the INT( ) of PW/256 is taken, the result will be the number to be POKEd into the high register. Once this is evaluated the low register value (LP) is calculated quite simply as:

$$LP = PW - HP * 256$$

Putting these parts together yields:

PROGRAM 9.3(bb)

```
25300 PRINT"<CLR>" ;:SYSH:PRINTR$"<3DCRS
R>DEFINE PULSE WIDTH<DCRSR>"
25310 PRINTR$"ENTER PERCENTAGE OF PULSE
":PRINTR$"WIDTH ON (0-100%)"
25320 PRINT"<DCRSR><7RCRSR>" ;:E=3:GOSUB
13200:PP=VAL(B$)
25330 IFPP<0ORPP>100THEN25160
25340 PW=0.01*PP*4095
25350 HP=INT(PW/256)
25360 LP=PW-HP*256
25370 G=1:GOSUB30300:IFPF=1THENPRINT"<H
OME>"TAB(26)"<3DCRSR> <4LCRSR>
"PP ; "%":GOTO25430
25380 PRINT"<CLR>" ;:SYSH:PRINTR$"<3DCRS
R>CURRENT PULSE WIDTH= <6LCR
SR>" ;PP ; "<LCRSR>%<DCRSR>"
25390 PRINTR$"DO YOU WISH TO:<DCRSR>"

25400 PRINTR$"<DCRSR>INCREASE PULSE WID
TH...I<DCRSR>"

25410 PRINTR$"DECREASE PULSE WIDTH...D<
DCRSR>"
25420 PRINTQ$"<UCRSR><LCRSR>.....X"

25430 GOSUB11500
25440 IFA$="X"THEN25000
25450 IFA$="I"THENPP=PP+1:IFPP>100THENP
P=100
25460 IFA$="D"THENPP=PP-1:IFPP<0THENPP=
0
25470 PF=1:PRINT"<10UCRSR>":GOTO25330
```

## Options 5.5: Filters

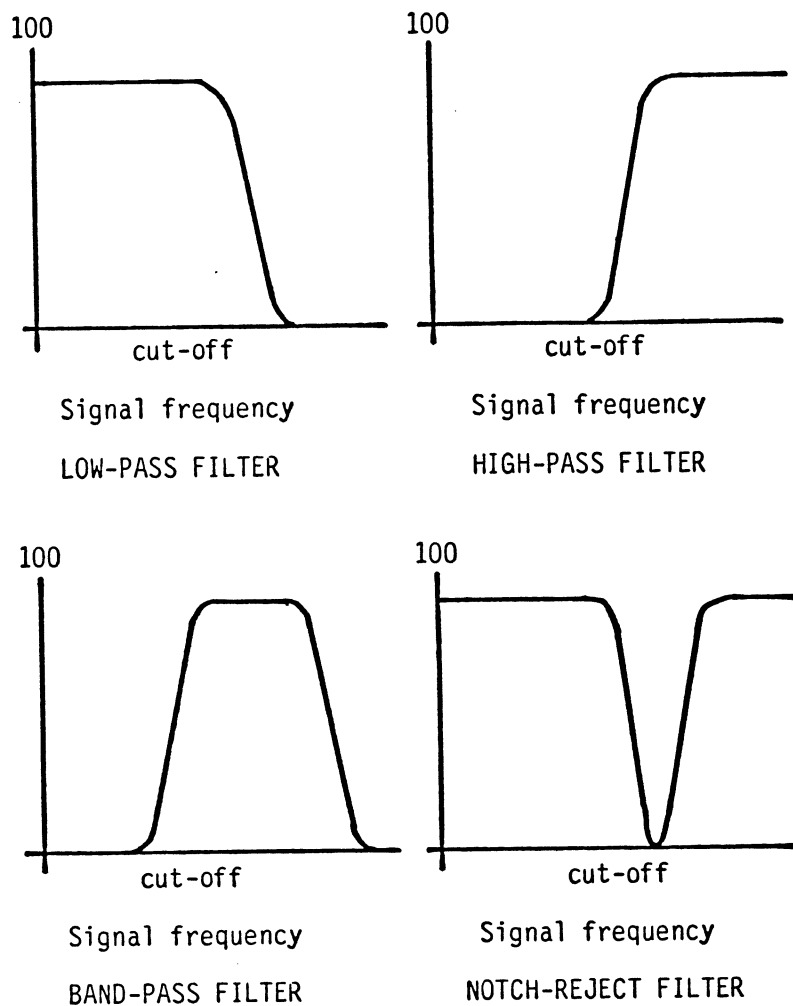
This option provides the user with the facility to set the SID chip filters as required. First things first though, let's look at the facilities provided on the SID chip.

### The filters

Any sound, other than a pure tone, is made up of a range of frequencies. When put through a filter, however, some of the frequencies are eliminated or simply made quieter (attenuated). Different types of filter act in different ways. Some filter cut or attenuate just the lower frequencies of the signal while others attenuate just the higher frequencies. Another type can allow a band of frequencies to pass while one other attenuates this band. On all these filters a frequency needs to be defined, above which, below which or around which the frequencies are attenuated. This is known as the 'cut-off' frequency. Summarising those available on the SID chip:

- Low-pass filter : passes frequencies BELOW the cut-off.
- High-pass filter : passes frequencies ABOVE the cut-off.
- Band-pass : passes a band of frequencies around the cut-off, attenuates all others.
- Notch-reject : attenuates a band of frequencies around the cut-off, passes all others.

The effect of these is displayed graphically in Figure 9.15.



AVAILABLE FILTER TYPES ON SID CHIP

FIGURE 9.15

In order to use the filters, the first stage is to set the filters to ON for voice one, or 'enable' them by a POKE+23,1. Following this it is necessary to select the filter type. Register 24 contains 3 bits which enable three filter types and allow the fourth type to be created. Figure 9.16 shows the structure of this byte.

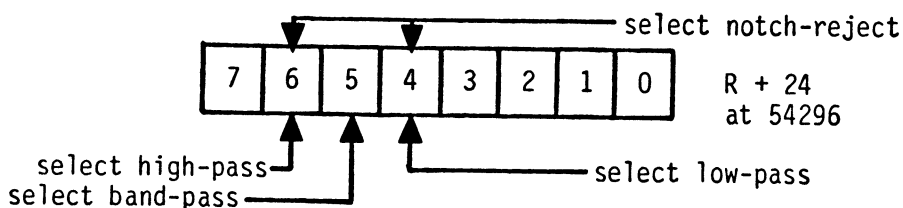


FIGURE 9.16

This routine, the first part of which is shown in Program 9.3(cc), therefore presents a MENU and once the choice is made, sets the appropriate bits of register 24. Having cleared the screen it then prints the chosen filter type on to the top of the screen.

PROGRAM 9.3(cc)

```

26000 PRINTM$"5":SYSH
26010 PRINTS$"LISTEN FILTERS<2DCRSR>":R
      EM POKE54295,1
26020 PRINTR$"SELECT FILTERS....S<DCRSR
      >"
26030 PRINTP$:PRINTQ$:GOSUB11500
26040 IFA$="N"THEN27000
26050 IFA$="X"THEN28000
26060 G=1:GOSUB30300
26070 PRINT"<CLR><2DCRSR>":SYSH:PRINTTA
      B(8)"SELECT FILTER TYPE":POKER+23
      ,1
26080 PRINTTAB(8)"-----"
26090 PRINTR$"<2DCRSR>LOW PASS.....
      .L
26100 PRINTR$"<DCRSR>HIGH PASS.....
      H
26110 PRINTR$"<DCRSR>BAND PASS.....
      B
26120 PRINTR$"<DCRSR>NOTCH REJECT.....
      N<DCRSR>
26130 PRINTQ$:GOSUB11500
26140 PRINT"<CLR><3DCRSR>"TAB(14);:SYSH

26150 IFA$="X"THEN26000
26160 IFA$="L"THENFT=16:PRINT"LOW PASS
26170 IFA$="H"THENFT=64:PRINT"HIGH PASS

26180 IFA$="B"THENFT=32:PRINT"BAND PASS

26190 IFA$="N"THENFT=80:PRINT"NOTCH REJ
      ECT

```

Once the filter type is defined, SID needs to be told what the relevant cut-off frequency is and, once this is decoded it needs to be stored in the appropriate place in memory. This frequency needs to have a large enough range to cover the whole of the SID chip's frequency spectrum and, hence, requires 11 bits of memory for storage. As with other values of this size, this needs to be broken down into an eight bit low-byte and three bit high-nybble. This can be done by dividing the value down as was done with the pulse-width calculation or, perhaps more elegantly, by means of logical operators. Stripping off the high part of the value can be achieved by ANDing it with 255. When this is done any 1's that exist in the first eight bits will survive but the others higher than bit 7 will be stripped off, i.e:

	1	1	1	0	1	0	1	1	0	1	1	1	0	1	1	0
ANDed with	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
gives	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0

Writing this into the program gives:

PROGRAM 9.3(dd)

```

26210 PRINT"<3DCRSR>"TAB(2)"ENTER FREQU
      ENCY FOR CUT-OFF<2DCRSR>
26220 PRINT"<4RCRSR>";:E=4:GOSUB13200:C
      O=VAL(B$)
26230 PRINTTAB(2)"<2DCRSR>CURRENT CUT-O
      FF FREQUENCY="          <9LCRSR>";C
      O;"HZ"
26240 CI=INT(CO*2047/12000):IFCI>2047TH
      ENCI=2047
26250 CO=INT(CI*12000/2047)
26260 LO=CIAND7
26270 HI=(CI-LO)/8:FV=1
26300 PRINTR$"<2DCRSR>DO YOU WISH TO:"
26310 PRINTR$"<DCRSR>INCREASE FREQ.....
      I
26320 PRINTR$"<DCRSR>DECREASE FREQ.....
      D<DCRSR>
26330 PRINTQ$:GOSUB11500
26350 IFA$="X"THEN26070
26360 IFA$="I"THENCO=CO+7+INT(0.01*CO)
26370 IFA$="D"THENCO=CO-7-INT(0.01*CO)
26380 PRINT"<13UCRSR>";:G=1:GOSUB30300:
      GOT026230

```



## Option 5.6: set resonance

The resonance of a sound-cavity system is simulated on the SID chip by means of a value (R) set to between 0 and 15. What, the SID chip is, in fact, doing when setting different resonance values, is to simulate the effect of the defined sound being emitted in rooms which differ in their resonance characteristics. Such features as size, number of reflective/non-reflective surfaces, position of the sound source, all affect the resonance of an enclosed cavity.

SID chip provides a range of resonance settings from zero to 15, so the routine that carries this out is none to complex, ie:

### PROGRAM 9.3(ee)

```
27000 PRINTM$"6":SYSH
27010 PRINTS$"SET RESONANCE<2DCRSR>"
27020 PRINTR$"SET RESONANCE.....S<DCRSR
>"
27030 PRINTP$:PRINTQ$:GOSUB11500
27040 IFA$="X"THENG=0:GOSUB30500:GOTO28
000
27050 IFA$="N"THEN28000
27060 IFFV<>0THEN27110
27070 PRINT"<CLR><4DCRSR>":SYSH:PRINTR$
;"NO FILTER SET:
27080 PRINTR$"<2DCRSR>RESONANCE FEATURE
27090 PRINTR$"<DCRSR>NONE OPERATIONAL
27100 PRINTR$"<3DCRSR>PRESS ANY KEY TO
CONTINUE":GOSUB11500:GOTO27000
27110 PRINT"<CLR><2DCRSR>":SYSH:PRINTTA
B(10)"SET RESONANCE(0-15)
27120 PRINT"<2DCRSR><7RCRSR>";:E=2:GOSU
B13200:RS=VAL(B$):RP=RS*16
27130 IFRS<ORRS>15THEN27020
27140 G=1:GOSUB30300
27150 PRINTTAB(5)"<3DCRSR>CURRENT RESON
ANCE SETTING= <6LCRSR>";RS
27160 PRINTR$"<3DCRSR>DO YOU WISH TO:"
27170 PRINTR$"<2DCRSR>INCREASE RESNCE..
.I
27180 PRINTR$"<DCRSR>DECREASE RESNCE...
D<DCRSR>
27190 PRINTQ$:GOSUB11500
27200 IFA$="X"THEN27000
27210 IFA$="I"THENRS=RS+1:IFRS>15THENRS
=15
27220 IFA$="D"THENRS=RS-1:IFRS<0THENRS=
0
27230 IFA$="I"ORA$="D"THENRP=RS*16:PRIN
T"<16UCRSR>";:GOTO27140
27240 IFA$<>"N"THEN27040
```

### Option 5.7: choose parameter

This option simply allows the user to step backwards and forwards between options. It consists mainly, therefore, of a MENU and its related decoding statements:

PROGRAM 9.3(ff)

```
28000 PRINT M$"7":SYSH
28010 PRINT"<2DCRSR><2RCRSR>WHICH PARAM
      ETER DO YOU WISH TO SET?<2DCRSR>"

28020 PRINTR$"PITCH.....P"
28030 PRINTR$"VOLUME.....V"
28040 PRINTR$"ENVELOPE....E"
28050 PRINTR$"WAVEFORM...W"
28060 PRINTR$"FILTERS.....F":PRINTR$"RE
      SONANCE...R"
28070 PRINTR$"DISPLAY.....D"
28075 PRINTR$"EXIT.....X":GOSUB11500

28080 IFA$="P"THEN22000
28090 IFA$="V"THEN23000
28100 IFA$="E"THEN24000
28110 IFA$="W"THEN25000
28120 IFA$="F"THEN26000
28130 IFA$="R"THEN27000
28150 IFA$="D"THEN31000
28160 IFA$="X"THENRETURN
28170 GOTO28080
```

## Option: Display default state

This option reports back on the current settings of the SID chip and provides a screen display.

PROGRAM 9.3(gg)

```
31000 PRINT"<CLR><DCRSR>":SYSH:PRINTTAB
      (8)"DISPLAY CURRENT DEFAULT
31010 PRINT"<DCRSR><4RCRSR>FREQUENCY ="
      ;FR
31020 PRINT"<DCRSR><4RCRSR>ENVELOPE:
31030 PRINT"<DCRSR><4RCRSR>ATTACK =" ;AT
      ;" DECAY =" ;DE
31040 PRINT"<DCRSR><4RCRSR>SUSTAIN =" ;S
      U;"RELEASE =" ;RE
31050 PRINT"<DCRSR><4RCRSR>WAVEFORM: "
      ;
31060 IFWF=16THENPRINT"TRIANGLE"
31070 IFWF=32THENPRINT"SAWTOOTH"
31080 IFWF=64THENPRINT"PULSE OF WIDTH "
      PP;"%"
31090 IFWF=128THENPRINT"RANDOM"
31100 IFRP=0THENPRINT"<DCRSR><4RCRSR>FI
      LTERS NOT SET:NO RESONANCE":GOTO3
      1170
31105 PRINT"<DCRSR><4RCRSR>FILTER:";
31110 IFFT=16THENPRINT"LOW PASS"
31120 IFFT=64THENPRINT"HIGH PASS"
31130 IFFT=32THENPRINT"BAND PASS"
31140 IFFT=80THENPRINT"NOTCH REJECT"
31150 PRINT"<DCRSR><4RCRSR>CUT-OFF FREQ
      UENCY ="CO"HZ"
31160 PRINT"<DCRSR><4RCRSR>RESONANCE="R
      S
31170 PRINT"<HOME><21DCRSR>"TAB(7);"PRE
      SS ANY KEY TO CONTINUE"
31180 GOSUB11500
```

## Menu 1; Option 6: Exit Compositune

This option provides the orderly way out of Compositune. Its major task is in resetting SID's registers so that the next program doesn't get mixed up with old noises. The routine is called by line 12240 and resides at 29000 onwards.

PROGRAM 9.3(hh)

```
12240 IFAS<>"X" THEN GOSUB 11500:GOTO 12070
12250 GOTO 29000
```

```
29000 PRINT"<CLR>":FOR CN=RTOR+25:POKER,
      0:NEXT:END
```

Postlude:

So that's Compositure. It's written for you to learn from as well as being a, hopefully, useful utility. You'll get the most from this by taking the bits that you need most and modifying them to what you want. It's all yours!

## CHAPTER

# 10

**I**n this chapter, some of the more advanced features of the C-64 are explored. These are 'advanced' in several ways, some require a greater knowledge of mathematical techniques than has previously been assumed while some involve the use of assembly language - the next stage up (or down!) from BASIC. Each of the four parts of Chapter 10 are self-contained so, if the maths is not to your liking, don't be put off, move on to Part 2.

### PART ONE: SUMS 'n' THINGS

#### Circles

In this, a small program is examined which will allow the user to program, display and draw a circle. This will use some of Honey.Aid's graphics commands, although all the calculations performed will use Commodore's built in mathematical functions. It will of course be necessary to explain these mathematical functions, and we shall do so as we go along. The first of these is:

SQR( )

The Commodore 64 like most computers has simple to use commands (functions) that perform mathematical calculations. For example, type in:

```
PRINT SQR(16)
```

You will get a reply of '4'. Thus the SQR command returns the square root of a number. Other built-in functions like RND() and INT() have already been discussed in earlier chapters.

## SIN( ) and COS( )

For a circle, the mathematical commands we need are our old friends from school, SIN and COS. These two useful functions will be used to the full when we draw a circle. Let's first see what they are and how they can help in this task.

Figure 10.1 shows a pair of axes with a segment one unit long fixed to the origin and at an angle 'A' from the 'X' axis:

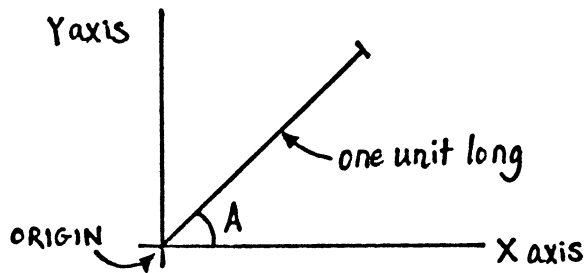


FIGURE 10.1

If this segment were to be rotated about the origin, the tip would trace out a circle:

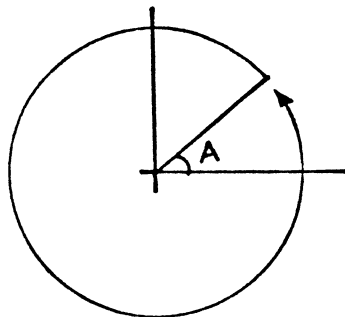


FIGURE 10.2

In order to use these ideas to draw a circle on the computer we will need to know the 'X' and 'Y' co-ordinates of the end point of the segment. This, as you may have guessed, is where SIN and COS come in. For a particular angle 'A' the 'X' and 'Y' co-ordinates are:

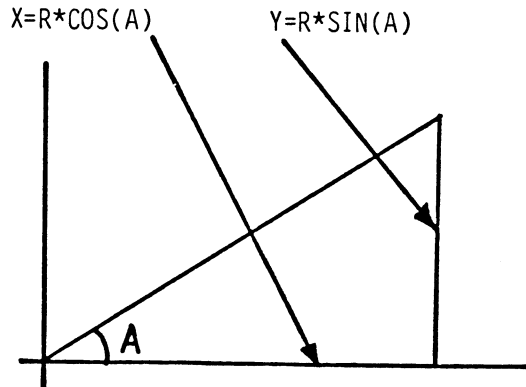


FIGURE 10.3

This value 'A' would be in radians, there being  $2\pi$  radians in 360 degrees (a circle). Pi (pronounced 'pie' as in 'Apple pie') is a number which is the ratio of the radius of a circle to its circumference - the circumference of a circle is  $2\pi R$  ( $R$ =radius). The actual value of pi is 3.14159... but you don't have to worry about it because there is a pi character on your keyboard. It is located on the same key as the up-arrow or 'exponentiation' sign next to <RESTORE>. Check this by typing:

PRINT  $\uparrow$

This demonstrates that the C-64 recognizes this symbol and treats it as a number - you don't have to type in the number 3.14159... itself, the computer works out the number for you. An angle in degrees can be changed to radians if the angle is multiplied by  $\pi/180$ .

If we let 'A' be all the angles in a circle, i.e. 0 to  $2\pi$  then SIN(A) and COS(A) give all the 'X' and 'Y' co-ordinates of the points around the circle. Well that's all very well for a circle with a radius of '1' but what about a circle with a radius of '100'? If the radius, or segment, is 100 times longer, then everything is 100 times bigger, so the co-ordinates of 'X' and 'Y' become:

$$X=100*\text{COS}(A) \quad Y=100*\text{SIN}(A)$$

Because the HIRES screen has two hundred pixels of space in the vertical direction the largest radius we could possibly draw is '100'. Let's use this highest value for our program.

To draw a circle we need to plot every point on the 'X' and 'Y' axis i.e. a FOR..NEXT..loop will be needed. It will loop from 0 to 360 degrees, or rather 0 to  $2\pi$  radians (a complete circle). (Make sure you have Honey.Aid loaded to use the HIRES command).

PROGRAM 10.1(a)

```
10 HIRES 0,1,1
20 R=100
30 FOR A=0 TO 360 STEP 0.5
40 X=R*COS(A*PI/180):Y=R*SIN(A*PI/180)
50 PLOT 150+X,100+Y,1
60 NEXT A
```

How's that? A wonderful circle created with SIN and COS. Now that we have plotted a circle using SIN and COS, let's look at another way of tackling the problem, using TAN.

TAN

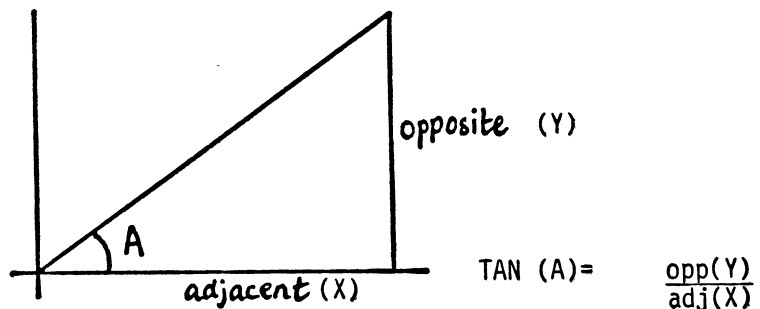


FIGURE 10.4

As demonstrated in Figure 10.4, we now have the values of three things: - 'R' is the radius, 100; 'A' is the value of our loop (0 to 360), and 'X' is  $R \cdot \cos(A \cdot \pi / 180)$ . So to find 'Y' using TAN instead of COS, the calculation becomes:

$$Y = X \cdot \tan(A \cdot \pi / 180)$$

So now change line 40 of Program 10.1(a) to that of Program 10.2(a) and then run it.

PROGRAM 10.2(a)

```
40 X=R*COS(A*PI/180):Y=X*TAN(A*PI/180)
```



Oh dear! we get an error: ' Division by zero error in 40'. Well that's okay. The computer is just telling us that it (and for that matter we) can't divide by zero. The cosine value is zero at 90 and 270 degrees; therefore the TAN is undefined at these values. All we have to do is to check for values of 90 and 270 and skip them. That way the computer doesn't have to divide by zero and it will plot a complete circle. So add line 35 to Program 10.2(b).

PROGRAM 10.2(b)

```
35 IF A=90 OR A=270 THEN 60
```

ATN

Out of interest you might like to know about a simple way of converting a TAN value back into radians. You can do this by using the ATN command, which takes the form:

ATN(X) where 'X' is a TANGent value

We will now take a look at some of the other functions on the Commodore 64. One of these functions is ABS:

ABS ( )

ABS(X) will give you what is called the 'absolute value' of X. In other words it makes X positive. There are three possible cases:

If X is positive, ABS(X) is positive - e.g. ABS(9)=9

If X is negative, ABS(X) is still positive - e.g. ABS(-7)=7

If X is zero, ABS(X) is zero

## SGN ( )

Another useful function is 'SGN'. This function tells you whether a numeric value is negative or positive:

```
SGN(-4)=-1
SGN(4)=+1
and SGN(0)=0
```

## LOG( ) AND EXP( )

The C-64 LOG( ) function finds the natural logarithm of a number - i.e. the log to base e, where  $e = 2.71828183\dots$

EXP( ) is the opposite function to LOG( ). Given the LOG( ) of a number, EXP( ) finds out what the number was. In other words, EXP( ) is the antilog function. What EXP(X) actually does is find  $e^X$ , that is,  $e^x$ , so you can check out the value of e for yourself by typing:

```
PRINT EXP(1)
```

To find the LOG to base 10 of a number, you divide the LOG to base e of the number by LOG(10). Similarly, to get a base 10 antilog of a number, you can multiply by LOG(10) before taking the EXP( ) of the number:

$Y = \text{LOG}(X)$  gives LOG to base e of X

$X = \text{EXP}(Y)$  gives ANTILOG to base e of Y i.e.  $e^Y$  or  $e^Y$

and

$Z = \text{LOG}(X)/\text{LOG}(10)$  gives LOG to base 10 of X

$X = \text{EXP}(Z*\text{LOG}(10))$  gives antiLOG to base 10 of Z i.e.  $10^Z$  or  $10^Z$

An analogous method can be used for LOG'S and EXP'S in any base.

LOG( ) works for numbers greater than (not equal to) zero, and the function EXP( ) works for numbers up to slightly over 88.

However, possibly the most useful functions of all will be those which you invent yourself, as explained below.

## DEF FN

Often, when programing you will find that you have to perform the same complicated calculation in more than one place in the program. It is, however, rather wasteful to type the calculation in each time. Also, if the calculation is to appear as part of some other statement, each time, e.g. in the middle of a bigger calculation, a GOSUB can be inconvenient as you have to go through the rigmarole of assigning the result of the GOSUB to a variable before you do the calculation. Also, in a complicated program, GOSUB's and GOTO's can become rather hard to follow. Therefore, we have FuNctions. Commands like ASC( ), CHR\$( ), EXP( ) etc. are functions built into the machine, but you can DEFine your own using DEF FN.

For example, supposing you want to solve some quadratic equations - e.g.:

$$AX^2+BX+C=0$$

Where A, B, and C are known constants (numbers). The general formula for solving a quadratic equation is:

$$X = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

We can put this into a program to find the real roots of a quadratic quite easily:

### PROGRAM 10.3

```
10 DEF FNQR(V)=(-B+V*SQR(B^2-4*A*C))/(2*A)
20 INPUT "A,B,C";A,B,C
30 IF 4*A*C>B^2 THEN PRINT "NO REAL ROOTS": END
40 V=1
50 PRINT "THE FIRST ROOT IS";FNQR(V)
60 V=-1
70 PRINT "THE SECOND ROOT IS";FNQR(V)
```

Line 10 is the function definition. Function definitions are positioned at the beginning of a program (or, at least, are defined prior to the first function call), and are of the form:

DEF FN<variable name>(<variable name>)=expression.

In the function definition on line 10, notice that the variable name V in brackets in the statement DEF FNQR(V) also appears in the formula on the right-hand side of the equals sign. When the function is called, the value of V at that time will be substituted into the 'V' where it appears on the right - i.e. that value will be used by the right-hand side V's.

A function is called just by using it as though it was a variable. Thus we could add lines 80 and 90 to the above program:

```
80 Z=FNQR(A)*16.31*EXP(7)
90 PRINT Z
```

Notice that in line 80 we have FNQR(A) rather than the FNQR(V) which we had on lines 50 and 70. What will the program do with this 'A' when it evaluates the function on line 10? Well, it will treat the V's in the function definition as though they were 'A's. That is, it will substitute the value of 'A' for V on the right-hand side of the equals sign. It can do this because the V in "DEF FNQR(V)" is a "dummy variable" - it doesn't really exist! The 'V' and 'A' in lines 50, 70 and 80, however, are real, and are known as "parameters". They are slotted into the DEF FNQR(V) statement whenever it is called, and are substituted for the dummy variable whenever it appears in the expression on the right-hand side of the definition.

Another feature of a function is that the dummy variable doesn't actually have to appear on the right-hand side of the equals sign at all! That's another reason for calling it a dummy!

Here is a function definition which you may find useful sometime: it generates normally distributed random numbers with a mean of MU and a standard deviation of SD (this function is very useful in the field of statistics):

```
10 DEF FNRN(X)=SQR(-2*LOG(RND(1))*COS(2*PI*RND(1))*SD+MU
20 INPUT "MEAN,STD.DEVIATION";MU,SD
30 INPUT "HOW MANY RANDOM NUMBERS DO YOU WANT"; N
40 FOR C=1 TO N
50 PRINT FNRN(X);
60 NEXT C
```

Two functions which can be useful are the C-64's functions:

VAL ( ) & STR\$( )

VAL ( ) is the opposite function to STR\$( ). VAL( ) turns a string into a number, whereas STR\$( ) turns a number into a string.

Thus

VAL("12.34")=12.34

but

VAL("H7") will not work, as "H" can not be converted directly to a number, even though "7" can be turned into the number 7. What will happen is that it will return the value 0, as the first character in the string is not one of 0-9. A couple more examples are in order:

VAL ("65.4JOHN17") = 65.4

VAL ("-3827") = -3827

also

STR\$(67.93) = "67.93"

- this is useful for storing numbers in string variables.

## PART TWO: IMPROVING THE HANGMAN GAME

### Sound

You can LOAD in the Hangman program and follow this section to improve it. Then, you can save the new Hangman program on another disk.

Any program is enlivened by sound and the hangman game is no exception. The question is what sound and where? As there are several things going on at once in the game, the sound will be kept fairly simple, two basic features being used: beeps at inputs and a chord at the end of the game. Naturally, a correct input should receive a more friendly beep than an incorrect one and an unsuccessful game is best rewarded with a 'raspberry'! Thus the four features provided are:

```
INPUT:   Successful - friendly beep
         Unsuccessful - less friendly beep
END:     Successful - rising chord
         Unsuccessful - falling chord
```

Whatever the situation, the first necessity is to set up the sound at the beginning of the program. All SID's registers need to be cleared and the waveforms set up, i.e.

### Initialisation

This is written as a subroutine located at 6000 and called once the initial screen has been set up:

#### PROGRAM 10.4

```
475 GOSUB 6000
.
.
.

6000 R=54272:WF=32:AD=0:SR=245
6010 FOR X=R TO R+24:POKE X,0:NEXT
6020 FOR V=1 TO 3:POKE R+S+((V-1)*7),
AD:POKE R+6+((V-1)*7),SR
6030 NEXT:RETURN
```

Next, a SID poke section needs to be set up to actually turn on the sound:

#### PROGRAM 10.5

```
6100 H1=INT(FI/256)
6110 L1=FI-256*H1
6120 POKE R,L1
6130 POKE R+1,H1
6140 POKE R+4,WF+G
6150 POKE R+24,(G*15)
6160 RETURN
```

As on composatune, it will only be necessary to enter this routine with the variables set appropriately in order to get SID to speak. However, on Hangman two variables will need setting, the note frequency (F1) and the gate value (G).

#### First: Successful Inputs

With successful inputs a 'reward' note will be given, the frequency being set to 8583(8910).

#### PROGRAM 10.6

```
2400 G=1:FI=8583:GOSUB6100:FOR D=1 TO 100:
NEXT: G=0: GOSUB6100
```

#### Then: Unsuccessful Inputs

Here the routine is very similar to that for successful ones but a somewhat lower note is used:

#### PROGRAM 10.7

```
4205 G=1:FI=2145:GOSUB 6100
4215 G=0:GOSUB 6100
```

A similar process is carried out when a letter is tried for a second time:

#### Next: Successful game

When the word has been guessed, a chord is played and, as this requires all three voices to be set a new SID POKE routine is provided i.e.:

#### PROGRAM 10.8

```
4334 G=1:FI=8583:GOSUB 6100:FOR Z=1 TO
200:NEXT:G=0:GOSUB 6100
4335 G=1:FI=10814:GOSUB 6100:FOR Z=1 TO
200:NEXT:G=0:GOSUB 6100
4336 G=1:FI=12860:GOSUB 6100:FOR Z=1 TO
100:NEXT:G=0:GOSUB 6100
4337 G=1:F(1)=8583:F(2)=10814:F(3)=12860
:GOSUB 6200
4338 FOR Z=1 TO 600:NEXT:G=0:GOSUB 6200
.
.
.
6200 FOR V=1 TO 3
6210 H(V)=INT(F(V)/256)
6220 L(V)=F(V)-256*H(V)
6230 POKE S+((V-1)*7),L(V)
6240 POKE S+1+((V-1)*7),H(V)
6250 NEXT V
6260 FOR V=1 TO 3
6270 POKE S+4+((V-1)*7),WF+G
6280 NEXT V
6290 POKE S+24,G*15
6300 RETURN
```

#### Finally: Unsuccessful game

When a word is guessed wrongly, a chord is played to commiserate:

#### PROGRAM 10.9

```
4920 G=1:F=6430:GOSUB 6100:FOR Z=1 TO
200:NEXT:G=0:GOSUB 6100
4930 G=1:F=5407:GOSUB 6100:FOR Z=1 TO
200:NEXT:G=0:GOSUB 6100
4940 G=1:F=4291:GOSUB 6100:FOR Z=1 TO
100:NEXT:G=0:GOSUB 6100
4950 G=1:F(1)=4291:F(2)=3608:F(3)=2864
:GOSUB 6200
4960 FOR Z=1 TO 1000:NEXT:G=0:GOSUB 62
00
```



### PART THREE : 6510 Machine Language

As you are probably aware, microprocessors speak only in numbers and, for mere mortals that's none too easy. However, because only numbers are used, the 6510 finds machine-language or machine-code programs much easier to execute and so it can run them much more quickly. To get over the problem of handling all the numbers, a further language has been devised called 'Assembly language'.

With this, the actual numbers are replaced by groups of letters which suggest the action that any particular command does. These groups of letters are known as MNEMONICS, pronounced 'Nemoniks'. One mnemonic, for instance is STA which stands for STORE the contents of the Accumulator (one part of the 6510 chip itself). Thus, STA 1024 means store the contents of the accumulator in memory location 1024 (i.e. on the screen). In machine-code 'STA' becomes 141 in decimal notation or 8D in hexadecimal. If you are unsure about these mathematical notations (and want to know!) have a read of Appendix One.

We have seen the command 'STA' which is used to store (or copy) the contents of the accumulator somewhere else but what about getting it there in the first place? This is done by means of the command LDAIM or Load the Accumulator Immediately. Thus: LDAIM 90 means load the accumulator with a '90'.

With just one more command or 'instruction' we could actually write a machine-code program. The instruction that tells the 6510 that the program's finished is RTS or ReTurn from Subroutine. Putting these into a short assembly language program that will print a Commodore zero (i.e. an "@" sign onto the screen) yields:

#### PROGRAM 10.10

In assembly language

```
LDAIM 0  Load Accumulator IMmediately (i.e. in Immediate
        mode) with a zero.
STA 1024  STore the contents of the Accumulator in 1024 (i.e.
        on the screen).
STA 55296 STore the contents of the Accumulator in 55296
        (i.e. set the color RAM).
RTS      ReTurn from Subroutine (i.e. tell the chip that the
        program is finished).
```

Changing this into both hexadecimal (Hex) and decimal gives:

Mnemonic	Data	Mnemonic hex/decimal	Data hex/decimal
LDAIM	0	A9/169	00/0
STA	1024	8D/141	00 04/0 4
STA	55296	8D/141	00 D8/0 216
RTS	none	-	-

(Notice that data items larger than 255, e.g. 1024, are stored as two separate numbers in the computer. Thus 1024 decimal is 400 in hexadecimal, which the computer stores as two numbers, 00 and 04, in that order, i.e. least significant number or byte first).

Writing this out gives:

Assembly language	Hexadecimal	Decimal
LDAIM 0	A9 00	169 0
STA 1024	8D 00 04	141 0 4
STA 55296	8D 00 D8	141 0 216
RTS	60	96

The next task is to store the program in memory and, as each number in the hexadecimal or decimal listing requires one byte, the overall program will require 9 bytes for storage. One convenient area for storing short (up to 200 byte long) programs is the cassette buffer, from memory location 828 onwards. Therefore, POKE the program into memory from 828 to 836, using the decimal form.

#### PROGRAM 10.11

```

10 FOR X=0 TO 9
20 READ A
30 POKE 828+X,A
40 NEXT A
50 DATA 169,0,141,0,4,141,0,216,96

```

When this is run, the machine-code program will have been stored in the cassette buffer. To get it to actually run from BASIC, the command used is:

**SYS**

This tells a program to jump to the memory location specified and execute the machine-code program stored there. Try it out in direct mode with:

SYS 828

This will execute the machine-code program and print an '@' sign in the top left-hand corner of the screen. Be careful that the 'RUN' command is not at the bottom of the screen when you press RETURN, otherwise the screen may scroll and you will lose the result of your labors.

A program such as this may appear trivial but it illustrates three of the 6510 chip's 'instructions' or commands and gives a taste of how machine-code works. When programming in 6510 language, one can ignore the 0's and 1's to some extent and make use of an 'assembler'. This is a program, written in BASIC or machine code (or a mixture) which allows the user to enter a program in assembly language and then changes it into machine-code and stores that in memory.

If you wish to explore the world of machine code more deeply, then see Commodore 64 Assembly Language Programming, published by Hayden Book Company. This comes complete with a "two-pass" assembler on tape or disk and takes you through the whole 6510 instruction set. In fact, the program we have examined above, is Program 1.1 from that book and is examined in much more detail in Chapter 1 of the book.

#### PART FOUR: A Screen Border: a machine-code utility.

In Compositune and several other programs on tape, a border was printed around the screen using a machine-code routine. This is provided on the disk, numbered from 60000 onwards. Two actions are needed to operate this:

- (i) Load the machine-code program into memory.
- (ii) Call the program with a SYS command.

The subroutine is loaded into memory by a simple GOSUB 60000. This causes the DATA to be READ and then POKEd into memory. This program firstly reads the top of memory by PEEKing 55 and 56 and then resets these pointers 120 bytes lower (line 60010). It then POKEs the machine-code routine into this newly-created space and RETURNS. As line 60010 sets the variable 'H' to the start of the machine-code routine, it can be called readily by the statement SYS H. (or SYSH).

As the routine moves the top-of-memory pointers down, it's a good idea, on leaving the program to reset these to where they were. Line 60199 does this and is activated by a GOSUB 60199. Once this is done, the pointer is reset to where it was originally.

#### PROGRAM 10.12

```
60000 REM BORDER ROUTINE
60010 H=4096*6:I=7*17
60015 READA$:IFA$<>"ZXC"THEN60015
60020 FORZ=HTOH+I:READA:POKEZ,A:NEXT:SYSH:RETURN
60030 DATAZXC,162,38,169,192,157,0,4,157,192,7,169,6,157,
0,216,157,192,219
60040 DATA202,208,237,162,240,169,221,157,0,4,157,240,4,
157,224,5,157,168
60050 DATA6,157,39,4,157,207,6,157,223
60060 DATA5,157,239,4,169,6,157,0,216,157,240,216,157,224,
217,157,168,218
60070 DATA157,39,216,157,207,218,157,223
60080 DATA217,157,239,216,138,56,233,40,240,4,170,24,144,
194,169,240,141
60090 DATA0,4,169,253,141,231,7,169,237,141,192,7,169,238,
141,39,4,169
60100 DATA6,141,0,216,141,231,219,141,192,219,141,39,216,96
```

## CHAPTER

# 11

### SOLUTIONS TO EXERCISES

#### CHAPTER ONE

##### EXERCISE 1.1

Move the cursor to the W of 'WHAT'

Type in 'PLEASE'

Space over the 'S' with the space bar

Insert seven spaces

Type in 'TYPE IN'

Move cursor on to the N in 'NAME'

Insert five spaces

Type in 'FULL'

Press 'RETURN'

You have now finished. Line 10 should look like this:

```
10 INPUT "PLEASE TYPE IN YOUR FULL NAME";  
A$
```

##### EXERCISE 1.2

Move the cursor to the A of 'A\$'

Insert 16 spaces

Type in "YOUR NAME IS ";

Press "RETURN"

And now line 30 looks like this:

```
30 PRINT "YOUR NAME IS ";A$
```

### EXERCISE 1.3

```
10 INPUT "WHAT IS YOUR FULL NAME AND AGE"; A$, A
30 PRINT "YOUR NAME IS ";A$
40 PRINT "YOUR AGE IS";A
```

## CHAPTER TWO

### EXERCISE 2.1

```
10 C=1
30 RV=INT(RND(0)*100)+1
35 PRINT RV
60 C=C+1
70 IF C<101 THEN 30
90 STOP
```

### EXERCISE 2.2

```
10 FOR X=0 TO 100 STEP 3
```

### EXERCISE 2.3

```
1 FOR X=10 TO 0 STEP-1
2 PRINT X
3 NEXT X
4 PRINT "FIRE!!!"
5 STOP
```

### EXERCISE 2.4

```
2 C=0
4 C=C+1
6 PRINT C
8 IF C>19 THEN 10
9 GOTO 4
10 PRINT "FINISHED"
12 END
```

## EXERCISE 2.5

```
2 C=1
4 PRINT C
6 C=C+1
8 IF C>=31 THEN 10
9 GOTO 4
10 PRINT "FINISHED"
12 END
```

## EXERCISE 2.6

First we need to change line 60 so that it goes to line 105 instead of line 110. 60 IF G=RV THEN PRINT "WELL DONE - GUESS CORRECT." :GOTO 105  
105 PRINT "YOU TOOK";C; "GOES".

## CHAPTER 3

### EXERCISE 3.1

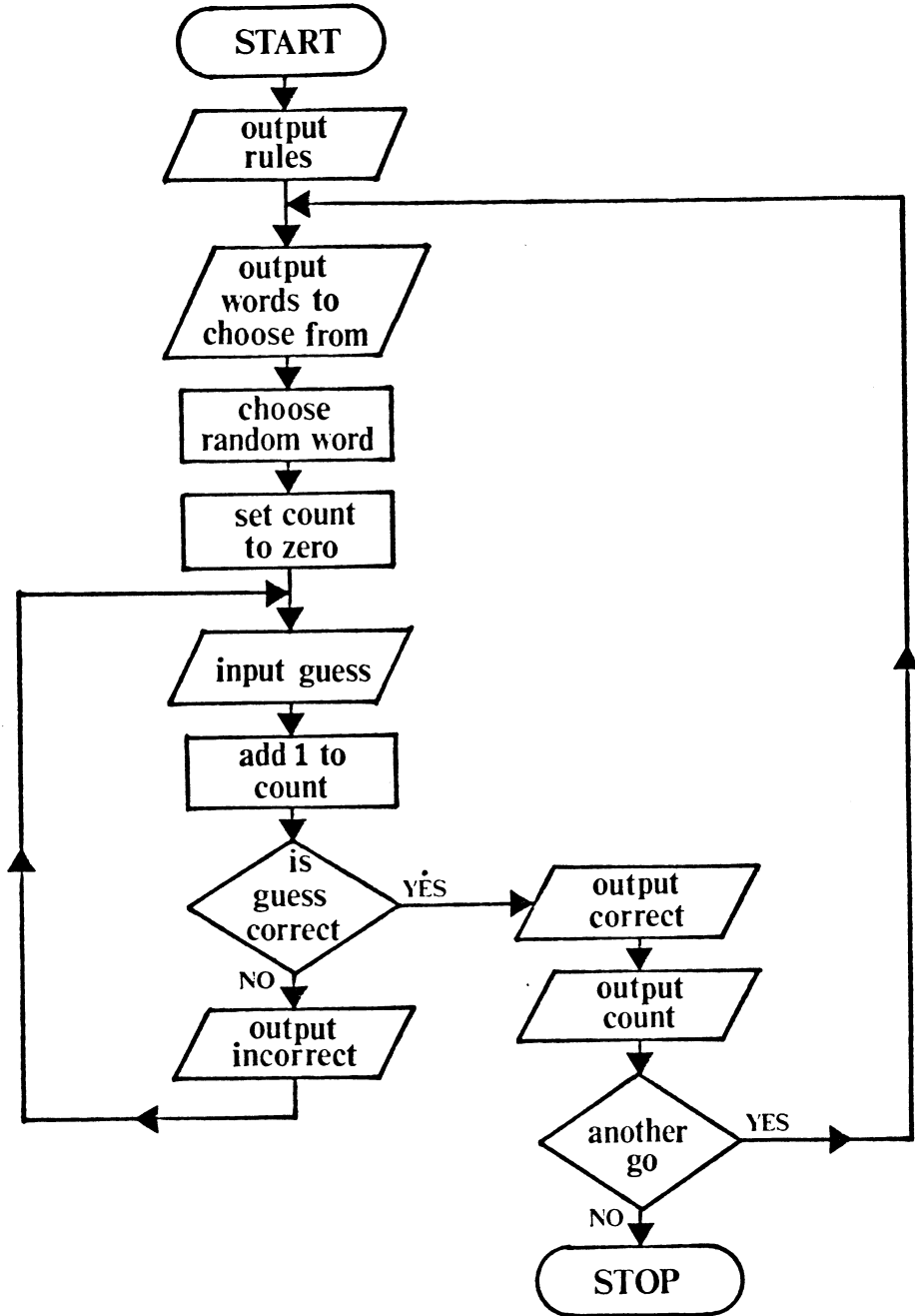
```
1 PRINT "<CLR><2DCRSR><14RCRSR>
ETCHA SKETCHA"
2 PRINT "<2DCRSR><2RCRSR>INSTRUCTIONS"
3 PRINT "<1DCRSR><4RCRSR>PRESS '4' TO MOVE LEFT"
4 PRINT "<4RCRSR>PRESS '5' TO MOVE DOWN"
5 PRINT "<4RCRSR>PRESS '6' TO MOVE UP"
6 PRINT "<4RCRSR>PRESS '7' TO MOVE RIGHT"
7 PRINT "<2DCRSR><10RCRSR>PRESS ANY KEY TO BEGIN"
8 GET A$: IF A$="" THEN 8
```

## CHAPTER 4

### EXERCISE 4.1

```
10 FOR X=1 TO 4
20 READ A,A$
30 PRINT A,A$
40 NEXT
50 DATA 1,ONE,2,TWO
60 DATA 3,THREE,4,FOUR
```

EXERCISE 4.2





```

10 REM WORD GUESSER
20 PRINT "<CLR><2DCRSR><4RCSR>
THESE ARE THE RULES"
30 PRINT "<DCRSR><4RCRSR>THERE ARE 10 WORDS TO GUESS"
40 PRINT "<4RCRSR>AND I WILL GUESS ONE OF THEM"
60 PRINT "<4RCRSR>YOU MUST GUESS WHICH ONE I HAVE GUESSED"
70 PRINT "<4RCRSR>THE WORDS TO GUESS ARE:"
80 RESTORE
90 FOR X=1 TO 10
100 READ A$:PRINT A$,
110 NEXT X
120 R=1+INT(RND(1)*10)
130 RESTORE
140 FOR X=1 TO R
150 READ A$
160 NEXT X
170 PRINT "I HAVE CHOSEN MY WORD"
180 PRINT "NOW YOU MUST GUESS IT"
190 C=0
200 INPUT "GUESS="; Z$
210 IF Z$=A$ THEN PRINT "CORRECT": GOTO 250
220 C=C+1: PRINT "WRONG"
230 GOTO 200
250 PRINT "THAT TOOK YOU";C;"GOES"
260 INPUT "DO YOU WANT ANOTHER GO";A$
270 IF A$= "Y" THEN 10
280 PRINT "BYE!"
290 END
300 DATA "COMMODORE","FRED","MUSIC","COMPUTER"
310 DATA "FLOWER","BASIC","LOVE","TRAIN"
320 DATA "MAGIC","DREAM"

```

## CHAPTER FIVE

### EXERCISE 5.1

The extra lines needed to give a continuous display of time are listed here:

```

250 FOR P=1 TO 575:NEXT P
260 GOTO 200

```

The pause on line 250 is added so the display changes approximately once per second instead of the display flashing continually.

## CHAPTER SEVEN

EXERCISE 7.1: The logical operation used to find the most significant BYTE is AND, i.e. AND with 240.

### EXERCISE 7.2

```
10 FOR X=0 TO 39
20 FOR Y=0 TO 15
30 POKE 1024+X+Y*40,83
40 POKE 55296+X+Y*40,Y
50 NEXT Y
60 NEXT X
```

### EXERCISE 7.3

```
10 PRINT "<CLR>"
20 FOR Y=0 TO 4
30 FOR X=0 TO 39
40 POKE 1024+X+Y*40,160
50 POKE 55296+X+Y*40,Y+3
60 NEXT X,Y
```

### EXERCISE 7.4

```
10 PRINT "<CLR>"
20 FOR X=0 TO 39
30 POKE 1984+X,81
40 POKE 56256+X,2
45 POKE 1984+X-1,32
50 NEXT X
```

### EXERCISE 7.5

```
10 PRINT "<CLR>"
20 FOR X=20 TO 0 STEP-1
30 POKE 1024+X+X*40,81
40 POKE 55296+X+X*40,2
50 POKE 1024+X+1+(X+1)*40,32
60 NEXT X
```

### EXERCISE 7.6

In order to give the ball a random start position we need only to change the values of 'X' and 'Y' in line 30

```
30 X=INT(RND(1)*38)+1:Y=INT(RND(1)*23)+1
```

### EXERCISE 7.7

```
21 FOR Y=0 TO 23
22 POKE 1033+Y*40,160: POKE 55305+Y*40,0
23 NEXT Y
116 IF A=0 THEN XI=XI*-1:GOTO 100
```

### EXERCISE 7.8

```
10 GET A$: IF A$="" THEN 10
20 GET B$: IF B$="" THEN 20
30 IF A$="O" AND B$="K" THEN PRINT
"CORRECT": STOP
40 PRINT "INCORRECT":STOP
```

### EXERCISE 7.9

To add an extra character to the bat size we need to add another two POKE commands and change lines 40 and 50.

```
40 IF A$="9" AND BC<37 THEN BC=BC+3:GOTO 70
50 IF A$="1" AND BC>0 THEN BC=BC-3:GOTO 70
85 POKE 1984+BC+2,160:POKE 56256+BC+2,0
75 POKE 1984+CB+2,32
```

### EXERCISE 7.10

```
10 REM DRAW THE WALL
20 FOR X=0 TO 39
30 FOR Y=2 TO 7
40 POKE 1024+X+Y*40,160
50 POKE 55296+X+Y*40,Y
60 NEXT Y,X
70 REM DESTROY THE WALL
80 FOR X=0 TO 39
90 FOR Y=2 TO 7
100 POKE 1024+X+Y*40,32
110 NEXT Y,X
```

### EXERCISE 7.11

To get the ball to start in a random position we can set 'X' and 'Y' to random values.

```
110 X=INT(RND(1)*39)+1:Y=INT(RND(1)*10)+12
```

By adding twelve to 'Y' we ensure the ball will start below the wall

## CHAPTER 8

### EXERCISE 8.1

0,24,0  
0,102,0  
1,129,128  
6,24,96

## APPENDIX ONE

### EXERCISE A1.1

- i)  $0000\ 0011_2 = 0+0+0+0+0+0+2+1$   
 $= 3_{10}$
- ii)  $0000\ 0100_2 = 0+0+0+0+0+4+0+0$   
 $= 4_{10}$
- iii)  $1000\ 0000_2 = 128+0+0+0+0+0+0+0$   
 $= 128_{10}$
- iv)  $1000\ 0011_2 = 128+0+0+0+0+0+2+1$   
 $= 131_{10}$
- v)  $1011\ 0111_2 = 128+0+32+16+0+4+2+1$   
 $= 183_{10}$
- vi)  $0111\ 0011_2 = 0+64+32+16+0+0+2+1$   
 $= 115_{10}$

### EXERCISE A1.2

- i)  $0009_{16} = 0 \times 409 + 0 \times 256 + 0 \times 16 + 9 \times 1$   
 $= 0 + 0 + 0 + 9$   
 $= 9_{10}$
- ii)  $0013_{16} = 0 \times 4096 + 0 \times 256 + 1 \times 16 + 3 \times 1$   
 $= 0 + 0 + 16 + 3$   
 $= 19_{10}$
- iii)  $00A5_{16} = 0 + 0 + 10 \times 16 + 5 \times 1$   
 $= 160 + 5$   
 $= 165_{10}$
- iv)  $0AAE_{16} = 0 + 10 \times 256 + 10 \times 16 + 14 \times 1$   
 $= 2560 + 160 + 14$   
 $= 2734_{10}$

$$\begin{aligned}
 \text{v)} \quad & 000E_{16} = 0+0+0+14 \\
 & = 14_{10} \\
 \text{vi)} \quad & 011A_{16} = 0+256+16+10 \\
 & = 282_{10} \\
 \text{vii)} \quad & 00EA_{16} = 0+0+14 \times 16 + 10 \\
 & = 224 + 10 \\
 & = 234_{10} \\
 \text{viii)} \quad & FOA3_{16} = 15 \times 4096 + 0 + 10 \times 16 + 3 \\
 & = 61440 + 160 + 3 \\
 & = 61603_{10}
 \end{aligned}$$

#### EXERCISE A1.3

$$\begin{aligned}
 \text{i)} \quad & 4_{10} = 0100_2(\text{BCD}) \\
 \text{ii)} \quad & 10_{10} = 1 \times 10 + 0 \\
 \text{iii)} \quad & 77_{10} = 7 \times 10 + 7 \\
 & = 0111 \ 0111_2(\text{BCD}) \\
 \text{iv)} \quad & 97_{10} = 9 \times 10 + 7 \\
 & = 1001 \ 0111_2(\text{BCD}) \\
 \text{v)} \quad & 53_{10} = 5 \times 10 + 3 \\
 & = 0101 \ 0011_2(\text{BCD}) \\
 \text{vi)} \quad & 102_{10} = 1 \times 100 + 0 \times 10 + 2 \times 1 \\
 & = 0001 \ 0000 \ 0010_2(\text{BCD}) \\
 \text{vii)} \quad & 953_{10} = 9 \times 100 + 5 \times 10 + 3 \times 1 \\
 & = 1001 \ 0101 \ 0011_2(\text{BCD}) \\
 \text{viii)} \quad & 2579_{10} = 2 \times 1000 + 5 \times 100 + 7 \times 10 + 9 \times 1 \\
 & = 0010 \ 0101 \ 0111 \ 1001_2(\text{BCD})
 \end{aligned}$$

#### EXERCISE A1.4

$$\begin{aligned}
 \text{i)} \quad & 0000 \ 0001_2(\text{BCD}) = 0 \times 10 + 1 \times 1 \\
 & = 1_{10} \\
 \text{ii)} \quad & 0000 \ 1001_2(\text{BCD}) = 0 \times 10 + 9 \times 1 \\
 & = 9_{10} \\
 \text{iii)} \quad & 0001 \ 0101_2(\text{BCD}) = 1 \times 10 + 5 \times 1 \\
 & = 15_{10}
 \end{aligned}$$

iv)  $0010\ 0000_2(\text{BCD}) = 2 \times 10 + 0 \times 1$   
 $= 20_{10}$

v)  $0100\ 1001_2(\text{BCD}) = 4 \times 10 + 9 \times 1$   
 $= 49_{10}$

vi)  $1010\ 0011_2(\text{BCD})$

\*\*\* This is not a valid BCD number as the first nybble,  $1010_2 = 10_{10}$ , i.e. is greater than allowed in BCD.

vii)  $1001\ 0111_2(\text{BCD}) = 0 \times 10 + 7 \times 1$   
 $= 97_{10}$

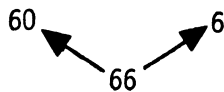
viii)  $1000\ 1000_2(\text{BCD}) = 8 \times 10 + 8 \times 1$   
 $= 88_{10}$

## APPENDIX

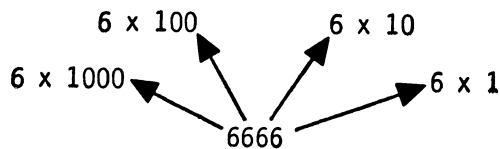
# 1

### BINARY, BINARY-CODED DECIMAL AND HEXADECIMAL NOTATION

Counting systems in general use throughout the world use the decimal system and this has been developed to count up to and beyond 10 and also below the value 1. In this standard the digits to the left of a number are of greater value than those to the right. For instance, in the number 66, the first 6 has a value 10 times the second, i.e.

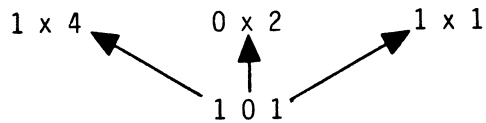


This is extended in larger numbers where digits to the left are successively greater by a multiple of ten, i.e.



A system where the position or place of a digit in a number affects its value is known as a PLACE-VALUE numbering system. In the decimal system, the values of digits increase in multiples of 10 and this is known as the BASE for that system. Other systems use different bases but follow the same pattern as the decimal system, i.e. the place to the left is greater by being multiplied by the base.

The computer, being basically electronic in operation, works better if it is told to only recognize two states, on or off or '0' and '1', and thus uses the Binary system - base 2. Thus, any number in binary consists simply of 0's and 1's, or electronically, zero volts (off) and some volts (on). To count past one, the binary system must resort to place-value notation and, as with other cases, the multiplying factor is the base, i.e. 2. Thus, the number 101 in base 2 or binary represents:



i.e.  $4+0+1=5$ . Clearly the plethora of bases presents a problem when representing numbers as in base 10, '101' represents one hundred and one while in binary (base 2) '101' represents 5. To overcome this ambiguity, a convention exists when representing numbers in that the base is written to the right of the number, just below the line. Thus, the two numbers discussed above become:

$101_{10}$  = One hundred and one in base ten.

$101_2$  = Five in base two.

The present-day generation of personal computers (1983-style) use eight bit registers or memories and can thus represent numbers up to  $11111111_2$  i.e. in base 10:

128	+64	+32	+16	+8	+4	+2	+1	=255	$_{10}$
1	1	1	1	1	1	1	1	Digit Equivalent in base 10	
128	64	32	16	8	4	2	1		

Fig. A1.1





From this point, the program will simply count, every time that you press the space bar, both the decimal and the binary boxes will index one. Try pressing the space bar once and the boxes should contain a 2<sub>10</sub> and a 10<sub>2</sub>. If you carry on indexing then you will see how binary counts. When you get to the stage where the decimal shows 15<sub>10</sub> the binary should read 1111<sub>2</sub>. Now index one further and the binary will change to 10000<sub>2</sub>. One way of looking at this is to lay out the addition:-

$$\begin{array}{r} 1111 \quad A \\ + \quad 1 \quad B \\ \hline \end{array}$$

On adding the 1 (A) to the 1 (B) this gives '2' i.e. 0, carry 1. This carry then produces another '0' plus another carry, and so on.

If you continued to press the space bar long enough, then eventually the binary register would become full. However, this would take an awful lot of pressing, so we will take a short cut to this state of affairs. Instead of pushing the space bar, press the RETURN key instead. This will return you to the menu where you can select the 'H' option again. This time, when asked "at what number do you want to start?" type a fairly high value which is less than 255, say 240. Off you go again until the binary register is full i.e. 11111111<sub>2</sub>. The addition of a further one, now, will clock all the binary register back to zeros and 256 will be lost. However, with the 6510, all is not lost as the 6510 has a carry flag that stores the fact that a carry has occurred. Clocking past 255<sub>10</sub> with the Binary/Hex tutor will show this happening. This is a handy feature of the 6510 but it must not be relied on as more than a temporary store of the carry. The carry flag is just as easily reset as it is set to 1!

In order to make sure that you really understand the binary notation, you may wish to try some of the exercises which are provided by the BIN/HEX exercises. Select 'E' at the main menu. This will provide you with a menu of exercises and you can select '2' to try the exercises converting decimal numbers into binary or '5' to try converting them back again. When you are running the exercises, by the way, typing a space (instead of a digit) will delete the last entry that you made, thus providing you with a correction facility. When you are satisfied that you have done enough, pressing the <return> key will take you back to the main menu.

While the 0's and 1's are convenient for the computer, they are much less so for the mere human so a compromise is sought. Decimal notation is of little use as, apart from  $1_2$  and  $1_{10}$  there is no other correspondence. A further idea would be to take the whole eight binary bits as a digit (i.e. up to  $255_{10}$ ) and use a base of 256! What would you see as the objection to this? That's apart from the idea itself being a bit mind-bending! Time to think ... The answer comes from an examination of the base 10 case in which ten digits (0 to 9) are needed to represent the ten steps up to 10. In the base 2 system, two digits are needed so base 256 would need 256 digits!

A compromise system adopted splits the eight bits up into two parts and represents these separately. Thus, the largest number to be represented is  $1111_2$  or  $15_{10}$  and this requires, along with the 0, sixteen different symbols. The ones adopted for this job are:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Decimal number
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Symbol

Fig. A1.2

Using this notation, any eight bit number can be represented by two symbols, one for the most significant four bits and one for the least significant four bits. To avoid the rather long description of these two halves of a byte, they are given the term NYBBLES. Thus a byte consists of two nybbles, a most significant nybble (MSN) and a least significant nybble (LSN) - see Fig. A1.3.

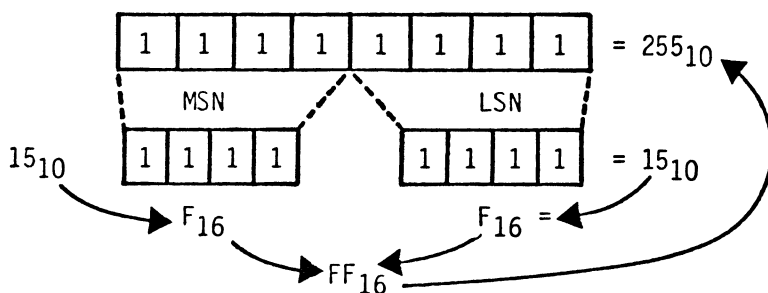


Fig. A1.3

The system described, which uses sixteen symbols is, of course, given the name HEXADECIMAL - usually abbreviated to HEX. Its major advantage, as far as computers are concerned, is that it is compatible with binary. Any eight bit binary number can be represented by two hexadecimal characters.

You are now in a position to look at the Binary/Hex tutor program again. The third row of boxes, which we ignored last time round, contains the Hex numbers. While the counting is going on in the binary boxes, so it is in the Hex boxes also. The comparability between binary and HEX shows wherever a major carry occurs - take for instance  $1111_2$ ,  $15_{10}$  or  $F_{16}$ . One index past this clocks the binary ones to zeros and adds a one to the left, i.e. to  $10000_2$  or  $10_{16}$ . These major points of correspondence occur at

$$\begin{aligned}
 1_2 &= 1_{16} = 1_{10} \\
 0001\ 0000_2 &= 10_{16} = 16_{10} \\
 0000\ 0001\ 0000\ 0000_2 &= 100_{16} = 256_{10} \\
 0001\ 0000\ 0000\ 0000_2 &= 1000_{16} = 4096_{10}
 \end{aligned}$$

Up to 9, the hex characters coincide with the decimal ones and between 10 and 15 the single letters correspond to the decimal numbers. After 15, Hex to decimal conversion becomes a little more tricky, as the use of two numbers together, e.g.  $FF_{16} = 255$ , once again calls for place-value notation. This time, as the base is 16 the ratio between any place and its neighbor is 16.

The values, in base 10 of the places in hexadecimal are:

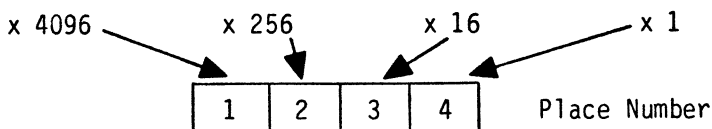


Fig. A1.4

Using Fig. A1.4 the way that  $E92F_{16}$  makes up  $59695_{10}$  is explained below in Figure A1.5.

$$E(14)x4096 + 9x256 + 2x16 + F(15)x1 = 59695$$

Fig. A1.5

Now that hex is totally mastered(!) try the following; the first two are explained fully on page 11-8.

EXERCISE A1.2

Calculate the value in decimal of the following:-

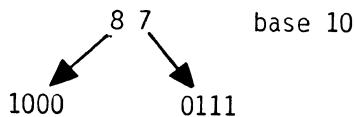
- |                  |                   |
|------------------|-------------------|
| i) $0009_{16}$   | v) $000E_{16}$    |
| ii) $0013_{16}$  | vi) $011A_{16}$   |
| iii) $00A5_{16}$ | vii) $00EA_{16}$  |
| iv) $0AAE_{16}$  | viii) $FOA3_{16}$ |

Answers on page 11-8.

BINARY CODED DECIMAL

As well as decimal, binary and hexadecimal notations, one other system is used in computing - binary-coded decimal. As its name suggests it is a hybrid form with elements from binary and decimal. It is commonly used where an output is required in digital format, e.g. a digital clock, or when great precision is required and no bits can be dropped.

In BCD the normal decimal base is retained, i.e. one place is a factor of 10 times its neighbor but each individual digit is represented in binary. Thus the number  $87_{10}$  would be represented as:



i.e. BCD = 1000 0111 (or in eight bits  
10000111)

Fig. A1.6

As the largest digit required in decimal notation is 9, only four bits of binary are needed to represent this, i.e.  $9_{10} = 1001_2$ , thus a BCD digit can be represented by a nybble and two digits by a byte. Figure A1.6 shows this, where  $87_{10}$  is represented in BCD as  $10000111_2$ . This can give rise to ambiguity in that  $10000111_2$  in binary is  $135_{10}$ . To overcome this, BCD representations will be given the notation  $10000111_2(\text{BCD})$ .

Using four bits of binary, it is possible to count up to  $15_{10}$  (i.e.  $1111_2=15_{10}$ ) but in BCD the largest digit used is 9, so inevitably BCD is less economical in its use of space. Its largest digit, 9, is  $1001_2$  and when one is added to this it clocks over to  $0000_2$  and carries the 1 to the next nybble, i.e.

$8_{10}$	=	0000 1000	(base 2 BCD)
$9_{10}$	=	0000 1001	" " "
$10_{10}$	=	0001 0000	" " "
$11_{10}$	=	0001 0001	" " "

Fig. A1.7

It would probably be helpful at this point if you load and run the Binary/Hex tutor program again. This time, select 'B' at the main menu, and when asked "At what number..." enter a 1 <return>.

The display will then show three rows of boxes again but this time they will contain decimal, binary and BCD. If you press the space bar as before, to index from '1', you will notice that up to  $9_{10}$ , binary and BCD are identical.. However, as you index from  $9_{10}$  to  $10_{10}$  keep an eye on the BCD box and you will see the 1 carried over to the most significant nybble. From  $10_{10}$  upwards BCD becomes a true hybrid representing the decimal number in a binary form.

As the number increases, the uneconomical nature of BCD will become apparent as  $99_{10}$  changes to  $100_{10}$ . When  $99_{10}$  indexes to  $100_{10}$  you will see the BCD generate a carry from its most significant nybble to the carry flag.

This carry is only a short term expedient and must be picked up at the earliest possible moment if it is not to be lost. This carry is generated on the BCD boxes at  $99_{10}$  while the binary boxes will store up to  $255_{10}$ . BCD is therefore fairly uneconomical in memory usage, but it has its uses in particular situations. In the past, microcomputers have always been dogged by their lack of memory and consequently BCD has been little used. However, the new generation of microcomputers have much larger memories and it is quite likely that BCD will be used much more frequently than it was in the past. Perhaps it is a sign of the times, that, although all COMMODORE computers have had BCD capability, your C-64 is the first to make use of BCD, albeit in a very small way. The 24 hour time of day clocks which are built into the C-64's two input/output chips (6526's) do make use of BCD.

As you know all about BCD now(!) try the following:-

#### EXERCISE A1.3

Convert the following decimal numbers into BCD:

- |         |            |
|---------|------------|
| i) 4    | v) 53      |
| ii) 10  | vi) 102    |
| iii) 77 | vii) 953   |
| iv) 97  | viii) 2579 |

Answers on page 11-9.

#### EXERCISE A1.4

Convert the following BCD numbers into decimal:-

- |                 |
|-----------------|
| i) 0000 0001    |
| ii) 0000 1001   |
| iii) 0001 0101  |
| iv) 0010 0000   |
| v) 0100 1001    |
| vi) 1010 0011   |
| vii) 1001 0111  |
| viii) 1000 1000 |

Answers on page 11-9.

In the explanations given of the value of places in place-value notation a simplification was adopted in order to make these explanations clearer for our less mathematically inclined brethren. However, if you wish to see a slightly more mathematical explanation, please read on. Otherwise - END OF APPENDIX ONE.

With binary numbers it was said that the places increase their value in multiples of 2, but the least significant bit of the binary number was equivalent to the same symbol base 10 (or for that matter base 3, or whatever). In actual fact the multiplying factor is the base raised to the power of its place starting with zero at the left. i.e. in binary:

7	6	5	4	3	2	1	0	Place
128	64	32	16	8	4	2	1	Previously stated multiplication factor
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	Mathematically more precise factor.

Thus the least significant bit is multiplied by  $2^0$  or 1. (If you are not sure of this try the direct program PRINT 2 0.) The next bit is multiplied by  $2^1$ , and so on.

This rule holds for ANY base; let's apply it for hex, i.e. base 16:

Least significant bit factor	=	$16^0$	=	1
2nd most significant bit factor	=	$16^1$	=	16
3rd most significant bit factor	=	$16^2$	=	256
Most significant bit factor	=	$16^3$	=	4096



## APPENDIX

# 2

This appendix is a guide to the Honey.Aid utility package which is supplied with this book.

### LIST OF HONEY.AID COMMANDS

#### Toolkit Commands

The Toolkit commands are intended to assist with the creation of a working BASIC program and subsequent amendment etc..

Toolkit commands can be used only as 'direct' commands i.e. they may not be included as commands within a BASIC program. An attempt to use one of the Toolkit commands within a program will result in a SYNTAX ERROR message and the abandonment of the program in which they were included. As will be realized, this is not really a serious restriction as the only use ever likely to be required of the Toolkit commands will be as 'direct' commands.

APPEND	CHANGE	HMEM	OLD
AUTO	DELETE	KILL	REPEAT
BRK	EXTEND	LMEM	RESET
CBM	FIND	NUMBER	

All other Honey.Aid commands, may be used either as 'direct' commands or as BASIC program commands.

## Graphics Commands

COLOUR (or COLOR)  
HIRES

LINE  
NRM

PLOT

## Synthesizer Commands

ENVELOPE  
PLAY  
PULSE

SOUND  
TEMPO  
VOL

WAVE

## DESCRIPTION OF HONEY.AID COMMANDS

### 1. Toolkit Commands

#### APPEND (abbreviation A <SHIFT> P)

Used in conjunction with the HMEM command (see below). Appends the BASIC program placed in high memory (using the HMEM command) to the program preserved in low memory. APPEND effectively issues an LMEM command (see below) on completion of the APPEND.

APPEND simply attaches the high memory program to the low memory program. The renumbering command, NUMBER, may/should be used prior to the APPEND to adjust the line numbers of the two programs so that the combined program's line numbers are in sequence. In many cases, renumbering after the APPEND will not produce problems.

#### AUTO (<SHIFT> A or A <SHIFT> U)

Creates line numbers automatically, using the specified increment.

	AUTO 100	creates increments of one hundred
and	AUTO 20	creates increments of twenty
while	AUTO	(no increment) is equivalent to AUTO 10.

Once an AUTO command has been issued, Honey.Aid will respond to the entry of any numbered BASIC line by issuing the next line number and waiting for the remainder of the line to be typed in; Honey.Aid will then issue the next line number and so on. The line numbers which is issued by Honey.Aid may be changed if required and the next line number issued by Honey.Aid will reflect the changed line number. AUTO is switched off by typing an empty line i.e. a line number followed by a <RETURN>.

### BRK (<SHIFT> B or B <SHIFT> R)

Performs a machine code BRK instruction. Unless a program has been loaded which responds to a machine code BRK, then the result of the BRK command will be a BASIC warm start. (BASIC pointers will be reset and the READY message will be given - in other words 'not a lot'). However, if a Machine Language Monitor (MLM) has been loaded, then the BRK command will cause the MLM to be entered.

### CBM (<SHIFT> C or C <SHIFT> B)

Prevents Honey.Aid from creating a non-CBM BASIC program. If either the special Honey.Aid Graphics or Synthesizer commands are included in a BASIC program, then the program will not be runnable by a C-64 which is not equipped with Honey.Aid. However, it is obviously very handy to be able to use the Honey.Aid Toolkit commands when writing BASIC programs which do not include Honey.Aid Graphics/Synthesizer commands. Use of the CBM command before creating the BASIC program will prevent Honey.Aid commands from being encoded and any accidental use of a Honey.Aid command in the program may be detected when the program is tested (a SYNTAX ERROR message will appear). Note that the CBM command does not prevent a previously saved Honey.Aid program from being run (provided it was not created with CBM in force, of course).

The CBM command will also cause a 'bleep' to be issued and the color of the border to be changed to remind the user that CBM is in force. CBM is cancelled by the EXTEND command (see below).

### CHANGE (C <SHIFT> H)

CHANGE is similar, in many ways, to the FIND command (see below). It will search the program for all the occurrences of one particular string of characters and then replace these by a second string of characters. Optionally, the search and change may be confined to one section of the program only. The string of characters being sought is defined by using almost any character which is not itself a part of the sought string (or the replacement string) as a delimiter. The delimiter must immediately precede and follow the strings, unless you intend the space to be considered part of the string. Full stop and comma are convenient characters to use as delimiters because of their position on the keyboard, but the slash sign (/) is not so good (see below).

## Examples:

CHANGE.PRINT.PRINT#3. will replace all occurrences of PRINT by PRINT#3 (but see "Using Quotes as a delimiter"). The character "." has been used here as a delimiter.

CHANGEQA\$QAI\$Q,100-860 will replace the variable name A\$ by the name AI\$ within the range of lines 100 to 860 inclusive. The letter Q has been used as a delimiter here.

The second example illustrates using a letter as a delimiter. However, most users find it less confusing when non-alphanumeric delimiters are used. The quote sign, when used as a delimiter, has some special effects; see below.

## Using quotes as a delimiter:

If you do search for PRINT, say, using ordinary (non-quote) delimiters, then some occurrences of PRINT may not be found. The 'missed' PRINTs will be those which appear within quotes in the program (e.g. the PRINT in 100 INPUT "PRINTER OR SCREEN";A\$ would not be found). Strings within quotes which contain a Commodore BASIC keyword (e.g. PRINT, GOTO, TO...) will only be found by CHANGE when the delimiters used are quotes. Similarly, strings which contain Commodore BASIC keywords which are not within quotes will not be found if the delimiters used are quotes. Hence, CHANGE "PRINTER"SCREEN" will find all occurrences of PRINTER which occur within quotes (and will change these to SCREEN), but it would not find for instance, the string in 220 PRINTER;S\$ (i.e. 220 PRINT ER;S\$). Lastly, you cannot use CHANGE to replace a string which is only a part of a BASIC keyword, unless the string is within quotes. FIND.RINT. will fail to find the RINT's in the PRINT and PRINT# commands.

Although +, -, \* and / may be used as delimiters without problems most of the time, they do not work if you are looking for strings which contain the BASIC REM or DATA keywords.

## DELETE (<SHIFT> D or D <SHIFT> E)

DELETE enables the deletion of a number of adjacent lines from a BASIC program.

Examples:

DELETE 100            deletes line number 100, not very useful: a 100 by itself would have the same effect.

DELETE 100-            deletes all lines from line 100 (inclusive) to the last line of the program.

DELETE 100-250        deletes all lines from 100 to 250 (inclusive).

DELETE -250            deletes all lines from the start of the program up to and including line 250.

Warning: DELETE - (the hyphen is present but the start and finishing line numbers are missing) will delete the whole program. If this is done, in error, then the situation cannot be recovered by use of the OLD command (see below).

## EXTEND (<SHIFT> E or E <SHIFT> X)

Will restore Honey.Aid following a CBM command so that Honey.Aid Graphic and Synthesizer commands used during the creation of a BASIC program will be accepted. Honey.Aid will sound a 'bleep' and restore the border to the original color as an indicator that Honey.Aid commands may again be included in BASIC programs.

## FIND (<SHIFT> F or F <SHIFT> I)

FIND provides a method of finding all the lines in the program containing a particular string of characters. Optionally, the search may be confined to a particular range of lines. The string of characters which is being sought is defined by using almost any character which is not itself part of the string as a delimiter.

Examples:

- FIND.PRINT A\$. finds and lists all lines in the program which contain the characters PRINT A\$. The character "." has been used here as a delimiter.
- FIND\$NEXT\$,100-580 finds all occurrences of NEXT in the line range 100 to 580 inclusive.
- FINDQLETQ,560- finds all occurrences of LET from line 560 to the end of the program. This example uses the letter Q as a delimiter.

The use of quotes as a delimiter in the FIND command mirrors the way in which they are used by the CHANGE command. Similarly the use of +, -, / or \* as a delimiter is not recommended as it will fail to find a string containing a REM or a DATA.

#### HMEM (H <SHIFT> M)

Seals off and preserves the existing program in memory and permits the memory which follows the existing program to be used for other programs. Possible uses are:

- i) To examine a program held on disk or cassette without the need to save the existing program, or
- ii) as a preparation to appending a program held on disk or cassette with the existing program. Loading the program which is to be APPENDED into high memory provides the opportunity to edit the program prior to the APPEND, or
- iii) examining the Directory of a disk.

When the HMEM command is issued, Honey.Aid will respond with a 'bleep' and will change the color of the border as a reminder to the user that HMEM is in force.

If the C-64 is already in the HMEM state when the HMEM command is used, a SYNTAX ERROR message will be printed to the screen.

## KILL (<SHIFT> K or K <SHIFT> I)

The KILL command is provided to enable Honey.Aid to be switched off. One reason for wishing to switch Honey.Aid off might be to remove the small overhead in the running speed of a program which is not making use of the Honey.Aid Graphics or Synthesizer commands. This overhead is considerably less than is normal with several other BASIC 'Toolkit' systems, owing to the method used to link Honey.Aid to Commodore 64 BASIC. Even so, the facility to switch off is provided should you need it.

Note that Honey.Aid is not removed from the C-64's memory by the KILL command and may be reactivated at a later time, if required. The procedure for reactivating Honey.Aid is explained in the introductory message put on to the screen when Honey.Aid is initialized, and a further reminder is given when the KILL command is used. In the current version of Honey.Aid, this procedure is to type SYS 750 followed by <RETURN>.

## LMEM (<SHIFT> L or L <SHIFT> M)

Releases the high memory space made available by HMEM and reverts to normal BASIC space complete with its original contents. Any program which is currently held in high memory will not be preserved, although an immediate HMEM following a LMEM should reinstate the high memory program. LMEM is unnecessary if APPEND is used, as automatic reversion to normal memory follows its execution.

If the C-64 is in the LMEM state when the LMEM command is issued, then a SYNTAX ERROR message will result.

## NUMBER (<SHIFT> N or N <SHIFT> U)

NUMBER enables the whole of a program to be renumbered. The line number which is to be given to the first line of the renumbered program may be specified, if omitted the value 100 will be assumed. The increment value to be added to each line number to create the next may also be specified, if omitted the value 10 will be assumed. All commands which refer to other line numbers such as GOTO, GOSUB, THEN, RUN and LIST will automatically have the line numbers adjusted.

### Examples:

NUMBER 200,25 will renumber the program so that the new line numbers will be:

200 ...  
225 ...  
250 ...

etc

NUMBER is equivalent to NUMBER 100,10  
NUMBER 200 is equivalent to NUMBER 200,10  
NUMBER ,20 is equivalent to NUMBER 100,20

Whenever Honey.Aid adjusts the line numbers referred to by a GOTO, GOSUB, THEN, RUN or LIST statement, the new line number is printed to the screen, e.g. if the line 220 GOTO 150 has been adjusted to 250 GOTO 190 then Honey.Aid will print 190 on the screen. These numbers should always be examined to check that there have been no problems during the renumbering process. The problem most likely to occur is when Honey.Aid discovers a GOTO, GOSUB, RUN, THEN statement which refers to a non-existent line number (the sort of problem which causes BASIC to complain of an UNDEF'D STATEMENT). Honey.Aid will replace the silly line number with a line number which is even sillier: 65535. It is these 65535 numbers that should be looked for in the list of changes that Honey.Aid produces. Clearly, only the programmer will be able to sort out what is his/her own error, but at least the FIND command is available to assist in clearing up the problem.

Warning: It is possible to obtain peculiar results if the start line number or increment value are such as will cause the line numbers at the end of the program to exceed 63999 (C-64 BASIC does not recognize program line numbers greater than this). Line numbers greater than 65535 cannot be created anyway. If this happens, it is usually possible, but not always, to sort the problems out by reNUMBERing with less extravagant values.

### OLD (<SHIFT> O or O <SHIFT> L)

The purpose of the OLD command is to undo a careless NEW command. OLD will reinstate a program to the state that it was in immediately prior to the NEW command. Note that OLD can not undo a careless DELETE command.



## REPEAT (<SHIFT> R or R <SHIFT> E)

When the C-64 is initialized, only the SPACE bar, the cursor keys and the INST/DEL keys will repeat if held down for longer than 0.5 seconds. The REPEAT command can be used to change the range of keys that will repeat in this manner.

Examples:

```
REPEAT 0 no keys to repeat
REPEAT 1 all keys to repeat
REPEAT revert to standard repeat pattern, i.e. SPACE,
. cursors and INST/ DEL.
```

## RESET (R E <SHIFT> S)

This is a very drastic command. The RESET command causes the C-64 to be reset (the so-called 'cold restart'). Exactly the same result is achieved as if the C-64 had been switched off and then switched on again: everything within sight is initialized including the RAM memory.

The occasions when this might be required are when the 64 has got itself in a complete mess and you wish to restart, or perhaps you might want to clear out every program from memory (even Honey.Aid!!).

Since this is such a drastic command, Honey.Aid will ask for confirmation before doing the RESET. Honey.Aid will warn "System Reset - Are you sure y/n?", and any response other than y will cause the RESET to be abandoned.

## 2. Graphics Commands

One of the great advantages of the Honey.Aid Graphics system over other systems on offer is that the Graphics commands can be used in 'direct' mode. Thus the user can experiment with different commands and see the result immediately, before building them into a BASIC program. A second advantage, not always provided by competing systems, is that Honey.Aid will only clear the high resolution screen when it is told to do so; it does not clear the screen automatically at the beginning of the execution of a program as many systems do. Therefore a graphics picture may be built up in stages by several runs of the same, or different programs.

The high resolution graphics screen is maintained in the RAM which lies under the C-64's Operating System ROM. The other part of the high resolution screen system, the color screen, uses the bottom 2000 bytes of the C block. The consequence of this is that the use of high resolution graphics subtracts not one single byte from the normal BASIC memory space and this means that very large high resolution graphics BASIC programs may be written. This is made possible, of course, by the unique characteristics of the C-64's 6510 processor, and Honey.Aid sets out to extract the full benefit from this chip for the user.

The system of defining the X, Y axes for plotting which is used by Honey.Aid conforms to the conventional cartesian system as taught in all schools i.e. the origin of the axes is in the bottom left hand corner of the screen and Y displacements are measured upwards. The majority of Graphic systems designed for microcomputers defy this convention - the origin is in the top left hand corner and Y displacements are measured downwards (graphic systems for mini and main-frame computers stick to the conventional Cartesian system however).

Finally, it should be noted that Honey.Aid allows high resolution graphics to be used alongside the synthesizer with no ill effects on either. With skill and patience, the all singing AND all dancing picture show is yours for the programing.

#### COLOUR or COLOR (abbreviation C <SHIFT> 0)

This command changes both the color of the normal text screens background (paper) and of the border. The first parameter specifies the screen background color and the second (optional) parameter controls the border color, e.g.

COLOR 0,1

would change the screen background to black (0) and the border to white (1).

COLOR 15

would change the screen background to grey 3 and leave the border color unchanged.

## HIRES (H <SHIFT> I)

Any graphics work that we wish to do using Honey.Aid's advanced commands, must be done in the high resolution screen area. This is an area '320' points across and '200' points up. The HIRES command switches the display from the normal text screen to the high-resolution screen, optionally setting/changing the drawing (ink) color, setting/changing the background (paper) and clearing screen or not.

HIRES i,p,c

Where i is the ink color, the color we will draw in,  
p is the background or paper color  
and c indicates whether or not we clear the HIRES screen. A value of '1' will clear the HIRES screen and a zero will retain the previous picture.

so: HIRES 2,8,1

will allow us to draw in red on an orange background, with the screen cleared of any previous drawings.

All arguments are optional so:

HIRES switches on the high-resolution screen using the same colors as before, retaining previous picture.

HIRES 6 would change the 'ink' color to blue, leaving the 'paper' color unchanged and retaining previous picture.

One of the oddities of the C-64 is that the colors of the high resolution screen can only be changed in chunks which correspond to the character spaces of the normal text. Thus the pixels belonging to the 8 by 8 square of pixels which make up a single character space have their own paper and ink colors. If HIRES is called with the 'c' value set to 1 then all of the chunks will be initialized to the same 'paper' and 'ink' colors. If HIRES is then called with a different set of colors and 'c' value set to zero, or omitted, then no immediate changes will be observed. However, any future plotting which is carried out will change the two colors of the character space in which the plotting takes place - all 64 pixels, even if only a single pixel is plotted. This can produce rather unexpected results but used with care can produce the effect of a full sixteen color high resolution graphical display.

## LINE (LI <SHIFT> N)

The LINE command allows one to draw a line from two sets of co-ordinates, the first co-ordinates being the start point and the second set being the end position.

```
LINE x1,y1,x2,y2,n
```

'n' controls the type of LINE that we will draw. There are three possible values of 'n', and these are:

- 0 This tells the computer to draw the line in the background (paper) color, (or erase the line).
- 1 Tells the computer to draw the line in the ink color.
- 2 Tells the computer to draw the line in inverse. The line will be drawn in the ink color except where it crosses a line or a point which is already drawn in the ink color. At this point it will 'switch off', revert to background color. At the point of intersection, and all other points will be drawn in the ink color.

So:           LINE 10,20,5,100,1

will draw a line from the eleventh point across (the points are numbered from zero) and twenty one points up to the sixth point across and the one-hundredth and first point up, in the ink color.

The value of the co-ordinates must be in the range of '0' to '319' for 'X' and '0' to '199' for 'Y', any other value will result in an ILLEGAL QUANTITY ERROR message.

## NRM (N <SHIFT> R)

The NRM command returns the computer to normal text use after using the HIRES graphics area. There are no parameters in this command.

In practice, this command is little used. If a program simply goes into HIRES mode and draws a pretty picture, without needing to revert to the text screen during the run, then the NRM command is unnecessary. Following the successful completion of a graphics run the program will stop with the graphics picture on display for all to admire. Pressing any key will automatically switch the screen to text mode (i.e. will do a NRM). With Honey.Aid, therefore, there is no need for delay loops at the end of a graphics run to allow the final picture to be displayed - it is automatic.

If a graphics program collapses during a run with an error, the screen will automatically revert to the text screen to show the error message (note that the high resolution picture, as it was at the time of the error, is still present, and may be examined by issuing a HIRES command).

The only time that a NRM command is necessary is when the program is halted with a STOP command. This is deliberate as this is possibly the effect that is required. If the text screen is required at STOPS then a NRM command should be placed before the STOP. If, in error, a STOP command without a NRM command is encountered, then NRM can be typed in as a 'direct' command.

### PLOT (P <SHIFT> L)

The PLOT command allows one to 'set', 'clear' or 'invert' each individual point on the HIRES screen. The PLOT command (like LINE) can 'plot' in three forms and these are:

- 0 This PLOTS the point in the background (paper) color, or if you like, erases the point.
- 1 This PLOTS the point in the 'ink' color.
- 2 PLOTS that point in the inverse. Thus if there was a point already there, it would then become the same as the background color. If there wasn't, then the point would appear in the 'ink' color.

So:                    PLOT 53,78,0

sets a point on the fifty-fourth point across (remember the points are numbered from zero) and the seventy-ninth point up, in the same color as the background color.

and                    PLOT 319\*RND(1),199\*RND(1),1

will plot a random point in the ink color, somewhere on the screen.

The value of co-ordinates must be in the range of '0' to '319' for 'X' and '0' to '199' for 'Y'; any other value will result in an ILLEGAL QUANTITY ERROR message.

### 3. Synthesizer Commands

The Honey.Aid Synthesizer system has been designed with compactness and ease of use as prime objectives. For example, suppose that a middle C crotchet is played with voice 1. Honey.Aid will 'remember' that it has played a middle C crotchet. Now if a request is made to play the same note with voice 2, it is not necessary to re-specify either the note, the octave or the duration as Honey.Aid will automatically use the 'remembered' values if new values are not supplied. In other words the values are 'sticky'.

Honey.Aid has been tuned to international 'concert pitch' (A4 = 440 Hz). There are two versions of the C-64 available intended for use with either NTSC TV standards (USA and others) or with PAL TV (Europe and others). The two versions of the C-64 have slightly different clock rates to cope with the different TV standards and consequently these clock rates have an effect on the pitch of the sound commands.

Honey.Aid automatically provides the appropriate compensation for this problem and will maintain concert pitch on either of the two versions of the C-64.

#### ENVELOPE (E N <SHIFT> V)

Sets the envelope characteristics of a specific voice.

The general form of the command is:

```
ENVELOPE v,a,d,s,r
```

where v specifies which of the three voices is to have its envelope modified: 1, 2, or 3,

a specifies the attack rate : 0 to 15  
d specifies the decay rate : 0 to 15  
s specifies the sustain level : 0 to 15  
and r specifies the release rate : 0 to 15.

If any of the a, d, s, or r values is not being changed then it is not necessary to include a value in the list.

Examples:

- ENVELOPE 1,0,1,8,2 sets the envelope characteristics of voice 1 to attack 0, decay 2, sustain 8, release 2.
- ENVELOPE 1,,,4 changes only the release rate of voice 1 to a value of 4.
- ENVELOPE 1,,5 changes the decay rate of voice 1 to a value of 5.

PLAY (P L <SHIFT> A)

This is a very powerful command which is used to play a sequence of musical notes. A single PLAY command could, in the ultimate, play a piece of music of 6630 notes!

In its simplest form PLAY takes a single string parameter. e.g. PLAY A\$ or PLAY "CDEFG". The string specifies the actual notes which are to be PLAYed. If desired, additional strings of notes may be specified.

e.g PLAY A\$,B\$,C\$ or PLAY "CDEFG",X\$.

The largest number of strings that could be specified with one PLAY command on one double line is 26, and as each of these strings could specify up to 255 notes, the maximum number of notes is 26 x 255 or 6630. A maximum of four symbols is required to represent a single note, but in certain circumstances it may be possible to reduce this to a single symbol.

One symbol is used to specify the voice which is to play the note: <BLK> (or <CTRL>1) specifying voice 1, <WHT> (or <CTRL>2) specifying voice 2 and <RED> (or <CTRL>3) specifying voice 3.

A second symbol is used to specify the note to be played: C D E F G A B specify the natural and shifted C D F G A are used to specify the sharps. An R or a hyphen is used to represent a rest. It is easier to see what is being specified if the C-64 is set to lower case mode, of course, and the examples below assume the mode.

A third symbol is used to represent the octave, and the international convention of numbering is used i.e. 0 to 7.

Lastly, the fourth symbol is used to represent the duration of the note. The function keys are used for this purpose : <f1> specifies a breve, <f3> a semi-breve, <f5> a minim, <f7> a crotchet. The 'shifted' function keys are used to represent the shorter notes : <f2> a quaver, <f4> a semi-quaver, <f6> a demi-semi-quaver and <f8> a hemi-demi-semi-quaver is!

Example:

```
PLAY "<BLK>C4<f7>" voice 1 to play a middle C crotchet
```

Any of the four symbols may be omitted (but not all, of course). If the voice symbol is omitted then the same voice as was last specified will be used. Similarly omitting the note, octave or duration symbol will result in the last specified note, octave or duration being used. To make this 'sticky' value process work, it is essential to remember that the order of specifications must be in the sequence: voice followed by note followed by octave followed by duration. If Honey.Aid finds a specifier out of sequence it will assume that this is the start of a new note. Playing chords is extremely easy. If Honey.Aid is told to play a note with a particular voice it will do so immediately if the voice is currently inactive or if it is in its release phase. If the voice is active in the attack, decay or sustain phase then Honey.Aid will wait. Thus

```
PLAY "<BLK>C4<f7><WHT><RED>3"
```

will play a chord - voice 1 will start to play a middle C crotchet and then Honey.Aid will find the voice 2 note, and since voice 2 is inactive it will be started immediately playing a middle C crotchet. The delay between the two starts is, to all intents and purposes imperceptible. Similarly, voice 3 will start playing a C below middle C crotchet without noticeable delay.

```
PLAY "<BLK>C4<f5><WHT>E<f7><RED>G<WHT><RED>"
```

Would play a middle C semi-breve in synchronism with two chords consisting of voice 2 playing E and voice 3 playing G crotchets. They should all finish together.



Off-beat effects with multiple voices are also simple to achieve by using 'rests' to delay the start of a voice, thus:

```
PLAY "<BLK>C4<F3><WHT>-<F2><RED>-<F7><WHT>E<F3><RED>"
```

Produces a sort of rippled chord.

Note that PLAY may be interrupted by the STOP key between notes.

#### PULSE (P <SHIFT> U)

This sets the pulse width of a specific voice to a given value. The effect of this will, of course, only be observed if the voice is using the pulse wave form.

The form of the command is:

```
PULSE v,w
```

Where 'v' specifies the voice (1,2 or 3) and 'w' specifies the pulse width (valid values are 0 to 4095)

#### SOUND (S <SHIFT> O)

May be used to play a single note

The form of the command is:

```
SOUND v,o,n,d
```

where v specifies the voice : 1 to 3  
o specifies the octave : 0 to 7  
n specifies the note : 0 to 12  
and d specifies the duration : 1 to 8

The voice and the octave specifications are the same as for the PLAY command. A zero value for the note creates a 'rest'. The other values 1 to 12 correspond to C, C#, D, D#, E, F, F#, G, G#, A, A# and B respectively.

The duration values 8 to 1 correspond to breve, semi-breve, minim, crotchet, quaver, etc.

As with the ENVELOPE command, it is only necessary to include values for those parts which need to be changed, in fact no values at all is perfectly acceptable, thus:

```
SOUND 1,4,1,6  
SOUND
```

would cause VOICE 1 to play two middle C crotchets.

The only advantage that SOUND has over PLAY is that parameters may be variables or expressions (the characters within the PLAY string are simply literals, of course). Thus SOUND could be used to issue a note whose pitch varied with the proximity of a missile from it's target, for instance, the octave and the note being computed from the distance.

#### TEMPO (T E <SHIFT> M)

Sets the tempo of all voices to the specified number of crotchets per minute.

The general form of the command is:

```
TEMPO t
```

Where 't' may be any positive value greater than or equal to 1. TEMPO with no 't' value specified will default to the standard March tempo of 120 crotchets per minute. Be warned, however, a tempo value of 1 means one crotchet per minute which means a very long crotchet indeed. Although Honey.Aid allows you to interrupt the music with the STOP key, the interruption takes place between notes. With a tempo setting of 1, you could have to wait four minutes for a breve to finish before Honey.Aid allowed your interruption.

#### VOL (V <SHIFT> O)

Sets the volume for all voices to the specified value (0 to 15)

The general form of this command is:

```
VOL v
```

where v may be any value between 0 (silent) and 15 (maximum). VOL with no 'v' specified defaults to 15.

## WAVE (W A <SHIFT> V)

Sets the specified voice to a wave form.

The general form of the command is:

WAVE v,w

'v' specifies the voice and 'w' the waveform. 'w' lies in the range of 0 to 8. A value of 0 disconnects the voice from all oscillators. Values of 1, 2, 4 or 8 sets the voice to the triangular, sawtooth, pulse or random (white noise) wave form respectively. It is possible (but not always very profitable) to connect the voices to two or even three wave forms, thus a value of 3 (1+2) connects to the triangular and sawtooth wave forms (with little result). Note that since the wave value may not exceed 8, the random wave form cannot be mixed with any of the others. This is deliberate, as such combinations are likely to cause the SID chip to lock up solid.

There are no reasons why all these voices cannot be set to the same waveform, with or without the same envelope.

## Initial Defaults

When Honey.Aid is first loaded or following a STOP/RESTORE panic, the standard defaults shown below are set up. This enables simple sounds to be generated without the need for setting up VOL, TEMPO, WAVE, ENVELOPE etc. If any changes are made to these values they remain at the new value until changed again.

### standard values

VOL	initially set to 15
TEMPO	initially set to 120 (120 crotchets per minute)
VOICE1	initially set to wave form 1 (triangular)
VOICE2	initially set to wave form 4 (pulse) with a pulse width of 255 (6.225%)
VOICE3	initially set to wave form 8 (random noise)
OCTAVE	initially set to 4
NOTE	initially set to C natural, hence middle C is the default octave/note combination.
DURATION	initially set to 4 (semi-quaver)



## APPENDIX

# 3

Table 1 - Commodore 64 Character Set

Table 2 - Hex to Decimal Conversion

Table 3 - ASCII Character Set

Table 4 - ASCII and CHR\$ Codes

Table 5 - C-64 Character Symbol Representation Convention

Table 6 - VIC II (Video Interface Controller) Registers

Table 7 - SID (Sound Interface Device) Registers

TABLE 1

64 Character Set

poke set1 set2	poke set1 set2	poke set1 set2	poke set1 set2
0 @	33 !	65  A	97
1 A a	34 "	66  B	98
2 B b	35 #	67  C	99
3 C c	36 \$	68  D	100
4 D d	37 %	69  E	101
5 E e	38 &	70  F	102
6 F f	39 .	71  G	103
7 G g	40 (	72  H	104
8 H h	41 )	73  I	105
9 I i	42 •	74  J	106
10 J j	43 +	75  K	107
11 K k	44 ,	76  L	108
12 L l	45 -	77  M	109
13 M m	46 .	78  N	110
14 N n	47 /	79  O	111
15 O o	48 0	80  P	112
16 P p	49 1	81  Q	113
17 Q q	50 2	82  R	114
18 R r	51 3	83  S	115
19 S s	52 4	84  T	116
20 T t	53 5	85  U	117
21 U u	54 6	86  V	118
22 V v	55 7	87  W	119
23 W w	56 8	88  X	120
24 X x	57 9	89  Y	121
25 Y y	58 :	90  Z	122
26 Z z	59 ;	91	123
27 [	60 <	92	124
28 £	61 =	93	125
29 ]	62 >	94	126
30 ↑	63 ?	95	127
31 ←	64	96 <space>	
32 <space>			

Codes from 128 to 255 are reversed images of codes 0-127.

TABLE 2

Hexadecimal to Decimal Conversion Chart																
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

TABLE 3

ASCII CHARACTER SET									
HEX	MSB	0	1	2	3	4	5	6	7
LSB	BIN	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	-	o	DEL

TABLE 4

PRINTS	CHR\$	PRINTS	CHR\$	PRINT\$	CHR\$	PRINTS	CHR\$
	0	.	46	#	92	f4	138
	1	/	47	J	93	f6	139
	2	0	48	↑	94	f8	140
	3	1	49	←	95	SHIFT<RET'N>	141
<WHT>	4	2	50	☐	96	<UCASE>	142
	5	3	51	☐	97		143
	6	4	52	☐	98	<BLK>	144
<NOCBM>	8	5	53	☐	99	<UCRSR>	145
<OKCBM>	9	6	54	☐	100	<RVSOFF>	146
	10	7	55	☐	101	<CLR>	147
	11	8	56	☐	102	<INST>	148
	12	9	57	☐	103	<BRWN>	149
<RET'N>	13	:	58	☐	104	<LTRED>	150
<LCASE>	14	;	59	☐	105	<GRY1>	151
	15	<	60	☐	106	<GRY2>	152
	16	=	61	☐	107	<LTGRN>	153
<DCRSR>	17	>	62	☐	108	<LTBLU>	154
<RVSON>	18	?	63	☐	109	<GRY3>	155
<HOME>	19	@	64	☐	110	<PUR>	156
<DEL>	20	A	65	☐	111	<LCRSR>	157
	21	B	66	☐	112	<YEL>	158
	22	C	67	☐	113	<CYN>	159
	23	D	68	☐	114	<SPACE>	160
	24	E	69	☐	115	☐	161
	25	F	70	☐	116	☐	162
	26	G	71	☐	117	☐	163
	27	H	72	☐	118	☐	164
<RED>	28	I	73	☐	119	☐	165
<RCRSR>	29	J	74	☐	120	☐	166
<GRN>	30	K	75	☐	121	☐	167
<BLU>	31	L	76	☐	122	☐	168
<SPACE>	32	M	77	☐	123	☐	169
!	33	N	78	☐	124	☐	170
"	34	O	79	☐	125	☐	171
#	35	P	80	☐	126	☐	172
\$	36	Q	81	☐	127	☐	173
%	37	R	82	☐	128	☐	174
&	38	S	83	<ORNG>	129	☐	175
'	39	T	84		130	☐	176
(	40	U	85		131	☐	177
)	41	V	86		132	☐	178
*	42	W	87	f1	133	☐	179
+	43	X	88	f3	134	☐	180
,	44	Y	89	f5	135	☐	181
-	45	Z	90	f7	136	☐	182
		[	91	f2	137	☐	183



PRINTS	CHR\$	PRINTS	CHR\$	PRINT\$	CHR\$	PRINTS	CHR\$
<input type="checkbox"/>	184	<input type="checkbox"/>	186	<input checked="" type="checkbox"/>	188	<input type="checkbox"/>	190
<input type="checkbox"/>	185	<input checked="" type="checkbox"/>	187	<input type="checkbox"/>	189	<input checked="" type="checkbox"/>	191

CODES	192-223	SAME AS	96-127
CODES	224-254	SAME AS	160-190
CODES	255	SAME AS	126

TABLE 5  
CHARACTER SET CONVENTION

MEANING	(OBTAINED BY)	CONVENTION
Cursor up	(<SHIFT><CRSR ↑ >)	<UCRSR>
Cursor down	(<CRSR ↓ >)	<DCRSR>
Cursor right	(<CRSR ⇒ >)	<RCRSR>
Cursor left	(<SHIFT><CRSR ⇐ >)	<LCRSR>
Clear screen	(<SHIFT><CLR/HOME>)	<CLR>
Home	(<CLR/HOME>)	<HOME>
Reverse off	(<CTRL>0)	<RVS OFF>
Reverse on	(<CTRL>9)	<RVS ON>
Upper case	(<SHIFT><CBM>)	<UCASE>
Lower case	(<CTRL>N) or <SHIFT> <CBM>)	<LCASE>
CBM disable	(<CTRL>H)	<NOCBM>
CBM enable	(<CTRL>I)	<OKCBM>
Black	(<CTRL><1>)	<BLK>
White	(<CTRL><2>)	<WHT>
Red	(<CTRL><3>)	<RED>
Cyan	(<CTRL><4>)	<CYN>
Purple	(<CTRL><5>)	<PUR>
Green	(<CTRL><6>)	<GRN>
Blue	(<CTRL><7>)	<BLU>
Yellow	(<CTRL><8>)	<YEL>
Orange	(<CBM><1>)	<ORNG>
Brown	(<CBM><2>)	<BRWN>
Light red	(<CBM><3>)	<LTRED>
Grey 1	(<CBM><4>)	<GRY1>
Grey 2	(<CBM><5>)	<GRY2>
Light green	(<CBM><6>)	<LTGRN>
Light blue	(<CBM><7>)	<LTBLU>
Grey 3	(<CBM><8>)	<GRY3>

## VIC II (Video Interface Controller) Registers

No	Address		Bits	Description
	Hex	Decimal		
0	D000	53248		Sprite 0 X position
1	D001	53249		Sprite 0 Y position
2	D002	53250		Sprite 1 X position
3	D003	53251		Sprite 1 Y position
4	D004	53252		Sprite 2 X position
5	D005	53253		Sprite 2 Y position
6	D006	53254		Sprite 3 X position
7	D007	53255		Sprite 3 Y position
8	D008	53256		Sprite 4 X position
9	D009	53257		Sprite 4 Y position
10	D00A	53258		Sprite 5 X position
11	D00B	53259		Sprite 5 Y position
12	D00C	53260		Sprite 6 X position
13	D00D	53261		Sprite 6 Y position
14	D00E	53262		Sprite 7 X position
15	D00F	53263		Sprite 7 Y position
16	D010	53264		Sprites 0-7 X position (ms bit of X coordinate)
17	D011	53265		VIC Control Register:
			7	Raster compare (ms bit)
			6	Extended colour text mode, 0 = disabled, 1 = enabled
			5	Bit -map mode, 0 = disabled, 1 = enabled
			4	Blank screen to border colour, 0 = blanked, 1 = normal
			3	Display 24/25 row text, 0 = 24 rows, 1 = 25 rows
			2-0	Smooth scroll to Y dot position
18	D012	53266		Read/write raster value
19	D013	53267		Light pen X position
20	D014	53268		Light pen Y position
21	D015	53269		Sprite display enable/disable
			7-0	0 = disable, 1 = enable
22	D016	53270		VIC Control Register
			7-6	Unused
			5	Reset bit, 0 = normal status, 1 = stopped
			4	Multi-colour mode, 0 = disabled, 1 = enabled
			3	Display 38/40 column text, 0 = 38 columns, 1 = 40 columns
			2-0	Smooth scroll to X dot position

No	Address		Bits	Description
	Hex	Decimal		
23	D017	53271	7-0	Sprite vertical expansion 0 = normal, 1 = expanded
24	D018	53272	7-4	VIC Memory Control Register Video matrix base address
25	D019	53273	3-0	Character dot data base address
			7	VIC Interrupt Flag Register VIC interrupt flag, 0 = no interrupt, 1 = interrupt
			6-4	Unused
			3	Light-pen interrupt, 0 = none, 1 = interrupt detected
			2	Sprite to sprite collision, 0 = none, 1 = collision detected
			1	Sprite to background collision, 0 = none, 1 = collision detected
			0	Raster compare, 0 = not matched, 1 = matched
26	D01A	53274	7	VIC Interrupt Enable Register 1 = set interrupt enable for..
			6-4	Unused
			3	light-pen, 0 = disable, 1 = enable
			2	sprite to sprite collision, 0 = disable, 1 = enable
			1	sprite to background collision, 0 = disable, 1 = enable
			0	raster match with D012, 0 = disable, 1 = enable
27	D01B	53275	7-0	Sprite to background priority 0 = background, 1 = sprite
28	D01C	53276	7-0	Sprite multi-colour mode 0 = standard mode, 1 = M.C.M.
29	D01D	53277	7-0	Sprite Y expansion 0 = normal, 1 = expanded
30	D01E	53278	7-0	Sprite to sprite collision detect 0 = do not detect, 1 = detect
31	D01F	53279	7-0	Sprite to background collision detect 0 = do not detect, 1 = detect
32	D020	53280		Border colour
33	D021	53281		Background colour (text)
34	D022	53282		Background colour 1
35	D023	53283		Background colour 2
36	D024	53284		Background colour 3

No	Address		Bits	Description
	Hex	Decimal		
37	D025	53285		Sprite multi-colour register 0
38	D026	53286		Sprite multi-colour register 1
39	D027	53287		Sprite 0 colour
40	D028	53288		Sprite 1 colour
41	D029	53289		Sprite 2 colour
42	D02A	53290		Sprite 3 colour
43	D02B	53291		Sprite 4 colour
44	D02C	53292		Sprite 5 colour
45	D02D	53293		Sprite 6 colour
46	D02E	53294		Sprite 7 colour

## SID (Sound Interface Device) Registers

### Voice 1 Registers

No	Address		Bits	Description
	Hex	Decimal		
0	D400	54272		Frequency control - low byte
1	D401	54273		Frequency control - high byte
2	D402	54274		Pulse waveform width - low byte
3	D403	54275		Pulse waveform width - high byte
			7-4	Unused
4	D404	54276	3-0	Bits 11-8 of waveform width
				Voice Control Register
			7	Random noise waveform, 0 = not selected, 1 = selected
			6	Pulse noise waveform, 0 = not selected, 1 = selected
			5	Sawtooth noise waveform, 0 = not selected, 1 = selected
			4	Triangle noise waveform 0 = not selected, 1 = selected
			3	Test bit for oscillator, 0 = normal, 1 = disable
			2	Ring modulate with oscillator 3 0 = don't modulate, 1 = do
			1	Synchronise with oscillator 3 0 = don't synchronise, 1 = do
			0	Gate bit, 0 = start release, 1 = start attack
5	D405	54277		Envelope Attack/Decay control
			7-4	Attack duration
			3-0	Decay duration
6	D406	54278		Envelope Sustain/Release control
			7-4	Sustain level
			3-0	Release duration

### Voice 2 Registers

7	D407	54279		Frequency control - low byte
8	D408	54280		Frequency control - high byte
9	D409	54281		Pulse waveform width - low byte
10	D40A	54282		Pulse waveform width - high byte
			7-4	Unused
			3-0	Bits 11-8 of waveform width

No	Address		Bits	Description
	Hex	Decimal		
11	D40B	54283	7	Voice Control Register Random noise waveform, 0 = not selected, 1 = selected
			6	Pulse noise waveform, 0 = not selected, 1 = selected
			5	Sawtooth noise waveform, 0 = not selected, 1 = selected
			4	Triangle noise waveform, 0 = not selected, 1 = selected
			3	Test bit for oscillator, 0 = normal, 1 = disable
			2	Ring modulate with oscillator 1, 0 = don't modulate, 1 = do
			1	Synchronise with oscillator 1, 0 = don't synchronise, 1 = do
			0	Gate bit, 0 = start release, 1 = start attack
12	D40C	54284		Envelope Attack/Decay control
13	D40D	54285	7-4	Attack duration
			3-0	Decay duration
			7-4	Envelope Sustain/Release control
			3-0	Sustain level Release duration

### Voice 3 Registers

14	D40E	54286		Frequency control - low byte
15	D40F	54287		Frequency control - high byte
16	D410	54288		Pulse waveform width - low byte
17	D411	54289		Pulse waveform width - high byte
			7-4	Unused
			3-0	Bits 11-8 of waveform width

No	Address		Bits	Description
	Hex	Decimal		
18	D412	54290	7	Random noise waveform, 0 = not selected, 1 = selected
			6	Pulse noise waveform, 0 = not selected, 1 = selected
			5	Sawtooth noise waveform, 0 = not selected, 1 = selected
			4	Triangle noise waveform, 0 = not selected, 1 = selected
			3	Test bit for oscillator, 0 = normal, 1 = disable
			2	Ring modulate with oscillator 2 0 = don't modulate, 1 = do
			1	Synchronise with oscillator 2 0 = don't synchronise, 1 = do
			0	Gate bit, 0 = start release 1 = start attack
19	D413	54291	7-4	Envelope Attack/Decay control
			3-0	Attack duration
			3-0	Decay duration
20	D414	54292	7-4	Envelope Sustain/Release control
			3-0	Sustain level
			3-0	Release duration

### SID General Registers

21	D415	54293	7-3	Filter cutoff frequency low byte		
			2-0	Unused		
22	D416	54294		Filter cutoff frequency high byte		
23	D417	54295	7-4	Filter resonance control		
			7-4	Resonance level		
			3	External input, 0 = don't filter, 1 = filter		
			2	Voice 3 output, 0 = don't filter, 1 = filter		
			1	Voice 2 output, 0 = don't filter, 1 = filter		
			0	Voice 1 output, 0 = don't filter, 1 = filter		



No	Address		Bits	Description
	Hex	Decimal		
23	D418	54296	7	Filter mode/Volume
			6	Cut-off voice 3, 0 = normal, 1 = cut-off
			5	Select high-pass mode, 0 = don't select, 1 = select
			4	Select band-pass mode, 0 = don't select, 1 = select
			3-0	Select low-pass mode, 0 = don't select, 1 = select
25	D419	54297		Output volume
26	D41A	54298		Analogue to digital conversion 1 (game paddle 1)
27	D41B	54299		Analogue to digital conversion 2 (game paddle 2)
28	D41C	54300		Oscillator 3 Random Number Generator Output
				Envelope Generator 3 Output

Note: Registers 0 to 24 (54272 to 54296, hex D400 to D418) are write-only, 25 to 28 (54297 to 54300, hex D419 to D41C) are read-only.



# INDEX

(H)=Honey.Aid

## A

ABS function 10-6  
Accumulator 10-13  
Addition 1-5  
ADSR 3-12, 9-4, 9-39 et.seq., 9-47  
AND operator 2-26, 6-7  
APPEND(H) A2-2  
Argument 2-2  
Arithmetic expressions 1-8  
Arrays 5-11, 6-21  
ASC function 5-18  
ASCII character codes A3  
Assembly Language 10-13  
ATN(Arc-TaNgent) function 10-5  
Attack 3-12  
AUTO(H) 3-16, A2-2  
Auto-repeat on keys 6-22

## B

Balltrap game 7-20 et.seq.  
BASIC 1-5  
    numeric functions 10-1 et.seq.  
    variables 5-11, 6-21  
Binary arithmetic A1  
Binary Coded Decimal A1  
Bit 6-2  
Bitwise operation 6-9  
Blockade game 7-27 et.seq.  
Breakout game 7-1 et.seq.  
BRK(H) A2-2  
Bug 2-18  
Buildasound 9-8, 9-32  
Byte 1-8, 6-2

## C

Cassette Buffer 10-14  
CBM(H) A2-3  
CHANGE(H) 3-20, A2-2  
CHR\$ codes 6-21, A3  
CHR\$ function 5-17  
CIRCLES 10-1  
CLR statement 1-7  
CLR/HOME key 1-4  
Clock 5-1, 6-15  
Colon (:) 2-16

Color  
   adjustment 1-4  
 CHR\$ codes A3  
   (H) A2-10  
   keys 1-4, 3-14  
   PEEK and POKE 7-2, 7-10  
   RAM 6-14,7-2  
   screen and border 5-6 et.seq., 7-1 et.seq.  
 Columns on screen(40) 4-9  
 Commodore key 1-3  
 Comparisons (<,>,>=<=<>) 2-13,2-14,2-25  
 Compositune 9-1 et.seq  
 Control variable 2-9  
 Concatenation of strings 5-3  
 Conditional statements 2-4  
 CONT command 7-8  
 CTRL key 1-4,1-10  
 COSine function 10-2  
 CRSR (CuRSOR) keys 1-4,2-27  
 Correcting errors 1-13  
 Cursor 1-2,1-4,2-27

## D

Data pointer 4-3  
 DATA statement 4-1  
 Decay 3-12  
 Defaults(H) A2-19  
 DEFine statement 10-7  
 Delay loop 4-17, 5-9  
 DELETE(H) 6-26, 6-33, A2-5  
 DELEte key 1-2, 1-15  
 Delimiters(H) 3-19, A2-4  
 DIMension statement 5-12  
 Direct entry commands 1-9  
 Disk 2-23  
 Dummy variables 10-8

## E

Editing programs 1-2, 1-13, 2-9  
 END statement 4-20, 6-27, 6-29, 6-30  
 Envelope 9-39 et.seq., 9-49  
 ENVELOPE(H) 3-12, A2-14  
 Error messages 1-2  
 Etcha-sketcha game 3-5  
 Expanded Sprites 8-3  
 EXPonent function 10-6  
 Exponentiation 10-3, 10-6  
 Expression, arithmetic 1-8  
 EXTEND(H) A2-5

F  
False 6-8  
Filters 9-58  
FIND(H) 3-18, A2-5  
Flags 4-9  
Flow charts 2-5  
FN Statement 10-7  
FOR statement 2-8  
FRE function 6-5  
Frequencies of notes 9-5, 9-36  
Function key codes 6-21  
Functions 10-1, 10-7

G  
Gate bit 9-41  
GET statement 3-6  
GOSUB stack 4-10  
GOSUB statement 4-10  
GOTO (GO TO) statement 2-3  
Graphics 6-1 et.seq  
Graphic symbols A3  
Greater than (>) 2-8, 2-13

H  
Hangman 4-1, 10-10  
Hexadecimal A-1  
HIRES(H) 3-2, A2-11  
HMEM A2-6  
Honey.Aid 3-1 et.seq., A2

I  
IF...THEN statements 2-4, 2-13  
Immediate mode commands 1-9  
Initialising SID 9-3  
INPUT statement 1-9, 1-13  
INST(INSerT) key 1-15  
INTEger function 2-2  
Interactive 7-1

J  
Jiffy clock (TI) 5-1, 5-5, 5-10

K  
Keyboard 1-1  
Keyboard buffer 5-10, 6-5, 6-28  
Keyboard on/off 6-10  
KILL(H) 3-1, A2-7

## L

LEFT\$ function 4-5  
LEN function 4-7  
Less than (<) 2-8,2-13  
LET statement 1-6  
LINE(H) 3-3, A2-12  
LIST command 1-10  
LMEM(H) A2-7  
LOGarithm function 10-6  
Logical operation 6-7  
Loops 2-3,2-8  
Loop variable 2-9  
Lower case characters 1-3

## M

Machine code 6-12,10-13  
Machine language 10-13  
Mathematical expressions 1-8  
Memory locations  
    SID 9-1, 9-2  
    sprites 8-3  
Memory mapped screen 7-1  
MID\$ function 4-5  
MOBs 8-1  
Module Testing 9-25,9-28  
Moveable Object Blocks (MOBs) 8-1  
Multi-statement lines 2-16  
Music 9-1 et.seq.

## N

Names  
    program 2-23  
    variable 1-7  
Nested subroutines 7-25  
NEXT statement 2-8  
Note  
    duration 9-4  
    frequencies 9-5, 9-36  
    PEEK values 9-26  
Not equal to (<>) 2-25  
NRM(H) 3-5, A2-12  
NUMBER(H) 3-17, A2-7  
Numeric variables 1-6, 1-7  
Nybble 9-44

**O**  
 OLD(H) 3-18, 2-8  
 ON statement -2<sup>1</sup>  
 Operators  
   arithmetic 1-5  
   logical 2-25, 2-26  
   relational 2-13, 2-14, 2-25  
 OR 2-25

**P**  
 Parameters 10-8  
 PEEK function 5-7, 7-10  
 PEEK value of notes 9-26  
 Pi 10-3  
 Pitch 9-2, 9-5, 9-36  
 Pixel 6-1  
 PLAY(H) 3-14, A2-15  
 Play-back of tune 9-30  
 PLOT(H) 3-2, A2-13  
 Pointer  
   Data 4-3  
   Top of Memory 10-16  
 POKE codes A3  
 POKE statement 5-6, 7-2  
 PRINT statement 1-2, 1-5, 1-6, 1-10  
 Program  
   editing 1-13  
   line numbering 1-9  
   loading/saving (disk) 2-23  
   names xi, 2-23  
 Prompts 1-13  
 PULSE(H) A2-17

**Q**  
 Quadratic equations 10-7  
 Quotation marks  
   as delimiters(H) 3-19, A2-4  
   printing onto screen 6-27

## R

Radians 10-3  
RaNDom function 2-1  
Random numbers 2-1, 10-  
Reaction Tester 5-1 et.seq.  
READ statement 4-1  
Release 3-12  
Relocating character set 6-4  
REMark statement 2-22  
REPEAT(H) A2-9  
Repeat on keys 6-22  
RESET(H) A2-9  
RESTORE statement 4-3  
Resonance 9-62  
Return key 1-2, 1-5  
RETURN statement 4-10  
RIGHT\$ function 4-5  
ROM 4-1  
Rows on screen (25) 4-9  
RND (RaNDom) function 2-1  
RUN command 1-9  
RUN/STOP key 2-3  
RVS OFF key 1-4  
RVS ON key 1-4

## S

SAVE command 2-23  
Saving programs (disk) 2-23  
Screen border utility 10-16  
Screen columns (40) 4-9  
Screen POKEing 7-1 et.seq.  
Scrolling screen 1-10  
Self modifying programs 6-27  
Separators 1-11,1-12,1-13  
SGN function 10-6  
Shift key 1-2  
SINE function 10-2  
Sixteen bit arithmetic 9-2  
Sound 9-1 et.seq.,  
SOUND(H) 3-10, A2-17  
Sound effects 10-10  
SID 9-1  
Sound Interface Device 9-1  
Space symbol 1-11, 4-17  
Sprite  
  graphics 8-1 et.seq.  
  memory 8-3  
  multicolor 8-4  
  precedence 8-19



SQR(Square Root function) 10-2  
 Stack, GOSUB 4-10  
 Stave 9-19  
 STEP 2-9  
 STOP command 2-4  
 STOP key 2-3  
 Storing programs on disk 2-23  
 String concatenation 4-13, 5-3  
 Strings 1-8  
 String variables 1-8  
 STR\$ function 10-9  
 Subroutines 4-10  
 Subroutine testing 9-25, 9-28  
 Subscripted variables(Arrays) 5-11, 5-12  
 Sustain 3-12  
 Syntax error 1-2  
 SYS statement 10-15, A2-7

## T

TAB function 4-9  
 TANgent function 10-4  
 Tempo 9-22  
 TEMPO(H) 3-16, A2-18  
 Testing routines 9-25, 9-28  
 THEN 2-4, 2-6  
 TI variable 5-1, 5-5, 5-10  
 TI\$ variable 5-1  
 Time clock 5-1 et.seq.  
 Tone color of note 9-4, 9-50  
 Top of memory pointer 10-16  
 TO statement 2-9  
 True 6-8

## U

Upper/Lower Case mode 1-3  
 User defined functions 10-7  
 User defined graphics 6-1 et.seq

## V

VALue function 10-9  
Variables 1-7  
  array 5-11  
  control 2-9  
  dimensions 5-12  
  dummy 10-8  
  integer 2-2  
  loop 2-9  
  names of 1-7  
  numeric 1-7, 1-8  
  string 1-8  
VERIFY command 2-24  
VIC II chip 6-12  
Voices 3-10, 9-1, 9-5  
VOL(H) A2-18  
Volume 9-3, 9-39

## W

WAVE(H) A2-19  
Waveforms 9-50  
Write only registers 9-14, 9-46

## SYMBOLS

: (Colon) 2-16  
  (Exponentiation) 10-3, 10-6  
> (Greater than) 2-8, 2-13  
< (Less than) 2-8, 2-13  
>= (Greater than or equal to) 2-14  
<= (Less than or equal to) 2-14  
<> (Not equal to) 2-25  
  (Pi) 10-3  
  (space) 1-11, 4-17



# Commodore 64<sup>®</sup> BASIC Programming

**Peter Holmes & Derek Bush**

Offers the beginning programmer a complete course in Commodore 64 BASIC. In addition to the book, the programs on disk give the user greater control in creating graphics and sound programming capabilities. Features four invaluable programs:

- Honey.Aid—a unique extension to Commodore 64 BASIC that adds 27 new statements to the language, including automatic line numbering, search and replace, plotting lines, changing tempo, and more.
- SPRITE.GEN—allows you to create sprites easily and display your sprite data at the touch of a single key.
- CHAR.GEN—allows you to easily redesign the Commodore character set.
- COMPOSATUNE—allows you to create a song and display the data for that song at the touch of a single key.

In addition to these programs, there are also games programs such as Hangman (with color and screen display techniques), Breakout, Balltrap, Blockade, and more. The book also includes a binary/hexadecimal tutor.

Requires: A Commodore 64, a TV (color preferred), and one disk drive.

*Also of interest . . .*

## Commodore 64<sup>™</sup> Assembly Language Programming

Peter Holmes and Derek Bush

Successfully master the challenge of assembly language programming. Includes a full-featured assembler plus a complete self-instructional course.

#7620, Book/Software (Tape). #7623, Book/Software (Disk).



**HAYDEN BOOK COMPANY**

a division of Hayden Publishing Company, Inc.  
Hasbrouck Heights, New Jersey

ISBN 0-8104-7630-4