THE COMPLETE GUIDE FOR THE COMMODORE 64

SPENCER·BATESON

THE COMPLETE GUIDE FOR THE COMMODORE 64

SPENCER BATESON

First published in Great Britain in 1984 by
PHOENIX PUBLISHING ASSOCIATES
14, VERNON ROAD, BUSHEY, HERTS. WD2 2JL

ISBN 0 9465 7621 1

Printed in Great Britain by
Billing and Sons Limited, Worcester
Cover Design by
Ivor Claydon Graphics
Typesetting by Prestige Press (UK) Ltd, Chesham
and Sunsetters, Rickmansworth
Production by Denis Gibney Graphics, Chesham

CONTENTS

To my truly wonderful mother, Sally.

INTRODUCTION

Essentially we have set out to create a book that will serve as both an introduction for the newcomer and a source of reference material for readers with some programming experience.

The decision to produce a volume centred around these rather ambitious objectives was prompted by one of the more curious mysteries of popular microcomputing. There can be little doubt that the Commodore 64 has been and will continue to be a huge commercial success. However, the indisputable qualities of the 64 (specifically its sound and graphics potential) can only begin to be investigated by programmers with a firm grasp of the **BASIC** dialect used by Commodore machines. This being the case, it is difficult to understand why the documentation provided with the machine (the User Manual) offers such a cursory introduction to the methods of harnessing the 64's power.

In short we have written this book in the hope that it will be used to supplement the information provided by the 64's manual.

Such a project has inevitably resulted in some compromises. The sound and graphics potential of the Commodore 64 is something close to spectacular, but mastering these facilities requires a relatively advanced understanding of how computers perform their miracles. Consequently we have devoted a large percentage of the text to these subjects in an effort to provide 64 owners with an introduction to sound and graphics that will hopefully make up for the many omissions and ambiguities of the User Guide.

Although the commands, functions and structures of 64 **BASIC** are itemised in the User Guide, no attempt has been made to provide anything like a systematic guide to learning the language. Thus our opening chapters provide a brief **BASIC** tutorial, followed by a fairly lengthy section explaining each of the individual **BASIC** keywords in detail.

So how have we compromised? Well, by attempting to cram an introduction to a computer language into five short chapters we have had to skimp somewhat on the chatty asides and flashy but irrelevant programs that characterise many of today's introductions to popular micros. We have had to restrict ourselves to short program examples which attempt to demonstrate a command or principle as clearly and simply as possible.

Equally, we have made no attempt to pretend that the complexities of sound and graphics can be mastered at the drop of a hat. Since Commodore **BASIC** offers no customised sound and graphics commands, the 64 programmer needs a clear understanding of the way in which the machine creates its sounds and generates its graphical displays before the methods of accessing these facilities can make sense. We want this handbook to be of enduring value to you and have consequently struggled to provide as much background material as is digestible in a book of this size.

If the tone of this introduction appears a trifle daunting, please accept our apologies. It is our firm conviction that programming computers is one of the most engrossing and consistently satisfying activities open to us humans. Computers are fun, and you couldn't do better than start your computing life on a Commodore 64. If we sound somewhat missionary in this statement of intent, it's simply an over-reaction to many of the introductions to programming that are currently cluttering the bookshops. Many of today's writers try to convince their readers that learning to program a computer is as simple as plugging in a games cartridge. It is not! Neither is it the magical mystery that the old school mainframe programmers made it out to be.

Despite any appearances to the contrary, these are meant as words of encouragement. If this is your first encounter with **BASIC**, work your way methodically through the first five chapters. Try all the examples.

Invent some of your own, and if you don't understand something don't move on and don't give up! Go back over the programs and explanations again (and again) until you've cleared up any problems. Refer to the keyword section (Chapter 5) if you don't understand a **BASIC** command, structure or a principle related to the command. **BASIC** is a tried and tested learning language, and certainly one of the least painless introductions to the micro era. If you get stuck don't blame the book (touch wood!), your micro (the 64's got a pretty good track record), or least of all yourself. You're almost certainly grappling with a concept you've never encountered before, so give yourself time. With a little determination all will become clear.

Enough words of wisdom, let's make a start. Just keep it in mind that if you play your cards right computers and programming will become a way of life. Humans may lose their appeal, but why worry? You'll never be alone now you own a 64!

Do not be confused by the use of quote marks in the sample programs which have been typeset. They will appear as " " and not as the" " which you will find on your computer keyboard. This is simply a peculiarity of the typeface we have used for this book.

You will also see that the exponential signs will appear in two different forms but, again, this is because different typefaces have been used. You will see ↑ and ^ but they both mean the same thing in programming terms.

CHAPTER ONE
THE SET-UP

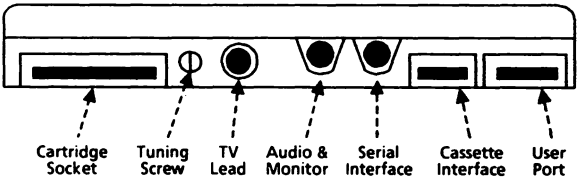
As you have undoubtedly realised, in order to complete your basic microcomputer system your 64 needs to be connected to some kind of VDU (Visual Display Unit) such as a television. If possible, you should try to lay your hands on one that can be permanently linked to your 64. There are a number of reasons why this somewhat luxurious situation is desirable. Apart from the obvious advantages of avoiding conflict with the other members of the family who are more interested in soap opera than computing, there are other, slightly more serious considerations.

The aerial socket at the back of your TV was not designed to have plugs constantly pushed in and pulled out of it. With a little care (and a total absence of brute force), damage to connectors can probably be avoided, but a permanent micro-workstation will ensure that problems of this nature simply do not arise.

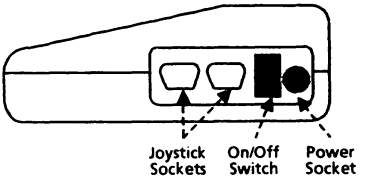
Like most micro systems, the basic Commodore 64 system is very easy to put together. The I/O (Input/Output) facilities on the machine are relatively robust and clearly identified in the *User Guide*. You'll only run into difficulties if you try forcing connections without first bothering to discover what you're plugging where!

Unfortunately not all the 64's I/O facilities (the sockets on the back) are labelled on the machine itself, so when you're just

starting it might be wise to avoid mistakes by making use of the diagrams below.



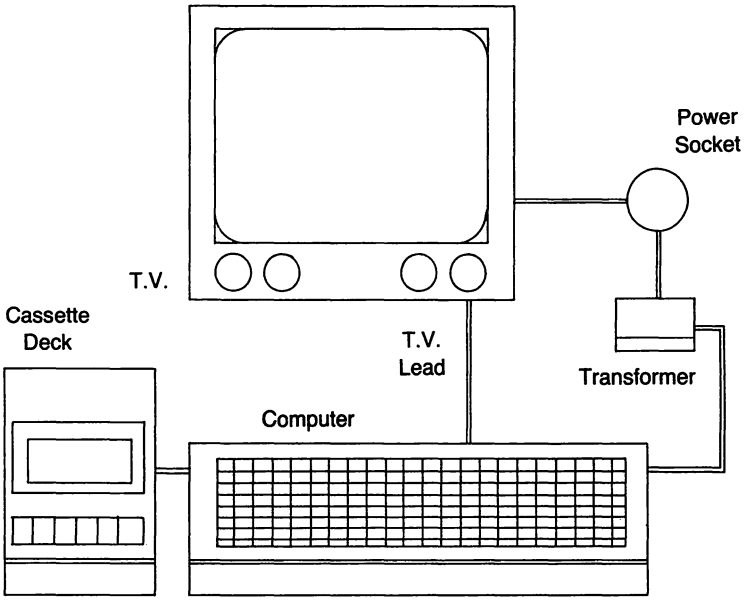
The Commodore 64 Connectors – Rear View



The Commodore 64 Connectors – Side View

Before going on to provide a step-by-step guide to setting up your system, this is as good a time as any to say a few words about the cassette recorder which makes up the final component of your basic system. Now, like most micros, the section of memory which stores the programs you key in is what is known as 'volatile'! This means that for as long as your 64 is powered up this part of memory will hold the program or code in question, but once the machine is turned off the program will be lost. Now, it is obviously impractical to type in a program every time you want to use it (a fifty line program will take you the best part of an hour to key in), so some method of 'off-line' storage is required, and the 64 has facilities which allow you to store information onto either magnetic tape or disc. Although the latter is far more reliable and efficient, at this stage we will assume that you are going to use the 64's cassette storage facilities which have the advantage of being considerably cheaper than using a disc drive.

The following diagram shows a rather idealised version of a basic 64 system:



The power pack that comes with your machine converts the household power supply into the low DC voltage required by the micro. It should be plugged into the power socket on the right-hand side of the 64 and the power pack attached to the mains.

In the best of all possible worlds a good quality colour portable TV should play the part of the VDU in your system. However, since we have already established the advantages of using a TV dedicated to the system, it is quite likely that you will start your computing life using a second-hand black and white portable.

A TV lead is supplied with the system, attach one end to the aerial socket of your TV and the other to the second port on the left at the rear of your micro.

Unlike many microcomputer manufacturers, Commodore use high quality components throughout their system, which means that if a plug doesn't easily fit you're almost certainly doing something wrong. Commodore have also made it their policy to produce custom made components for the complete system. Thus, the only cassette recorder that you can use directly with your micro is the one manufactured by Commodore which, although slightly more expensive than the cheapest domestic machines, is extremely reliable. The recorder plugs directly into the cassette port, which is second from the right at the rear of the machine. Apart from its proven compatability with the 64, the recorder has the added advantage of not requiring an independent power supply as it takes its power directly from the micro.

THE 64 KEYBOARD

In the last section of this chapter we will take a close look at the intricacies and layout of the keyboard. The keys are laid out in the standard QWERTY typewriter format, so if you are accustomed to using a typewriter you won't find typing on the 64 a problem.

Try typing a few words and characters and press the <**RETURN**> key (which tells the computer that you have finished the line). You will probably be confronted with a screen displaying the characters that you have keyed in and ?SYNTAX ERROR with READY underneath. This will be fully explained in the following chapter, for now suffice to say that all micros are fussy about the information that they will accept. Nothing that you can type in will actually hurt the machine, so should anything untoward happen, such as the keyboard not responding when you press the keys, hit the **RUN/STOP** key and the **RESTORE** keys simultaneously or switch the 64 off and then on again and the keyboard response will be returned.

Up until now all the letters that you have entered have been in uppercase (capitals). To access the lower case set press the **SHIFT** and Commodore key simultaneously. *(The Commodore key is bottom left of the keyboard bearing the Commodore logo.)* On doing this you will notice that the entire screen has changed to lower case. Try typing in a few more characters in this mode, pressing the shift key to access capitals. When you return to the capitals mode (by pressing the **SHIFT** and Commodore key) you will notice that all the characters that were **SHIFT**ed, in order to create capitals when you were in lower case, radically change their appearance to totally incomprehensible symbols.

On the front of the majority of the keys are a couple of strange symbols, these will be fully explained later on in the book, but let's take a quick look at how they are accessed. The symbol on the right-hand side of the key is obtained by simultaneously pressing **SHIFT** and the key in question (thus the strange symbols we just witnessed) and the left-hand character is accessed by pressing the Commodore key at the same time as the desired key.

We will now run through all the labelled keys one by one giving a detailed explanation of their function(s).

ARROW OR CURSOR KEYS

These keys are found below the **RETURN** key, at the bottom right-hand corner of the QWERTY keyboard and are used to move the cursor around the screen in the direction indicated by the arrow. By **SHIFT**ing either key the cursor moves in the direction of the top arrow. So if you **SHIFT** the key on the right the cursor will move left and the cursor will move up the screen if the left-hand key is **SHIFT**ed.

CLR/HOME

This key, situated next to the **INST/DEL** key at the top right of

the keyboard, has two functions. When you press it on its own it **HOME**s the cursor, returning the **PRINT** position to the top left of the screen. If it is pressed in conjunction with the **SHIFT** key, the screen is totally cleared and the cursor is, once again, **HOME**d to the top of the screen.

COMMODORE KEY

This key is at the bottom left-hand corner of the keyboard and bears the Commodore logo. As we have already seen it has two uses: either to access the left-hand symbol on the front of the keys, or, when used in conjunction with **SHIFT**, to obtain lower case characters.

CTRL

CTRL (Control) is always used in conjunction with another character and can produce many interesting effects. Try pressing the **CTRL** key and one of the colour keys (1-8) and you will see that the cursor changes colour and anything that you key-in will now appear in this colour. Now try pressing **CTRL** and **9** followed by any character(s) and the display will be reversed. To turn the reverse effect off press **CTRL** and **0** together.

FUNCTION KEYS

Standing on their own on the far right of the **64** are four keys, labelled **f1** through to **f8**. These are the function keys. Once you have learnt how to program, these keys will come into their own since you can use them to define your own customised functions. You can do anything from getting your mothership to shoot at the oncoming aliens to printing out a listing.

INST/DEL

This key is situated in the top right-hand corner of the QWERTY KEYBOARD. It is used to edit any text on the screen, either **DEL**eting characters or, when **SHIFT**ed, **INS**er**T**ing them. Type in some characters and press <**RETURN**>. Ignore the ?SYNTAX ERROR message and using the arrow keys move the cursor back to the beginning of the line. Press the **INST/DEL** key and **SHIFT** simultaneously and you will see that a space has been created, allowing you to **INS**er**T** a single character. By pressing **INST/DEL** and **SHIFT** twice you will create two spaces and so on. Move the cursor along the line using the arrow keys until it is over the character to the right of the one you want to remove. Press **INST/DEL** (without using the **SHIFT** key) and the appropriate character will be deleted. It is advisable to familiarise yourself with this key as you will find that you use it all the time!

RETURN

The **RETURN** key does a similar job to the carriage return key on a typewriter, returning the cursor to the left-hand margin. However, its most important function is to either implement a command or consign **BASIC** statements to memory, depending on whether the code in question is a direct command or a program line. The full use of **RETURN** is explained in the following chapter.

RESTORE

This key is always used in conjunction with the **RUN/STOP** key. It is used to break into a program and reset the screen back to normal. For instance if a program stops and the 64 seems to have 'hung up', pressing these keys together will return the machine to a usable state without losing any program that is in the memory.

RUN/STOP

This key is found on the left-hand side of the keyboard above the Commodore key.

As we have already seen, when used in conjunction with **RESTORE** it resets the 64. If it is used un**SHIFT**ed it will **STOP** the execution of a program. Its third function, when **SHIFT**ed, is to **LOAD** a program and then **RUN** it automatically.

SHIFT AND SHIFT LOCK

As we have already seen, these keys are used either to access the right-hand symbols on the front of the keys or, to access capitals when in the lower case mode.

The final point that remains to be made whilst we are on the subject of the keyboard is the different arithmetical operators used by micros. Most of them follow standard notation but a couple take a slightly different format. For example, instead of using the standard multiplication sign (x) an asterisk is used (*), hence avoiding any confusion with an x in a program. A complete list of all the arithmetical operators follows:

()	Brackets
*	Multiplication
/	Division
+	Addition
–	Subtraction
^	Exponentiation (raising to the power of)
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

CHAPTER TWO
BASIC COMMUNICATION

So your fingers are poised over the keyboard, and your brand new 64 is **READY** for action, but how do you communicate your needs to the willing but silent machine? Well, in order to provide an answer to this loaded but perfectly reasonable question, we must first dispel an ancient computer myth.

If you consider your Commodore 64 as an up-market calculator you will be far closer to the truth than if you set it on a pedestal as some sort of super-intelligent mastermind. Like calculators, in the final analysis, all computers are primarily concerned with the processing of numeric data. They perform operations on numbers at fantastic speeds, and this is the quality that lies at the heart of their power, and often gives them a semblance of intelligence.

Essentially then, computers have the ability to perform simple tasks very efficiently. The problem that we humans are faced with is how to communicate our needs to a machine in a language that it will be able to understand. There are two languages available to 64 users which enable us to communicate with our micro. One is known as a machine code language, and a little later we shall be taking a look at the version that can be used on the Commodore 64. However, there is a far simpler alternative available for the first time programmers and this language is called **BASIC** (which stands for Beginner's All-purpose Symbolic Instruction Code).

BASIC, as its name implies, is a learning language, and it enables us to issue the 64 with instructions using a vocabulary that is very close to the English language. Thus, it is very easy to translate a human demand into a **BASIC** instruction. As you probably know, a collection of these **BASIC** instructions entered in a numbered list and stored in the 64's memory is known as a program. The next few chapters will provide you with an introduction to programming on your Commodore 64.

One of the first lessons you learn (usually the hard way), when you start programming is that computers are incredibly fussy about the kind of information they will deal with and the manner in which it is presented. Micros are painfully literal minded, and unless an instruction is presented in precisely the right way it simply will not be processed. (If you take a look at the Error Message appendix at the back of this book you'll get an idea of the number of different ways that the 64 can say no!). So whilst **BASIC** is both easy to understand and use, its grammar must be rigorously respected if your demands are to be understood by the micro. Like the process of learning any language, it is unlikely that you'll be especially impressed by any individual step on the path to mastering **BASIC**. However, by the end of this section you will hopefully have reached the same conclusions about **BASIC** as Shakespeare did about English – the value of the language far exceeds the sum of its parts!

MAKING A STATEMENT

One of the central concepts behind the language is that it should be used as it is learned, which is why you should have your 64 at hand while you read through this section of the book. Try all the examples and experiment with each new command as it is introduced. There is very little that can harm the machine, so if you can think of a variation to the format of a command try it out and see what happens. At worst, the 64

will 'hang-up' on you (the keyboard will not respond to data you enter), in which case simply switch off the machine and try again.

The time has come to learn our first **BASIC** keyword. There are 57 such words in Commodore **BASIC**, and it seems sensible to kick off with one of the most important – **PRINT**. As one might assume, **PRINT** is the command that enables us to **PRINT** characters to the screen. It can be used in a variety of formats that determine how and what gets **PRINT**ed where. Let's start with the simplest form of the **PRINT** statement. So that we can see exactly what's going on we must clear the screen by using the procedure we discussed in the last chapter: press the **SHIFT** and **CLR/HOME** keys simultaneously. (If you have any problems using the 64 keyboard refer back to the previous chapters). Now type in the following statement exactly as it appears below:

```
PRINT "HI THERE"
```

and then press the <**RETURN**> key. Unless you have made a typing error, you'll find that the message in quotes, HI THERE, has been **PRINT**ed on the line below the one you entered. Below the message there should be a blank line, and under this should appear the READY prompt. The prompt appears each time the 64 successfully completes an instruction (or a series of instructions), and is there to let you know that the machine is awaiting further commands. Since we have already noted how fussy computers are about the format of a command, let's try a few variations and see what kind of response we get from the machine. Try typing in the following:

```
PRINT HI THERE
```

or

```
PRINT 'HI THERE'
```

In the first instance, we have tried to **PRINT** a list of characters without the double quotes. The 64 will consider these characters to be a *numeric variable* (a name that represents a number), and searches its memory to discover which number has been given the name **HI THERE**. Since it is unable to find any number with this name, it will simply **PRINT** a zero to the screen. We'll worry about variables a little later; the important point here is that because we didn't follow the rules, the 64 wasn't able to carry out the task we intended it to accomplish. It **PRINT**ed a 0 instead of our message. Our second attempt at modifying the **PRINT** statement has thrown up our first error message. Instead of **PRINT**ing the message we entered, the 64 has displayed ?SYNTAX ERROR on the screen. This is because although our statement uses quotes, it uses the wrong kind of quotes (single instead of double quotes). ?SYNTAX ERROR tells us that although the computer has received an instruction, it is not in a form it recognises.

So now we know how to **PRINT** letters to the screen. But what about numbers? Type:

```
PRINT "8-4"
```

and press <**RETURN**>. Once again our **PRINT** statement has followed our instructions, 8-4 has been displayed on the screen. However, because we have put the numbers in quotes, the 64 has treated them in the same way as it treated the letters in our HI THERE message. If we want numbers to be dealt with as a calculation we must omit the quotes. Type in:

```
PRINT 8-4
```

and then <**RETURN**>. Now the result of this numerical **PRINT** statement is displayed and 4 appears on the screen. Each time you want the computer to carry out one of these "direct commands", you must press the <**RETURN**> key.

From now on we'll assume that you will follow this procedure for each complete statement. Let's try another example that uses both of the **PRINT** statements we have looked at so far. Type:

```
PRINT "3+2="3+2
```

This is slightly more complicated, but if we split it into sections we can see how it works. Following the command we have some characters in quotes ("3+2="), which, as we have seen will appear on the screen exactly the way they appear in the statement. After the final quote we have entered a numeric expression, 3+2, which the 64 calculates, then **PRINT**s to the screen. Thus we get 3+2= 5. You'll notice that the computer has included a space before the 5. This is known as a leading space, and there is one before and after every positive value printed by the 64, since it allows room for a minus sign in case we want to **PRINT** a negative value. To test this out try the following:

```
PRINT "4-7="4-7
```

This time when 4-7=-3 is displayed the leading space has disappeared and been replaced by the minus sign (-). Now type:

```
PRINT 2;1;Ø;-1;-2
```

This will display both leading and trailing spaces, i.e.:

```
2  1  Ø -1 -2
```

This statement introduces the semi-colon into our **PRINT** statement. A semi-colon (;) ensures that the characters or calculation that follows it will be **PRINT**ed next to whatever preceded it. In the example we have just typed in, the semi-colon makes sure that the numbers we want displayed are **PRINT**ed individually, rather than their result calculated and then **PRINT**ed. Without the semi-colons we would have to

enter: **PRINT** 2 1 0 -1 -2, which the 64 would calculate as 210-1-2, and thus **PRINT** 207 to the screen. Experiment with the various **PRINT** formats we have encountered so far. Mix statements in quotes with ones without quotes and make sure you understand the results of your research. Then take a deep breath because the time has come to take our first look at how your 64 does its arithmetic!

GETTING TO THE CRUNCH

As we have seen, the 64 is quite happy to be used as an (extremely expensive) calculator. Straightforward addition and subtraction present no problems. For example: **PRINT** 6+4 will display 10 to the screen and **PRINT** -6-2 will offer up a result of -8. Since **PRINT** is one of the commonest **BASIC** keywords, and it can get a little tiresome having to type in the full word each time you want to use it, Commodore have thoughtfully provided us with an abbreviation for the command. Instead of **PRINT** you can use a question mark **(?)**. Thus ? 6+4 and ? -6-2 will produce exactly the same results as the full versions of these statements.

The multiplication process is also much as you might expect, except that like most computers the 64 uses an asterisk (*) instead of a standard multiplication sign. This is to avoid any confusion with an x.

Thus: **PRINT** 5*4 will display 20 on the screen. Division uses a standard slash: **PRINT** 25/5 will return a 5.

If we want to dabble in a little exponentiation (in other words, raising a number to the power of another), then we must use the up arrow (↑). Thus 2 raised to the power of 4 (2*2*2*2) will be calculated by:

```
PRINT 2^4
```

and **PRINT** 16 to the screen.

However, if we want to persuade the 64 to calculate slightly more complex expressions, we have to be careful about the form in which they are presented. For example, with an expression like:

```
PRINT 3+2*5
```

the solution that we come up with depends on which of the two arithmetic operands (+ or *) are considered to have the highest priority. Are we expecting the 64 to calculate the sum of 3 plus 2 and multiply the result by 5 to get 25? Or do we mean 2 times 5 plus 3 which would return an answer of 13? In order to avoid this sort of confusion, the 64 assigns a different priority to each of its arithmetic operators. The table below lists each operator, with the highest priority at the top, the lowest at the bottom:

( )	Brackets
↑	Exponentiation
*=/	Multiplication and division (same priority)
+=-	Addition and subtraction (same priority)

So what does all this mean in practice. Well, any expression contained in brackets will be calculated before anything outside them. Then the 64 will process any calculations involving exponentiation. It will then tackle multiplication and division (assigning an equal priority to each), and finally calculate any addition or subtraction (once again, assigning an equal priority to these operations). Thus, depending on what we actually want the 64 to calculate, the expression could take one of two possible formats:

```
PRINT 3+2*5
```

In which the 2*5 as multiplication has a higher priority than addition, and the result (10) is added to 3 to **PRINT** 13. However, if we wanted to first add 3 and 2 together and then multiply the result by 5 we would have to type:

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

in which 3 will be added to 2 (as brackets have the highest priority) to return 5 which is then multiplied by 5 to **PRINT** 25.

In the case of multiplication/division and addition/subtraction, where two operations have the same priority, the expression will be evaluated from left to right. For example:

```
PRINT 5/2*6
```

will first divide 5 by 2 to get 2.5, and then go on to multiply the result by 6 to **PRINT** 15.

By paying close attention to the order in which the 64 assigns priorities to its arithmetic operations, we can construct dauntingly complex expressions, yet still be confident that the answer we end up with corresponds to the problem we are trying to resolve. For example:

```
PRINT 2^2+(6-4)*2
```

is telling the 64 to first calculate 6 minus 4 (brackets), which then produces an expression to read:

```
PRINT 2^2+2*2
```

then performs the exponential calculation 2 ^ 2, which will give us:

```
PRINT 4+2*2
```

Since the multiplication has the next highest priority, the 64 will next calculate 2*2 which gives us:

```
PRINT 4+4
```

and finally the addition will be performed and an 8 will finally

be **PRINT**ed to the screen. Without an inflexible system of priorities the expression we have just examined could have produced any one of an enormous variety of results, including 8, 12, 32 and 256! We'll take a break from number crunching for a while, and go on to see how we can radically alter the effects of a **PRINT** statement by tinkering with its punctuation.

GETTING INTO PRINT

Having looked at the use of **PRINT** as a direct command (i.e. a command that is entered directly into the 64), we can now go to examine how it can be used within a program. Programs are simply lists of **BASIC** statements whose order of execution is determined by the line numbers that precede them. For example, type in the following statements, pressing the <**RETURN**> key before starting a new program line:

```
10 PRINT "THIS IS LINE 10"
20 PRINT "THIS IS LINE 20"
30 PRINT "THIS IS THE END"
```

Notice that although you have pressed <**RETURN**> after each line, none of the instructions have been processed by the computer. This is because the 64 has noted that there are line numbers before each statement and concluded that you are entering a program rather than a series of direct commands. Your micro now needs to be told to **RUN** the program. Let's not keep it waiting. Beneath the program you have just typed in the following should have appeared:

```
THIS IS LINE 10
THIS IS LINE 20
THIS IS THE END
```

The mechanics of this process should be quite clear. When the program is **RUN**, the 64 searches for the statement with the lowest line number (line 10), executes the command it contains, then moves on to the next lowest number (line 20), executes this statement, and so on until it reaches the end of the program. As our first program demonstrates, each time the 64 encounters a new **PRINT** statement, it will start displaying the specified **PRINT** items on a fresh line. However, if we want to write two **PRINT** statements on different program lines but require the result to be displayed on the same line, we can use the following format:

```
10 PRINT "THIS LINE ";
20 PRINT "IS JOINED TO THE NEXT ";
30 PRINT "WITH A  SEMI COLON (;)"
```

As you can see, after the double quotes in lines 10 and 20 we have added a semi-colon (;). This informs the 64 that the contents of the **PRINT** statement in the line in question must be joined to the content of the next **PRINT** statement encountered. Notice that before the final quote mark in these lines we have included a space (" "), this ensures that our message will not read:

```
THIS LINEIS JOINED TO THE NEXTWITH A  SE
MI COLON (;)
```

In the two short examples above we have used line numbers that go up in steps of 10. We could have just as easily used 1,2 and 3 as line numbers or 5,6 and 7. However, it is standard programming practice to use steps of 10 so that if you need to insert an extra line you have enough space to do so. For example, suppose we write the following piece of code:

```
10 PRINT "ONE"
20 PRINT "TWO"
30 PRINT "FOUR"
```

If we now decide we want to include a THREE amongst our **PRINT** statements, we could simply add:

```
25 PRINT "THREE"
```

Obviously this would not have been possible if our original program had used 1,2,3 as line numbers.

Commas can also be used to effect the format of a **PRINT** statement. If you imagine each **PRINT** line as being divided into four sections, by using commas it is possible to **PRINT** a character at the beginning of each section. The following example should help to clarify this format:

```
10 PRINT 1,2,3,4
```

This line will **PRINT** at the beginning of each of the four **PRINT** fields, with space for nine characters between each of the numbers displayed. Try and use this format to **PRINT** more than four single characters of the screen:

```
20 PRINT 1,2,3,4,5
```

and the fifth character will be **PRINT**ed at the beginning of the first **PRINT** field on the next line. You can use more than one comma between each **PRINT** item:

```
30 PRINT 1,,,4
```

which will format in exactly the same way as line 10, except that two and three will not be displayed.

A more flexible method of determining **PRINT** positions is by adding another keyword to your **PRINT** statements. **TAB** works in much the same way as the **TAB** function on a typewriter. It takes the following form:

```
40 PRINT TAB(10)2
```

and in this case will **PRINT** a character at the eleventh **PRINT** position counting from the beginning of the first **PRINT** field, since **TAB**'s first **PRINT** position is 0. (In other words, it will **PRINT** a 2 at the beginning of the second **PRINT** field). You can use more than one **TAB** in a **PRINT** statement:

```
50 PRINT TAB(10)2;TAB(20)3
```

But it is not possible to **TAB** backwards:

```
60 PRINT TAB(20)3;TAB(10)2
```

since the 64 will simply **PRINT** the second item in the next available position after the first **TAB** position.

SPC is an alternative to the **TAB** function and simply **PRINT**s a specified number of **SP**a**C**es to the screen:

```
70 PRINT SPC(10)2
```

Before ending this section on the **PRINT** command we must mention one final extention to the statement, which we shall not explain until a little later. The **CHR$** (character string) command extends the power of **PRINT** statements still further. The number in brackets following this keyword contains a code which determines the effect of the command. For example, you can clear the screen at the beginning of the program of **PRINT** statement examples you have just entered by typing:

```
5 PRINT CHR$(147)
```

More about **CHR$** and its codes later. For the time being we'll leave **PRINT** statements with a brief summary of what we have learnt:

i) Any characters you require to be **PRINT**ed literally to the screen must appear within quotes (eg **PRINT "23%*ABC"**)

ii) Any arithmetic expression whose result you want to display must not be enclosed in quotes (eg **PRINT 6*2**)

iii) A semi-colon causes the **PRINT** items of the next **PRINT** statement to be displayed on the same line as the first. For example:

```
1Ø PRINT "HELLO AND ";
2Ø PRINT "GOOD MORNING"
```

will display HELLO AND GOOD MORNING.

iv) A comma will cause the next **PRINT** item to be displayed at the beginning of the next **PRINT** field (eg **PRINT 1,3,5**)

v) **TAB** (x) creates a **PRINT** position x characters to the right of the last.

vi) **PRINT SPC**(y) **PRINT**s y spaces to the screen.

LET THERE BE VARIABLES

We have considered the **PRINT** command in depth not only because it is such a well-used and flexible instruction, but also because it allows us to introduce the concept of a program as simply as possible. This is because we can demonstrate most **PRINT** formats without having to rely on any other **BASIC** word. However, like the words of any language, the power of most **BASIC** keywords can only be fully demonstrated in the context of other keywords. Thus it will be necessary for some of the example programs which follow to utilise commands and structures that will not be explained until later in the text. On these occasions you can either look up the new statement in the keywords section (Chapter 5), or else concentrate on the word under discussion and contain your curiosity until the keyword in question crops up in this brief tutorial.

At the beginning of this chapter we discovered that if we attempt to **PRINT** letters without quotes the 64 will **PRINT** a zero to the screen. However, before we can start typing in some examples to help us understand why this happens, we

must get rid of our example program for **PRINT**. (Assuming you haven't switched off the power to your 64 in the meantime.)

Whenever you want to start a new program, it is important to ensure that your computer's memory is not storing any program lines from a previous program. Up until now we have got around this problem by using the same line numbers for all our examples. If you type in two different statements with the same line number, the second will replace the first which will be erased from memory. In fact you can get rid of unwanted lines by simply typing in the line number in question and pressing the <**RETURN**> key. Obviously this is an impractical way of deleting an entire program. To see whether your 64 is storing any program lines, type **LIST** and then <**RETURN**>. If any lines are in memory they will now be **LIST**ed to the screen. The **LIST** command is almost always used as a direct command and holds the key to the 64's powerful editing facilities. It can be used in a variety of formats (see keyword section) which allow single lines or specified sections of program to be displayed on the screen. **LIST** on its own displays the complete program.

NEW is also primarily used as a direct command. It wipes out the program in memory and all the 'variables' that may have been created when executing the program. So you should always think twice before entering this command. If the program you are about to destroy is likely to be needed again, make sure you have **SAVE**d it to tape or disc as there is no going back with the **NEW** command. Once **NEW** has been entered, there is no way of getting the program back again.

Once you have used **NEW** to clear the 64's memory, enter the following as a direct command to convince yourself of the effect of **PRINT**ing letters without quotes:

```
PRINT ABC
```

In **PRINT**ing a zero to the screen the 64 has done its best to complete a procedure with a total absence of co-operation from us humans. When executing our **PRINT** statement, it has tried to treat **ABC** as a numeric 'variable'. Up until now we have decided what values we want to include in a statement, and typed them in each time they are required. However, by using a variable name to represent a particular value, we can create statements in which the variable takes the place of the value each time it is required. For example, suppose we decide to give the value 2.5 the variable name A. We can do this by using the **BASIC** keyword **LET** which takes the following format:

```
10 LET A=2.5
```

When it encounters this statement, the 64 reserves a space in its memory into which it places the value 2.5 and calls this location A. For the rest of the program (or until it is given instructions to the contrary) each time it encounters an A without quotes it will regard it as the value 2.5. The creation or assignment of variables is the most important stage in the coding of any **BASIC** program, and it is always wise to place these statements at the beginning of the section in which they are processed. One of the many advantages of this facility is demonstrated by the following simple example:

```
10 PRINT 6*(3.231*3/4)+6
20 PRINT 1.2*(3.231*3/4)+1.2
30 PRINT 5.6*(3.231*3/4)+5.6
```

As you can see, each line in this example multiplies and adds a different value to an identical expression (3.231*3/4). By assigning this expression to a variable name we can make each calculation condiserably easier to follow, and save ourselves the bother of having to type in a lengthy expression each time it is required in the program. With the inclusion of a **LET** statement our example can be modified in the

following way:

```
10 LET V=3.231*3/4
20 PRINT 6*V+6
30 PRINT 1.2*V+1.2
40 PRINT 5.6*V+5.6
```

RUN both examples to make sure that the results **PRINT**ed are the same.

When we looked at the **PRINT** statement we emphasised that you must enclose text in quotes if you want it treated as a string of characters and not a numeric variable. We must now go on to explain that if you want to **PRINT** a string variable that has already been created it does not require quotes. String variables work in much the same way as numeric variables, except that the variable name must be followed by a dollar ($) sign and that the characters that are being assigned to the variable must be in quotes. For example:

```
10 LET N$="JIM"
20 LET AGE=34
30 PRINT N$;" IS";AGE;"YEARS OLD"
```

will **PRINT** JIM IS 34 YEARS OLD.

Notice that although the string JIM requires quotes when it is assigned the variable name N$ in line 10, the string variable does not require quotes when it is **PRINT**ed in line 30.
When creating variable names, it is useful to come up with names which indicate their role (eg AGE or N$ for a Name). Variable names can be of any length (although practically speaking it is wise to limit yourself to three or four characters as a maximum), but the 64 will only recognise the first two characters of a variable name. Thus NA$ and NAME$ will be considered as the same variable.

The use of the word **LET** in a **LET** statement is optional in 64 **BASIC**. So both **LET** A=3 and A=3 are acceptable forms of the same instruction. Experienced programmers tend to leave out the keyword itself to save memory, however when you are starting out it's worth writing the statement in its full version to remind yourself of what you are doing.

EDITING ON YOUR 64

We've got to the stage where our example programs are becoming longer, so this is probably as good a time as any to take a look at the 64's editing facilities.

In the last section we saw how the **LIST** command displays the current program to the screen. The programs we have looked at up until now have only been a few lines long, but when you **LIST** a full sized program it normally fills more than one screen. Under these circumstances the **LIST**ing scrolls up the screen until it reaches the final lines. If you want to examine particular sections of the **LIST**ing you will want to **STOP** the scrolling or at least slow it down. To **STOP** the scrolling you simply press the **RUN/STOP** key; to slow down the scrolling press the **CONTROL** key.

The program below is essentially a simple example of variables in action. However, we have allowed a number of typing errors to creep into the **LIST**ing to give us an opportunity to unveil the 64's editing facilities.

```
10 LET RETAIL=20
20 LET WHOLESALE=10
30 LET SOLD=40
40 LET PROFIT=(RETAIL-WHOLESALE)*SOLD
50 PRINT "TISS WEEK'S PROFIT IS":PROFIT
```

As you can see, line 50 is a bit of a mess. Thankfully, the 64 is equipped with powerful, easy to use editing facilities. In this instance we've got two options open to us. We can either edit

line 50 as it appears in the listing itself, or separate the offending line from the rest of the program by entering **LIST** 50. In either case, use your cursor keys to move the cursor until it covers the I of TISS. Now press the **SHIFT** and **INST/DEL** keys simultaneously. This will create a space between the T and I which enables us to insert an H into the **PRINT** statement. Now move the cursor along the line until it covers the second S of TISS and press the **INST/DEL** key (unshifted). This will remove the superfluous S. Finally, we must replace the colon with a semi-colon. Move the cursor until it covers the colon, then simply type in the semi-colon. Line 50 should now read:

```
5Ø PRINT "THIS WEEK'S PROFIT IS";PROFIT
```

Remember, if you want to delete an entire line, all you need to do is to type in the line number in question and press the <**RETURN**> key.

We can now include one final statement which will add the finishing touch to our example program. Enter the following line:

```
5 REM*VARIABLE EXAMPLES*
```

This line is known as a **REM** statement. **REM** stands for **REM**inder or **REM**ark, and has no effect on the execution of a program. So, what's the point of it? Well, the purpose of **REM** statements is to allow the programmer to add titles or a commentary to a **LIST**ing. This facility is invaluable when developing lengthy or complex sections of code, since the conscientious use of **REM**s clarify programs when you come back to them at a later date.

Up until now we have restricted ourselves to one **BASIC** statement per program line. However, by using a colon it is possible to create multi-statement lines. For example:

```
10 LET A$="A STRING":PRINT A$
```

When using this construction there are a couple of points that should be borne in mind. Some **BASIC** commands use line numbers to change the order in which a program is executed (these commands are **GOTO** and **GOSUB**, and we'll be coming to them shortly). However, if you have used multi-statement lines it is only possible to access the entire line using such a command, not individual statements within a line. You've also got to be careful if you use a **REM** statement within a multi-statement line. For example:

```
50 REM*EXAMPLE*:PRINT "EXAMPLE ONE"
60 PRINT "EXAMPLE TWO"
```

will only display EXAMPLE TWO, because as soon as the 64 encounters a **REM** statement it moves on to the next line, and in the case above would never actually get to the **PRINT** "EXAMPLE ONE" statement. So line 50 would have to be written as follows:

```
50 PRINT "EXAMPLE ONE";REM*EXAMPLE*
```

Another point that should be taken into consideration is that, whilst the multi-statement facility can sometimes be convenient, excessive use of this format tends to make a listing difficult to follow, so use multi-statements in moderation.

GETTING INTO INPUTS

Up until now the program examples we have looked at have not required any participation from the user. Of course the majority of programs are interactive, and the most important **BASIC** command that facilitates user participation is **INPUT**.

When the 64 encounters an **INPUT** statement it stops processing the program and will not continue until the user

has keyed-in an appropriate response. The format of the statement can allow for strings, numbers or both strings and numbers, but if the wrong sort of data is entered the program will throw up a ?RE-DO FROM START message. Let's take a look at the simplest form of the **INPUT** statement.

```
5 REM*RADIUS*
10 PRINT"ENTER A RADIUS"
20 INPUT R
30 PRINT CHR$(147)"THE AREA OF A CIRCLE
WITH RADIUS";R;"IS"
40 PRINT π*R↑2
```

When this program is **RUN** it will stop at line 20 until the user **INPUT**s a number. When the number is entered (by pressing <**RETURN**>) it is stored in the **INPUT** variable R and the computer continues processing from the next line (line 30). **INPUT** statements provide a variety of prompts which indicate that data is required of the user. In the case of the example above, a question mark prompt will be displayed at the current cursor position. However, a question mark doesn't give the user of the program much of an indication of what kind of **INPUT** is required, which is why the **PRINT** statement in line 10 is necessary. A more economic alternative is to incorporate a customised prompt within the **INPUT** statement itself. Enter 10<**RETURN**> (to delete line 10) and then type in the following:

```
20 INPUT "ENTER A RADIUS ";R
```

which will print the **INPUT** prompt just like **PRINT** statement, except that a question mark will also be displayed to indicate that data is required from the user. String **INPUT**s can be programmed in exactly the same way, and two or more **INPUT** variables can be incorporated into a single **INPUT** statement, so long as they are separated by commas. The program will only continue after both of the required values have been entered. The sample program that follows

demands both string and numeric **INPUT**s:

```
5 REM*INPUT*
10 PRINT"ENTER FIRST YOUR NAME"
20 INPUT"AND THEN YOUR AGE";NAME$,AGE
30 PRINT"YOUR NAME IS ";NAME$
40 PRINT"AND YOU ARE";AGE;"YEARS OLD"
```

When this program is **RUN** the textual prompts are **PRINT**ed along with usual question mark prompt. When you enter the string input **(NAME$)** the 64 then **PRINT**s a second double question mark prompt (??) to show that a further **INPUT** is required before the program will continue.

Now let's try a slightly more serious example that is an extension of our variable demonstration program.

```
10 REM*INPUT2*
20 LET RETAIL=120
30 LET WHOLESALE=70
40 INPUT "HOW MANY ITEMS DID YOU SELL
THIS WEEK";SOLD
50 LET PROFIT=(RETAIL-WHOLESALE)*SOLD
60 PRINT"THIS WEEK'S PROFIT IS";PROFIT
```

As you can see, the **INPUT** command has added an extra dimension to the earlier program in that it allows us to **INPUT** a variable for the number of items SOLD.

When we described multi-statement lines we mentioned that **GOTO** and **GOSUB** commands enable us to alter the flow of control of a program. In other words, rather than the program simply **RUN**ning from the lowest line number executing the statements in order until reaching the highest line number, and then stopping, **GOTO** and **GOSUB** can re-direct control.

Suppose we want to calculate **PROFIT** for an unspecified number of weeks of trading. Rather than having to re-**RUN** the program each time we want to enter a new **INPUT**, we

could add the following line to our listing:

```
70 GOTO 10
```

With this minor modification, rather than a single **INPUT** causing the program to stop at line 60, the 64 is forced to **GO** back **TO** line 10 and start the process all over again. In fact the program will **RUN** endlessly (so long as the user continues to **INPUT** data), and the only way it can be stopped is by breaking into it by using the **RUN/STOP** key.

The use of **GOTO** is explained at some length in the keyword section, and you should refer to the appropriate entry when you have finished this chapter.

Both **GOTO** and **GOSUB** are extremely powerful commands as they are the main method of establishing efficient program structures when coding in **BASIC**. **GOSUB** works in much the same way as **GOTO**, in that it diverts control to the line number specified by the statement in which it appears. However, the **GOSUB** facility is far more sophisticated than a **GOTO** statement. The 64 has been designed to remember the line number in which the **GOSUB** statement appears and, once the process called by **GOSUB** has been executed, will **RETURN** to the line that follows the original **GOSUB** statement.

The section of code to which control is diverted when a **GOSUB** statement is used is known as a subroutine. Subroutines are usually collections of **BASIC** instructions which will be used more than once in the course of a program. Thus, rather than having to repeat the same set of instructions each time a specific routine is required, the **GOSUB** statement means that the routine in question only needs to be written once, and called by a **GOSUB** statement at the appropriate point in the program. Every time you create a subroutine the final statement in the routine must be **RETURN** in order to send control of the program back to the

line following the **GOSUB** statement. This all sounds very complicated, so let's have a look at an example that will hopefully clarify matters.

```
1 REM*GOSUB EXAMPLE*
5 :
10 GOSUB 700:REM*CLEAR SCREEN*
20 :
30 GOSUB 100:REM*CALL SUB1*
40 :
50 GOSUB 200:REM*CALL SUB2*
60 :
70 GOSUB 300:REM*CALL SUB3*
80 :
90 END:REM*END OF PROGRAM*
95 :
100 REM*SUBROUTINE 1*
110 GOSUB 500
120 PRINT TAB(9);"THIS IS SUBROUTINE 1"
130 GOSUB 600
140 RETURN
150 :
200 REM*SUBROUTINE 2*
210 GOSUB 500
220 PRINT TAB(9);"THIS IS SUBROUTINE 2"
230 GOSUB 600
240 RETURN
250 :
300 REM*SUBROUTINE 3*
310 GOSUB 500
320 PRINT TAB(9);"THIS IS SUBROUTINE 3"
330 GOSUB 600
340 RETURN
350 :
500 REM*FORMAT SCREEN*
510 FOR A=1 TO 7
520 PRINT
530 NEXT A
540 RETURN
550 :
```

```
600 REM*PAUSE LOOP*
610 FOR A=0 TO 3000
620 NEXT A
630 GOSUB 700
640 RETURN
650 :
700 REM*CLEAR SCREEN*
710 PRINT CHR$(147)
720 RETURN
730 :
800 REM*END OF SUBROUTINES*
```

As you can see, an intelligent use of subroutines can make a listing easy to read, since it is possible to create discrete modules of **BASIC** instructions for each major process in a program. The role of the main body of the program also becomes simplified when programs are structured in this way, since its role is less concerned with actually processing data than with directing control to the appropriate subroutines.

It is worth stressing that although on the face of it **GOTO** is a simpler command than **GOSUB**, in the final analysis it causes more programming problems. This is because it is tempting to use **GOTO** to jump around a listing like a bed bug, rather than organise the structure of your program systematically from the start. Apart from the fact that such programs will almost certainly produce unpleasantly surprising results, it also makes the listing difficult to unravel when you get around to seeking out such bugs. So use **GOTO**s with care, or you'll end up with the following kind of nightmare in coding:

```
10 REM*SPAGHETTI GOTOS*
20 GOTO 65
30 PRINT CHR$(147);"HELLO ";NAME$
40 GOTO 110
50 PRINT"AND YOU ARE";AGE;"YEARS OLD"
```

```
60 GOTO 140
65 PRINT CHR$(147)
70 INPUT "WHAT IS YOUR NAME";NAME$
80 GOTO 30
90 PRINT CHR$(147);"SO YOUR NAME IS ";NA
ME$
100 GOTO 50
110 PRINT CHR$(147)
120 INPUT "HOW OLD ARE YOU";AGE
130 GOTO 90
140 END
```

Believe it or not, this program **RUN**s in spite of itself, but it is so difficult to follow and inefficient, that you almost wish that it didn't bother. The only worthwhile aspect of this example is the **END** statement in line 130. As you might expect, this command indicates the **END** of processing in a program. When the 64 encounters this statement it stops processing and displays the **READY** prompt on the screen to indicate that it is awaiting further instructions. If you include an **END** statement in the middle of a program (which can sometimes be useful when you are trying to locate errors in a program), processing can be **CONT**inued by typing **CONT** <**RETURN**>. If you want the 64 to display the line number at which processing has **STOP**ped, you must use **STOP** instead of **END**, which will have exactly the same effect, but produce a **BREAK IN In** report (where **In** is the line number in question).

CHAPTER THREE
DEALING WITH DATA

WHAT'S THE PROBLEM?

In the last chapter we demonstrated how **GOTO** can be misused to produce inefficient and incomprehensible programs. One of the main reasons that this particular statement causes so many problems is that there is a tendency, particularly amongst inexperienced programmers, to use **GOTO**s to patch-up badly designed programs. It is always tempting to start developing a program directly on the 64, but this should be strenuously resisted. This isn't to say that programs should be drafted in long hand and then keyed-in to the computer. However, what all programmers must be clear about before they start coding is how the problem they are trying to solve can be broken down into manageable tasks that can be dealt with by self-contained units (or modules) of **BASIC** instructions. Once this has been done, it's easy enough to decide the order in which these modules should be processed to produce an efficient program. Let's take a simple problem and run through the factors that should be taken into consideration before we start writing our program.

Say we want to work out the area and circumference of a circle whose radius is to be provided by the user of the program. In addition, we also require the program to correct the results of its calculations to two decimal places. We first need to work out how many different operations we require the 64 to

perform. These can be itemised as follows:

1. Display function of program.
2. Provide **INPUT** facility so that the user can enter the radius of the circle.
3. We need to create a variable that holds the formula for calculating the circumference of a circle.
4. And a variable for the area of a circle.
5. We need a routine to correct the results of the calculations to two decimal places.
6. We need a method of displaying the results to the screen.

By comparing our list of tasks with our original brief, we can make sure that we have identified every aspect of our problem.

What is clear from our task list is that one process must be performed twice for each radius that is entered. Although the calculations to find the area and circumference of our circle will obviously be different, the results of both calculations have to be rounded to two decimal places. The rounding procedure can obviously be coded as a subroutine, so we can avoid having to write the same set of instructions for the result of each of the two calculations.

Having decided that process number five can be used twice, we are immediately faced with another problem. The results of our two calculations are held in two different variables, but must both be applied to the same process. In other words, if A is the variable which is the result of calculating the area of a circle, and C is the circumference of a circle, how can we use two different variables in the same routine to round to two decimal places? To make this problem a little clearer we must jump ahead and take a look at the subroutine we will have to create to round to two decimal places. This routine contains a new keyword which will be fully explained later in this chapter. For the moment, however, don't worry about the calculation involved, concentrate on the problem of using two

variables in the same calculation. The new keyword we must use is the **INT** function. The keyword chapter contains a full description of this function, and if you're interested you should take a look at this entry before reading any further. However, all you really need to know at the moment is that **INT** rounds a decimal number down to the next lowest whole number (thus if v=INT(2.5) then v=2).

Let's construct our rounding subroutine so that it rounds the result of calculating the area of our circle to two decimal places. Remember the variable A = Area of circle.

```
200 REM*ROUNDING SUB*
210 LET A=INT(100*(A+.005))
220 LET A=A/100
230 RETURN
```

This will round our circle area (A) to two decimal places (you'll have to take our word for it). However, the problem with variables should now be clear. Whilst the result of the Area calculation can now be rounded, we cannot use this routine to round the circumference results (because this is represented by the variable C not A). The only way around this problem is to avoid using either C or A as a variable in the rounding routine. Let's say that Z is any value that must be rounded to two decimal places. Thus we can add two processes to our list:

7. Convert Area variable into rounding variable.
8. Convert Circumference variable into rounding variable.

Now that our list of tasks is complete, we can turn to the keyboard and start coding. Task one simply **PRINT**s the purpose of the program to the screen:

```
1 REM*CIRCLE CALC*
5 REM*DISPLAYINTENT*
```

```
10 PRINT CHR$(147)
20 PRINT"THIS PROGRAM CALCULATES THE"
30 PRINT"CIRCUMFERENCE AND AREA"
40 PRINT"OF ANY CIRCLE TO TWO DECIMAL PL
ACES"
45 :
```

This part of the program is quite clear. Lines 20 and 30 display the facilities offered by the program, and line 10 is a statement to clear the screen. Now we need to allow the user to **INPUT** the radius value to be used in subsequent calculations.

```
50 REM*USER INPUT*
60 PRINT
70 INPUT "ENTER CIRCLE RADIUS ";R
75 :
```

As soon as the radius has been entered we can perform our first calculation. Let's calculate the circumference of a circle with radius R. The formula for this is 2*π*R. So our next lines will read:

```
80 REM*CALCULATE CIRCUMFERENCE*
90 LET C=2*π*R
```

Since we are going to require the result of this calculation to be processed by our rounding subroutine we are going to have to change our C into a Z:

```
100 LET Z=C
105 :
```

Next we need to call the rounding subroutine and, once the rounding process has been completed, **PRINT** the result to the screen.

```
110 REM*ROUND & PRINT RESULT*
120 GOSUB 300
130 PRINT CHR$(147)
```

```
140 PRINT"CIRCUMFERENCE OF A CIRCLE WITH
 RADIUS";R;"IS";Z
145 :
```

Now we need to assign to a variable the expression to calculate the Area of a circle with Radius R:

```
150 REM*CALCULATE AREA*
160 LET A=π*R↑2
165 :
```

A now needs to be made equal to the rounding variable Z:

```
170 REM*ROUNDING VARIABLE*
180 LET Z=A
185 :
```

Once again the subroutine must be called and the result **PRINT**ed:

```
190 REM*ROUNDING*
200 GOSUB 300
205 :
210 REM*DISPLAY*
220 PRINT"AREA OF CIRCLE WITH RADIUS";R;
"IS";Z
```

We have now completed all the processing modules of our program. Lines 0-180 provide the means of **INPUT**ing data, assigning variables and displaying results. The subroutine at line 200 performs the rounding operation for both calculations. We now need to include a closing statement to prevent the main program module from running to the subroutine:

```
230 END
240 :
```

When we initially wrote the subroutine it was coded to deal with the result of the Area calculation. Now we have created the rounding variable Z it is necessary to make a couple of minor changes to the routine:

```
300 REM*ROUNDING SUB*
310 LET Z=INT(100*(Z+.005))
320 LET Z=Z/100
330 RETURN
340 REM*END OF SUB*
```

So our complete program now looks like this:

```
1 REM*CIRCLE CALC*
5 REM*DISPLAYINTENT*
10 PRINT CHR$(147)
20 PRINT"THIS PROGRAM CALCULATES THE"
30 PRINT"CIRCUMFERENCE AND AREA"
40 PRINT"OF ANY CIRCLE TO TWO DECIMAL PL
ACES"
45 :
50 REM*USER INPUT*
60 PRINT
70 INPUT "ENTER CIRCLE RADIUS ";R
75 :
80 REM*CALCULATE CIRCUMFERENCE*
90 LET C=2*π*R
100 LET Z=C
105 :
110 REM*ROUND & PRINT RESULT*
120 GOSUB 300
130 PRINT CHR$(147)
140 PRINT"CIRCUMFERENCE OF A CIRCLE WITH
 RADIUS";R;"IS";Z
145 :
150 REM*CALCULATE AREA*
160 LET A=π*R↑2
165 :
170 REM*ROUNDING VARIABLE*
180 LET Z=A
185 :
```

```
190 REM*ROUNDING*
200 GOSUB 300
205 :
210 REM*DISPLAY*
220 PRINT"AREA OF CIRCLE WITH RADIUS";R;
"IS";Z
230 END
240 :
300 REM*ROUNDING SUB*
310 LET Z=INT(100*(Z+.005))
320 LET Z=Z/100
330 RETURN
340 REM*END OF SUB*
```

If you compare the final program with our original list of tasks, you'll see that each list item is represented by an appropriate section of **BASIC** code. Ideally you should always tackle a programming problem in this way. When the program's objectives are as straightforward as those of our example such an approach is almost inevitable since, apart from the variable problem the coding is so simple that the program almost writes itself. When the problem is more complex, it is considerably more difficult to hammer out a task list that relates so literally to the finished **BASIC** program. The main reason for this is that when problems become more complex each process in the initial task list may well be as involved as the complete program we have just written. Under these circumstances it's easy to overlook a problem like the need for variable conversion. On the other hand, as long as the major processing requirements have been correctly identified, a slight adjustment to a single module hardly constitutes a disaster.

The moral of all this is quite simple. The biggest mistake any programmer can make is to start churning out instructions which attempt to solve the entire problem in one fell swoop. The art of writing efficient and readable programs is primarily determined by the ability of the programmer to break down a problem into a series of smaller problems. As you have probably realised, individual **BASIC** commands can only be applied to very specific problems. The flexibility of the language as a whole is only apparent when the commands are creatively combined within a program. This said, it is precisely the specificity of the role of each command which explains the importance of breaking down a problem into manageable sub-problems before it is coded. In this way individual tasks can be matched with **BASIC** keywords on a one to one basis.

A NUMBER OF NUMBER SYSTEMS

If we look back at the task list for the program we have just written, we can identify three essential elements which are common to any program. In order to solve any problem your 64 requires data to process, instructions which determine how this data is processed and a method of displaying or communicating the results of the processing to the user. Before we can go to look at the sophisticated sound and graphics potential of your 64 we must be clear about the kind of data it will accept, and the way in which such data can be processed.

Our profit program in the previous chapter first set up the data in the form of variables, then defined a variable called PROFIT which calculated profit and finally **PRINT**ed the profit to the screen. In this program we dealt solely with decimal numeric data (as opposed to both string and numeric). There are however, many different types of number systems, and one important method of presenting numeric data on micros is in the form of exponential numbers. The exponential number system is used to express very large and very small

numbers. If a number exceeds 999,999,999 (nine characters long) it will be presented in exponential notation taking the format of 1.02E+09 or 1.02E-09. When the exponential number is positive you move the decimal point to the right to obtain the value in its normal format (in this case nine times, giving the number 1,020,000,000) and if the exponential number is negative you move the decimal point to the left (in this case nine times giving the number 0.00000000102). The biggest number that the 64 can hold is 1.70141183E+38 and the smallest is 2.93873588E-39.

The 64 can deal with numbers between 999,999,999 and -999,999,999 in the normal way, but if a number is greater or smaller it will be returned in the exponential format we have just outlined.

The other number system that the 64 (and all computers) deals with is the binary system. Binary numbers use a sequence of zeroes and ones to hold all forms of data in memory. Binary representation is used by micros because the electronic switches at the heart of all computers can only be either on or off, a binary 1 represents the switch being on and 0 represents off.

The number system with which most of us are familiar is decimal or base 10. The binary system is base 2. Let's do a bit of counting in binary and then see how it works.

DECIMAL		BINARY
1	=	1
2	=	10
3	=	11
4	=	100
5	=	101
6	=	110
7	=	111
8	=	1000
9	=	1001

10	=	**1010**
11	=	**1011**
12	=	**1100**
13	=	**1101**
14	=	**1110**
15	=	**1111**
16	=	**10000**
17	=	**10001**
18	=	**10010**
19	=	**10011**
20	=	**10100**

Before we take a look at the binary system we'll quickly run through the basic principles of the decimal system. When we have a number such as 1,987 we are actually saying:

1×10^3	=	**1000**
9×10^2	=	**900**
8×10^1	=	**80**
7×10^0	=	**7**
		1987

Binary numbers work on exactly the same principle, the binary number 11011 can be broken down thus:

BINARY			**DECIMAL**	
1	**$1x2^4$**	=	**10000**	**16**
1	**$1x2^3$**	=	**1000**	**8**
0	**$0x2^2$**	=	**0**	**0**
1	**$1x2^1$**	=	**10**	**2**
1	**$1x2^0$**	=	**1**	**1**
			11011	**27**

There will be more about the binary system in a later chapter so we will leave it for now.

All the characters and symbols on your 64 are stored in the memory as numbers. These numbers are called **ASCII** codes and you will find a complete list of these character codes in Appendix 7. (**ASCII**, pronounced as-key, is a acronym for American Standard Code for Information Interchange.) We can use the **BASIC** keyword **ASC** to find out the **ASCII** code of any given character.

```
PRINT ASC("X")
```

This statement **PRINT**s 88 to the screen as 88 is the **ASCII** code for x. The following program **PRINT**s out the **ASCII** code of your **INPUT**.

```
 1 REM***ASC***
10 INPUT "PRESS ANY KEY";A$
20 PRINT "THE ASCII CODE FOR ";A$;"IS";ASC(A$)
30 GOTO 10
```

To break out of this program you will have to press the **RUN/ STOP** key. Note that when **PRINT**ing the **ASCII** code of a character the argument must be within brackets and quotes ("x") when it is literally quoted, but if it is a variable it only needs to be inside brackets (A$).

We could have written this program using **GET** instead of **INPUT**. **GET** scans the keyboard to see if a key is being pressed and if it is stores the appropriate value in the variable in the **GET** statement. When **GET** is used you don't need to press <**RETURN**> as the 64 automatically scans the keyboard, but unlike **INPUT** it will not stop the program and will only read a value into its variable if a key is actually being pressed. So let's re-write our program using **GET.**

```
 1 REM ***GET***
10 PRINT "PLEASE PRESS A KEY"
20 GET A$:IF A$="" THEN GOTO 20
```

```
30 PRINT "THE ASCII CODE OF ":A$;"IS ";CHR$(a)
40 GOTO 10
```

The second statement in line 20 simply tests to see a key has been pressed and if it hasn't, repeats the **GET** scan until input is received. Using the keyword **CHR$** we can perform the same function in the opposite direction instead of converting the character to its **ASCII** code we can convert an **ASCII** code to its character. The number that follows the **CHR$** must be between 0 and 225.

The next command we shall look at is **VAL**. In the previous chapter we saw how the **PRINT** statement does not look at the information between the quotes but just **PRINT**s it to the screen. Consequently you cannot perform a calculation between any numbers that are quoted as strings or assigned to a string variable. However, **VAL** enables us to convert such a string into its numeric value. Key in the following program and we will then run through it and see how this is done:

```
1 REM*VAL*
5 PRINT CHR$(147):PRINT
10 PRINT "ENTER A NUMBER"
20 GET A$:IF A$=""THEN 20
30 LET A=VAL(A$)
40 PRINT"VAL WILL NOW TURN YOUR INPUT"
50 PRINT"INTO A NUMBER"
60 PRINT"ENABLING US TO PERFORM A CALCUL
ATION"
70 GOSUB 200
80 PRINT"THUS:4*";RIGHT$(A$,2);"=";4*A
```

```
90 GOSUB 200
100 GOTO 10
110 :
120 REM*PRESS RUN/STOP*
130 REM*TO EXIT PROGRAM*
140 :
200 REM*DELAY LOOP*
210 FOR C=0 TO 3000:NEXT C
220 PRINT CHR$(147)
230 RETURN
```

You are asked to input a number, the value of which is stored in A$ (string variable). Line 30 **LET**s A (numeric variable) equal **VAL**(A$), this turns A$ into a numeric variable thus enabling us to perform a calculation at the end of the program. The first character of **VAL**'s argument must be a plus (+ or minus (-) sign, a decimal point, a space, or a number, else the statement will return to zero. Any non-numeric characters in the middle of the argument will be ignored.

The complementary function to **VAL** is **STR$**. Where **VAL** turns strings into numbers, **STR$** converts numbers into strings! For a full explanation of the function, turn to its entry in the keywords section. It's action is actually self-explanatory, as you'll see if you **RUN** the following example which shows **STR$** used in conjunction with **VAL**:

```
10 REM*VAL & STR$*
20 LET A$="1.65"
30 PRINT STR$(VAL(A$)*1.3/2.4)
```

We'll be looking at the rest of the 64's numeric functions a little later, but for the moment we'll concentrate on the rest of the string functions. When you're attempting to format screen displays from user **INPUT** it's often essential to know the length of a particular string. This can be discovered by

applying **LEN** to the string in question, since this function returns the number of characters in its argument (the string or string variable in brackets).

For example:

```
10 A$="STRING"
20 PRINT LEN(A$)
```

will return 6 (the **LEN**gth of A$). You can create tidy screen displays by using the function with **TAB** statements:

```
10 REM*LEN*
20 PRINT CHR$(147)
30 INPUT "ENTER TWO NAMES";A$,B$
40 PRINTCHR$(147)
50 PRINT TAB(20);"NAME":PRINT
60 PRINT TAB(24-LEN(A$));A$
70 PRINT TAB(24-LEN(B$));B$
```

When displaying tables and charts you'll often want to apply **LEN** to numeric variables. This is only possible if first you convert them into strings by using **STR$**. For example:

```
10 REM*LEN2*
20 PRINT CHR$(147)
30 INPUT "ENTER TWO NUMBERS";A,B
40 LET A$=STR$(A):LET B$=STR$(B)
50 PRINT CHR$(147)
60 PRINT TAB(25-LEN(A$));A
70 PRINT TAB(25-LEN(B$));B
```

BASIC has a wide range of commands which allow you to manipulate strings with ease. Strings can be sliced and generally mutilated, then painlessly joined together again.

The following examples demonstrate the functions which facilitate such savagery, and the circumstances in which they can be usefully employed.

MANIPULATION OF STRINGS

Having created a string and assigned it to a variable, it is possible to link it to other previously created strings. String addition (or concatenation) joins strings together using the plus sign (+). Key-in the following example to see this facility in action:

```
5 REM*CONCATENATION*
10 PRINT CHR$(147)
20 INPUT"WHAT'S YOUR NAME";NAME$
30 LET SPACE$=" "
40 LET QUERY$="HOW ARE YOU?"
50 LET HI$="HELLO"+SPACE$+NAME$+SPACE$
60 PRINT CHR$(147)
70 PRINT HI$+SPACE$+QUERY$
80 INPUT ANSWER$
90 PRINT CHR$(147)
100 LET REPLY$="SO YOU ARE FEELING"
110 PRINT REPLY$+SPACE$+ANSWER$
```

This program joins all the strings together using the addition sign (+).

Using either **LEFT$**, **RIGHT$** or **MID$** we can extract a specified character or characters from a string as the following program demonstrates:

```
10 REM*LEFT$*
20 LET A$="PATTERN"
30 LET B$="INTERESTING"
40 LET C$="TINGLE"
50 LET L$=LEFT$(A$,2)
60 LET M$=LEFT$(B$,2)
70 LET R$=LEFT$(C$,4)
80 PRINT L$+M$+R$
```

This example extracts the specified number of left-most characters from A$, B$ and C$. The number of characters that are extracted from these strings is specified by the

number in the brackets following **LEFT$**. For instance, line 50 (displayed below) assigns **LEFT$**(A$,2) to the variable called L$. In simple terms this is saying, extract the 2 left-most characters from A$ (these are P and A) and assign them to the variable L$.

```
50 LET L$=LEFT$(A$,2)
```

RIGHT$** works the same way as **LEFT$, except that the number in the brackets (let's call it 2) extracts the 2 right-most characters from the string. In the above program we always assigned **LEFT$** to a variable, this is not essential, as the following example of **RIGHT$** demonstrates:

```
10 REM*RIGHT$*
20 LET A$="SPRINTER"
30 LET B$="DREG"
40 LET C$="DUODENUM"
50 PRINT RIGHT$(A$,5)+RIGHT$(B$,3)+RIGHT
$(C$,3)
```

This program **PRINT**s "interregnum" to the screen by extracting the 5 right-most characters from A$, the 3 right-most characters from B$ and the 3 right-most characters from C$. These are all joined together in line 50 and the result is **PRINT**ed to the screen.

The last of the string functions that we will look at in this section is **MID$**, which, though similar to **RIGHT$** and **LEFT$**, is slightly more sophisticated in its action. **MID$** is usually followed by two numbers, taking the following syntax:

```
MID$(A$,X,Y)
```

Where A$ is the string from which the characters are extracted (the "source" string), X is the first character to be extracted of a "sub-string" which is Y characters in length. So

if we have:

```
1 REM*MID$1*
10 LET A$="POTENTIAL"
20 PRINT MID$(A$,3,4)
```

"TENT" will be **PRINT**ed to the screen, since this program forces the 64 to extract four characters from A$ starting with the third character. This command can also take the same two parameters formats as **RIGHT$** and **LEFT$**:

```
MID$(A$,X)
```

where X is the first character to be extracted from A$. By missing out the second parameter (Y) the 64 will extract the rest of the string. Thus:

```
1 REM*MID$2*
10 LET A$="INFIDELITY"
20 PRINT MID$(A$,3)
```

will **PRINT** "FIDELITY". In essence the **MID$** statement in line 20 is saying extract an unspecified number of characters from A$ starting with the third character. Because the number of characters to be extracted hasn't been defined the program **PRINT**s out the entire string. **MID$** will also return the entire string if asked to extract more characters than there are in the string.

Having briefly cantered through the string functions available in Commodore **BASIC**, let's itemise the numerical functions available on the machine.

ABS(n)	Returns the **ABS**olute value of n
ATN(n)	Returns **A**rc**T**a**N**gent of n
COS(n)	Returns **COS**ine of n
EXP(n)	Returns e raised to the power of n

INT(n)	Returns n truncated to **INT**eger
LOG(n)	Returns natural (base 10) **LOG**arithm of n
RND(n)	Returns a **RaNDom** number
SGN(n)	Returns 0 if n is zero 1 if n is positive -1 if n is negative
SIN(n)	Returns **SIN**e of n
SQR(n)	Returns the square root of n
TAN(n)	Returns the **TAN**gent of n

All of these numerical functions are fully explained in the keyword section, so we won't devote a great deal of space to them in this chapter. However, it is important to understand the circumstances under which these commands can be employed, so let's go back to school and have a brief maths lesson.

GETTING NUMBERS TO FUNCTION

If the truth be told, it's highly unlikely that you're going to be able to use your 64's numerical functions unless you've a fairly sound grasp of what **COS**, **SIN** and all the rest of them are about. This said, it could be that your memory simply needs jogging, so prepare yourself for a swift numeric jolt!

We'll look at the simplest functions first. As the following example demonstrates, it's sometimes necessary to force the 64 to return a positive value, even if there's a chance of a calculation resulting in a negative value. Under these circumstances, the **ABS** function can be applied, and this will return its argument's **ABS**olute value, that is the number ignoring the + or - sign.

```
10 REM*ABS TO PRINT*
20 REM*YRS 500BC TO 1000AD*
30 FOR Y=-500 TO 1000 STEP 50
40 PRINT ABS(Y);
50 IF Y<0 THEN PRINT "BC",:GOTO 70
```

```
60 PRINT "AD",
70 NEXT Y
```

Another straightforward numeric function is **INT**. Some parts of a program might require a particular value to be a whole number or integer. The **INT** function rounds values down to the next lowest integer. Thus:

```
PRINT INT(7.89)          displays 7
PRINT INT(-9.57)         displays 10
```

Note that the next lowest integer to -9.59 is, of course, -10 not -9! On the subject of signs, the 64's **SGN** function returns a result which indicates the status of the value to which it is applied. In the statement **SGN**(x), if x is positive the function will return 1, if x is zero it will return 0, and if x is a negative value **SGN** will produce -1.

As we have already seen the up arrow (↑) is the way that we raise a number to the power of another on the 64. Well, the **EXP** function raises its argument to the value of e (which is approximately 2.71828183). This is extremely valuable if you happen to perform calculations with the 64's natural **LOG** function, since **EXP** (**LOG**(n)) returns the antilog of its argument. The **LOG** function itself can be useful when working with very large numbers that might potentially throw up an overflow error. Under such circumstances you can ensure that:

```
V<7^N
```

by making sure that:

```
LOG(V)N*LOG(7)
```

and although it's possible that V 7 could cause an overflow, the **LOG** formulation performs the test without risking an error report.

The 64's **RaNDom** number function is fully explained in the keyword section, so here we will simply provide an outline of the possible formats for a **RND** statement. If you want to create a random number in the range 0 (which it can equal) to 1 (which it will never quite reach), the format is:

```
r=RND(1)
```

where r is the random value returned.

If we need the random number to fall in the range 0-45 the format is:

```
r=RND(1)*46
```

If we want to set both lower and upper limits for our random number we must use the following format:

```
r=RND(1)*(260-45)+45
```

which will generate a number in the range 45-260.

If the function's seed value (the N in **RND**(N)) is 1 then the resulting value is the same as the last one. If you use:

```
FOR A=1 TO 10: PRINT RND(1): NEXT A
```

to create your first sequence of random numbers, the sequence of numbers will always be the same. However, if you add a line such as:

```
r=RND(-TI)
```

you can avert such random predictability. The use of a negative value for the function's argument forces a new sequence of numbers.

Before we race onto the 64's trigonometric functions, **SQR** needs a mention. The square root of a number is the number which, when multiplied by itself, produces the original value. Thus the square root of 9 is 3. The **SQR** function simply returns the square root of its argument. Thus:

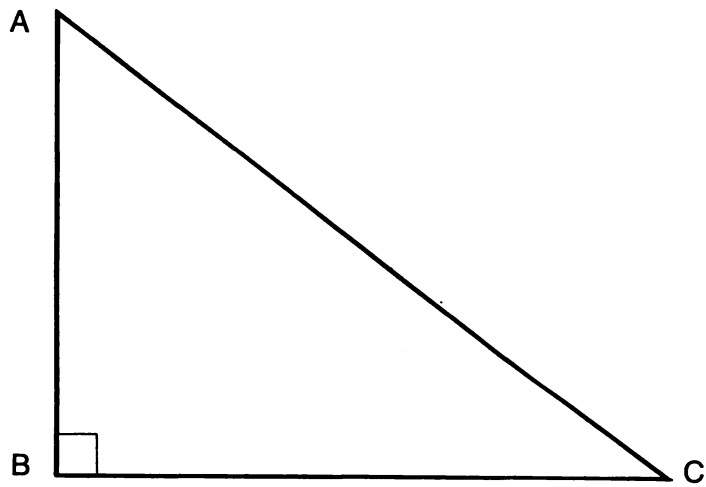
```
PRINT SQR(9)
```

will return 3, and:

```
PRINT SQR(81)
```

will display 9.

Your micro has three major trigonometric functions, **SIN**, **COS** and **TAN**. In order to explain the values returned by each of these functions, we must resort to the use of a right-angled triangle:



Each of the three functions represents a specific ratio of the sides of a right angled triangle. Taking x to be angle opposite the right angle of the triangle ABC, the various ratios can be

expressed as follows:

$$\text{COS(X)} = \frac{\text{BC}}{\text{AC}} \quad \begin{matrix}\text{(ADJACENT)} \\ \text{(HYPOTENUSE)}\end{matrix}$$

$$\text{SIN(X)} = \frac{\text{AB}}{\text{AC}} \quad \begin{matrix}\text{(OPPOSITE)} \\ \text{(HYPOTENUSE)}\end{matrix}$$

$$\text{TAN(X)} = \frac{\text{AB}}{\text{BC}} \quad \begin{matrix}\text{(OPPOSITE)} \\ \text{(ADJACENT)}\end{matrix}$$

The value of recognising this set of relationships is that, given partial information about such a triangle, we can use these ratios to fill in the gaps. For example, suppose we know the angle x and the length of BC, we can go on to work out the lengths of the other two sides by calculating:

```
AC = TAN(X)*BC
AB = BC/COS(X)
```

While the theory might be tedious, the would-be graphics programmers amongst you will immediately see that these functions will be invaluable for future projects. The format for each of the 64's trigonometric functions is as follows:

```
r = TAN(x)
r = COS(x)
r = SIN(x)
```

When you use these functions you must remember that, in common with all micros, the Commodore 64 does not measure angles in degrees, but in radians. However, since:

360 degrees = 2*PI radians

and the fact that you can **PRINT** an acceptable approximation of PI directly from the 64's keyboard using the

(π) key, converting degrees into radians and vice versa is no problem. You'll find the formula in the keywords section (see **SIN**).

Now suppose we know the lengths of the sides of our triangle, how can we calculate its angles. Well, the 64's final trigonometric function makes this possible. **ATN**(Y) returns the arc tangent of its argument (Y). The easiest way to think of this function is to regard it as the inverse of tangent. To make things a little clearer let's go back to our trusty diagram. Say we know the lengths of AB and BC, by using **ATN** we can now calculate angle X using the formula:

X=ATN(AB/BC)

One of the shortcomings of Commodore 64 **BASIC** is that it does not include a full set of inverse trigonometric functions. However, using the ones we do have at our disposal you can create your own user **DEF**ined inverse **FuN**ctions. The following formulae simulate arc sine and arc cosine functions:

```
ASN(BC)=ATN(BC/SQR(1-BC*BC))
ACS(AB)=π/2-ATN(AB/SQR(1-AB*AB))
```

If the prospect of typing in expressions like this every time they are required in a program fills you with dread, fear not! User **DEF**ined **FuN**ctions are the answer to your prayers.....

A FUNCTION TO DEFINE

In addition to the 64's built-in functions like **COS** and **TAN**, there is also a way of defining your own functions using the **DEF FN** statement. This enables the programmer (that's you!) to provide the arithmetic expression of the required function using the following format:

DEF FNv(z) = numeric expression

The v is the name of the function (in the same as **SIN** is the name of the sine function), and z is a standard numeric variable that the **DEF**ined **FuN**ction will use when calculating the specified expression. The expression itself can be any numeric expression and can use any other numeric function, but must obviously include the operative variable (z).

When the customised function is required in a program it is accessed in the same way as any other numeric function, except that the name of the function (v) must be preceded by **FN**. Let's suppose we want to **DEF**ine a **FuN**ction that converts feet into metres. The formula for this conversion is:

x feet = x*0.3048 metres

We'll call the function that performs this conversion **M** (for metres) and it can be defined with the following statement:

```
DEF FNM(FEET) = FEET*0.3048
```

If we define this function in a program it can be called at any subsequent point by using **FNM(x)**, where **x** is any value that needs converting from feet into metres. The FEET variable in the **DEF FN** statement is known as a dummy variable, since it isn't actually used when the function is called (with the **FNM** statement), but merely establishes the role of the function's argument in the expression it calculates. The dummy variable can be of any length (since it's a standard numeric variable), but the name of the function must be a single letter.

The example program below demonstrates the value of user **DEF**ined **FuN**ctions. It creates a function which calculates the area of a circle from a radius in inches, and then converts the result into square centimetres.

```
5 REM*DEF FN*
10 DEF FNA(V)=V*2.5
20 PRINTCHR$(147)
30 PRINT"THIS PROGRAM RETURNS THE AREA"
40 PRINT "OF A CIRCLE IN CMS FROM A"
50 PRINT "RADIUS ENTERED IN INCHES"
60 PRINT:PRINT
70 INPUT "ENTER RADIUS(IN INCHES";R
80 LET I=π*R↑2
90 LET C=FNA(I)
100 PRINT CHR$(147)
110 PRINT"THE AREA OF A CIRCLE WITH A"
120 PRINT "RADIUS OF"R"INCHES"
130 PRINT"IS"C"CMS"
135 END
140 REM*DEFINED FUNCTION*
150 DEF FNA(V)=V*2.5
```

CHAPTER FOUR
LOOPS AND/OR DECISIONS

As you have probably realised, one of the main strengths of any computer lies in its ability to endlessly repeat a process or a sequence of processes without complaint. Up until now we've used **GOTO** and **GOSUB** to create processing "loops" which allow us to execute a set of instructions more than once. We're now going to introduce an equally important **BASIC** structure which provides programmers with a flexible and efficient means of repeatedly executing a series of commands a specific number of times. Such operations can be coded using **FOR...NEXT** loops.

The importance of the **FOR...NEXT** structure was hammered home to us when we were planning the early chapters of this book. We realised how difficult it was to create meaningful example programs without recourse to the versatile **FOR...NEXT** facility. In the next few pages we'll try and illustrate the power of this construction.

Let's take a very simple example. Say we wanted to **PRINT** out the numbers 1 to 10. Depending on the type of display we required, we'd be faced with a variety of fairly tedious options were it not for the **FOR...NEXT** facility. If we wanted to **PRINT** across the screen we would have to use:

```
10 PRINT 1;SPC(3);2;SPC(3);
20 PRINT 3;SPC(3);4;SPC(3);
```

and so on until we reached 10. Or if we wanted each number on a different line we would have to create an equally clumsy section of code:

```
1Ø PRINT 1
2Ø PRINT 2
3Ø PRINT 3
```

until we churned out the tenth **PRINT** statement. Obviously this sort of program is hopelessly inefficient (quite apart from devouring unacceptable amounts of memory), and totally ridiculous if you want to generate a display of numbers from 1-1000).

However, such a problem is easily resolved if we employ a **FOR...NEXT** loop. Enter and **RUN** the following example, then we'll take a look at how it works.

```
5 REM*FOR...NEXT*
1Ø FOR A=1 TO 1Ø
2Ø PRINT A
3Ø NEXT A
```

This simple loop **PRINT**s the numbers 1 to 10 to the screen. The loop statements are contained in lines 10 and 30, whilst the code to be processed is the **PRINT** statement in line 20. Since there is only one **PRINT** statement and the program has **PRINT**ed ten numbers to the screen, with each number on a different line, it should be clear that the loop has been executed ten times. All this is accomplished by line 10. When the 64 first encounters this statement it assigns the value 1 to the loop variable A (which is also known as a loop counter), which is then **PRINT**ed in line 20. The computer will continue processing the lines which follow (in this case there is only line 20), until it encounters a **NEXT** statement (line 30). The program will then loop back to line 10 and the entire process will be repeated, except the value of the loop counter A will be incremented by 1 with each pass of the loop. This will

continue until A reaches its end value, which in this case is 10 (the second parameter in line 10).

So essentially line 10 is instructing the 64 to create a variable called A which must then be assigned a sequence of values from 1 to 10. The first value assigned to A is determined by the statement's first parameter (in this case 1), and since it hasn't been instructed otherwise, the value of A will be incremented in steps of 1. However, before the value of A can be altered, the 64 must first execute the statements in the body of the loop. When it reaches the **NEXT** statement, the computer checks to see whether A has reached its end value (10), and if it hasn't processing recommences with the first statement of the loop. When A=10, processing moves onto the next line following the **NEXT** statement (or ends, as in the example above).

The start and end parameters of a **FOR...NEXT** loop can be integers, floating point values, variables or calculated expressions. It is even possible to use negative values. When the first statement of a loop takes the format above, the value of the loop variable will be incremented in steps of 1. However, by adding a **STEP** statement we can increase (or decrease) the value of the loop counter by a specified amount. For example:

```
10 REM*FOR...NEXT...STEP*
20 FOR A=10 TO 100 STEP 10
30 PRINT A
40 NEXT A
```

will execute line 30 ten times, and **PRINT** 10, 20, 30...100, since line 20 instructs the 64 to increment the value of A by **STEP**s of 10 with each pass of the loop.

Like most popular micros, the 64 allows us to omit the loop counter in the **NEXT** statement. Thus line 40 could read:

```
40 NEXT
```

However, the inclusion of the variable name does make a program considerably easier to follow, particularly if there are a lot of program lines between the loop's opening and closing statements. An absence of variable names in **NEXT** statements is also confusing when a loop is placed within another loop or series of loops (these are known as "nested" loops). As the simple example below demonstrates, in a nested loop structure the inner loop is completely processed with each pass of the outer loop. Since our example is intended to clarify this construction, it's very easy to follow. However, when loops are nested three or four deep, and the code within the loop is lengthy and/or complex, listings can become difficult to follow. Under these circumstances it is always advisable to use the complete form of the **NEXT** statement.

```
10 REM*NESTED LOOP*
20 :
30 REM*OUTER LOOP START*
40 FOR A=20 TO 5 STEP -5
50 :
60 REM*INNER LOOP START*
70 FOR B=5.25 TO 10.25 STEP .25
80 LET X=B*4.3/A
90 PRINT X
100 NEXT B
110 REM*END INNER LOOP*
120 :
130 NEXT A
140 REM*END OUTER LOOP*
150 :
160 END
170 REM*END PROGRAM*
```

You'll find examples of the **FOR...NEXT** structure in action throughout the rest of the text, and additional details about these statements in the keywords section. However, one

important use of the **FOR...NEXT** construction that deserves a mention is the creation of empty loops to pause a program for a specified period of time. Suppose you want a message to appear on the screen for as long as it takes to be read and then erased. Since Commodore BASIC has no customised pause command, empty **FOR...NEXT** loops are often used to circumvent this omission. For example:

```
10 REM*DELAY LOOP*
20 PRINT CHR$(147)
30 PRINT TAB(15)"THE SCREEN WILL"
40 PRINT TAB(15)"CLEAR IN 2 SECONDS"
50 FOR D=1 TO 1000:NEXT D
60 PRINT CHR$(147)
```

As you can see, the 64 takes approximately 2 secs to process an empty loop of 1-1000, and thus a pause of about 10 secs can be achieved by a loop that runs from 1-5000.

Having introduced the final control structure available to 64 programmers, we can now go on to look at the last remaining method of introducing data into BASIC programs. We have left this explanation of DATA statements until now because they are commonly used in conjunction with **FOR...NEXT** loops.

READING DATA

We've seen how to feed the 64 with information from the outside world using **INPUT** and **GET**, and how we can assign string and numeric values to individual variables. **DATA** statements are another means of storing data within a program, and they are normally used when a large amount of data needs to be accessed in a specific order.

This method of storing information is both straightforward to code and easy to use. The data is held in a program line (or

a series of lines) which can appear anywhere in a program. Let's take a look at some typical statements:

```
100 DATA MON, TUES, WED
110 DATA 23,56,2.6
120 DATA THURS,12,FRI,34
```

As you can see, both string and numeric data can be stored in this way; it is also possible to mix both types within the same program line. You'll notice that the string data does not require quotes, and that each item of data is separated by a comma. (However, if you want to use commas, spaces or quotes as DATA items they must be enclosed in quotes.)

So how is this information accessed? Well, **DATA** is always used in conjuction with **READ**, which must be accompanied by the appropriate variable type. For example:

```
10 REM*DATA*
20 FOR C=1 TO 6
30 READ A
40 PRINT A
50 READ A$
60 PRINT A$
70 NEXT C
80 DATA 12,RED,14,BLUE,16,CAT
90 DATA 45,MAN,34,NEXT,2,DOG
100 END
```

As soon as the 64 encounters **READ** for the first time it searches for the first item of the first **DATA** statement, against which it places a pointer. So in our example, line 30 **READ**s line 80 and places the first data item (12) into the numeric variable A which it then **PRINT**s out. The next **READ** statement is in line 50, this time accompanied by a string variable, which is just as well since our data pointer has moved on and is now placed against a string (RED). This is **PRINT**ed out, and the program loops back and repeats the

process until all twelve items of **DATA** have been **READ** and **PRINT**ed.

You must make sure that for every execution of a **READ** statement there is a corresponding number of data items, otherwise the program will be halted by an ?OUT OF DATA error. You can test this by changing line 20 to:

```
20 FOR C=0 TO 6
```

The program will also stop (this time with a ?SYNTAX ERROR) if a **READ** statement with a numeric variable is used to access a string data item.

A data item can only be **READ** once in the course of a program unless a **RESTORE** statement is used. Change line 20 back to its original form again, then make the following modification:

```
100 GOTO 20
```

If you **RUN** the program again you'll find that when the loop is executed for the second time the program is halted with an ?OUT OF DATA message. Now add:

```
95 RESTORE
```

and you'll see that each time the loop is completed **RESTORE** sets the data pointer back to the first data item and the program continues **RUN**ning indefinitely.

It is possible to include more than one variable in each **READ** statement. Thus our example could have been written:

```
10 REM*DATA*
20 FOR C=1 TO 6
30 READ A,A$
40 PRINT A:PRINT A$
50 NEXT C
60 DATA 12,RED,14,BLUE,16,CAT
```

```
70 DATA 45,MAN,34,NEXT,2,DOG
80 END
```

Needless to say, when using **READ** with multiple variables it is important to be especially careful to ensure that they are applied to the appropriate type of **DATA** item.

Now the forward thinking among you will have realised that using **READ** and **DATA** in this way means that an item of data is only stored until the next **READ** statement is encountered. In the example above, we are redefining the values of A and A$ with each pass of the loop. So how can we access all the items of data without creating a different variable for every **READ** statement? There obviously has to be an answer, otherwise **DATA** statements might just as well be replaced by line after line of **LET** statements. The solution to this particular **BASIC** mystery lies in another **DIM**ension!

ARRAY FOR DATA!

You may be relieved to learn that we are about to take a look at the last method of holding data on the 64. Say we want to **PRINT** out a calendar. Now obviously our program is going to require the days of the week and the months of the year to be stored as data somewhere in the code. We could clearly construct a different **PRINT** statement for each day and each month, or else we could add a little flexibility by assigning strings to variables. Whilst the latter is certainly the better of the two options, it nevertheless requires the creation of nineteen different variable names (twelve for the months in the year, and seven for the days of the week). Think how much simpler it would be if we could consider all the days of the week as a variable we could call DAY$, and all the months in a year as a list we could store as a variable called MONTH$. In the best of all possible worlds we could then **PRINT** out January by telling the 64 to **PRINT** MONTH$(1) and Monday by creating a line like **PRINT** DAY$(1). Well necessity being the mother of invention, precisely such a

facility is available in BASIC, and it goes by the name of an array.

The storage of data in arrays is one of the most important tools available to the BASIC programmer. The creation of arrays is essentially a means of allocating a specified amount of memory space for the storage of a list (of numbers or strings). The list itself is known to the computer by a single variable name (the array variable), and each item of the list can be assigned a number (the subscript) by which it can be accessed. In other words, in the case of our list of months, the array variable is MONTH$ and January is accessed by MONTH$(1). Thus 1 is the subscript of an array called MONTH$.

Before we can store such a list we must tell the 64 how much memory it is likely to occupy. In other words, what are the **DIM**ensions of the memory space that are going to be given over to a list of twelve strings. We do this by using a **DIM** statement.

Since storing data in arrays plays such an important role in BASIC programming, and the concept of **DIM**ensioning is often confusing to new programmers, we have included a fairly substantial entry in the keyword section. Before continuing with the rest of this chapter turn to the **DIM** entry and make sure that you understand the process of **DIM**ensioning an array.

So now you know how quantities of data can be held in variables, without an individual variable name being created for each data item. It should now be clear that the ability to access items by their subscript enables complex operations to be performed on large quantities of data with simple code that uses memory economically. The following example demonstrates the power of this storage method, when arrays are used in conjunction with **FOR...NEXT** loops.

```
5 REM*DATA EXAMPLE*
10 GOSUB 500
20 PRINT S$;TAB(15);"MENU":PRINT
30 PRINT"DO YOU REQUIRE:":PRINT
40 GOSUB 300
50 PRINT TAB(12);"ENTER 1,2 OR 3"
60 GET A$:IF A$="" GOTO 60
70 LET A=VAL(A$)
80 IF A<1 OR A>3 GOTO 60
90 GOSUB 400
100 PRINTTAB(10);"ENTER DAY OF WEEK"
110 PRINTTAB(9);"EG:SUN=1,TUES=3 ETC"
120 GET F$:IF F$="" GOTO 120
130 LET F=VAL(F$):IF F<1 OR F>7 GOTO 120
140 GOSUB 200
150 END
160 :
180 REM*SUBROUTINES*
190 :
200 REM*DAY*
210 PRINTS$
220 FOR V=1 TO 7
230 PRINT
240 READ V$(V)
250 NEXT V
260 PRINT Q$;"TODAY IS ";V$(F);"DAY AND YOUR ";X$(A);" IS:
270 PRINT:PRINT TAB(15);CHR$(156);Y$(A)
280 RETURN
290 :
300 REM*MENU*
310 FOR X=1 TO 3
315 LET P=P+5
320 READ X$(X)
330 PRINT CHR$(P);TAB(12);X;X$(X)
340 NEXT X
350 PRINT CHR$(144):PRINT
360 RETURN
370 :
400 REM*CHOICE*
410 FOR Y=1 TO 3
420 READ Y$(Y)
430 PRINT
440 NEXT Y
450 PRINT CHR$(147)
460 RETURN
470 :
500 REM*VARIABLES*
510 LET P=143:LET Q$=CHR$(145)
520 LET S$=CHR$(147)
530 RETURN
540 :
600 REM*DATA*
610 DATA GREETING, CURSE,FAREWELL
620 DATA HELLO SAILOR!,GET LOST!,GO IN PEACE
630 DATA SUN,MON,TUES,WEDNES,THURS,FRI,SATUR
640 DATA GREETING, CURSE,FAREWELL
READY.
```

LOGICAL CONTROL

By now you should be clear about the essential elements that go to make up any BASIC program. We've looked at ways data can be stored and displayed, and how the 64's string and numeric functions can be used to process the data. We've also seen how the order in which processing occurs in a program can be directed by using loops, subroutines and **GOTO** statements. Now it's time to look at the way in which the 64 can be programmed to test data to determine the order and nature of processing by using the **IF...THEN** statement.

Most computers-in-the-future horror stories centre around the intelligence of the machines that take over the world. From what we have seen so far, your 64 does not constitute much of a threat as regards world domination, but with the **IF...THEN** statement it is possible to create programs in **BASIC** that simulate intelligence.

In our discussion of **FOR...NEXT** loops we saw how the 64 tested the value of the loop counter to see whether or not it equalled the loop's end value before deciding whether to re-execute the loop or move on to the next statement in the program. It is this ability to logically compare values that provides the basis for the **IF...THEN** construction.

In spite of appearances to the contrary, the world seen through the eyes of the Commodore 64 is very simple Something is either True of False, and it has no capacity to come to any other form of conclusion when assessing a particular condition. **BASIC** uses this somewhat uncompromising quality to good effect when testing and comparing values in order to determine a course of action.

Like so many **BASIC** commands, the **IF...THEN** statement has a self-explanatory format: **IF** (condition) **THEN** (action).

The condition after **IF** can be one of a variety of alternatives. The table below contains all the conditional operators that

can be used to determine the relationship between values:

= **equal to**
< **less than**
> **greater than**
<= **less than or equal to**
>= **greater than or equal to**
<> **not equal to**

Thus the **IF...THEN** statement is split into two distinct sections. The first part contains the expression which establishes a relationship between values, and tests for its truth or falsity. **IF** the expression is false, the 64 doesn't even bother to look at the second half of the statement, but simply moves on to the next program line. **IF** the expression is true, **THEN** the instructions in the latter half of the statement are implemented.

The instructions which follow **THEN** can be any legal BASIC statement. Thus if a condition is True a value can be assigned to a variable:

```
IF x>10 THEN y=15
```

Or a program can be terminated:

```
IF D$="N" THEN END
```

One of the most powerful applications of the **IF...THEN** construction is when it is used to direct control in a program:

```
IF V<>2 THEN GOSUB 500
```

If the consequence of a True expression is the execution of a **GOTO** statement, the format is:

```
IF V<=4 THEN 50
```

which means that the program must **GOTO** line 50, although the keyword **GOTO** is omitted.

One of the more tedious aspects of creating interactive programs in **BASIC** is that a large percentage of the code must be devoted to guarding against human stupidity. As a general (and fairly inflexible) rule, **INPUT** statement should always be followed by some form of **INPUT** check. For example:

```
10 REM*INPUT CHECK*
20 INPUT "ENTER A NUMBER (1-10) ";N
30 IF N<1 OR N>10 THEN 20
40 PRINT "CONDITION SATISFIED"
```

Just because a user is asked to enter a number between 1 and 10 it does little in the way of ensuring that a value in the correct range will actually be keyed-in. (Unlike computers, humans excel in the contradiction of instructions!) However, the inclusion of line 30 in the example gives the user of the program very little choice in the matter. **IF** the number entered (N) is less than (<) 1 or greater than (>) 10, the program will simply return to the **INPUT** statement in line 20.

You'll notice that we've used the logical condition operator **OR** to combine two expressions in the first part of line 30. Commodore **BASIC** provides us with three such operators (the other two being **AND** and **NOT**), all of which are fully described under their respective entries in the keyword chapter. For now we'll simply look at a few examples which demonstrate how they can be used to combine conditions in **IF...THEN** statements.

BOOLE WITHOUT BOVVER

The importance of logical (or Boolean) operators should be clear from our **INPUT** check example. If the check is to serve its purpose it is obviously essential that both expressions are

True before the program continues. Thus the statement in line 30 uses **OR** in exactly the same way as the word is used in English: IF N is either less than 1 **OR** if N is greater than 10 return to line 20. But we're jumping the gun a little. Let's look at each of the operators in turn, and explain the consequences of the Truth or Falsity of the expressions they combine. The important point to remember is that although more than one expression can be tested by an **IF...THEN** statement, there can only be a single evaluation of the statement as a whole. The testing of logical conditions reveals another difference between micros and men. For the 64 there is no such thing as a half truth.

AND

Having repeatedly insisted that the 64 can only evaluate a conditional expression as either True or False, it's time to back-pedal a little in the interests of precision. Whilst the outcome of a logical test can only be True or False, the 64 does not actually have these words in its vocabulary. When asked to pass judgment on an expression, the computer returns either -1 or 0, which are its representations of True and False respectively. You can test this out by entering a couple of statements as direct commands. For example:

```
LET A=5:PRINT A>10
```

will return a 0, since the expression that A is greater than 10 is False (0). Try:

```
LET A=5:PRINT A<>10
```

Since A is clearly not equal to 10, the expression is True (-1) and thus -1 is returned.

So the 64 is obviously capable of logically assessing the Truth or Falsity of an expression. However, it is often necessary to relate a number of expressions to one another

to discover the logical status of the combined expression. When **AND** is used to relate expressions the final result will only be True if each element is True. For example:

```
PRINT 4=4 AND 3=2
```

will **PRINT** 0 (False), because although the first part of the expression (4=4) is True, the second part (3=2) is False and thus the expression as a whole is False. When such an expression is used with **IF...THEN** the complete statement resembles an English sentence whose meaning is quite clear:

```
10 REM*AND1*
20 INPUT"ENTER A NUMBER";N
30 IF N>=5 AND N<=10 THEN 60
40 PRINT "FALSE"
50 END
60 PRINT "TRUE"
```

Obviously the 64 will only get the chance to reach line 60 and **PRINT** out True if the number entered falls between 5 and 10. We can also use this sort of construction to relate string expressions:

```
100 IF S$="F" AND M$="Y" THEN A$="MRS"
```

The traditional (and indeed clearest) method of describing the effects of combining expressions is by means of a Truth table. What follows is a Truth table for **AND**, in which the results listed under the headings Condition1 and Condition2 define the logical status of the component expressions related by **AND**:

Condition1		Condition2	Result
TRUE	AND	TRUE	TRUE
TRUE	AND	FALSE	FALSE
FALSE	AND	TRUE	FALSE
FALSE	AND	FALSE	FALSE

As the table graphically demonstrates, every element of a multiply conditional expression whose components are related by **AND** must be True before the expression as a whole can be deemed True (-1).

OR

OR is used as a relational operator when a conditional statement must return True if any one **OR** all of its component expressions are True. Thus:

```
100 IF A=10 OR B=-1 THEN 450
```

will pass control to line 450 if either A=10 **OR** B=-1 **OR** if both conditions are True. As with **AND, OR** can also be used to test conditions relating expressions using strings:

```
200 IF A$="Y" OR B$="N" THEN 10
```

Thus the Truth table for **OR** is:

Condition1		Condition2	Result
TRUE	OR	TRUE	TRUE
TRUE	OR	FALSE	TRUE
FALSE	OR	TRUE	TRUE
FALSE	OR	FALSE	FALSE

since **OR** ensures that a conditional statement in which it appears is only False if all its component expressions are False.

NOT

This is neither the time nor the place to discuss the perennial debate on whether **NOT** should be used at all as a relational operator. Suffice it to say that **NOT** should always be used with care since its use does not always produce such immediately predictable results as its logical colleagues **AND** and **OR**. Essentially **NOT** inverses the logical status of the expression it precedes. However, whilst **NOT**(False) will always return True, **NOT**(True) will not always return False. We'll deal with this little dichotomy along with the mysteries of logical bitwise operations in the keyword chapter. For the moment, let's take a look at **NOT**'s disconcertingly simple Truth table:

NOT (Expression TRUE) = FALSE
NOT (Expression FALSE) = TRUE

Thus:

```
PRINT 6=6 AND 5=5
```

will return True(-1). And:

```
PRINT NOT (6=6 AND 5=5)
```

will return False(0). Predictably enough **NOT -1** will return False (0) and **NOT 0** True(-1).

So when used within conditional **BASIC** statements, the effects of the three relational operators is quite straightforward. Indeed **IF...THEN** statements combining expressions with the operators are close enough to their counterparts in English to be virtually self-explanatory. However, the operators can also be used to logically relate the binary representations of numbers in the 64's memory.

As we'll hopefully clarify in the appropriate entries in the keyword chapter, these operations are no less logical than their condition testing counterparts, but initially their results are considerably less predictable. For example try:

```
PRINT 45 AND 1Ø
```

or

```
PRINT 6 OR 5
```

or

```
PRINT NOT 65
```

Don't worry! All will be revealed. For the moment take some time out to construct some demonically complex expressions using one or more of the operators. Use the Truth tables if you are at all surprised by the results the 64 comes up with.

CHAPTER FIVE
THE KEY TO BASIC

One of the most conspicuous weaknesses of the User Guide is that it fails to provide programmers with anything like complete definitions of each of the Commodore **BASIC** keywords. Now that we have offered a schematic introduction to the logic and constructions available to 64 programmers, we shall endeavour to systematically detail all of the 64 keywords (or "reserved" words).

The keywords are presented alphabetically in a standard format. Each entry comprises a format example using the following standard symbols:

V= numeric variable name
A$= string variable or expression
i= integers or whole numbers
n= floating point or decimal value
c= conditional logical expression
ln= program line number
addr= a memory location address
x,y= represent screen coordinates

Any special symbols required to clarify particular keywords will be defined within the entry in question.

When appropriate, we have included a short example program designed to show the keyword in action and a list of related keywords which should be referred to as a means of

situating commands in their group context. Finally, all the keywords are accompanied by their **BASIC** token values, which is the means by which the 64 stores commands in memory in a single byte form.

ABS

SYNTAX: v = ABS(n)

TOKEN: 182

Function which returns the positive value of a numeric variable or expression.

There are occasions when it is essential to ensure that particular calculations return a positive result irrespective of the kind of data they have to process. Without the **ABS** function this would be something of a problem. Suppose the computer has to calculate the sum of (x-y), and yet always return a positive result. Obviously this will only be possible as long as x is greater than y. However, if we use a statement like this :

```
10 PRINT ABS(x-y)
```

the answer will always be positive. Thus **ABS**(-637) will return 637 and **ABS**(5-6) will return 1.

```
5 PRINT CHR$(147)
10 FOR A=1 TO 5
20 INPUT "AVALUE";B(A)
30 D=INT(ABS(B(A)))
40 L(A)=LEN(STR$(D))
50 IF D=0 ANDB(A)<>0THEN L(A)=(A)-1
60 NEXT
70 PRINT  CHR$(147)
80 FOR S=1 TO 5
```

```
90 PRINT TAB(18-L(S))B(S)
1000 NEXT
```

The program above uses **ABS** along with **INT** to produce a tidy table of numbers, regardless of whether they are positive or negative, integer or floating point. As well as returning positive results, **ABS** also rounds up to seven decimal places. Thus :

PRINT ABS(12.12345678)

will return :

12.1234568

RELATED KEYWORDS: SGN

AND

Syntax: condition **AND** condition
expression **AND** expression

AND is one of the logical operators available in CBM64 **BASIC**. The others are **OR** and **NOT**. **AND** performs logical conjunction on conditional expressions or numeric expressions, as described below. The two operations are, from the point of view of the **BASIC** programmer, separate, although the operations performed by the **ROM** routine of the 64 is the same.

Conditional statements used in **IF...THEN** statements are evaluated as 'True' or 'False', the statement(s) following **THEN** being executed if the condition is 'True'. The 64 uses the value -1 for 'True' and 0 for 'False'. Conditions may be combined using **AND**. Each condition is first evaluated as True or False, and the results combined as follows:

Condition 1	Condition 2	C1 AND C2
True	True	True
True	False	False
False	True	False
False	False	False

Thus the result of **AND**ing two conditions is true only if both conditions are true, and is false if either or both are false.

Multiple expressions may be constructed. If we have a program line:

```
200 IF A=B AND X$="N" AND D<93 THEN 900
```

then the expressions are evaluated in sequence, left to right, as with arithmetic expressions. **A=B AND X$="N"** is evaluated, and the resulting 'True' or 'False' **AND**ed with the true or false result of D93. If all three expressions are true control will pass to line 900. We can alter the sequence of evaluation using brackets:

C1 AND C2 AND C3

evaluates as (C1 **AND** C2) **AND** C3, as above, whereas

C1 AND (C2 AND C3)

will be evaluated with the result of C2 **AND** C3 being found first, and this then used to **AND** with C1. Whilst this makes not the slightest difference with a sequence of three conditions **AND**ed together, conditions may be combined using both **AND** and **OR**. (See the entry on **OR**.) The expression :

```
IF A>2 AND B$<"ZZ" OR M=N
```

will produce entirely different results with the addition of brackets to force the evaluation of the second pair of

conditions to occur first:

```
IF A>2 AND (B$<"ZZ" OR M=N)
```

Combined conditions can help in error checking. For example, any mutually exclusive pair of conditions is easily checked using **AND**:

```
200 INPUT"END";E$
210 IF E$<>"Y" AND E$<>"N" THEN PRINT "E
RROR Y OR N ONLY! TRY AGAIN":GOTO 200
```

The second use of **AND** is in logical bitwise operations. When used with numeric expressions, the 64 first evaluates the expression, and then converts the result to a signed two-byte integer value, -32768 to +32767. The highest bit of the 16 bits is used as a sign bit, being set to 1 if the number represented by the remaining 15 bits is negative. **AND**ing then takes place on each bit of the two numbers, with the result being a one if both the bits in this position of the numbers being **AND**ed were one, and zero otherwise. Thus 17 **AND** 62, with the result 16, is performed as follows:

	Sign bit	**High Byte**	**Low Byte**
17	**(0)**	**0 0 0 0 0 0 0**	**0 0 0 1 0 0 0 1**
AND			
62	**(0)**	**0 0 0 0 0 0 0**	**0 0 0 1 1 1 1 0**
16	**(0)**	**0 0 0 0 0 0 0**	**0 0 0 1 0 0 0 0**

The principal use of bitwise operations is in bit masking, in order to specify the state of a particular bit or bits within a byte. However, a useful function of **AND** simulates the **MOD** function found in some **BASIC**s, but only on powers of two.

The expression X **AND** 255 will give the remainder after X has been divided by 256, X **AND** 7 gives the remainder when X has been divided by 8, and so on for the other powers of two.

RELATED KEYWORDS: NOT, OR

ASC

SYNTAX: v=ASC(a$)

TOKEN: 198

A function which returns a number between 0 and 255 which corresponds to the ASCII code of the first character of the statement's argument.

The 64 does not store the actual characters as they are seen on your screen, but holds them as numbers known as ASCII character codes. ASCII stands for American Standard Code for Information Interchange, and each character in the Commodore's character set is identified by its own unique ASCII number. Hence, the letter R is not stored as the letter itself, but as its ASCII code value which is 82. **ASC** complements **CHR$**, which returns the character represented by the code of its argument. If **ASC** is applied to a null string an **ILLEGAL QUANTITY** error will be generated.

The program which follows demonstrates the role of **ASC** in input checks. Line 40 tests to see whether the code of the character that has been entered falls within the range 48-57, which represent the numbers in the 64's character set. Unless the input is a number, it will not be printed to the screen.

```
10 REM*ASC*
20 PRINT CHR$(147)
30 INPUT "ANY STRING";A$
```

```
40 IF ASC(A$)>=48 AND ASC(A$)<=57 THEN
PRINT TAB(25) A$
50 GOTO 30
```

RELATED KEYWORDS: CHR$, STR$, VAL

ATN

SYNTAX: v=ATN(n)

TOKEN: 193

Returns the radian whose **A**rc**T**a**N**gent is n. The value returned is given in radians, and will fall between -PI/2 and PI/2.

Like all the trigonometric functions on the 64, **ATN** is, of course, invaluable when attempting any serious graphical work on the machine. It can be regarded as the inverse of the **TAN** function, and can be used to simulate inverse functions to complement the 64's other trigonometric functions (**COS** and **SIN**). Note that the value returned by this function is always measured in radians. To convert radians to degrees simply multiply the value by 180/PI (since 2*PI radians = 360 degrees).

RELATED KEYWORDS: COS, SIN, TAN

CHR$

SYNTAX: A$=CHR$(N)

TOKEN: 199

String function which converts the ASCII code specified by its argument (N) into string form.

CHR$ is the string function which complement **ASC**. While the latter returns the ASCII code of a string character, **CHR$** performs the reverse operation. For example :

```
10 A=ASC("A")
20 PRINT A
30 A$=CHR$(65)
40 PRINT A$
```

Line 10 assigns the code for the character "A" to the numeric variable A, which is **PRINT**ed in line 20. Thus 65 is displayed on the screen. Line 30 then uses **CHR$** to convert 65 into a string, and thus "A" is **PRINT**ed in line 40. So we can use the following program to display the 64's upper-case alphabet :

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

As well as providing access to the 64's standard character set, **CHR$** can also be used to control facilities such as cursor movement and text colour. Add the following lines to the example above and the upper case characters will be **PRINT**ed in a variety of colours :

```
5 FOR B=149 TO 154
40 PRINT CHR$(B)
50 NEXT B
```

Adding a further line will clear the 64's screen :

```
1 PRINT CHR$(147)
```

There is a full list of ASCII codes in an Appendix. A little experimentation with **CHR$** will reveal the value of this flexible function.

RELATED KEYWORDS: ASC, STR$, VAL

CLR

SYNTAX: CLR

TOKEN: 156

This command voids the values of all variables currently in use.

The execution of **CLR** leaves the **BASIC** program itself intact, but clears all variable assignments and makes available the sections of **RAM** occupied by values which are no longer required by the program.

In the example program below, a series of values are assigned to the array C(A) and then **PRINT**ed out (lines 20-40). The variables are then **CL**ea**R**ed in line 50. When we try to **PRINT** out the values of C(A) again (lines 60-70) we find that the 64 simply returns 0 for each element of the array.

```
1Ø REM*CLR*
2Ø FOR A=1 TO 1Ø
3Ø LET C(A)=A+2
4Ø PRINT C(A):NEXT A
5Ø CLR
6Ø FOR B=1 TO 1Ø
7Ø PRINT C(B):NEXT B
```

RELATED KEYWORDS: RUN, NEW

CMD

SYNTAX: CMD i,(a$)

TOKEN: 157

Input/Output statement which forces the 64's main output

device to divert screen output to the file specified by its argument (i).

The optional string (the a$ which appears in brackets in the syntax example, but does not require them when the statement is actually used) is sent to the specified file when the statement is executed.

CMD is primarily of value when listing data to a printer. Under these circumstances the optional string (a$) which can follow the specified file number (i) is useful as a means of labelling a printout.

CMD statements can also be used to transfer files to disks, tapes or other I/0 peripherals (e.g. modems). Before a **CMD** statement can be used a file number must be established. This is achieved by means of an **OPEN** statement e.g. **OPEN** 3,1,1,"CODE":CMD 3, where 3 is the name assigned to the file. Once a **CMD** statement has been executed, the contents of all subsequent **LIST** and **PRINT** statements will be sent to the specified file, rather than displayed on the screen. To reverse the effects of **CMD** an empty line must be sent to the output device (using a **PRINT** statement), followed by **CLOSE**. Any error reports will also force a return to screen display.

RELATED KEYWORDS: OPEN, CLOSE, PRINT

CONT

SYNTAX: CONT

TOKEN: 154

CONTinues the processing of a program from the point that it has been halted by **END**, **STOP** or the **RUN/STOP** key.

When developing a program it is often important to know the value of a particular variable at specific points in its execution. By using the **RUN/STOP** facility the program can be halted, the values checked, and then re-started using the **CONT** command. The 64 will re-commence processing at the precise point at which it was interrupted.

CONT can also be used to re-start a program that has been halted by an **END** or **STOP** statement. Once again, processing will continue from the line following the **STOP/ END** statement. However, you cannot **CONT**inue a program if any alterations have been made.

RELATED KEYWORDS: END, STOP, RUN, GOTO

COS

SYNTAX: v = COS(n)

TOKEN: 190

Returns the cosine of its argument.

It should be remembered that the 64's trigonometric functions all return values measured in radians, and that in order to calculate the angle in degrees, v must be multiplied by 180/PI.

RELATED KEYWORDS: ATN, SIN, TAN

DATA

SYNTAX: DATA v,v,v...
DATA v$,v$,v$...

TOKEN: 131

A statement used to store data within the body of a program.

This statement is used in conjunction with **READ** and **RESTORE** to access string and numeric data as and when required by a program. Here's an example of **DATA** statements in action – it produces an example of a 64 **DATA** statement!

```
1 REM*DATA 1*
5 PRINT CHR$(147)
10 FOR A=0 TO 10
20 READ A$: PRINT A$
30 NEXT A
40 DATA AN,EXAMPLE,OF,A
50 DATA 64,"DATA STATEMENT:","
"
60 DATA 10,DATA,"1,","2,",GREEN
```

One of the many wonders of Commodore **BASIC** is that **DATA** statements can be positioned anywhere in a program, although what passes for standard practice normally dictates that the statements are coded at the end of a program or routine (as in our example). So how do **DATA** statements work? Well, in the program above the loop in lines 10-30 **READ**s and then **PRINT**s **DATA** stored in lines 50 to 70 (there are eleven items of data, and so the loop must make eleven passes, **READ**ing a single **DATA** item at each pass). Thus, whilst the position of the **DATA** statements is of no consequence, the point at which the program must **READ** the statements is critical.

Under normal circumstances **DATA** can only be **READ** once. This is because each time **DATA** is **READ** a pointer in the 64's memory (Zero-page) moves on to the next data item, until the store of **DATA** is exhausted. To test this out change line 10 to read:

```
10 FOR A=1 TO 11
```

and **RUN** the program again. All will be well until the 64 tries

to process the loop for the twelfth pass. Since there are only eleven items stored, the 64 throws up an Out of Data error. However, it is possible to re-use the **DATA** with the use of the **RESTORE** statement which returns the data pointer to the first **DATA** item. Change line 10 back to the original version and add the following line:

```
35 GOTO 1Ø
```

If you **RUN** the program again the 64 once again generates the same error message. However, if you now add:

```
34 RESTORE
```

the computer will cheerfully **PRINT** out the data until the program is halted.

RELATED KEYWORDS: READ, RESTORE

DEF

SYNTAX: DEF FN v(z)=exp

TOKEN: 150

Allows the creation of user defined functions.

This statement facilitates the creation of functions which, once defined, can be used repeatedly at subsequent points in the program. For example, let's assume that we want to use a relatively complex expression (like the one defined in line 10 of our example program), and apply a variety of values to this expression. The **DEF FN** statement allows us to carry out such an operation without having to repeat the expression each time we need to use it. **RUN** the following example:

```
1 REM*DEF FN*
```

```
5 PRINT CHR$(147)
10 DEF FNB(V)=V/45*(π/.40)
20 FOR A=1 TO 10
30 PRINT FNB(A)
40 NEXT A
```

The **FuNction** is **DEFined** in line 10. The loop (lines 20-40) then uses its variable which takes the place of V in the expression, which has been called by **FN** in line 30.

Let's look at each component of the statement. In the format line at the top of the page, v (a one or two letter variable name) completes the name by which the **FuNction** can be recalled (using the **FN** statement), later in the program. The z in brackets is the name which is used in the argument in the function and is what is known as a dummy variable. The (exp) following the equals sign represents an expression using z (the dummy variable).

RELATED KEYWORDS: FN

DIM

SYNTAX: DIM v(i,i,...),v(i,i...)
DIM v$(i,i,...),v$(i,i,...)

TOKEN: 134

A statement which allows space to be allocated for the storage of arrays.

An array allows us to store related data – integers, floating point numbers or strings – as elements of a list (in the case of a one dimensional array), or as elements in lists (in multi-dimensional arrays). The simplest type of array is a one dimensional numeric array. For example:

DIM V(12)

will set aside enough memory space for the storage of thirteen numbers (0-12). The first example program shows how this statement could be programmed:

```
5 PRINT CHR$(147)
10 DIM A(12)
20 FOR B=0 TO 12
30 LET A(B)=B+5
40 NEXT B
50 FOR E=0 TO 12
60 PRINT A(E)
70 NEXT E
```

The array A is **DIM**ensioned in line 10. The number in brackets (which is known as the subscript), informs the computer that it must make space in memory for up to a maximum of thirteen numbers. Each value created by line 30 is assigned to an element of the array, whose subscript is determined by the loop variable. Thus the assignment of values to each element of the array is A(0)=5, A(1)=6, A(2)=7...A(12)=17. These values are then **PRINT**ed out by the second loop at line 60, where on this occasion the subscript is determined by loop variable E.

If an array is created but not **DIM**ensioned, the 64 automatically creates space for a maximum of 11 elements (0 to 10). Thus:

```
10 FOR A=0 TO 10
20 LET B(A)=A*A
30 PRINT B(A)
40 NEXT A
```

is a perfectly acceptable construction which will **PRINT** out

the values of B(0) to B(10) without complaint. However, if we now make the following change:

```
1Ø FOR A=Ø TO 11
```

all will be well until the twelfth and final pass of the loop, at which the program will halt with a "?BAD SUBSCRIPT" error. This can be rectified with the addition of:

```
6 DIM B(11)
```

which ensures that the 64 will now allocate enough space for a twelve element array (0-11).

It is also possible to create arrays which allocate space for more than one list of numbers. This can be achieved by the use of multi-dimensional arrays. The format for this is as follows:

```
DIM A(i,j)
```

in which i is the number of elements per list, and j the maximum number of lists. Thus:

```
DIM A(13,Ø)=4
```

sets the fourteenth location of the first list equal to 4. The use of multi-dimensional numeric arrays can be seen in this program:

```
1Ø PRINT CHR$(147)
2Ø DIM A%(3,4)
3Ø FOR X=Ø TO 3
4Ø FOR Y=Ø TO 4
5Ø A%(X,Y)=X*3+Y
6Ø NEXT Y: NEXT X
7Ø FOR M=Ø TO 4
```

```
80 FOR L=0 TO 3
90 PRINT A%(L,M)
100 NEXT L: PRINT: NEXT M
```

If you key-in and **RUN** this example you will be confronted with a graphic representation of the concept of multi-dimensional arrays – five columns of four figures. What about the % following the array variable A? Well, it signifies that the values that are going to be sorted will be integer numbers rather than floating point values. The point of creating integer arrays is that they save memory as the 64 sets aside less space for this type of array than it does for a non-integer array.

We can also use arrays for the storage of strings. Let's have a look at a one dimensional string array:

```
5 PRINT CHR$(147)
6 DIM A$(15)
10 FOR V=0 TO 15
20 READ A$(V)
30 NEXT V
40 DATA CAT,MAT,TRADE,ROUGH,DIVE
50 DATA SIN,ROLL,%%,&&,20,111
60 DATA 54,IN,OUT,$$,245
70 FOR X=0 TO 15
80 PRINT A$(X)
90 NEXT X
```

As you can see, the dimensioning of one dimensional string arrays is exactly the same as with its numeric counterpart – except that the array variable must be followed by the $ sign. You can store multiple lists of strings in multi-dimensional string arrays which, once again, work in exactly the same way as numeric arrays.

When allocating space for arrays you are, in practice only

restricted by the amount of available **RAM**. Any number of **DIM**ensions are permitted, with up to a maximum of 255 subscripts.

It is important to note that a **DIM** statement can only be executed once in a program. If a **DIM** statement is processed more than once the program will be halted with a "REDIM'D ARRAY" error.

It is worth remembering that once an array has been **DIM**ensioned, the specified amount of memory will be put to one side by the 64 whether or not it is actually used. Hence it is important to ensure that your **DIM**ensioning is precise, or valuable memory will be wasted, Finally, although an array can be **DIM**ensioned at any point in a program before it is actually used, it is good practice to **DIM**ension at the beginning of a program to minimise the chances of accidental re-**DIM**ensioning.

RELATED KEYWORDS: LET

END

SYNTAX: END

TOKEN: 128

A command that halts a program.

This command tells the 64 to **END** a program. It works in much the same way as the **STOP** command, but, when **END** is used there is no ?BREAK IN In message. You can then run the program from the line following the **END** command by using **CONT** or **GOTO** In.

```
10 REM*END*
20 PRINT"WHEN THE PROGRAM ENDS TYPE CONT
```

```
, AND IT WILL CONTINUE FROM LINE 40"
30 END
40 PRINT "THIS WILL BE PRINTED WHEN THE
PROGRAM ISCONTINUED"
```

RELATED KEYWORDS: STOP, CONT

EXP

SYNTAX: v=EXP(n)

TOKEN: 189

This function raises e to the power of its argument (n).

As you probably know, if we talk of raising 3 to the power of 4 (3 ↑ 4), we are calculating:

3*3*3*3

and the **EXP** function calculates the value of 2.71828183 raised to the power of the value contained in brackets. The value of e is a magic number well known to mathematicians, not least because:

EXP(n)

returns the natural antilog of n. The example program demonstrates **EXP** along with a number of other 64 functions in the evaluation of complex mathematical expressions:

```
1 REM*EXP*
5 PRINT CHR$(147)
10 INPUT "INPUT A NUMBER";X,Y
```

```
2Ø IF LOG (X)<Ø THEN X=1/X
3Ø PRINT SQR(X)*EXP(Y)
4Ø INPUT "ANOTHER VALUE (Y/N)";N$
5Ø IF N$="Y" THEN GOTO 5
6Ø STOP
```

RELATED KEYWORDS: LOG

FN

SEE DEF

TOKEN: 165

FOR...TO...NEXT...STEP

SYNTAX: FOR v=n TO n (STEP n) NEXT v

TOKEN (FOR...TO...STEP): 129...164...169
TOKEN (NEXT): 130

A statement which creates a loop structure that processes the instructions contained within the body of that loop (the amount of times that were specified in its opening line). The loop is terminated by a **NEXT** statement.

In the final analysis, the most important quality of any microcomputer lies in its ability to endlessly carry out boring, repetitious tasks without complaint. **BASIC** offers a number of structures which enable the heartless programmer to force his/her micro to carry out such tasks efficiently, and the **FOR...NEXT** loop is one of the most important.

Loops are easy to use and, once mastered, provide the programmer with one of the most valuable facilities of the **BASIC** language. Take a simple example. Say we want to

PRINT the numbers 0 to 10. Without a loop the coding would be hideously tedious, requiring an individual **PRINT** statement for each number. Loops make life considerably easier:

```
1Ø FOR A=Ø TO 1Ø
2Ø PRINT A
3Ø NEXT A
```

What could be simpler? Let's take it line by line. In line 10 the number of times that the instructions within the loop are to be executed is established. In this case it is eleven times (0-10). Thus the first time the loop is processed A (the loop counter) will be equal to 0, the second pass and A=1, the third A=2, and so on until A=10. In line 20 the value of the loop counter is **PRINT**ed out. When it reaches line 30 the **NEXT** statement tells the computer to check to see if the loop counter has reached the maximum value specified in the first line of the loop (in this case 10). If A is not yet equal to 10, the **NEXT** statement forces the computer to return to the beginning of the loop and start the process all over again.

In this example, the value of A is incremented in steps of 1. This will always be the case, unless we make use of the optional **STEP** command. Key-in the following modification to our example:

```
1Ø FOR A=Ø TO 1ØØ STEP 1Ø
```

If we **RUN** the program again, instead of **PRINT**ing out 0,1,2,..100, the 64 will **PRINT** in **STEPS** of 10,20,...100. It is also possible for the **STEP** size to be negative. For example:

```
1Ø FOR A=3Ø TO 5 STEP -5
```

The possibilities are virtually endless. There's no reason why the **STEP** size should necessarily be a whole number. Try this:

```
10 FOR A=20 TO 0 STEP -.25
```

In fact, the **FOR...NEXT** loop is so flexible that all its values can be non-integer, variables or even calculated expressions. Add these lines to test these wild claims:

```
5 X=56:C=π/2
10 FOR A=2*4 TO X/3 STEP C
```

If you use a **FOR...TO** statement but neglect to close the loop with a **NEXT**, the program will halt with a **FOR WITHOUT NEXT** error message.

The inclusion of the loop variable in the **NEXT** statement is optional, but it is advisable to leave it in since it makes a program much easier to read, particularly if you have created a series of loops nested within each other.

An 'empty' **FOR...NEXT** loop can be used to pause a program:

```
FOR A=0 TO 500: NEXT A
```

This statement will pause the program for 1 second (0-1000 will pause it for 2 seconds and so on). Using the **FOR...NEXT** statement in this manner forces the 64 to execute the loop the specified number of times, thus pausing the program until it has performed this task. This is very useful when you want to display text to the screen or slow down a particular process. When it is used in this manner it is referred to as a delay loop.

Finally, never jump out of a loop with a **GOTO** or **GOSUB** statement. If a crisis forces you into a position where a loop jump is unavoidable you must always return to the loop and allow it to complete its specified number of passes. Failure to comply with this rule will sooner or later result in the program crashing.

RELATED KEYWORDS: none

FRE

SYNTAX: FRE(v)

TOKEN: 184

A function which returns the number of available bytes in **RAM**. its argument (v) is a dummy variable.

If you are developing a lengthy program it may be important to discover how much memory the 64 has free. The **FRE**(v) function returns the number of bytes yet to be consumed by the current program in memory. If the number of available bytes exceeds 32767 then **FRE** will return a negative value. (Under these circumstances there is probably very little to worry about. However, if you still want a memory report simply add 65535 to the value returned by **FRE**.)

RELATED KEYWORDS: none

GET #

SYNTAX: GET A$,A...

TOKEN:

Command which facilitates single character input that can be entered without requiring <**RETURN**>.

One of the drawbacks of **INPUT** statements is that they require <**RETURN**> before the user-generated data is entered into the computer. Under certain circumstances, the application of **GET** statements offers programmers a way round this problem.

The example below demonstrates a standard application of the command:

```
10 REM*GET1*
20 PRINT CHR$(147);"THE PROGRAM IS RUNNI
NG"
30 PRINT:PRINT"BUT HAS BEEN STOPPED BY A
 GET STATEMENT"
40 PRINT:PRINT"PRESS ANY KEY TO CONTINUE
"
50 GET A$:IF A$="" GOTO 50
60 PRINT:PRINT"NOW THAT YOU'VE PRESSED A
 KEY"
70 PRINT:PRINT"GET HAS ALLOWED THE PROGR
AM TO END"
80 END
```

RELATED KEYWORDS: INPUT

GET

SYNTAX: GET i,(variable list)

TOKEN: 161

Input/Output statement which enables single bytes of data to be read from the specified file (i).

When a **GET** is executed, the 64 reads data from a specified file into the variables following the file number (i). If no characters are received, the variable in question is assigned a null string (in the case of string variables) or zero (for numeric variables).

RELATED KEYWORDS: INPUT, OPEN, CLOSE, CMD

GOSUB

SYNTAX: GOSUB ln
GOSUB v
ON v GOSUB ln

TOKEN: 141

A statement which transfers control from the main body of a program to a subroutine starting at line number ln. Control is returned to the main program by means of a **RETURN** statement.

The creation of subroutines utilising **GOSUB...RETURN** statements provide **BASIC** programmers with a means of creating code which is not only processed efficiently, but is also easy to follow. A **GOSUB** statement resembles **GOTO** in as much as it forces the 64 to execute instructions from the specified line number (rather than following the line numbers in sequence). However, **GOSUB** is considerably more powerful than **GOTO** because as well as directing control to a specified part of the program, it also stores a return address in memory. Thus once it has processed the instructions contained in the subroutine, a **RETURN** statement forces control back to the line that follows the original **GOSUB** statement.

The line number specified by a **GOSUB** can be literally quoted, a variable or the result of a calculated expression.

One important rule to remember when using subroutines is that they should never be jumped out of with a **GOTO**, but should always **RETURN** to the main body of the program. Apart from being the height of bad programming practice, jumping out of subroutines and not **RETURN**ing will soon corrupt the 64's memory stack and crash the program. This said, it is quite acceptable to create subroutines that call other subroutines (such structures are known as nested subroutines), but care must be taken to ensure that ultimately

control is systematically **RETURN**ed to the line that follows the original **GOSUB** statement.

As well as prompting the creating of clear and efficient code, **GOSUB**s are obviously an important means of saving memory. In much the same way as **DEF FN** uses memory economically by allowing the single definition of an expression which is repeatedly required by a program, **GOSUB**s enable a process to be coded once, but executed any number of times.

Before moving on to our example program, one final word of warning. Since it is standard practice to place subroutines at the end of a program, it is important to make sure that when the main body of the program has been processed that it is halted before the 64 reaches the subroutines (which must only be called or entered with a **GOSUB** statement).

```
5 REM*GOSUB*
10 PRINT CHR$(147)
20 INPUT "KEY IN A RADIUS";R
30 LET C=2*π*R
40 LET Z=C
50 GOSUB 200
60 PRINT "CIRCUMFERENCE IS";Z
70 LET A=π*R↑2
80 LET Z=A
90 GOSUB 200
100 PRINT "AREA IS";Z
110 END
200 REM*ROUNDING SUBROUTINE*
210 LET Z=INT(100*(Z+.005))
220 LET Z=Z/100
230 RETURN
240 REM*END OF SUBROUTINE*
```

Finally, multiple branching in a program can be coded by coupling **GOSUB** with an **ON** statement. This structure is commonly used to transfer control in menu-driven programs, in which the user's **input** determines which subroutine is called.

```
10 REM*GOSUB*
20 REM
30 REM*CONTROL PROGRAM*
40 REM
50 GOSUB 200
60 GOSUB 300
70 GET N$:IF N$="" GOTO 70
80 LET N=VAL(N$)
90 IF N<1 OR N>3 GOTO 70
100 GOSUB 400
110 ON N GOSUB 500,600,700
120 PRINT S$;S2$;"THE ";M$(N);" IS";V
130 GOSUB 900
140 IF D$<>"N" THEN RUN
150 END
160 :
170 REM*END OF MAIN PROGRAM*
180 :
200 REM*VARIABLE SUB*
210 LET S$=CHR$(147)
220 FOR M=1 TO 3
230 READ M$(M),K(M)
240 NEXT M
250 RETURN
260 :
300 REM*MENU*
310 PRINT S$;SPC(15);"MENU"
320 PRINT CHR$(47);"THIS ROUTINE WILL CALCULATE:"
330 FOR O=1 TO 3
340 PRINT CHR$(47);SPC(10);O;M$(O)
350 NEXT O
360 PRINT CHR$(47);SPC(10);"ENTER YOUR CHOICE (1,2 OR 3)"
370 RETURN
380 :
400 REM*INPUT*
405 PRINT S$
410 PRINT CHR$(47);SPC(10);"NOW ENTER A VALUE"
420 PRINT SPC(10);
430 PRINT "SO WE CAN CALCULATE YOUR ";
440 PRINT LEFT$((M$(N),K(N))
450 INPUT X
460 RETURN
470 :
500 REM*AREA*
510 LET V=PI*X^2
520 RETURN
530 :
600 REM*COS*
610 LET V=COS(X)
620 RETURN
630 :
700 REM*SQR*
710 LET V=SQR(X)
720 RETURN
730 :
800 DATA AREA OF CIRCLE,4
810 DATA COSINE OF AN ANGLE,6
820 DATA SQUARE ROOT,11
```

```
830 :
900 REM*AGAIN*
910 PRINT CHR$(47);"ANOTHER GO (Y/N)?"
920 GET D$:IF D$="" GOTO 920
930 RETURN
940 :
950 REM*END OF SUBROUTINES*
```

In our example program (which makes extensive use of **REM** statements to clarify the structure of the program), the **ON...GOSUB** statement is used to determine whether the N **INPUT** in line 70 is processed by subroutine 1 or 2. If the value of **INPUT** N in line 140 is 1, the program will **GOSUB** 500 and use X to calculate the area of a circle. If N=2 **GOSUB** 600 will be called upon to calculate the **COS**ine of the value entered and if N=3 the **SQR** subroutine at line 700 will be executed. The **INPUT** check in line 90 is necessary to ensure that C is neither greater than 2 nor less than 1. Without this line incorrect **INPUT**s would cause the program to move on to the next line in the program without ever reaching a subroutine.

RELATED KEYWORDS: GOTO, ON, RETURN

GOTO

SYNTAX: GOTO ln
GOTO v
ON v GOTO ln

TOKEN: 137

Statement which passes control to the line number specified by ln which can be literally quoted, a variable or the result of a calculation.

As you probably know, when a program is **RUN** the 64 will execute statements according to the order specified by the program's line numbers, unless this process is disrupted by either a **GOTO, GOSUB, RETURN, ON GOTO** or **ON**

GOSUB statement. The only other facilities in Commodore **BASIC** capable of diverting a program's linear flow are **FOR...NEXT** loops and the **SYS** statement (which passes control to a machine code subroutine).

All of these statements are important as in different ways each expands the programmer's capacity to code the flow of a program to suit the needs of a particular problem. As soon as you start writing your own programs it will immediately become clear why linear processing by line number would simply not be practical.

A **GOTO** statement is one of the simplest means of re-directing control, and it performs in much the same way as a **GOSUB**. The major difference between the two statements is that instead of passing control to a subroutine and **RETURN**ing to the main program, **GOTO** simply jumps to the specified line number and continues processing in the normal way. No return address is stored when a **GOTO** statement is executed and hence there is no way that control can be returned to the code between the statement and line it specifies (except by using another **GOTO**).

In the final analysis, a **GOTO** statement is a powerful facility that should be used with care. As a general rule, if you find you are developing a program that seems to require a large number of **GOTO**s, it is almost certain that the most efficient solution to what ever problem you are tackling has eluded you. In other words, think again!

Let's take a look at an example which illustrates some of the problems presented by the commands:

```
5 REM*SPAGHETTI GOTO*
10 PRINT CHR$(147)
20 INPUT "ENTER A VALUE";N$
30 PRINTCHR$(147);"DO YOU REQUIRE:"
40 PRINT"1.THE SQUARE ROOT OF ";N$;"?"
```

```
50 PRINT"2.";N$;" RAISED TO THE POWER OF
 2?"
60 PRINT:PRINT:PRINT"ENTER 1 OR 2
70 GET C$:IF C$="" GOTO 70
80 IF C$<>"1" AND C$<>"2" GOTO 70
90 IF VAL(C$)=2 GOTO 200
100 PRINT CHR$(147);"THE SQUARE ROOT OF
";N$;" IS";
110 PRINT SQR(VAL(N$))
120 GOTO 250
200 PRINT CHR$(147);N$;" RAISED TO THE P
OWER OF 2 IS";
210 PRINT (VAL(N$)↑2)
250 PRINT:PRINT "ANOTHER GO(Y/N)?"
260 GET E$:IF E$="" GOTO 260
270 IF E$<>"N" GOTO 10
280 END
```

Now let's see how the same program could have been written. The key to appreciating the difference between two listings is to note how the second version exploits the distinctions between the facilities offered by **GOTO** and **GOSUB** statements. Since the code generated for our second example was planned rather than cobbled, it uses the two commands to produce complementary rather than alternative statements:

```
5 REM*GOTO&GOSUB*
10 GOSUB 150
20 INPUT"ENTER A VALUE";N$
30 IF ASC(N$)<48 OR ASC(N$)>57 GOTO 20
40 GOSUB 400
50 GET D$:IF D$="" GOTO 50
60 IF D$<>"1" AND D$<>"2" GOTO 50
70 ON VAL(D$) GOSUB 250,300
80 PRINT" IS ";V
90 GOSUB 500
```

```
100 IF E$="Y" THEN RUN
110 END
120 :
130 REM*END OF CONTROL PROGRAM*
140 :
150 REM*VARIABLES/CLS*
160 LET S$=CHR$(147)
170 LET R$="THE SQUARE ROOT OF "
180 LET Z$="↑2":PRINT S$
190 RETURN
200 :
250 REM*SQUARE ROOT*
260 LET V=SQR(VAL(N$))
270 PRINT S$;R$;N$;
280 RETURN
300 REM*↑2*
310 LET V=VAL(N$)↑2
320 PRINT S$;N$;Z$;
330 RETURN
340 :
400 REM*MENU*
410 PRINT S$;"DO YOU REQUIRE:"
420 PRINT "1.";R$;N$;"?"
430 PRINT "2.";N$;Z$;"?"
440 PRINT
450 PRINT "ENTER 1 OR 2"
460 RETURN
470 :
500 REM*AGAIN?*
510 PRINT
520 PRINT"ANOTHER GO(Y/N)?"
530 GET E$:IF E$="" GOTO 530
560 RETURN
570 :
580 REM*END OF SUBROUTINES*
```

In the final analysis, the safest way of planning the coding of **GOTO** statements is to consider the control structures they facilitate when used in conjunction with **GOSUB**s. Although (as in our example) the planned application of these statements tends to produce slightly longer programs, the resultant code will be considerably more efficient, much easier to follow and consequently simpler to debug.

RELATED KEYWORDS: GOSUB, ON, RETURN

IF...THEN

SYNTAX: IF c THEN ln
IF c GOTO ln
IF c GOSUB ln
IF c THEN statement

TOKEN: 139.....167

Statement which enables the 64 to test conditions and, contingent upon their outcome, execute or ignore specific instructions.

The **IF...THEN** statement is what is known as a decision structure. The condition (c) following **IF** can be any legal **BASIC** expression, and use variables, logical operators, strings or numbers. The second half of the statement can be any legal **BASIC** instruction. The **THEN** can be omitted if the consequence of the condition being satisfied is that the program **GOTO**s and specified line number or **GOSUB**s to a particular subroutine. (See syntax format above.)

If the specified condition(s) are satisfied (TRUE), the instructions which follow are executed. If the specified condition(s) are false (not satisfied), then everything following the condition is ignored, and processing moves on to the next program line. Thus multi-statement lines following an **IF...THEN** statement should be avoided.

```
10 INPUT "ENTER  YOUR SURNAME";A$
20 INPUT "ENTER YOUR SEX (M/F)";S$:PRINT
 CHR$(147)
30 PRINT "YOUR NAME IS ";
40 IF S$="M" THEN PRINT "MR. ";: GOTO 60
50 PRINT "MS ";
60 PRINT A$
70 END
```

In the example program above, line 40 will only **PRINT** "MR " and control pass on to line 60 if S$="M". If S$<>"M" **THEN** control will pass to line 50.

RELATED KEYWORDS: AND, OR, NOT, GOTO, GOSUB

INPUT

SYNTAX: INPUT v,v$,....
INPUT "PROMPT";v,v$,....

TOKEN: 133

A statement which allows the user of a program to enter string and numeric data which is then held in the **INPUT** variable(s).

INPUT statements are the most popular method of enabling the user of a program to enter data for subsequent

processing. When the 64 reaches an **INPUT** statement the program will stop until the user keys in the appropriate form of data (string or numeric) and presses **RETURN**. The type of data it will accept is determined by the statement's format. For example:

```
10 INPUT n
```

will stop the program and **PRINT** a prompt (in the form of a question mark) on the screen. Since n is a numeric variable, the 64 will only accept numeric **INPUT** which, as the operator types it in, will be **PRINT**ed to the screen alongside the prompt. The program will continue processing from the line following the **INPUT** statement as soon as **RETURN** is pressed. If string data is entered a ? REDO FROM START will appear and the user must type in the numeric data that the computer expects before the program can continue. As soon as the numbers have been entered and **RETURN**ed, the 64 assigns their values to whatever **INPUT** variable(s) have been used (in this case n).

Strings can be **INPUT** in much the same way. For example:

```
10 INPUT A$
```

will stop the program until characters are entered and **RETURN** pressed. This format will accept any keyboard data without throwing up an error message, however, any numbers that are entered will be stored as strings and can only be operated upon as such unless **VAL** is used.

Since the ? prompt gives the user of the program no indication of the form of **INPUT** that is acceptable to the 64, **INPUT** statements are normally accompanied by an additional prompt which indicates the kind of response that the program requires. This customised prompt can be coded in one of two ways:

```
10 PRINT TAB(19)"ENTER A NUMBER (1-19)"
20 INPUT N
```

or

```
10 INPUT "ENTER A NUMBER (1-19)";N
```

The choice of construction is determined by the programmer's screen display requirements. The first example allows the prompt to be positioned with a **TAB** statement, while the second simply **PRINT**s at the current cursor position.

INPUT statements can contain more than one variable. For example:

```
10 INPUT a,A$,NAME$
```

is a perfectly acceptable construction. Processing will restart only when the appropriate form of data has been entered for each of the **INPUT** variables. When using **INPUT** statements to obtain multiple entries of both string and numeric data, it is obviously essential to present explicit prompts if incorrect entries are to be avoided. For example:

```
10 PRINT "TYPE IN YOUR SURNAME FIRST"
20 PRINT "THEN YOUR AGE"
30 PRINT "AND FINALLY YOUR PHONE NUMBER"
40 PRINT "PRESS <RETURN> AFTER EACH
ENTRY"
50 INPUT NAME$,AGE,NUM$
60 PRINT NAME$,AGE,NUM$
```

If this program is **RUN** the question mark prompt appears on the screen and the program stops until a string has been keyed-in and **RETURN** pressed. The 64 then displays a double-question mark prompt (??), which indicates that a statement requiring multiple **INPUT**s has been used. It is possible to enter all the data required at once by using a

multiple **INPUT** statement, typing in the entries in the correct order and separating them from each other with commas. Thus the example program above will move on to line 60 if an entry such as the following is keyed in:

```
CHAPLIN,56,636,2101
```

followed by a single **RETURN**. Note that although the final **INPUT** (the phone number) is a number it has been assigned to a string rather than a numeric variable. This is to ensure that however the data is presented by the user (636 2102,636-2101,6362101) it will be accepted as valid by the 64.

INPUT can only be used within a **BASIC** program and will not be accepted as a direct command.

RELATED KEYWORDS: PRINT,GET

INPUT

SYNTAX: INPUT i,v,v

TOKEN: 131

Input/Output statement which enables multiple bytes of data to be read from a specified file (i).

INPUT can be used to assign up to eighty characters to an appropriate variable, and will continue assigning data to a single variable until a carriage return is encountered. Before the **INPUT** command can be used the specified file must be opened by means of an **OPEN** statement.

RELATED KEYWORDS: OPEN, CLOSE, GET, PRINT

INT

SYNTAX: v=INT(n)

TOKEN: 181

Function which returns the integer value of its argument (n).

INT is used when it is necessary to ensure that a value is an integer (a whole number) rather than a floating point value. It is important to remember that the value returned by **INT** will be less than or equal to the number to which the function has been applied. This is because the action of **INT** rounds n down to the next lowest integer. For example:

```
10 I=INT(49.991)
20 PRINT I
```

will **PRINT** out 49.

Keep your wits about you when applying **INT** to negative values. For example:

```
10 Y=INT(-25.9)
20 PRINT Y
```

will **PRINT** out -26 (which is of course a smaller value than –25).

RELATED KEYWORDS: none

LEFT$

SYNTAX: v$=LEFT$(a$,i)

TOKEN: 200

This command facilitates the extraction of the left-most section of a string of characters.

Along with its related commands, **RIGHT$** and **MID$**, **LEFT$** enables a string to be sliced into smaller groups of characters. In the syntax example above, a$ is what is known as the "target string", and **LEFT$** will return a "substring" of a$.

The target string can be quoted as a group of characters or as a variable. Thus:

```
PRINT LEFT$("BATMAN",3)
```

and

```
PRINT LEFT$(a$,3)
```

are both legal statements.

The number of characters to be extracted is specified by i (0-255). When executing a **LEFT$** statement, the 64 counts the characters within the string in question from left to right (with the left-most character in a$ counted as 1), until i characters have been dealt with. If i is greater than the number of characters in the string, then the entire string will be returned. If i is 0, then a null string will be returned and if i is negative an illegal quantity error will stop the program.

```
10 REM*LEFT$:
20 A$="COMMODORE"
30 FOR I=1 TO LEN(A$)
40 PRINT LEFT$(A$,I)
50 NEXT I
```

RELATED KEYWORDS: RIGHT$, MID$, LEN

LEN

SYNTAX: n=LEN(a$)

TOKEN: 195

A function which returns the number of characters held in its argument (a$).

The **LEN** function is used when it is necessary to know how many ASCII characters are contained in a particular string. Usually **LEN** is applied to a string variable, although the function can also calculate the number of characters in a string which is literally quoted (i.e. **PRINT LEN**("string")). The integer value returned by **LEN** will be in the range 0-225.

An important use of the **LEN** function is in the creation of tabular screen displays:

```
1 REM*LEN*
5 PRINT CHR$(147)
10 FOR A=1 TO 5
20 INPUT"ENTER A STRING";A$(A)
30 NEXT A
40 PRINT CHR$(147)
50 FOR B=1 TO 5
60 N=LEN(A$(B))
70 PRINT TAB(25-N)A$(B)
80 NEXT
```

Since **LEN** can only be applied to strings, numeric data can only be operated upon by this function in conjunction with the **STR$** function (which converts numeric data into strings).

If **LEN** is applied to a null string (a$=""), zero is returned.

RELATED KEYWORDS: MID$, LEFT$, RIGHT$, STR$

LET

SYNTAX: LET v=n
LET v$=a$

TOKEN: 136

A statement which is used to assign values to variables.

Since it is so rarely used in its complete form, it is easy to forget that **LET** statements are one of the most common constructions in **BASIC**. When we assign a value to a variable, for example:

```
10 V=2.6
```

what we are actually using is an abbreviation of a **LET** statement which in its complete form would read:

```
10 LET V=2.6
```

Since the word **LET** is optional most programmers tend to leave it out (to save memory). However, if you're new to programming the inclusion of the full statement will often help to clarify your listings.

The actual assignment of variables is fairly straightforward. For example:

```
20 LET W=19
```

simply instructs the 64 to hold the value 19 in a memory location which can be identified by the variable W. The value of W can be changed by a redefinition like:

```
50 LET W=W+10
```

which tells the 64 to locate memory location W and add 10 to

the value it already holds.

Assigning a string variable works on the same principle as a numeric variable, but the variable name must be followed by a dollar sign ($) and the text must be within quotes, thus:

```
60 LET A$="HELLO"
```

RELATED KEYWORDS: none

LIST

SYNTAX: LIST ln-ln

TOKEN: 155

A command which displays the line(s) specified by ln of the **BASIC** program currently held in memory.

LIST is essential during the developmment of a program since it allows you to examine a single program line, a series of lines or the entire program and can be used in conjunction with the 64's flexible editing facilities.

The command can be entered in a variety of formats. For example:

LIST

on its own will display the entire program currently in memory on the screen. If the program fills more than a screenful with code the display will scroll upwards until it reaches the end of the listing. If you want to stop the scrolling at any point you can do so by using the **RUN/STOP** key and the scrolling can be slowed down with the aid of the **CTRL** key.

LIST 120

will display line 120 on the screen.

LIST 230-390

will display all the lines from 230-490 inclusive.

LIST 500-

will list all the lines from 500 to the end of the program. Similarly:

LIST -120

will list everything between 1 and 120.

List 0, however, will list the entire program to the screen.

RELATED KEYWORDS: none

LOAD

SYNTAX: LOAD
LOAD "filename"
LOAD "filename",8

TOKEN: 147

A command which **LOAD**s a file into the 64's memory from an external storage system.

Let's run through the syntaxes of this command one by one to determine which format should be used when. '**LOAD**' and '**LOAD** "filename"' are used when you want to **LOAD** a file from cassette, and the final syntax "**LOAD** "filename",8' is used when you want to **LOAD** a program from disc.

When **LOAD** is used on its own as a direct command the 64 will **LOAD** the first program that it encounters on the cassette. However, if you use the **LOAD** "filename" syntax the 64 will scan the cassette looking for the program in question and when it has been found it will be **LOAD**ed into memory.

There is an alternative way of entering **LOAD** as a direct command: **SHIFT-RUN/STOP**. By pressing these keys simultaneously you will **LOAD** the first program encountered on the tape, but, using this method the program will **RUN** automatically.

RELATED KEYWORDS: SAVE, VERIFY

LOG

SYNTAX: v=LOG(n)

TOKEN: 188

Function which returns base e logarithm of its argument.

LOG (n) returns what is commonly referred to as the natural logarithm of n. The antilog of such a value can be calculated by using **EXP**(LOG(n)). When necessary, natural log operations can be used as with common logs. For example:

```
PRINT EXP(LOG(n)+LOG(v))
```

will print out the product of n and v.

If n is a zero or a negative value the program will be halted with an ?ILLEGAL QUANTITY ERROR MESSAGE.

RELATED KEYWORDS: EXP

MID$

SYNTAX: v$=MID$(a$,i,j)

TOKEN: 202

A string function which extracts a specified number of characters from a specified start point in a predefined string.

Like **LEFT$** and **RIGHT$**, the string function **MID$** is used to perform string slicing processes. In some respects the name of this keyword obscures its flexibility, since **MID$** actually permits the extraction of characters from any part of a string, not just the middle.

The numeric parameters that follow the string variable in brackets must fall in the range 0-255. The first parameter (i) establishes the start point of the string to be extracted, and the second (j) the number of characters to be removed.

In the syntax statement above, if i is greater than the length of the specified string of j is a zero, then a null string will be returned. If the second parameter (j) is left out altogether, the 64 will return the entire sub-string (starting at i). Finally, if the value of j is greater than the number of characters from i to the end of the source string, then **MID$** will return all of the remaining characters in the string.

Since i and j must be integers, it is worth setting their variables as such (by adding the % sign) since integer variables occupy less memory space than standard numeric variables. Let's take a look at some possible formats for **MID$**. Say our source string is:

STRING$="MICROWAVE OVEN"

Then:

SUB$= MID$(STRING$,1,5)

```
PRINT SUB$
```

will **PRINT** out **MICRO**.

And,

```
SUB$2=MID$(STRING$,11,4)
SUB$3=MID$(STRING$,3,1)
PRINT SUB$3+SUB$2
```

will **PRINT** out **COVEN.**

If we leave out the second (j) **MID$** parameter:

```
PRINT MID$(STRING$,11)
```

we'll get OVEN on the screen. For further explanation of this command refer to Chapter 3.

RELATED KEYWORDS: LEFT$, RIGHT$, LEN

NEW

SYNTAX: NEW

TOKEN: 162

A command which resets **TOP** and **LOMEM** and calls **CLEAR**.

As should be obvious from the definition above, **NEW** is probably the most uncompromising of the **BASIC** keywords, since it completely wipes any **BASIC** program from memory. It also clears all variables (other than the system variables).

It is advisable to enter **NEW** as a direct command before starting any new program to ensure that it is unaffected by any program lines left over from a previous listing.

RELATED KEYWORDS: RUN

NEXT

SYNTAX: NEXT v

TOKEN: 130

Statement which determines the end of a **FOR...NEXT**loop.

Commodore **BASIC** allows a single **NEXT** statement to terminate a series of nested loops. The format for such a construction is:

NEXT v1,v2,v3...

where v1,v2, and v3 are the loop variables of each loop to be ended (each variable must be separated from the next by a comma). However, if this format is adopted care must be taken to ensure that the loops are nested correctly so that statements are executed in the intended order.

The loop variable can be omitted from a **NEXT** statement, although its inclusion does clarify a listing (especially in the case of a nested loop structure).

For a more detailed description of **FOR...NEXT** statements see entry under **FOR**.

RELATED KEYWORDS: FOR, TO, NEXT, STEP

NOT

SYNTAX: NOT expression
NOT condition

NOT acts to negate (swap) the truth value of the logical condition upon which it acts, or if acting upon a numeric expression. Complements each bit of the two-byte integer value concerned.

With a conditional expression, **NOT** merely inverts the value of -1 ('true') or 0 ('false') assigned to a condition by the 64. The reason for these numbers being chosen is explained below, but try entering:

```
TRUE=-1: PRINT NOT TRUE
```

which will return 0 ('false'), and similarly

```
FALSE=0:PRINT NOT FALSE
```

will **PRINT** the value -1.

Strictly speaking **NOT** is never necessary in condition testing, since all conditions are evaluated using the relational operators (=, <>, >=, <=, <, >), and each of these has its opposite or complement:

=	**complements**	<>
<	"	>=
>	"	<=

This means that we can always change the logical expression to avoid the use of **NOT**. For example, the expression **NOT** (A=B) is equivalent to A<>B, and **NOT**(A<B) is the same as A>=B. This said, **NOT** can sometimes make the sense of a complex logical expression clearer. Note that **NOT** cannot be used to invert the sense of an expression used on its own, such as:

```
IF X THEN 600
```

This syntax relies on the fact that any non-zero value for X will cause it to be evaluated as 'true'. The expression is a short

form of **IF** X<>0 **THEN**..., but this cannot be inverted with **NOT**, since **NOT**(X) will give another non-zero ('true') value resulting from the bitwise operation of **NOT** (treated below), except where X=-1. In this instance, **NOT FALSE** is always true, but **NOT TRUE** may also be true! Flags, variables used to indicate a state of affairs, can alternate between 0 and -1, enabling **NOT** to be used to swap the values (FLAG=NOT FLAG), and accessible program lines to be used, such as:

```
IF NOT END THEN 100
```

NOT can be used to invert the sense of an expression. The use of brackets determines the sequence of evaluation of an expression, and must be used carefully. The result of:

```
NOT X=Y OR A$=B$
```

is produced by the truth value of **NOT** X=Y being **OR**ed with A$=B$, which is an entirely different matter from:

```
NOT (X=Y OR A$=B$)
```

which inverts the result of the two conditions **OR**ed together.

In logical bitwise operations on the numeric values of two-byte integer values, **NOT** inverts each bit, turning 1's to 0's and vice versa. This process includes the highest order bit (bit 15), which is used as a sign bit. The sign of any number operated on by **NOT** will thus change. The value returned for a number N is always -(X+1). Take **NOT** 156:

	Sign bit	High Byte	Low Byte
NOT 156	**(0)**	**0 0 0 0 0 0 0**	**1 0 0 1 1 1 0 0**
-1	**(1)**	**1 1 1 1 1 1 1**	**1 1 1 1 1 1 1 1**

RELATED KEYWORDS: AND, OR

ON

SYNTAX: ON v GOTO ln,ln...
ON v GOSUB ln,ln...

TOKEN: 145

Statement which, when used in conjunction with **GOSUB** or **GOTO**, allows the flow of control to be determined by the value of v and directed to one of a number of specified line numbers.

This statement often makes an appearance in menu-driven programs in which the user's **INPUT** determines which of a number of processes are executed by the 64. For example:

```
1 REM*ON*
1Ø INPUT "ENTER 1, 2 OR 3";V
2Ø ON V GOSUB 5Ø,6Ø,7Ø
3Ø END
4Ø REM*SUBROUTINES*
5Ø PRINT "ONE":RETURN
6Ø PRINT "TWO":RETURN
7Ø PRINT "THREE":RETURN
```

The use of the **ON** statement in line 20 will call the subroutine in line 50 if 1 is entered, line 60 if 2 is entered and if 3 is keyed in then line 70 is executed. If V is not an integer then the **INPUT** will be rounded to the next lowest integer. If the input is negative an ?ILLEGAL QUANTITY error will be generated, if it is zero control will pass to the next program line and the **GOTO**s/**GOSUB**s will be ignored. The rest of the **ON** statement will also be ignored if the value of V exceeds the number of specified line destinations.

For further examples see entries for **GOTO** and **GOSUB**.

RELATED KEYWORDS: GOTO, GOSUB, RETURN

OPEN

SYNTAX: OPEN file, peripheral, addr.,"filename, type, mode"

TOKEN: 159

OPENs the channel to an I/0 device.

This command is used to **OPEN** a channel to enable the user access to the specified peripheral. The first parameter is, in effect, a dummy parameter, it is used to label the file in question and link the I/0 commands together and must be in the range of 1 to 255. Thus if you **OPEN** 1,x,x,"x,x,x" you must use the same opening parameter with a different I/0 command (**CLOSE** 1,x,x"x,x,x").

The 64 allocates each peripheral its own number and this number is used in the second parameter of the command. These numbers are; 1 for a cassette player, usually 4 for a printer, but when two printers are being used the second printer is usually 5, and the disc drive is 8. However, you can change the disc drives 'peripheral' number to 9 when a second disc drive is in use.

The third parameter, addr., can be 0, 1, or 2 when it is used in conjunction with a cassette recorder (if it is omitted the 64 will assume that it is 0). When this parameter is set to 0 it makes the file **OPEN** enabling you to recall information from tape. When this parameter is set to 1 it enables you to send information to the cassette recorder and when it is set to 2 it leaves a 'mark' on the cassette which says 'end of tape'.

When you are using this parameter in conjunction with disc files you can choose the addr. number but each disc file should have its own addr. parameter. However, you can send, or recall information without using this parameter. The only number which does have a particular use is 15, this sends commands that read the error light on the disc drive.

RELATED KEYWORDS: CLOSE, CMD, GET, INPUT, PRINT

OR

SYNTAX: condition OR condition
expression OR expression

OR is a logical operator which performs logical disjunction on conditions or numeric expressions. The first operation uses the numeric values of -1 and 0, used by the 64 to represent 'true' and 'false' respectively. Note that any non-zero value is recognised as 'true' if used in a conditional expression. With numeric expressions, a logical bitwise operation is performed on the values of the numeric expressions when converted to two-byte signed integers.

For conditional expressions linked by **OR**, the combined expression is evaluated as true or false as in the truth table below:

Condition 1	Condition 2	C1 OR C2
True	True	True
True	False	True
False	True	True
False	False	False

Thus the result of two expressions combined using **OR** is true if either or both conditions are true, and false only if both conditions are false. **OR** is useful in checking whether values are within acceptable ranges. For instance, take the case of an **INPUT** for the month:

```
200 INPUT "MONTH (1-12)";M%
210 IF M%<1 OR M%>12 THEN PRINT "WHAT?!!
TRY AGAIN": GOTO 200
```

The following program demonstrates both the use of **OR** in **INPUT** checks and decision-making in programs:

```
5 REM *OR DEMO*
10 PRINT"HOW MANY DIN PLUGS ARE INSERTED
INTO THE 64 YOU'RE USING":INPUT L%
20 IF L%<1 OR L%>3 THEN PRINT"COME OFF I
T! TRY AGAIN":GOTO 10
30 INPUT"DO YOU USE A TV AS VDU(ENTER Y
OR N)";A$
40 IF A$="Y" OR A$="N" THEN 60
50 PRINT A$;" IS NOT A VALID ANSWER.":PR
INT"ENTER Y FOR YES,N FOR NO":GOTO30
60 IF(A$="Y" AND L%>1) OR (A$="N" AND L%
>2)THEN GOTO 80
70 PRINT"START SAVING FOR A DISC DRIVE!"
:END
80 PRINT"SO YOU'VE GOT A DISC DRIVE OR P
RINTER   ON YOUR SYSTEM AS WELL!"
```

The bitwise operation of **OR** takes the two-byte integer form of the value (-32768 to 32767) of each of the numeric expressions, and performs an **OR** operation on the respective bits of each number. For each bit position a 0 is taken as false and a 1 as true. The result is that if both bits are 0, then the result will have a 0 in that bit position, and any other combination (both 1's or a 1 and a 0) will result in a 1 being placed in that position in the result. The resultant value is of course again a two-byte signed integer value. If we take the operation 49 **OR** 156, for example, the result (189) is arrived at by comparing each of the 16 bits and **OR**ing each pair, thus:

	Sign bit	High Byte	Low Byte
49	(0)	0 0 0 0 0 0 0	0 0 1 1 0 0 0 1
OR			
156	(0)	0 0 0 0 0 0 0	1 0 0 1 1 1 0 0
189	(0)	0 0 0 0 0 0 0	1 0 1 1 1 1 0 1

Such bitwise **OR** operations are usually used in bit-masking, to set or unset particular bits in a byte.

RELATED KEYWORDS: AND, NOT.

PEEK

SYNTAX: PEEK (addr)

TOKEN: 194

Statement which examines the contents of a specified memory location (addr) and returns the value contained in that location.

The value returned by **PEEK**ing one of the 64's memory locations will fall in the range 0-255, while the address specified by the statement must be in the range 0-65535 if an ?ILLEGAL QUANTITY error is to be avoided.

For a detailed explanation of the value of both **PEEK** and **POKE** see the chapters on **SOUND, GRAPHICS,** and **MEMORY.**

RELATED KEYWORDS: POKE, USR, SYS

POKE

SYNTAX: POKE addr,i

TOKEN: 151

Statement which enables a single byte binary value (i) to be placed into the memory location specified by the statement's first parameter (addr).

For the majority of programming demands, the **BASIC** language is a perfectly adequate means of solving problems with your 64. However, there are circumstances in which a

greater level of control is required than **BASIC** can provide, but does not justify the development of customised assembly language routine. This is where **POKE** statements come into their own.

POKE is particularly valuable for 64 programmers, since although the machine offers dynamic sound and graphics facilities, they are unsupported by appropriate **BASIC** commands. The **POKE** command allows us to access any of the machine functions of the 64 and thus invalidates the short-sighted attacks on the Commodore's admirably condensed version of **BASIC**.

The first parameter (addr) which specifies the location must be in the range 0-65535, and the value to be placed into the location (i) must be an integer in the range 0-255. If either parameter falls outside this range the program will be halted with an ?ILLEGAL QUANTITY error.

For a detailed explanation of the uses of both **PEEK** and **POKE** statements, see the **SOUND** and **GRAPHICS** chapters for examples of their practical implementation and the section on **MEMORY** for a theoretical elaboration of their operation.

RELATED KEYWORDS: PEEK, USR, SYS

POS

SYNTAX: POS(0)

TOKEN: 185

The **POS** function returns the current cursor position and will generate an integer value in the range 0-79 (being the logical screen width of an 80 character screen). Thus when using **POS** to control formats on the 64's 40 character screen, it must be remembered that any value returned by the function

which falls in the range 40-79 will refer to cursor positions on the second screen line.

It should be noted that the value in brackets following **POS** is what is known as a dummy argument which must be included but will be ignored when the statement is executed.

RELATED KEYWORDS: PRINT

PRINT

SYNTAX: PRINT plist
PRINT TAB(x)plist

TOKEN: 153

Statement to display **PRINT** items to the screen.

The items to be displayed can be literally quoted strings, which must be centered within double quotes:

PRINT "A STRING"

string variables, literally quoted numeric data, which do not require quotes:

PRINT 2*4

and numeric variables. **PRINT** used by itself without any **PRINT** items displays a blank line on the screen.

The **PRINT** statement is described at some length in Chapter 2, so we shall merely outline the possible formats here:

i) **PRINT** items to be displayed literally to the screen must be enclosed within quotes, (eg **PRINT** "HHGH $12")
ii) Any arithmetic expression to be calculated must not be included within quotes (eg **PRINT** 2.4+3.5)

iii) Semi-colons after **PRINT** items leave the **PRINT** position set directly after the last character **PRINT**ed, so that the next item to be **PRINT**ed will follow on. For example:

```
1Ø PRINT "A STRING AND ";
2Ø PRINT "ANOTHER STRING"
```

will display A STRING AND ANOTHER STRING.

iv) The 64's screen is divided into four **PRINT** fields. Including a comma between **PRINT** items causes the item to start **PRINT**ing at the first position of the next **PRINT** field:

```
1Ø PRINT 1,2,3,4
```

v) **PRINT** can also be combined with **TAB** statements. For example, **PRINT TAB**(x) creates a **PRINT** position x characters to the right of the last.

vi) **PRINT** can also be used with the keyword **SPC**. For example, **PRINT SPC**(y) **PRINT**s y spaces to the screen.

RELATED KEYWORDS: TAB, SPC

READ

SYNTAX: READ v$,v$,v$...
READ v,v,v,...
READ v$,v,v$,v...

TOKEN: 135

This statement is used to load the next **DATA** item into the variable(s) specified by **READ**.

The variable names which follow a **READ** statement can be both string and numeric. However, the **DATA** which is loaded into the **READ** variable(s) must be appropriate to the variable in question (ie string **DATA** assigned to a string variable, numeric **DATA** to a numeric variable), or a ?SYNTAX ERROR will be generated.

When the **DATA** pointer in the 64's memory has reached the last **DATA** item, the computer will generate an ?OUT OF DATA error if any further attempt is made to **READ DATA** without prior use of the **RESTORE** statement. **RESTORE** will return the data pointer to the beginning of the first **DATA** statement.

For further details about the use of this statement, see the entry for **DATA**.

RELATED KEYWORDS: DATA, RESTORE

RND

SYNTAX: i=RND(i)

TOKEN: 187

Numeric function that generates floating point value between 0 and 1.0.

RND is one of the 64's most valuable numeric functions. For example, for obvious reasons, the creation of random (or at least apparently random) values is essential when coding games programs. (Think how tedious a game would be if the Invader/Ghost/Spaceship always emerged from exactly the same position on the screen!)

The values produced by **RND** are calculated by the 64's random number generator. As we will demonstrate, these values are actually only "pseudo-random", but for the most

part this qualification does little to detract from the value of the function.

The selection of the value returned by **RND** is initially determined by what is known as the function's "seed value", which is the number used as the basis of random number generator's calculations. The seed value is established by the function's argument. Before we look at the various situations and formats in which **RND** can be usefully employed, we'll endeavour to clarify why the function isn't "genuinely" random. Enter the following example, making sure that you switch your 64 off and then on again before entering the code.

```
5 REM*RND1*
10 FOR A=1 TO 5
20 PRINT RND(1)
30 NEXT A
```

The first time this program is **RUN** always produces the same sequence of numbers:

.185564016
.0468986348
.827743801
.554749226
.897233831

Now on the face of it **RND** appears to have literally done its job and thrown up a wholly random collection of numbers. It's only when you discover that you'll always get exactly the same sequence of values when **RUN**ning this program from power-up that it becomes clear that **RND** is obliquely predictable. The fact of the matter is that whenever the 64 is switched on the random number generator automatically selects a seed value and goes on to create a list of numbers betwen zero (which sometimes appears) and 1 (which is never quite reached). Now so long as **RND**'s argument is

positive, the selector will simply follow the order of the list, moving on to the next "random" value each time the function is used. However, if the argument is negative the selector will create a new list of random values each time **RND** is called upon. Let's see this in action!

```
10 LET RD=RND(-1)
20 FOR A=1 TO 5
30 PRINT RND(1)
40 NEXT
```

each time you switch on afresh and **RUN** this program the same sequence of numbers appears, although they're not the same as those generated by our first example. Hopefully, the reasons for this result are now clear. Since the first **RND** statement, (line 10), has a negative argument, the random number generator creates a different list of values than the positive value list. However, since the second **RND** statement controlled by the loop (line 30) is positive, the same sequence of values is generated each time the program is **RUN** from power-up.

By altering the negative value in line 10 you can force the generator to create a different list for each different negative argument, but the positive **RND** statement will ensure that the same sequence is produced each time a particular value is used. Test this out by using a variety of negative arguments in line 10. You might also want to check the constancy of the effects of a positive argument by trying different positive values in line 30. The details of the random number generator's calculations are of no interest to us in this chapter. However, it should be noted that when **RND**(0) is used the 64 uses its **TI** clock to create its random value. Indeed Commodore programmers often use this facility to randomise the seed selection process. The following example demonstrates such a construction.

```
5 REM*RND3*
10 LET RD=TI
20 LET V=RND(-RD)
30 FOR A=1 TO 5
40 PRINT RND(1)
50 NEXT A
```

Since the value of **TI** will always be different, the sequence generated by this program is very close to being "genuinely" random. Alternatively, we could exploit the properties of the **RND**(0) statement. Thus we could modify the last example by changing line 10 to:

```
10 LET V=RND(0)
```

and deleting line 20.

So the **RND**'s "pseudo" randomness can be improved upon fairly easily. Now let's look at methods of controlling the range of values that can be randomly generated by using **RND** in conjunction with other numeric functions.

As you've probably gathered, in the majority of cases it's unlikely that you'll want to use **RND** in its "raw state". After all, how often are you actually going to require a random floating point value that falls between 0 and 1? Let's say we want to produce a sequence of a dozen random integers in the range 0-255. This is simple to code:

```
5 REM*RND3*
10 FOR A=1 TO 12
20 PRINT INT(RND(0)*256)
30 NEXT A
```

On the other hand, maybe we don't want the values produced to fall into a range that starts at 0. The following example churns out values in the range 175-255.

```
5 REM*RND4*
1Ø FOR A=1 TO 12
2Ø PRINT (RND(Ø)*156)+1ØØ
3Ø NEXT A
```

An alternative method of establishing an upper and lower limit for the values generated is demonstrated in this final example. Let's assume that our upper limit must be 123 and the lower limit 22.

```
5 REM*RND5*
1Ø FOR A=1 TO 12
2Ø PRINT (RND(Ø)*(123-22)+22)
3Ø NEXT A
```

By now it is hopefully clear that whilst logically and theoretically **RND** is anything but random, its controlled application can produce results that are genuinely unpredictable.

RELATED KEYWORDS: none

REM

SYNTAX: REM statement

TOKEN: 143

A statement which allows **REM**arks to be inserted into a program without a syntax error being generated or the program being in any way affected.

These statements are of immense value to all programmers. Whilst their inclusion has no effect on a listing when it is actually **RUN**, the function of the statement is to provide an internal commentary which explains the action of a particular line or section of code. **REM** statements are normally

included at the beginning of a line, for example:

```
15Ø REM*SUBROUTINE1*
```

they can also appear at the end of a multi-statement line:

```
18Ø LET A=34:REM*AREA OF CIRCLE*
```

The important point to remember is that the computer will ignore all instructions following a **REM** statement for the duration of the line in which it is included.

There is a tendency to only use **REM** statements when a program becomes especially complex or inpenetrable, on the grounds that short code will be self-explanatory. However, it can be argued that **REM**s should always be used in programs, however brief or straightforward, since painful experience has convinced the author of this book than even a schematic commentary of **REM**s can avoid many wasted hours of searching through unmarked listings.

RELATED KEYWORDS: none

RESTORE

SYNTAX: RESTORE

TOKEN: 140

Statement which allows the contents of **DATA** statements to be **READ** more than once.

When processing **BASIC** programs, the 64 uses an internal pointer to keep track of the last **DATA** item **READ** into the **DATA** variable(s). Under normal circumstances, once a **DATA** item has been **READ** it cannot be **READ** again. However, the **RESTORE** statement resets the internal

pointer to the beginning of the **DATA** statements enabling the code to be used once again. For more details about this statement see the entries for **DATA** and **READ**.

RELATED KEYWORDS: DATA, READ

RETURN

SYNTAX: RETURN

TOKEN: 142

Statement used to mark the end of a subroutine and **RETURN** control to the main program.

When a subroutine is called from the main body of a program, control is passed to the line number specified by the **GOSUB** statement. When the instructions contained in the subroutine have been processed a **RETURN** statement is required to return control to the line that follows the original **GOSUB** statement. A **RETURN** without a **GOSUB** statement will halt the program with an error report. Nested subroutines require a **RETURN** statement for each **GOSUB** statement.

For further details about subroutines see the entries for **GOSUB** and **ON**.

RELATED KEYWORDS: GOSUB, ON

RIGHT$

SYNTAX: v$=RIGHT$(a$,i)

TOKEN: 201

A string function which enables a sub-string of a specific

length to be extracted from the right-most end of a predefined string.

Like **MID$** and **LEFT$**, **RIGHT$** facilitates the creation of a sub-string from a predefined string. It takes the form:

SUB$=RIGHT$(a$,i)

in which **SUB$** is the variable name to which the new string will be assigned, a$ is the source string from which i characters will be extracted from the right-most end of a$. i can be any value from 0-255, and a$ either a predefined variable or a literally quoted string within the brackets, (**RIGHT$**("RICHARD",4)) would return **HARD**. If i is 0 a null string will be returned, if i is greater than the length of the source string (a$) then the entire string will be returned.

The following example should clarify the operation of this function:

```
5 REM*RIGHT$*
10 LET a$="RIGHTMOST"
20 LET SUB$=RIGHT$(a$,4)
30 PRINT SUB$
```

will **PRINT** out MOST to the screen.

For more details about string slicing see the entries for **LEFT$** and **MID$**.

RELATED KEYWORDS: MID$, LEFT$, STR$

RUN

SYNTAX: RUN
RUN ln

TOKEN: 138

A command which clears all variables and commences execution of **BASIC** program currently in memory.

RUN is usually used as an unadorned direct command to start the execution of the **BASIC** program you have just **LOAD**ed or keyed in. However, the command can also be followed by a line number. When used in this way, all the variables are still cleared, but processing commences from the line number specified. If you don't want variables cleared, then processing can be re-started using **CONT** (which will re-commence execution from the line that follows the statement at which it was halted) or **GOTO** In which will enable processing to be started at any point in the listing.

If you enter **RUN** In and the line number in question does not exist, the 64 will generate an UNDEF'D STATEMENT error. The command can also appear within a program, as in the example below (which, incidentally is not intended to work but merely demonstrates this format of **RUN**):

```
500 PRINT "FINAL SCORE =";fs
510 PRINT CHR$(147)"ANOTHER GAME (Y/N)";
D$
520 IF D$="Y" THEN RUN
530 END
```

RELATED KEYWORDS: CONT, END, STOP, NEW

SAVE

SYNTAX: SAVE
SAVE "filename"
SAVE "filename",8

TOKEN: 148

A command that stores a file onto tape or disc.

This command is used to **SAVE** a program or file onto some form of external storage which can be either a cassette or disc. The final syntax above (**SAVE** "filename",8) tells the 64 to store the program onto disc and the other two examples are required when **SAVE** is used to store the program to tape.

The first syntax (**SAVE**) should never be used, since you should always give the program that you wish to **SAVE** a filename, and it is always wise to label the cassettes clearly as it is infuriating not being able to find a program.

RELATED KEYWORDS: LOAD, VERIFY

SGN

SYNTAX: i=SGN(n)

TOKEN: 180

A numeric function which returns a value which indicates the sign or signum of its argument.

The **SGN** function returns 1,0 or -1. It is used to indicate whether its argument is positive or negative. If n is positive then v=1, if n is negative then v=-1 and if n=0 then v=0. The following example demonstrates its use in a rather contrived routine:

```
5 REM*SGN*
10 PRINT CHR$(147) "ENTER A NUMBER"
20 INPUT n
30 PRINT "YOUR NUMBER WAS ";n
40 ON SGN(n)+2 GOSUB 100,200,300
50 INPUT "ANOTHER GO (Y/N)?";D$
60 IF D$="Y" THEN GOTO 10
70 END
```

```
100 PRINT "A NEGATIVE VALUE": RETURN
200 PRINT "ZERO": RETURN
300 PRINT "A POSITIVE VALUE": RETURN
400 REM*END OF PROGRAM*
```

By adding 2 to the value returned by the **SGN** statement in line 40, control is directed to the subroutine which prints the appropriate status of the value of the **INPUT** (ie −1+2=1=GOSUB 100;0+2=2=GOSUB 200;1+2=3=GOSUB 300).

RELATED KEYWORDS: ABS

SIN

SYNTAX: v=SIN(n)

TOKEN: 191

A numeric function which returns the sine of its argument (n).

Like all the 64's trigonometric functions **SIN** returns a value measured in radians. To convert radians into degrees use the following formula in which r equals the number of radians:

degrees=r*180/PI

The following program returns the **SIN** of angles from 0-360 in steps of five degrees:

```
1 REM*SIN*
5 PRINT CHR$(147)
10 FOR A=0 TO 360 STEP 5
20 RAD=A*π/180
30 PRINT A,SIN(RAD)
40 NEXT A
```

Notice the use of the degrees to radians conversion formula in line 20.

RELATED KEYWORDS: ATN, COS, TAN

SPC

SYNTAX: SPC(i)

TOKEN: 166

A statement which is used in conjunction with **PRINT** to **PRINT** to the screen the number of spaces specified by its argument (i).

When this command is executed the 64 locates the first available line where it **PRINT**s the number of **SPaCe**s specified by the integer in brackets following **SPC**. Many 64 programmers use **SPC** as an alternative to **TAB** when coding screen formatting statements, and the example program below demonstrates the command in this role:

```
10 REM*SPC*
20 PRINT CHR$(147)
30 FOR A=1 TO 4
40 PRINT CHR$(147)
50 PRINT SPC(8);"ENTER NAME";A
60 INPUT NAME$(A)
70 PRINTSPC(8);"ENTER AGE";A
80 INPUT AGE$(A)
90 NEXT A
100 PRINT CHR$(147)
110 FOR B=1 TO 4
120 LET S=LEN(NAME$(B))
130 PRINT SPC(4);
140 PRINT NAME$(B);SPC(15-S);AGE$(B)
150 NEXT B
```

The command can also be used when using a printer. You can force a carriage return by adding **SPC** to the last character position in a line, although no space will actually be printed.

RELATED KEYWORDS: PRINT, TAB

SQR

SYNTAX: v=SQR(n)

TOKEN: 186

Function which returns the square root of its argument.

Yet another of the 64's valuable numeric functions, **SQR** returns the square root of any positive number. However, if the value of its argument (n) happens to be negative processing will be halted with an ?ILLEGAL QUANTITY error.

The simple example below shows the function in action:

```
1 REM*SQR:
10 PRINT CHR$(147)
20 FOR I=45 TO 60
30 PRINT "SQR ROOT OF "I" IS ";SQR(I)
40 NEXT I
```

RELATED KEYWORDS: ABS

STEP

(See **FOR**)

STOP

SYNTAX: STOP

TOKEN: 144

Statement which halts the execution of a program and returns the computer to the direct mode.

If you include a **STOP** statement in a program, when the 64 processes that line the program will halt with a ?BREAK IN LINE ln report, in which ln is the line containing the **STOP** statement. Processing can be re-started by using **GOTO** ln (where ln is the line number from which you wish to re-start the program) or **CONT** (which will **CONT**inue running the program from the line that follows the **STOP** statement), as direct commands. The use of these commands will preserve the values of variables currently in memory.

The program can also be re-started with **RUN** or **RUN** ln, but using this command will **CL**ea**R** all variables currently in memory. A **STOP** statement has the same effect as an **END** statement, except that the latter halts the program without a ?BREAK IN LINE ln report.

STOP is primarily of value when de-bugging a program (when information about when and where processing has halted is critical), but can also be included in a program in which certain conditions must be fulfilled before processing can continue (although **END** is normally used in this role). For example:

```
500 PRINT "GAME OVER"
510 INPUT "ANOTHER GO (Y/N)";D$
520 IF D$="N" THEN STOP
530 GOTO 10
```

RELATED KEYWORDS: END, RUN, CONT, GOTO

STR$

SYNTAX: a$=STR$(v)

TOKEN: 196

A string function which returns a string representation of its numeric argument (v).

STR$ is the function which enables string functions to be applied to numeric values as it converts them into a string format. For example, when the 64 **PRINT**s a positive value to the screen it includes a leading and trailing space in the display. This can result in untidy displays when the following type of statement is used:

```
40 LET V=61
50 PRINT "VALUE="V"UNITS"
```

However, by using the **STR$** function the display can be tidied up by using one of the string slicing functions (**MID$** in this case):

```
20 LET V=61:A$=STR$(V)
30 LET B$=MID$(A$,2,2)
40 PRINT "VALUE"B$"UNITS"
```

The inclusion of leading and trailing spaces should be borne in mind when the **STR$** function is used in the comparison of strings. Thus if "6510" and **STR$**(6510) are compared, the 64 will include leading and trailing spaces in the result of the latter, and thus find the two strings unequal.

STR$ performs the opposite function to **VAL**, which converts strings of numbers into their numeric value.

RELATED KEYWORDS: VAL

SYS

SYNTAX: SYS addr

TOKEN: 158

A statement which passes control to a machine code subroutine which starts at the memory location specified by addr.

In the same way as control is passed to a subroutine starting at a line number specified by a **GOSUB** statement, a **SYS** calls a machine code subroutine whose first instruction is held in the memory location specified by addr (see syntax example above).

SYS statements are the simplest means of enabling programmers to write code in both **BASIC** and machine code, and they can be used both as direct commands or executed from within a program.

In the same way as a **BASIC** subroutine must end with a **RETURN** statement, a machine code routine called by **SYS** must close with a **RTS** instruction (**R**e**T**urn from Subroutine), which directs control to the **BASIC** program line following the **SYS** statement.

The memory location specified by the **SYS** statement (addr) can be any location in either **ROM** or **RAM**.

RELATED KEYWORDS: none

TAB

SYNTAX: PRINT TAB(i) print statement

TOKEN: 163

An optional element in a **PRINT** statement, which moves the cursor i spaces from the left-most position of the current **PRINT** line.

TAB is a way of formatting screen displays and can only be used in a **PRINT** statement (it cannot, for example, be used in conjunction with **INPUT**). The value of the **TAB** argument (i) must be in the range 0-255, or an ?ILLEGAL QUANTITY error will stop the program. Refer to the **PRINT** statement for further clarification of this command.

Many 64 programmers use **SPC** as an alternative to **TAB**.

RELATED KEYWORDS: PRINT, SPC

TAN

SYNTAX: v=TAN(n)

TOKEN: 192

A trigonometric function that returns the tangent of its argument (n).

Like all of the 64's trigonometric functions, **TAN** calculates angles in radians (and not degrees). Thus both its argument and the value returned when the **TAN** function is used are expressed in radians. To convert degrees to radians, and vice versa, the following formulae must be used:

Degrees = **r/π*180**
Radians = **d*π/180**

RELATED KEYWORDS: ATN, COS, SIN

THEN

(See **IF**)

TIME

SYNTAX: TI

A numeric function which enables the 64's system clock to be read.

The 64 has an internal "clock" which counts in 60ths of a second. The clock is set at zero when you first switch on (or reset) your micro and continues to operate until power is cut off (or tape I/O generated).

TI is the numeric variable that can be used to "read" the clock. However, since the timer counts intervals of 1/60th of a second, TI/60 must be used to return a value in seconds.

RELATED KEYWORDS: TIME$

TIME$

SYNTAX: TI$

A string function which reads and updates the 64's system clock which records the time since power-up or re-set (in hours, minutes and seconds).

TIME$**, which is abbreviated to **TI$, works in much the same way as an ordinary clock. It can be set in the same way as any value is assigned to a string variable, except that in this case the string must be comprised of six numeric characters (two for the hour, two for the minutes, two for the seconds). If **TI$** is not set, the 64 sets the timer to start at 00hrs:00mins:00secs and will update like an ordinary clock

until the 64 is switched off, re-set, or utilises a storage device (tape or cassette).

The **TI$** function can be used as a direct command, or within a program.

RELATED KEYWORDS: TIME

TO

(See **FOR**)

USR

SYNTAX: USR(v)

TOKEN: 183

Function which calls a machine code subroutine whose start address is established by a **POKE** statement which must precede the line containing the **USR** call.

The **USR** function performs in much the same way as **SYS**, except that it enables the value of its argument (v) to be placed in the 64's floating point accumulator, operated on by the specified machine code subroutine, and the result returned to the **BASIC** program. The start address of the machine code subroutine must be **POKE**d into memory locations 785-786 or else an ?ILLEGAL QUANTITY error will halt the program.

RELATED KEYWORDS: SYS

VAL

SYNTAX: v=VAL(a$)

TOKEN: 197

String function which returns the numeric value of the string of numbers contained in its argument.

VAL performs the opposite function to **STR$** (which converts a numeric value into its string representation). The first character of the string to be converted must be either a space, a minus sign a plus sign, or a number, otherwise the 64 will return a zero. If any non-numeric characters are included in the middle of the string, it will be ignored in the evaluation. Thus:

```
10 n$="65xy10"
20 v=VAL(n$)
30 PRINT v
```

will **PRINT** out 6510.

```
1 REM*VAL*
5 PRINT CHR$(147)
10 INPUT "ANY NUMBER ";V$
20 LET B$="5"
30 PRINT V$+B$: PRINT
40 V=VAL(V$):B=VAL(B$)
50 PRINT V+B
60 END
```

In the example program above, line 30 performs a simple string concatenation (and thus prints the input string next to the 5 contained in B$). However, after **VAL** has determined the numeric value of these strings (line 40), these values can be added together and **PRINT**ed out (line 50).

RELATED KEYWORDS: STR$

VERIFY

SYNTAX: VERIFY "filename",(i)

TOKEN: 149

An Input/Output command which makes it possible to ensure that a program has been correctly transferred to tape or disc.

There are few more frustrating surprises than the discovery that a program you have **SAVE**d has been incorrectly transferred to a tape or disc. The **VERIFY** command offers programmers a means of ensuring that the program that has been transferred to an external storage device is exactly the same as the one held in the 64's memory.

When the command is followed by a filename (which must be contained within quotes and be exactly the same filename as the one used in the **SAVE** statement), the 64 will search the tape or disc until it locates the specified file which it will then try to match with the program in memory. If the filename is left out, the 64 will take the first program it encounters on the tape/disc, and compare it with the one in memory. If the program has been incorrectly transferred (or it's the wrong program!), the 64 will generate a ?VERIFY ERROR message.

The second part of the **VERIFY** command (i) lets the 64 know what kind of storage device a program has been **SAVE**d on. If this parameter is omitted, the 64 will assume that Commodore's cassette recorder has been used. If a program has been **SAVE**d to disc, the second parameter must be 8 if **VERIFY** is to operate correctly.

RELATED KEYWORDS: SAVE, LOAD

WAIT

SYNTAX: WAIT,addr,X,(Y)

TOKEN: 146

Statement that halts program execution until the memory address specified (addr) recognizes a specific bit pattern determined by logical relational operations utilising X or X and the optional Y parameter.

The general consensus amongst 64 programmers is that with a couple of exceptions **WAIT** is more trouble than it's worth. Its effects are unique to Commodore **BASIC**, and the majority of programmers familiar with other **BASIC**s consider this to be one of the command's few redeeming features. Although such judgments are probably a trifle harsh, the fact of the matter is that there are few occasions in which a **WAIT** statement is of much practical use.

Perhaps the most important point to clarify is that (under normal circumstances) it doesn't pause a program for a specified period. (Some dialects of **BASIC** have a **WAIT** command that performs this function.) Before going on to chronicle (as opposed to elucidate) the process initiated by **WAIT**, here is an example of how the command can be usefully employed as a means of pausing a program until any key is pressed:

```
10 REM*WAIT*
20 PRINT CHR$(147);"HELLO"
30 PRINT"PRESS ANY KEY TO CONTINUE"
40 WAIT 197,64:WAIT 197,191
50 PRINT CHR$(147);"THIS IS WAIT IN ACTI
ON"
```

Used in this manner, **WAIT** statements actually have a very real value. Although we could obviously have used **GET** to perform much the same task, code using **WAIT** is more economic, and the 64's response much faster.

So how does it work? Well, for what it's worth when the 64 encounters a **WAIT** statement it exclusively **OR**s the contents of the memory location specified by the first parameter (addr) with the (optional) variable Y. The result of this operation is then **AND**ed with the statement's second parameter (X). If the result of this process is 0 the entire operation is repeated. The program will only continue when a non-zero result is obtained. Thus processing is halted until some external event causes a memory location to become equal to a specific value. All quite straightforward!

With the exception of the masochists amongst you, it should be clear that **WAIT** is usually only useful if you have a predilection for particularly esoteric code. However, there is one further demonstrable use of the command that goes some way towards justifying its existence. By using it in conjunction with the 64's clock it can be customised to simulate a pause command.

RELATED KEYWORDS: none

CHAPTER SIX
THE ART OF MEMORY

One of the virtues of **BASIC** is that it is a high-level language, which means that once we have stated what we wish the computer to do in correct **BASIC** grammar there is no need to consider how the instructions are carried out by the machine. However, the 64 has many capabilities which can only be utilised by dealing directly with certain parts of its hardware. In the absence of **BASIC** instructions to do these things for us, we need to delve around in the workings of the 64, using **POKE** and **PEEK** to interact with the computer memory. Some of the deficiencies of the 64's **BASIC** can be overcome with the use of Simon's **BASIC** or other software packages, but these will all illustrate the compromise inherent in designing computer languages, to whit, the trade off between memory available for programs and the facuilities of the built-in language, which competes for the limited amount of available memory.

MEMORIES ARE MADE OF THIS

Like all computers, the 64 contains nothing but the representations of numbers stored in it, so we better start at the bottom. Digital electronic circuitry deals in simple electrical terms which admit only two states, on (current flowing) and off (no current). So we're dealing with a system built up of switches. To use numbers with this sort of system, we start by defining the "on" state as 1 and the "off" state as 0. Not the greatest advance in terms of capacity for storing

numbers, but it's a start, and this start allows us to use the Binary System of notation for numbers, in which everything is written down in terms of 1's and 0's. Each memory location can be thought of as a bank of eight switches (known as bits), each of which may be on, representing one, or off, representing zero. Such a block of eight bits is known as a byte. Imagine we wish to represent the number using this type of switch bank. We would simply set the first (rightmost) switch to on and leave the rest off; similarly, for two we could simply turn the second switch on and all the others off. Now we come to the clever bit (excuse the pun) with the question of what we are going to do about three: if we simply turned on the third switch and turned off all the others then, following that train of thought, we would run out of switches after representing only eight numbers. If, however, we decide that since the first switch was used for representing one, we could turn it on again along with the second switch and add up the two switches, one and two, to make three, we postpone our use of the next switch till we need to represent 4. At this stage it becomes obvious that we can cope with all the numbers up to seven with just the combinations of ons and offs allowed by our first three switches, since we can make up totals for any lesser number out of the combinations of 1, 2 and 4. We brightly deduce that the next required "switch-valve" is eight and slowly the truth creeps on leaden feet into our burgeoning semi-consciousness, it is always twice the last "switch-valve". For eight switches we thus have a series of values 128,64,32,16,8,4,2,1, which can be used to provide representations for all the whole numbers up to their sum total of 255. If we write the ons and offs as 1's and 0's then we are writing in binary, and after a while you'll get the hang of seeing things like 00000111 as 7. (Honest, you will.)

This number system is called binary because it deals with just two numerals, 0 and 1, and as shown above, deals in powers of two. The eight bits in a byte are numbered 0 to 7 right to left, and each bit, moving leftward, represents a higher power of two:

Bit number:	7	6	5	4	3	2	1	0
Value when set:	128	64	32	16	8	4	2	1
Power of two:	2↑7	2↑6	2↑5	2↑4	2↑3	2↑2	2↑1	2↑0

The pattern of 0's and 1's in a byte is known, not unreasonably, as the bit pattern, and if we take this pattern:

10110010

we can evaluate it by thinking of it as:

128+0+32+16+0+0+2+0

giving us 178, in our normal human decimal system. If, however, you look at the binary number as:

(1*2↑7)+(0*2↑6)+(1*2↑5)+(1*2↑4)+(0*2↑3)+(0*2↑2)+(1*2↑1)+(0*2↑0)

you will more clearly see the similarities of binary (base 2) and decimal (base 10, or denary) number systems, since 178 decimal is:

(1*10↑2)+(7*10↑1)+(8*10↑0)

This complex state of affairs should be clarified by the following program, which grinds through, in intimate detail, the workings of the binary/decimal relationship.

```
10 REM*** BINARY/DECIMAL ***
20 REM*** CONVERSION     ***
30 PRINT"■ BINARY/DECIMAL TUTOR":PRINT
:PRINT
```

```
40 PRINT"ENTER CHOICE:"
50 PRINT"PRESS B FOR BINARY TO DECIMAL"
60 PRINT"PRESS D FOR DECIMAL TO BINARY"
70 GET A$:IF A$<>"B" AND A$<>"D" THEN 70
80 IF A$="D" THEN GOSUB 500
90 IF A$="B" THEN GOSUB 1000
100 PRINT"■■■TRY ANOTHER NUMBER? (Y/N)"
110 GET A$:IF A$="" THEN 110
120 IF A$="Y" THEN PRINT"■■":GOTO40
130 END
140 REM
490 REM*******************
491 REM*DECIMAL TO BINARY*
492 REM*   SUBROUTINE    *
493 REM*******************
500 PRINT"■■ENTER A DECIMAL NUMBER (0 TO
 32767)"
510 INPUT N%:PRINT:PRINT"  "N%"DECIMAL
520 BY$=""
530 I%=INT(N%/2)
540 BIT=N%-2*I%
550 IF BIT=0 THEN BY$="0"+BY$:GOTO 570
560 BY$="1"+BY$
570 N%=I%
580 IF I%<>0 THEN N%=I%:GOTO 530
590 PRINT:PRINT"IS "BY$" IN BINARY"
600 RETURN
610 REM
990 REM*******************
991 REM*BINARY TO DECIMAL*
992 REM*   SUBROUTINE    *
993 REM*******************
1000 PRINT"■■ENTER A BINARY SEQUENCE"
1010 INPUT BY$:L=LEN(BY$)
1020 N=0:P=0
1030 FOR B=L TO 1 STEP-1
1040 BIT$=MID$(BY$,L-P,1)
1050 N=N+VAL(BIT$)*2↑P
1060 P=P+1
1070 NEXT B
```

```
1080 PRINT:PRINT"[illegible]"BY$
1090 PRINT:PRINT"IS"N"DECIMAL"
1100 RETURN
```

Let's now move on to another number system, that of hexadecimal notation. We've just seen that we can use binary to represent any number that a computer uses but this does not mean getting used to long strings of 0's and 1's. Hexadecimal (base sixteen) numbers are an attempt to compromise between the computer's way of looking at things and our way of writing numbers. Of course this does mean that ideally we should all know our sixteen times table . . . well, we'll just have to get along without it then. Here's an example to start with:

DECIMAL:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18..

HEXADECIMAL:
1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12..

Comparing hexadecimal (let's call that hex for short) with decimal we can see from the list above that they're just the same up to 9. After 9 we carry on counting right up to fifteen by borrowing the first six letters of the alphabet to use as digits. Thus A is worth ten, B is eleven, and so on up to F, which is fifteen. When we get to sixteen we put a zero in the first column and place a one in the sixteens column (just as decimal numbers have tens and units the hex numbers have sixteens and units). A little example may help at this point: suppose we have the hex number A5 and we wish to know what its equivalent is in decimal; we have A (or ten) sixteens plus 5 units so we can see that the answer is 160+5 or 165. Whenever we need to write hexadecimal numbers we prefix them with a dollar ($) sign: this stops us from confusing $11 (seventeen) with 11 (eleven) and so on.

Now we compare hex with binary in the table below:

HEX	BINARY	HEX	BINARY	HEX	BINARY
00	00000000	10	00010000	20	00100000
01	00000001	11	00010001	21	00100001
02	00000010	12	00010010	22	00100010
03	00000011	13	00010011	23	00100011
04	00000100	14	00010100	24	00100100
05	00000101	15	00010101	25	00100101
06	00000110	16	00010110	26	00100110
07	00000111	17	00010111	27	00100111
08	00001000	18	00011000	28	00101000
09	00001001	19	00011001	29	00101001
0A	00001010	1A	00011010	2A	00101010
0B	00001011	1B	00011011	2B	00101011
0C	00001100	1C	00011100	2C	00101100
0D	00001101	1D	00011101	2D	00101101
0E	00001110	1E	00011110	2E	00101110
0F	00001111	1F	00011111	2F	00101111

What you should be able to see from this table is that the final four bits of the binary number are the same whenever the final digit of the hexadecimal number is the same. This four to one correspondence of binary to hex means that we can use hexadecimal as a sort of shorthand for binary numbers (when we have bytten off more than we can chew). A half-byte, four bits or one hex digit is known as a nybble (ho,ho).

Right now, where were we . . . oh yes, eight bits make a byte. 1024 bytes make a kilobyte eh? 1024? It's that binary stuff again. The nearest multiple of 2 to 1000 is 2*2*2*2*2*2*2*2*2*2 or 2 ↑ 10 or 1024, so we call 1024 a kilobyte, or Kbyte, (Kilo meaning 1000, as with kilograms). This Binary awkwardness is allowable because it simplifies the discussion of memory amounts. The 64K of your Commodore 64 is in fact 65,536 bytes!

DOWN MEMORY LANE

So, we now know that inside the 64's memory there is a mass of bytes, which can store numbers in binary format. The 6510 chip can specify any particular byte by means of an address. You can think of the memory as a long road, with 65536 single-roomed houses along it, each identified by its number (but remember the first house is numbered as 0) arranged in groups of 256 (or $FF) – the values in a single byte.

Each address that the 6510 chip can access is defined by the chip as a bit pattern on the 16 address lines it possesses. This corresponds to the bit pattern of two bytes, one sequence following on after the other to give 16 ones or zeros. If you followed the discussion of binary, you'll see that this means the largest value that can be held in two bytes considered as a 16 bit sequence is 2 ↑ 16–1 or 65535 ($FFFF). There are 65536 values available, just as there are 256 values available in a single byte, but one of them is zero, so the maximum positive number that can be represented is one less. For our purposes, we should note that this limitation on addressing capacity means we're restricted to the 64K of memory which the chip can access, hence Commodore 64.

We can specify any address using two bytes of memory. These are referred to as high byte and low byte, and are put together to give us a sixteen-bit value:

Address Bit:	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	<			**High Byte**				>	<		**Low Byte**					>
Bits:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

We get the decimal two-byte value by using the expression (High byte*256)+Low byte. Dealing with hexadecimal is easier, since the notation is a direct representation of binary, so that if the high byte value is $1F, and the low byte $D4, for example, the two byte value is simply $1FD4. Unfortunately,

the 64 requires us to work in decimal! To turn a decimal number N into the correct two values (HI and LO) to be **POKE**d into two bytes, we use:

HI = INT(N/256)
LO = N – (HI*256)

It's important to note that addresses (and other two-byte values used by the system) are always stored with the Low byte first, followed by the High byte. (The only exception to this is in the case of integer variable values, where the order is high/low). Given this, to store a two byte value derived as above at a memory location B:

```
POKE B,LO

POKE B+1,HI
```

Similarly , the address of value stored in memory locations B and B+1 is given by:

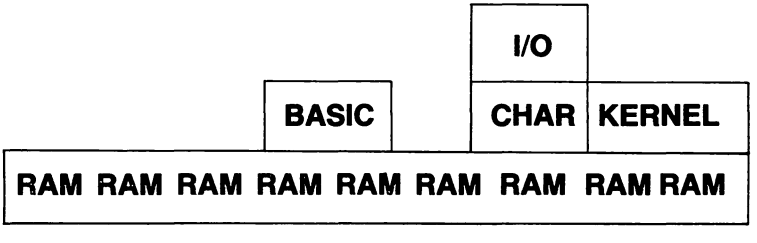
```
PEEK(B)+256*PEEK(B+1)
```

An address can be seen as specifying a particular location (0–255) with the low byte, and which area (0–255 again) of memory the location occupies with the high byte. This would be like saying, House 6 of Group 231, in terms of our street analogy. Each such 256-byte area specified by the high byte is known as a Page of memory.

Memory is of two types, Read Only Memory (ROM) and Random Access Memory (RAM). We can **PEEK** both types of memory and read the contents stored at any particular address location, but we can only insert values into a location if it is within **RAM**.

The 64 contains 64K of **RAM**, but it also contains 20K of **ROM**. How does it do this? Well, it does this by over-laying some areas of the **RAM** with **ROM**, so that they share the

same addresses. It's as if certain groups of houses along Memory Lane had a second storey built on top. The 6510 chip, although it can only 'see' 64K of memory at any one time, can alter its viewpoint so that it sees either the ground floor **RAM** or the first floor **ROM** in certain sections of memory. Altering the 6510's viewpoint in this way is done by setting control bits in special locations. Let's have a closer look at the standard memory organisation first. There is one area of memory that has three storey houses, of which the top storey is the Input/Output area, dealing with the interfacing to the outside world via the **CIA** chips, the **SID** sound chip, the **VIC** video chip, and screen colour data, which is the portion of memory that the 6510 chip normally 'sees'. Underneath this is the character data, which holds in **ROM** the patterns for all the 64's screen display characters, and the bottom level is the **RAM**. The character **ROM** is peculiar, in that it is not seen by the 6510, nor is it seen by the **VIC** chip (which uses the data to put characters on the screen!). The **VIC** chip actually 'sees' a different memory arrangement than does the 6510, limited to a 16K block of memory, and inside this 'window' onto the 64's memory the **VIC** chip sees an image of the character **ROM** block. In order to be able to alter the character set, we can switch out ('bank out') the **I/O** area of **ROM**, which enables us to **PEEK** the character **ROM** and copy it into a suitable area of **RAM**.The other 'second storey' portions of memory are the 8K of **ROM** holding the interpreter for the **BASIC** language, and the Kernel, also 8K of **ROM**, which has the routines which do the hard work. Looking down Memory Lane, we'd see something like this:



The 64K of memory is arranged as shown in the following memory map when the 64 is switched on:

Decimal		Hex
0		0000
	System Variables RAM	
1023		03FF
	Screen Memory RAM 40*25 locations	
2039		07E7
	Sprite Pointers	
2047		07FF
	BASIC Program Area RAM	
	User Program	
	Variable Storage	
	Array Storage	
	Free Memory	
	String Storage	
40959		9FFF
	BASIC Interpreter ROM	
49151		BFFF
	User Memory RAM	
53247		CFFF

	Input/Output Area for VIC and SID chips	
55295		**D7FF**
	Screen Colour Data RAM	
56295		**DBE7**
	Input/Output Area for CIA chips and future I/O	
57343		**DFFF**
	Kernel ROM	
65535		**FFFF**

Starting at the bottom of memory (location 0 ($0000)), the first two locations (0 and 1) are the ones which allow us to control which storey of the two or three levels of memory is seen at any time by the main processor chip (see below). Locations 2 to 1023 ($0002 to $03FF) are reserved for use by the operating system (the **BASIC ROM** and the Kernel) to store system variables which are required to keep track of the state of the operating system.

Above this is the screen memory, with 1000 bytes (one per character position) for the standard screen display, locations 1024 to 2023 ($0400-$07E7), each byte holding the character code to be displayed in the respective screen position. The 8 bytes of locations 2040-2047 ($07F8-$07FF) are the sprite pointers which locate sprite definitions in memory. If screen memory is moved, the sprite pointers move with it.

Next we have the **BASIC** program area. The divisions indicated within this area vary according to the requirements of the current program. Variable storage space is created as needed for numeric variables, and string pointers giving the address at which strings are stored, and this 'floats' on top of the **BASIC** program listing. Array storage created by **DIM**

statements similarly rides on top of the variable storage. This all grows up towards the top of the **BASIC** program area, and the contents of string variables are stored starting from the top of this area, so that the unused portion of memory is in the middle. An 8K **ROM** cartridge, when attached, replaces **RAM** in the area 32768-40959 ($8000-$9FFF).

The **BASIC ROM** comes next, followed by a 4K block of free **RAM** which is available to the programmer. This is separate from the **RAM** used by **BASIC**, and is not used for any purposes by the system, so forming a protected area for storage. 16K **ROM** cartridges overlay the top portion of the **BASIC** program area as above, plus the **BASIC ROM** area.

The **I/O** areas act like **RAM**, in that locations can be **PEEK**ed and **POKE**d, but the locations are dedicated to the purpose of communication, and consist of the registers of the various chips (other than the 6510) which deal with disc **I/O**, input from the keyboard, paddles, joysticks, or other external devices, and data output to screen, printer, or other external devices such as a modem. Our control over many of the 64's functions depends on the values we **POKE** into these control registers. In between the two **I/O** areas is an area of **RAM** which stores the colour data for the screen, with 1000 nybbles (four-bit values), one for each screen location. This colour **RAM** is fixed. An expanded memory map for this area shows the areas dedicated to each chip:

Decimal		Hex
53248	6566/69 VIC II chip	D000
53296		D02F
	VIC images	
54272		D400
	6581 SID chip	

54301		D41D
	SID images	
55296		D800
	Colour RAM	
56296		DBE8
	unused	
56320		DC00
	6526 CIA chip 1	
56336		DC10
	CIA 1 images	
56576		DD00
	6526 CIA chip 2	
56592		DD10
	CIA 2 images	
56832		DE00
	Reserved for future I/O	
57344		E000
	Kernel ROM	

The images are copies of the chip registers, which reflect their current values. The final section of memory is the Kernel **ROM**.

SWAPPING PARTS IN THE MEMORY THEATRE

The 64 allows us to reconfigure the memory map under software control. Addresses 0 to 1 of memory are the 6510 data-direction register and **I/O** port which control both the memory bank switching and cassette input/output. We can

switch memory banks with bits 0-3 of location 1, but must maintain the bit settings for the cassette control. The normal (default) settings for the bits are:

BIT	HOLDS	NAME	FUNCTION
0	1	LORAM	Switches 40960-49152 ($A000-$BFFF) BASIC ROM/RAM
1	1	HIRAM	Switches 57344-65535 ($E000-$FFFF) Kernel/RAM
2	1	CHAREN	Switches 53248-57343 ($D000-$DFFF) I/O /CHAR ROM
3	1	CASSWRT	Cassette write
4	0	CASSSNS	Cassette switch sense
5	1	CASSMTR	Cassette motor switch
6	0	Not used	
7	0	Not used	

To switch single bits in a byte whilst maintaining the values of other bits we use the binary operators **AND** and **OR**, a process known as bit masking. This requires you to know something of binary notation, hence our treatment above. The **AND** and **OR** operators function in accordance with the truth tables:

X	Y	X OR Y	X AND Y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

This applies to the bits in a byte. If we have say, X=17, which is binary 00010001, and Y= 240, binary 11110000, then:

X	**00010001**	**X**	**00010001**
Y	**11110000**	**Y**	**11110000**
XANDY	**00010000**	**XORY**	**11110001**

Thus, **AND**ing a memory location's contents with a value 0-255, which defines a bit pattern for the byte will leave unchanged any bit where both are 1, and set to 0 any bit where our chosen bit pattern has 0.

ORing, however, will set to 1 any bit which is 1 in our chosen bit pattern, and leave unchanged any bit where the pattern has 0.

To set a specific bit or bits in a location to 1, we **OR** with a value of 2 ↑ bit number. Thus:

```
POKE 400, (PEEK(400)OR 2↑1)
```

will set bit 1 in location 400 to 1 and leave the others as is. We can use the decimal value, of course, and can set more than one bit at a time:

```
POKE 1023, (PEEK(1023)OR(2↑1+2↑5))

POKE 1023, (PEEK(1023)OR 34)
```

are equivalent statements to set bits 1 and 5 of location 1023 to 1.

ANDing is used to find out whether a particular bit is set:

```
PRINT (PEEK(340)AND 2↑3)
```

will display 8 (2 ↑ 3 remember!) if bit 3 is already set to 1, and 0 if it is 0. We also use **AND** to set a bit to 0:

```
POKE 300, (PEEK(300)AND 255-2↑3)
```

will set bit 3 of location 300 to 0, since 255 (11111111 in binary) minus 2 ↑ 3 (or 8 decimal), binary 00000100) gives us 247, binary 11110111. If for example, location 300 held 107 decimal, binary 01101011, we have:

107	**01101011**
247	**11110111**
107 AND 247	**01100011**

We have maintained everything as was except bit 3, which is now 0.

"Toggling" a specific bit, without knowing what its current value is, is somewhat complex. If L is the location and B the bit number to be toggled, then:

```
(PEEK(L) AND 2↑B)/2↑B
```

gives us a 1 or 0 value for bit B. Then if this value is V, the expression:

```
POKE L,PEEK(L) AND (255-2 B) OR (1-V
)*2↑B
```

toggles bit B.

Having made sure you understand this process, let's look at the memory controls. **LORAM** switches between the 8K of **BASIC ROM** and the underlying **RAM**. If **BASIC** is switched out, the 64 cannot be used for **BASIC** programs, so this operation is used to replace **BASIC** with another operating system or when running machine code programs.

To switch out the **BASIC ROM** use

```
POKE 1, PEEK(1) AND 254
```

To switch in the **BASIC ROM** use

```
POKE 1, PEEK(1) OR 1
```

HIRAM banks in or out the Kernel **ROM**. Since **BASIC** uses the routines in the Kernel, banking out the Kernel means everything will stop dead unless some other system is operating.

To switch out the Kernel **ROM** use

```
POKE 1, PEEK (1) AND 253
```

To switch in the Kernel **ROM** use

```
POKE 1, PEEK (1) OR 2
```

CHAREN banks in and out the **I/O** section of memory, making available the 4K of character definition **ROM** to the 6510's view. This is done when it is required to copy some or all of the pre-defined character set into **RAM** in order to modify it. In order to do this it is necessary to perform another operation, that of switching off the keyboard scan, which is done by means of an interrupt generated by the 6510 chip every sixtieth of a second. (An interrupt, as the name suggests, stops all other processing whilst some task is carried out). Since the 6510 uses the 6526 chips, accessed through the **I/O** system, for this, we have to switch off the interrupt, or the system will hang up on us. This is done by means of location 56334. **POKE 56334, PEEK (56334) AND 254** will turn off the interrupt request, and **POKE 56334, PEEK (56334) OR 1** will turn it back on. (See the graphics section for the use of this in defining your own characters.)

To switch out the **I/O** area use

```
POKE 1,PEEK(1) AND 251
```

To switch in the **I/O** area use

```
POKE 1,PEEK(1) OR 4
```

THE BASIC MEMORY

Let's now take a look at the way the 64 organises **BASIC** programs in memory. In so doing, we'll deal with some of the system variables that hold current data required by the 64 in the course of its operations.

When you enter a program line into memory by pressing **RETURN**, the 64's operating system scans the line, and tokenises any **BASIC** keywords in the line, compressing each into a single-byte value from 128 to 202 inclusive, plus the value 255, used for π. The tokens are identified by having bit 7 set to 1, which distinguishes them from the screen characters (codes 32 to 95). If you look at the Keyword Tokens Appendix or enter the program given later, you will see that each **BASIC** keyword has a token value, as do the arithmetic operators (+,–,/,* and ↑) and the relational operators (<, > and =) plus the pi sign, π. These operators form part of the instructions which the **BASIC** interpreter has to execute, and this is indicated by their tokenised form in the program. When **LIST**ed the 64 "detokenises" the program to display it in comprehensible form. The 64 knows that anything not tokenised is a variable name, punctuation, or a string, if there are quotes around it. This is the reason reserved words cannot be included in variable names. Reserved words are the **BASIC** keywords plus the reserved variables **STATUS (ST)**, **TIME (TI)** and **TIME$(TI$)**, which do not have tokens, but are scanned for by the operating system in the same way.

This memory-efficient system compresses the **BASIC** program, and makes more efficient the syntax checks that are performed on each program line as it is **RUN**. As each **BASIC** program is entered, the interpreter runs through the program until it encounters a program line with the same

number, whereupon it replaces the old line with one just entered, or a higher line number, when it enters the new line in the correct position. The program area is then expanded, and the system resets the pointer values that it uses to keep track of the whereabouts of significant divisions of memory. The main pointers or system variables concerned with the **BASIC** program area are shown in the diagram below.

POINTER	BASIC MEMORY
55/56 Top of memory	
	STRING STORAGE AREA
51/52 Bottom of string space	
	FREE MEMORY
49/50 First byte of free memory (End of arrays plus one)	
	ARRAY STORAGE AREA
47/48 Start of array area (end of variables plus one)	
	VARIABLES STORAGE
45/46 Start of variables table (end program plus one)	
	BASIC PROGRAM SPACE
43/44 Start of BASIC (bottom of memory plus one)	

There are other system variables potentially useful to the **BASIC** programmer. We will deal with a few more here, and you will find a list of selected locations and their functions in The Selected Memory Locations Appendix. The start of the **BASIC** program area in memory is normally location 2049, and this address is held (as a two-byte value) in locations 43 and 44 decimal. Thus **PEEK(43)+256*PEEK(44)** will give us the value 2049, but since these are standard **RAM** memory locations, we will also **POKE** values into them, which enables us to shift the start of **BASIC**, in this instance, as we wish. We'll see reasons for doing this later, but we first need to look at the structure of a **BASIC** program. The next program examines itself and displays the contents of the sequence of memory locations which start at 2049:

```
10 B=2048:K=0
20 M=K+B:C=PEEK(M)
30 PRINTM,C;:IFC>32ANDC<96THENPRINT"   "C
HR$(C);
40 PRINT" ":K=K+1:IFK<21THEN20
50 GETA$:IFA$=""THEN50
60 K=0:B=B+21:PRINT"--------":GOTO20
```

The program is a simple loop which displays the address, its contents, and the corresponding alphanumeric character if it is within the specified (printable) range. The first couple of screenfuls of display will be this (without the comments!):

```
2048    0       Actual start of BASIC area (always 0)
2049    16      Program start. Low byte of link address
2050    8       High byte, link address (16+8*256=2064)
2051    10      Low byte of program line number
2052    0       High byte of program line number
2053    66  B   Variable name
```

2054	178		Token for =
2055	50	2	Number value
2056	48	0	Number value
2057	52	4	Number value
2058	56	8	Number value
2059	58	:	Colon statement separator
2060	75	K	Variable
2061	178		Token for =
2062	48	0	Numeric zero
2063	0		Zero byte value-end of program line
2064	33	!	Numeric 33, low byte of link address (2064=16+8*256)
2065	8		High byte of link address
2066	20		Low byte of program line number
2067	0		High byte of program line number
2068	77	M	Variable
2069	178		Token for =
2070	75	K	Variable
2071	170		Token for +
2072	66	B	Variable
2073	58	:	Colon
2074	67	C	Variable
2075	178		Token for =
2076	194		Token for PEEK
2077	40	(	Open bracket
2078	77	M	Variable
2079	41	)	Close bracket
2080	0		End of line
2081	65	A	Numeric value — low byte of link address

Location 2048 holds zero, as a marker value for the start of **BASIC**. Location 2049, the actual start of the program, is the address normally held in the pointer of locations 43 and 44. This holds the low byte, and 2050 the high byte of the link address, which points to the first byte of the next program line. The next two bytes store the line number, again in low/high format. In this case, **10+256*0** gives 10. **2053** holds B, stored as **ASCII** code, as a variable name. We then have our first token, for the = operator. If stored in a string, = would be held as its **ASCII** code, but since it's an operator here, it gets

tokenised. The next four bytes hold the value **2048**, stored as **ASCII** codes which are not evaluated until **RUN** time comes along. Note that numeric 0 is held as **ASCII 48**, and a zero value for the byte is either known to be part of an address, or serves as a marker for the end of a line – as with location **2063** in this case. Following the end-of-line zero is the address pointed to by the link address of the previous line, and **2064/5** point to the start address of the next line. This system of link addresses allows the 64 (and us) to step quickly through each line of the program.

If you step through the screens displayed by the program, you will find that the program ends at location **2185** (or somewhat later, if you've inserted any spaces into your program that don't appear in our listing). This is indicated by the link address pointed to by the first two bytes of the last line holding zeroes. When the 64 encounters this it drops back into command mode and gives you the **READY** prompt.

You will see that the first byte after the zeroes (which should be 2188) holds 66, the code for B. **STOP** the program, and enter:

```
PRINT PEEK(45)+256*PEEK(46)
```

This will give you the address of the byte containing the B. This is the first entry in the variable table produced when the program was **RUN**. As each variable is assigned in the program, an entry is placed in this table, and the array storage area moved up seven bytes in memory to accommodate this, with the pointers changed to indicate the new position. Integer and floating point variables are held directly in this area, as are the pointers to string storage locations and function definitions **(DEF FN)**. These all occupy seven bytes in this table, with the following formats:

INTEGER VARIABLES

Byte 1 First character of variable name, stored as ASCII code +128
2 Second character of variable name, as ASCII code +128(128=null)
3 Sign bit and high seven bits of integer value
4 Low byte of integer value
5 Not used, holds 0
6 Not used, holds 0
7 Not used, holds 0

FLOATING-POINT VARIABLES

Byte 1 First character of variable name, stored as ASCII code
2 Second character of variable name, as ASCII code (0=null)
3 Exponent value plus 128,0 for zero,else 255=+127,1=–127.
4 Sign bit and high bits (bits 24-32) of mantissa value
5 Bits 16-23 of mantissa
6 Bits 8-15 of mantissa
7 Bits 0-7 of mantissa

Exponent byte holds the value of the exponent plus 128, allowing a range of exponent values from +127 to –127, with exponent zero indicating zero. (All bytes should be zero, but rounding may cause errors). High bit of byte 1 is assumed always to be 1, and is used as a sign bit 0=positive,1=negative). The mantissa is treated as a 32-bit binary fraction (.111 . . . indicating 1/2+1/4+1/8 . . . etc).

STRING VARIABLES

Byte 1 First character of variable name, held as ASCII code
2 Second character of variable name, held as ASCII code+128 (0=null)
3 Number of characters in string
4 Low byte of storage address
5 High byte of storage address
6 Not used, holds 0
7 Not used, holds 0

Address points to location at which string starts. If a string

constant, this will be to the address within the **BASIC** program where the string is defined, otherwise to an address in the string storage area.

FUNCTION DEFINITIONS

Byte 1 First character of Function name stored as ASCII code plus 128
2 Second character of Function name stored as ASCII code
3 Low byte of address of function definition (in BASIC program)
4 High byte of definition address
5 Low byte of address of function argument variable
6 High byte of address of function argument variable
7 Not used, holds 0

Simply by first declaring a variable, finding the start of the variable storage area using locations 45 and 46 and then displaying the memory contents as in the program above you can investigate the way any particular value is stored. Remember that strings operate differently, and you will have to extract the pointer for a string address, and then display the memory contents. The program below defines **A$** and **B$** as the first and second variable definitions, so these will be the first two entries in the variable table, starting at location Z. The contents of the first fourteen locations of the variable table are printed out, and you can verify the description of the arrangement given below. The locations at which the strings are stored is then calculated from the values of the fourth and fifth bytes, and these are displayed. You will see that the location of **A$** is a few bytes after the start of **BASIC**, and that **B$** is stored at the top of memory as the first entry in the string storage area. Just to prove the point, **B$** is printed out from the sequence of memory locations, using the number of characters held in the third byte of the definition of **B$**.

```
5 REM*STRINGVAR*
10 A$="HELLO"
20 B$="OT"+A$
30 Z=PEEK(45)+256*PEEK(46)
40 FOR F=0 TO 13:PRINT Z+F,PEEK(Z+F)
```

```
50 NEXT
60 PRINT:PRINT"A$ STORED AT "PEEK(Z+3)+2
56*PEEK(Z+4)
70 B=PEEK(Z+10)+256*PEEK(Z+11):PRINT"B$
STORED AT "B
80 FOR F=0 TO PEEK(Z+9):PRINTCHR$(PEEK(B
+F));:NEXT
```

Array storage starts at the address given by **PEEK(47)+256*PEEK(48)**. Again the sequence of arrays in memory is in the order in which they were **DIM**ensioned in the program. The type of array is identified by the encoding of the variable name, just as for the different variable types. Each array has a header sequence with the following format:

Byte
1 First character of array name (ASCII,+128 if integer array)
2 Second character of name (ASCII,+128 if integer or string array)
3 Low byte of number of bytes in array (pointer to next array)
4 High byte of number of bytes in array
5 Number of array dimensions
6 Low byte of number of elements of last dimension specified
7 High byte, number of elements in last dimension
8 Low byte, number of elements in next to last dimension
9 High byte, number of elements in next to last dimension
....
....
– Low byte, elements in first specified dimension
– High byte, elements in first specified dimension

This array header is followed by the contents of the array. Integer elements occupy two bytes, floating point elements five, and string elements three (number of characters, high and low byte of storage address). Single dimensioned arrays have the elements in linear sequence, two-dimensional arrays are listed by running through the initial dimension values for each value of the second dimension. For example, an array A(2,3) is stored:

A(0,0) :A(0,1) :A(0,2) :A(0,3) :A(1,0) :A(1,1) :A(1,2) :A(1,3) :A(2,0) :A(2,1) . . . . . .

For arrays of higher numbers of dimensions, the same principle is followed. For an array **A(B,C,D)** the elements are listed by following first the **B** sequence, then the **C**, then the **D**. **A(1,2,2)** will have elements in the sequence:

A(0,0,0) :A(1,0,0) :A(0,1,0) :A(1,1,0) :A(0,2,0) :A(1,2,0) :A(0,0,1) :A(1,0,1) . . . . .

and so on.

For obvious reasons, keeping track of multi-dimensional arrays in order to perform manipulations directly in memory is rather more of a problem than a solution. However, single dimensional string lists are more efficiently sorted by swapping the values of the pointers to the string variables, rather than swapping the strings themselves. You can investigate the storage of arrays by means of a simple memory display program like that above, using the value stored in locations 47 and 48 to locate the start of array storage.

POINTING THE WAY

We can illustrate some of the potential of memory manipulation utilising system variables with the following fairly simple programs. The first produces a list of all the **BASIC** tokens. It uses a series of memory locations from 631 to 642 **($0277–$0280)** which form the keyboard buffer, holding characters entered at the keyboard but not yet processed. This occurs when a program is **RUN**ning, as the 64 can't do anything with them until a **GET** instruction is encountered or the program ends (when the characters will get displayed on the screen). The only place we can see tokens in intelligible form is in a program listing, as there is no way to simply **PRINT** them out. The program below fools the 64 into thinking a sequence of program lines have been entered, and then **POKE**s the tokens into each line (the number of which corresponds to the token code value) in turn:

```
5 REM *TOKENS DISPLAY*
10 I=128
20 POKE 640,I: REM KEYBD BUFFER LOC'N US
ED AS STORE
30 PRINT CHR$(147):PRINT:PRINT:PRINT I;"
REM":PRINT"GOTO 80"
40 PRINT CHR$(19):REM HOME BUT NOT CLEAR
50 POKE 198,2:REM NUMBER OF CHARS IN KBD
 BUFFER
60 POKE 631,13:POKE 632,13 :REM RETURN*2
 POKED INTO KEYBD BUFFER
70 END
75 REM LINE 80-TOKEN POKED TO PROGRAM LI
NE,REPLACING REM
80 I=PEEK(640):P=PEEK(45)+256*PEEK(46)-4
:POKE P,I
90 I=I+1:IF I=203 THEN I=255:REM 203-254
 UNUSED
100 IF I<256 THEN 20
110 END
```

The value of I, initially 128, the code for the first token, is **POKE**d into the keyboard buffer. Nothing happens to it here, but the keyboard buffer is a useful place to store values temporarily. Notice it is placed in the last location of the keyboard buffer, and as **GET** is a single character input anything placed in the buffer is fairly safe (unless the user abuses his keyboard) as long as it is not in the first couple of locations. (**INPUT** does not use the buffer – it goes directly to the screen so that editing can take place.)

The program next **PRINT**s the character string **147** (Clear/Home), drops down two lines, and **PRINT**s the value of I followed by **REM** on the fourth line, and then, on the next line, **GOTO 80**. **CHR$(19)** homes the cursor, without clearing the screen. Location 198 is the keyboard buffer pointer, and is incremented every time a character enters the buffer, pointing to the next free location. The value of two **POKE**d

in leads the 64 to believe that two characters are in the buffer, and line 60 provides them by **POKE**ing the code for **RETURN** into the first two buffer addresses. The program then **END**s – but the 64, after giving the **READY** prompt, has the cursor on the fourth line, which has a program line entered, and a **RETURN** character in the buffer, so it executes this, entering the line into memory, and on the next line finds a **GOTO 80** instruction, which is activated by the second **RETURN** character in the buffer.

Going to line 80, the current value of **I** is read from the buffer store, **P** is set to a value which is four less than the start address of the variable table, which points to the last byte of the last program line – which is the one just automatically entered. This location holds the token for **REM**, but this is replaced by the token with the code **I**. The process then repeats, line 90 merely missing out the unused codes. On listing the program, all the codes will be entered in program lines, the numbers of which are the corresponding token codes.

We can also define the top and bottom of **BASIC** memory using the 64's pointers. The start of **BASIC** is held in locations 43 and 44, and the top of memory (where strings are stored, growing downwards) in locations 55 and 56. Lowering the top of memory gives us protected storage space for data or machine code programs, and raising the bottom of memory can perform the same function. However, raising the bottom of memory is more useful as a method of having two programs in memory at the same time, or even joining two programs together.

The following program deletes blocks of program lines. It can be **LOAD**ed above the program from which block deletion is to occur, by shifting the bottom of **BASIC** above the first program, in which case it will only function to delete a single block, since line 210 resets the bottom of memory. You can, however, omit line 210, and choose several blocks for deletion, entering **POKE 44,8;CLR** yourself, and then

entering line numbers as per the messages . Make sure you take a note of the line numbers you wish to delete before you start, however! If you omit line 210 and either **LOAD** the program before starting to program, or give it high line numbers and join it onto a program using the method given below, you can use this facility as often as you wish, and even use it to delete itself from the finished program!

```
10 REM**BLOCKDELETE**
20 REM**LOAD ABOVE PROG FOR DELETION
30 REM**AFTER X=PEEK(46)+1:POKE X*256,0:
40 REM** POKE 44,X:NEW  TO SHIFT PAGE
50 REM**OR LOAD BEFORE STARTING TO
60 REM**PROGRAM,OMITTING LINE 210
70 B=2049
80 INPUT"DELETE BLOCK STARTING LINE";S
90 INPUT"LAST LINE TO BE DELETED";E
100 LN=PEEK(B+2)+256*PEEK(B+3):IF LN=S T
HEN LOC=B
110 IFLN=ETHEN POKE LOC,PEEK(B):POKE LOC
+1,PEEK(B+1):GOTO 200
120 B=PEEK(B)+256*PEEK(B+1):GOTO100
200 PRINT "ENTER"S"(RETURN) TO DELETE"
210 POKE44,8:CLR
```

Assuming you wish to load above an existing program, you must first enter:

```
X=PEEK(46)+1:POKE X*256,0:POKE 44,X:NEW
```

as a direct command. This extracts the current value of the top of the **BASIC** program (high byte), and then adds one, to give the start of the next page (block of 256 bytes) of memory. The first byte of this page must be set to 0, since the 64 expects the first byte of the **BASIC** program area to be 0, and then the new bottom of **BASIC** is set by **POKE**ing location 44. **NEW** then resets all the other **BASIC** program pointers. (This

process is not necessary if you're using it from within the same program from which you're deleting blocks). The block delete program may then be **LOAD**ed, and it will be held in memory above the first program. It will probably occupy the variables area for the first program, but this doesn't matter in this instance.

After the first and last line numbers of the block to be deleted have been entered, our current high memory program starts from the beginning of the first program (now "hidden" but still there), checking each line number in turn. The address of the first byte of the link address is stored if the line is the specified start of the block (line 100). The program lines continue to be stepped through (line 120 works out the link address and sets B equal to it) until the line number is found to be equal to E, the last line of the block (line 110). The link address of the start line (addresses **LOC** and **LOC+1**) is then **POKE**d with the values of the link address of the last line (pointing to the start of the line after the specified block). The prompt is displayed, and if the program is being used from above another, the **BASIC** start is reset to 2049 **(POKE 44,8)**. Since the link address of the start line of the block now indicates the first line after the specified block, when the 64 is instructed to delete a line (by entering just the line number followed by **RETURN**) the entire block is deleted, since the 64 deletes a line by wiping out everything held between the start of the deleted line and the link address given in the line.

The next program enables us to renumber lines, although it does not change the line numbers following **GOTO**, **GOSUB**, or **THEN** statements. With this program, we give a version for **LOAD**ing before you start to program. It could be used in the same fashion as the block delete program above, but shifting the **BASIC** as done above is more illustrative then efficient with utility programs. Such utility routines should be loaded before starting to program as a "toolkit". They might include sorting, input checking, print positioning and suchlike routines commonly needed in programs, as well as

programming aids. A block delete routine ensures any routines not required for a specific program can be swiftly eradicated.

```
60000 REM*RENUMBER UTILITY*
60010 REM** FOR USE WHEN PART OF
60020 REM* PROGRAM (WITH LINE NUMBERS
60030 REM* LESS THAN 60000!)
60040 REM* NOT GOTO'S AND GOSUB'S!!!!
60050 REM* CALL WITH RUN 60000    **
60060 INPUT"RENUMBER START LINE NO.";B
60070 INPUT"IN STEPS OF";S
60080 Z=0:A=2049:REM START NORMAL BASIC
60090 NXT=PEEK(A)+256*PEEK(A+1):IF PEEK(
NXT+2)+256*PEEK(NXT+3)=60000THEN Z=1
60100 LH=INT(B/256):LL=B-256*LH:REM NEW
LINE NUMBER
60110 POKEA+2,LL:POKEA+3,LH:IF Z=1 THEN6
0130:REM POKE NEW LINE NO.,END IF Z=1
60120 B=B+S:A=NXT:GOTO60090 :REM NEXT LI
NE NUMBER,NEXT LINE
60130 PRINT"RENUMBER FINISHED"
60140 LIST
```

After the user has **INPUT** the first line number and the step size a flag **Z** is initialised, and **A** set to the normal start of **BASIC**. Remember that if you're shunting memory around for one reason or another, you should **PEEK** locations 43 and 44 for the start of **BASIC**, rather than use the default value. Line 60090 evaluates the link address of the current line, checking to see if it is the start of the renumber program and setting **Z** to 1 if it is. The new line number is split into high and low bytes by the next line, and then **POKE**d into the appropriate locations. If the flag is set the program ends, otherwise the line number **B** is incremented by the step value, **A** is set to the start location of the next line and the process repeats.

Finally, here's how to join two programs together. This is not

a true merge procedure, since the second program must have higher line numbers than the first, but it is nonetheless extremely useful. We use locations 45 and 46 to determine the high and low bytes for the address of the first of the two zeroes which mark the end of the program in memory, **POKE**ing the values into the pointers for the start of **BASIC**:

```
POKE 43,PEEK(45)-2:POKE 44,PEEK(46)
```

The second program may now be **LOAD**ed, and the first line will follow on from the last line of the first program. Entering **POKE 43,1:POKE 44,8** will reset the start of **BASIC** to the normal value, and the 64 has a single program in memory.

There's a lot more **POKE**ing in memory required to deal with Graphics and Sound, but the information for this is given in the relevant chapters.

CHAPTER SEVEN
SOUND AND THE FURY

The Commodore 64 possess very powerful sound and music capabilities, provided by the dedicated sound chip incorporated in the machine, which allows a wonderful range of precisely controlled sounds to be produced. The overlord of this panoply of sound is the SID chip, or to give its full title, the 6581 Sound Interface Device. This complex chip provides facilities for three separate channels ("voices") of sound, each of which is programmable in terms of frequency, volume waveform and envelope characteristics. Other facilities provided by SID are programmable filters, to further control the sound produced, and ring modulation, plus the capability to synchronise the output of one voice with that of another.

Unfortunately, to control the power and complexity of the sounds which can be produced by SID requires equally complex operations from the programmer. The SID chip is controlled by a group of 29 eight-bit registers, each of which is memory mapped to a location in the 64's memory. The SID chip copies values from or writes values to these locations so that they mirror its own registers. **POKE**ing these locations with suitable values gives us control over all the SID chip functions. However, in some cases we are interested in a specific bit or group of bits within each byte, or the value of a nybble (half a byte, or four bits) and not simply the value of 0 to 255 that is **POKE**d. This requires us to perform preliminary calculations on the values before we can **POKE** the locations correctly. Since the memory locations are dedicated to the SID chip registers, they do not act in the same way as normal memory locations, and this also affects the way we make use

of them. Specifically, some register locations are **WRITE ONLY.** This means we can only **POKE** values into them. Attempts to **PEEK** their values will produce meaningless results. Some are **READ ONLY,** which means we can **PEEK** the values inserted into them by the SID chip, but we can not affect their values by **POKE**ing. The practical result of having some registers as write-only is that we must store current values in variables, since these cannot be ascertained by means of a **PEEK** instruction.

SID WAVES AN ENVELOPE

Let's first take a look at the essentials of the process of programming SID's sound synthesis options. We have three voices available, all of which are similar in their data requirements. The basic thing we need for a musical sound (as opposed to the noise the 64 can also produce), is the frequency or pitch. If we want to deal with musical notes, we have to define the frequency accurately, to maintain the correct relationships between notes and keep it all in tune. The simplest of systems, quite good enough for all but the most sensitive of ears, is known as the even- or equal-tempered scale, which is mathematically exact, and is that used on fretted instruments. With this system, there are twelve semi-tones per octave, so for example D flat is considered the same as C sharp, and the musical interval known as the major third does not sound right to the trained ear. For most of us, it's fine, and the table given later in this chapter provides the appropriate values for these frequencies. Should anybody want to work out other scales, which is most likely to involve oriental music rather than a differently tempered scale (the 64 has great sound capabilities for a computer, but is not a Stradivarius!), the formula required is:

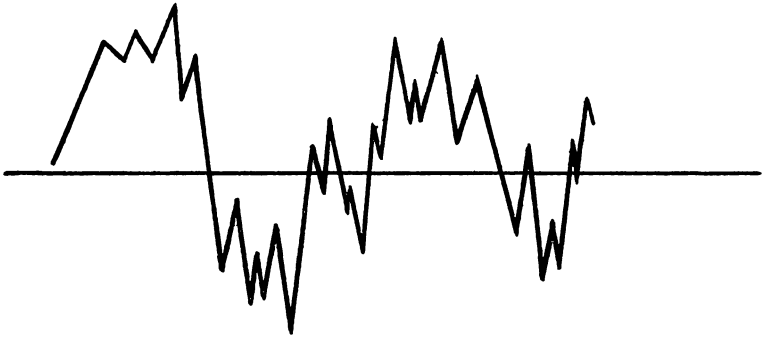
Frequency = Frequency Register Value ★ 0.05872548

This assumes that one is working with the European PAL

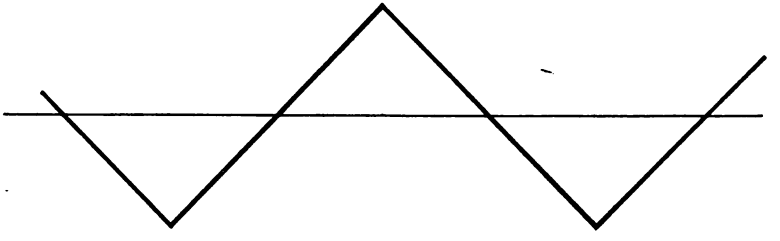
television display system, and the associated 0.98525 MHz system clock. For the US system (NTSC), the multiplier is 0.06095946. This US standard is used by Commodore in all its documentation.

The frequency register value is a 16 bit number stored in the first two bytes of the registers for each voice. Since two bytes can hold values up to 65535, we have a possible range of frequency from 0 to circa 4000 cycles per second for the fundamental frequency.

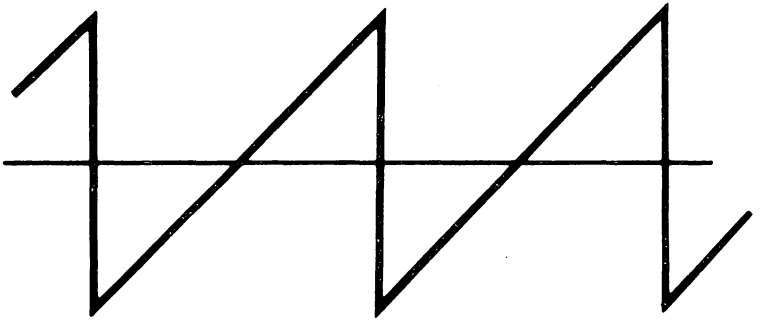
Having defined the frequency of a note we next need to specify the waveform, or frequency spectrum of the note. Realistic sounds produced by instruments have a characteristic timbre (which is how we recognise, say, a clarinet), due to the set of overtones produced. An overtone or harmonic is an integer multiple of the base frequency. The 64 has four types of basic waveform available, these being Triangular, Sawtooth, and Pulse for musical tones, and Noise for sound effects. Noise produces a chaotic mixture of random frequencies known as white noise, with no ordering principle.



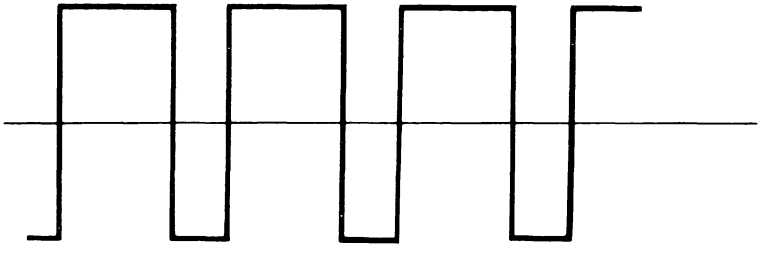
The Triangular waveform looks like this:



It has a similarity to a sine wave, and produces a smoother and more mellow sound than the sawtooth or pulse waveforms. The waveform contains only those overtones which are odd multiples of the base frequency. The sawtooth waveform differs in that it contains all the harmonics, and in different proportion. Where the triangular waveform has overtones with a volume setting proportional to the square of the harmonic, so that the frequency which is three times that of the fundamental frequency has a volume setting of 1/3★3 or 1/9 that of the fundamental, and so on, the sawtooth has a volume setting of 1/3 the volume of the base note for the same harmonic. The sawtooth waveform produces a brighter and brassier sound, and gets its name from the graph shape:

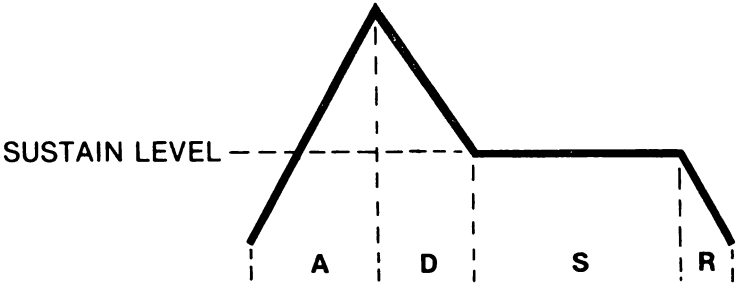


The fourth waveform is the pulse wave, of which the square wave is a special case:



The pulse wave contains only the odd harmonics, but the proportions vary with the width of the high portion of the pulse, which is specified separately from the base frequency. In the case of the square wave (equal high and low portions), the proportion of each harmonic is set by the reciprocal of the harmonic multiple, as with the sawtooth wave. The variability of the pulse width allows a large variety of sound qualities to be produced by the pulse waveform.

As well as the basic sound given by the waveform, the specific quality of the sound is generated by the sound envelope. This is the way the sound varies over time, and is divided into four sections. Attack is the period during which the sound increases to maximum volume, Decay the time taken for the sound to drop from the highest volume to the Sustain level (defined as a proportion of the maximum amplitude) at which the sound persists until switched off, when it reduces to zero over the period specified by the Release rate. Thus we have a cycle like this:



The values chosen for this set of **ASDR** values define the sound envelope. Different musical instruments have characteristic envelopes: drums have rapid attack rates and fast decay rates, with no appreciable sustain period, whereas a flute or a violin will have a gradual attack, little decay, and a long sustain period. Note that whilst Attack,

Decay and Release values specify periods of time, Sustain is specified in terms of a percentage of maximum volume, and is timed by switching the note on and off.

As well as waveform and envelope, the SID chip allows us further opportunities to modify the sound produced by means of filtering. There is only one set of filters, through which the output from any or all of the three voices may be passed, controlled by an additional group of registers. For all filters, a cut-off frequency needs to be specified as a reference point. The high-pass filter attenuates or reduces the frequencies below the cut-off point, and does not affect frequencies higher than this. The low-pass filter has the opposite effect, whilst the bandpass filter attenuates all frequencies above or below the cut-off frequency. A further facility of the SID chip is resonance, which, rather than diminishing the quantity of unwanted frequencies, emphasises the small range of frequencies close to the cut-off frequency.

The SID chip also allows two types of interaction between different voices. The first is synchronisation, which links the output of one voice to the frequency of another. The two waveforms are **AND**ed, and this may be used to modify the output of one voice by the (audible or silent) output of another. Ring modulation similarly takes two frequencies, and outputs the sum and difference frequencies, i.e. if one frequency is 600 cycles per second and the other 1000, the output will be one tone of 1000+600=1600 cps, and one of 1000 – 600 = 400 cps.

LOCATING SID'S SET OF VALUES

Phew! All a bit complex, this sound synthesis lark! Unfortunately, now that we've seen the processes involved, we have to get to grips with doing it in practice. The tables below give the 29 registers of the SID chip for reference, and details of the Voice 1 and filter registers. The register locations start at **54272 ($D400),** and are broken up into three groups of

seven bytes, each controlling one of the three voices, followed by a four-byte group controlling filtering. All these registers are **WRITE ONLY.**

The last group of four bytes are **READ ONLY.** The first two hold the current values of potentiometers (games paddles) connected to the control ports, which can be used as external controls for a synthesiser program. The final two registers hold the current values of the frequency oscillator and the envelope generator for Voice 3. The values held in these two bytes can be utilised to produce dynamic effects by **PEEK**ing the values and then using them to affect other parameters.

Table 1 The SID Chip: Registers

Voice 1			
0	Frequency Low Byte	54272	D400
1	Frequency High Byte	54273	D401
2	Pulse Width Low Byte	54274	D402
3	Pulse Width High Byte	54275	D403
4	Control Register	54276	D404
5	Attack and Decay nybbles	54277	D405
6	Sustain and Release nybbles	54278	D406
Voice 2			
7	Frequency Low Byte	54279	D407
8	Frequency High Byte	54280	D408
9	Pulse Width Low Byte	54281	D409
10	Pulse Width High Byte	54282	D40A
11	Control Register	54283	D40B
12	Attack and Decay nybbles	54284	D40C
13	Sustain and Release nybbles	54285	D40D
Voice 3			
14	Frequency Low Byte	54286	D40E
15	Frequency High Byte	54287	D40F
16	Pulse Width Low Byte	54288	D410
17	Pulse Width High Byte	54289	D411
18	Control Register	54290	D412
19	Attack and Decay nybbles	54291	D413
20	Sustain and Release nybbles	54292	D414
Filters			
21	Filter Frequency Low 3 bits	54293	D415
22	Filter Frequency high byte	54294	D416
23	Resonance and Filter nybbles	54295	D417
24	Mode bits and Volume nybble	54296	D418
Read Registers			
25	Paddle potentiometer 1	54297	D419
26	Paddle potentiometer 2	54298	D41A
27	Voice 3 Oscillator	54299	D41B
28	Voice 3 Envelope	54300	D41C

Table 2 The SID Chip: Register Details

Register	Bit:	7	6	5	4	3	2	1	0
Voice 1									
0: Frequency Low		F7	F6	F5	F4	F3	F2	F1	F0
1: Frequency High		F15	F14	F13	F12	F11	F10	F9	F8
2: PW Low byte		PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
3: PW Hi nybble		—	—	—	—	PW11	PW10	PW9	PW8
4: Control reg.		Noise	Pulse	Sawt'th	Triang.	Test	RingMod	Synch.	Env.gate
5: Attack/Decay		A3	A2	A1	A0	D3	D2	D1	D0
6: Sust'n/Rel'se		S3	S2	S1	S0	R3	R2	R1	R0
Filters									
21: Freq. low bits		—	—	—	—	—	FF2	FF1	FF0
22: Freq. hi byte		FF10	FF9	FF8	FF7	FF6	FF5	FF4	FF3
23: Res'ce/Filts		R3	R2	R1	R0	F.Ext'l	F.V3	F.V2	F.V1
24: F.Mode/Vol		V3 Off	HighP.	BandP.	LowP.	Vol 3	Vol 2	Vol 1	Vol 0

Registers for Voice 2 (7-13) and Voice 3 (14-20) are identical to those for Voice 1, save for the Synch and Ring Modulation functions of the Control Register:
Voice 2 synchronises and modulates with the Voice 1 oscillator
Voice 3 synchronises and modulates with the Voice 2 oscillator
Registers 25 to 28 are 8-bit registers.

Let's run through the registers for Voice 1. Except as noted, the registers for Voice 2 and Voice 3 are exactly the same, merely occupying different locations.

REGISTERS 0 AND 1: VOICE 1 FREQUENCY

Register 0 **(location 54272)** holds the low byte of the frequency, Register 1 **(54273)** the high byte. The value is thus given by **PEEK(54272)+PEEK(54273) *256.** The table below gives the low byte/high byte values which need to be **POKE**d into locations **54272** and **54273** respectively to give the correct frequency output for each of the eleven notes in all the seven octaves the 64 can cover.

Table 3 **Musical Note Values**

Sound	Musical Note		Frequency Register		
Hertz	Note	Octave	Freq.	High	Low
16	C	0	272	1	16
17	C#	0	289	1	33
18	D	0	306	1	50
19	D#	0	323	1	67
21	E	0	357	1	101
22	F	0	374	1	118
23	F#	0	391	1	135
24	G	0	408	1	152
26	G#	0	442	1	186
27	A	0	459	1	203
29	A#	0	493	1	237
31	B	0	527	2	15
33	C	1	561	2	49
35	C#	1	595	2	83
37	D	1	630	2	118
39	D#	1	664	2	152
41	E	1	698	2	186
44	F	1	749	2	237
46	F#	1	783	3	15
49	G	1	834	3	66
52	G#	1	885	3	117
55	A	1	936	3	168
58	A#	1	987	3	219

Sound	Musical Note		Frequency Register		
Hertz	Note	Octave	Freq.	High	Low
62	B	1	1055	4	31
65	C	2	1106	4	82
69	C#	2	1174	4	150
73	D	2	1243	4	219
78	D#	2	1328	5	48
82	E	2	1396	5	116
87	F	2	1481	5	201
92	F#	2	1566	6	30
98	G	2	1668	6	132
104	G#	2	1770	6	234
110	A	2	1873	7	81
116	A#	2	1975	7	183
123	B	2	2094	8	46
131	C	3	2230	8	182
139	C#	3	2366	9	62
147	D	3	2503	9	199
156	D#	3	2656	10	96
165	E	3	2809	10	249
175	F	3	2979	11	163
185	F#	3	3150	12	78
196	G	3	3337	13	9
208	G#	3	3541	13	213
220	A	3	3746	14	162
233	A#	3	3967	15	127
247	B	3	4206	16	110
262	C	4	4461	17	109
277	C#	4	4716	18	108
294	D	4	5006	19	142
311	D#	4	5295	20	175
330	E	4	5619	21	243
349	F	4	5942	23	54
370	F#	4	6300	24	156
392	G	4	6675	26	19
415	G#	4	7066	27	154
440	A	4	7492	29	68
466	A#	4	7935	30	255
494	B	4	8412	32	220
523	C	5	8905	34	201
554	C#	5	9433	36	217
587	D	5	9995	39	11
622	D#	5	10591	41	95
659	E	5	11221	43	213

Sound	Musical Note		Frequency Register		
Hertz	Note	Octave	Freq.	High	Low
698	F	5	11885	46	109
740	F#	5	12601	49	57
784	G	5	13350	52	38
831	G#	5	14150	55	70
880	A	5	14984	58	136
932	A#	5	15870	61	254
988	B	5	16824	65	184
1046	C	6	17811	69	147
1109	C#	6	18884	73	196
1175	D	6	20008	78	40
1244	D#	6	21183	82	191
1318	E	6	22443	87	171
1397	F	6	23788	92	236
1480	F#	6	25202	98	114
1568	G	6	26700	104	76
1661	G#	6	28284	110	124
1760	A	6	29969	117	17
1865	A#	6	31757	124	13
1976	B	6	33648	131	112
2093	C	7	35640	139	56
2217	C#	7	37751	147	119
2349	D	7	39999	156	63
2489	D#	7	42383	165	143
2637	E	7	44903	175	103
2794	F	7	47577	185	217
2960	F#	7	50404	196	228
3136	G	7	53401	208	153
3322	G#	7	56568	220	248
3520	A	7	59939	234	35
3729	A#	7	63498	248	10

To define a specific frequency not covered in the table for Voice 1, and then place the relevant values into these registers, we can use the following piece of coding:

```
10 INPUT F
20 FR=F/0.05872548
30 FHI=INT(FR/256)
40 FLO=FR-(FHI*256)
```

This takes the desired frequency F, scales it to give the frequency register value, and then divides this into high **(FHI)** and low **(FLO)** byte values for **POKE**ing. Note that we could use **FLO=FR AND 255** to derive **FLO** directly, rather than as done here. In the main, we will keep away from bit wise operations in this chapter, since they are a common, if deplorable, source of problems for the neophyte.

REGISTERS 2 AND 3: PULSE WIDTH

If the pulse waveform is selected then the pulse width must be specified. This is held as a 12 bit value. Register 2 **(54274)** holds the low byte, and Register 3 **(54275)** holds the high value, stored in the first four bits (low nybble) of the byte, and this is thus restricted to values 0-15. The width of the high portion of the pulse, as a percentage of the total width, is given by the value stored in the 12 bits divided by 40.95 and will be as near as makes no odds a square wave when the value stored is **2048 ($800).**

REGISTER 4: CONTROL REGISTER

This byte **(54276)** controls a variety of functions, switching them according to the status (0 or 1) of each bit, as follows:

Bit 0: Envelope On/Off

This gate bit controls the envelope generator for Voice 1. When set to 1 it triggers the output for Voice 1, starting the

Attack/Decay cycle. The Sustain cycle will persist (playing the sound) until this bit is set to zero, when the Release portion of the **ASDR** cycle starts. You must first ensure that the gate bit is set to 0, and then set it to 1, to start the note playing, as it is the change that acts as the trigger. Once the note is triggered, the attack and decay portions of the envelope will be executed, and the note will continue at the volume specified by the sustain parameter until the bit is changed to 0, when the release sequence starts. Terminating the envelope sequence at any point by ungating the sound starts the release cycle, which will itself be cut short if the sound is gated on again before the release period is over.

The easiest way to handle gating and ungating is to store the current value of the whole control register byte apart from this bit in a variable, and **POKE** with this variable plus one to gate, and the variable alone to ungate, since these registers are write-only, and may not be **PEEK**ed for the current value. Thus, if a variable CR holds the current value of the control register bits 1-7 then **POKE 54276, CR+1** will turn the sound on, **POKE 54276,CR** will turn it off.

Bit 1: Synchronisation On/Off

When set to 1, the frequency specified for Voice 1 is synchronised with that of Voice 3. This causes the waveforms to be **AND**ed together to give complex sound structures to the output of Voice 1 which vary at the frequency of Voice 3.

Note that the output from the Voice 3 oscillator does not have to be audible for synchronisation to occur, as there is a bit in the Resonance/Filter byte (bit 7 of register 24) which when set to 1 prevents Voice 3 from reaching the output stage of the circuit.

The equivalent bit in the registers for Voice 2 (bit 1 of location 54283) synchronises the Voice 2 output with that of Voice 1, and Voice 3, with the appropriate bit of 54290 set to 1, is synchronised to Voice 2.

Bit 2: Ring Modulation On/Off

When set to 1, this bit turns on the ring modulation function, provided that the triangular waveform has been selected. Voice 1 output is combined with that of the oscillator of Voice 3. The equivalent bit for Voice 2 modulates Voice 2 with the Voice 1 oscillator, and for Voice 3 links it with the Voice 2 oscillator.

Again, as with synchronisation, the Voice 3 oscillator can be used with no output by setting bit 7 of register 24 to 1, preventing any audible output from Voice 3.

Bit 3: Test Bit

This has no function in sound production, but may need to be set back to 0 if you make the mistake of selecting another waveform whilst Noise is on, since this may cause the noise output of the relevant voice to cease when this bit will be set to 1 by the system. Beware, since this effect is not entirely predictable!

Bit 4: Triangular Waveform On/Off

This bit and the next three select the current waveform. More than one may be selected (the outputs are logically **AND**ed), to generate additional waveforms, but note the warning above about noise output.

Bit 5: Sawtooth Waveform On/Off

Bit 6: Pulse Waveform On/Off

Selects pulse wave and accesses the pulse width registers to define waveform.

Bit 7: Noise Waveform On/Off

REGISTER 5: VOICE 1 ATTACK AND DECAY

Bits 4 to 7 (high nybble) hold the Attack rate, and bits 0 to 3 (low nybble) the Decay rate, with each being a value 0-15. These values specify the time taken to reach maximum volume after the sound starts, and the time for the volume to then drop to the selected Sustain volume respectively, as listed in the table below. The expression to **POKE** the values into the byte is: **POKE 54277,A*16+R**

ATTACK/DECAY RATES

Value	Attack	Decay
0	0.002	0.006
1	0.008	0.024
2	0.016	0.048
3	0.024	0.072
4	0.038	0.114
5	0.056	0.168
6	0.068	0.204
7	0.08	0.24
8	0.1	0.3
9	0.25	0.75
10	0.5	1.5
11	0.8	2.4
12	1.0	3.0
13	3.0	9.0
14	5.0	15.0
15	8.0	24.0

REGISTER 6: VOICE 1 SUSTAIN AND RELEASE

Bits 4 to 7 hold the Sustain volume level, a value 0-16 giving the volume to which the sound will decay, and at which the sound will continue to be output until switched off. The 16 levels define 16 equal steps between zero and the master

volume set by Register 24. Bits 0 to 3 store the Release rate, where the values 0 - 15 specify times for the Release cycle which are the same as those given above for Decay.

The chosen values are stored with the statement:

POKE 54278,16*S+R

Voice is controlled by Registers 7 to 13, which occupy locations **54279** to **54285.**

Voice 3 is controlled by Registers 14 to 20, which occupy locations **54286** to **54292.**

REGISTERS 21 AND 22: FILTER FREQUENCY

All filters when activated use the value held in these registers to specify the cut-off or reference frequency. The eight bits of Register 22 are followed by bits 3 to 0 of Register 21 to form an 11 bit number. Register 22 **(location 54294)** thus holds values 0-255, while Register 21 **(54293)** holds 0-7. The combination hold values 0 to 2047 **(2^11).** Multiplying this value by 6 gives you the approximate frequency value. For a filter cut-off frequency F, the following piece of code will store the correct values:

```
20  FR=INT(F/6)
30  FHI=INT(FR/8):FLO=FR-8*FHI
40  POKE 54293,FLO:POKE 54294,FHI
```

REGISTER 23: FILTER ON/OFF AND RESONANCE

This register controls the activation of the filter for each of the three voices, and also the external audio signal which can be passed through the 64's filters if desired. Bits 0, 1, and 2 pass the outputs from Voice 1, Voice 2 and Voice 3 respectively, through the filter when set to 1. Bit 3 passes external audio input (via pin 5 of the audio/video socket) through the filter

when set to 1. This allows you to use the filters to modify sound from a guitar, electric organ etc.

The high nybble (bits 4-7) of this byte stores the resonance setting (0 for no resonance to 15 for maximum resonance).

REGISTER 24: VOLUME AND FILTERS/VOICE 3 OFF

This register holds the master volume control for all three voices in the low nybble (bits 0-3). The volume range is 0-15, with 0 giving no output and 15 the maximum volume. Some level other than zero must be selected to produce any output from SID. Bit 4 switches in the low-pass filter when set to 1, bit 5 selects bandpass filter mode, and bit 6 the high-pass filter. Filter modes are additive, and more than one may be selected. Bit 7 switches off Voice 3 output when set to a 1, to allow the Voice 3 output to be used for synchronisation or ring modulation purposes without being audible.

The above registers are all **WRITE ONLY,** and can be **POKE**d but not **PEEK**ed for the current value. The following four registers are **READ ONLY,** and provide access to values which can be fed back into other SID chip registers for dynamic control of the sounds produced.

REGISTERS 25 AND 26: PADDLE VALUES

These two registers hold the values supplied by a pair of control paddles connected to Control Port 2. Each paddle will return a value from 0-255 in integer steps, depending on the potentiometer setting of the paddle. The paddle values can be read from the two registers, and these values used to change other SID chip values. However, **BASIC** is not fast enough to deal with rapidly changing paddle values, for use in smooth frequency changes and the like. Suitable scaling of the returned paddle values can help if we want to affect, say, the Attack parameter, since only 16 values need to be derived from the 0-255 range produced by the paddles. Real-time paddle control of frequency or pulse width using paddles

would require a machine-code routine.

REGISTER 27: VOICE 3 OSCILLATOR

This register reflects the current value of the oscillator for Voice 3. This output can be used to alter the values of other SID registers. The obvious use is with the modulation and synchronisation facilities, but the output can be scaled as necessary and used with any SID registers. The sequence and timing of numbers generated at this location depends on the waveform and frequency values current for Voice 3.

The Triangle waveform generates numbers which increase in integer steps from 0 to 255 and then decrement back to 0.

The Sawtooth waveform gives numbers which increment from 0 to 255 and then drop to zero to start the process again. The rate at which the numbers increment is determined by the frequency of Voice 3 specified.

The Pulse waveform produces only the values 0 and 255, at a rate dependant on the frequency of Voice 3 and in a time ratio specified by the pulse width register value.

The Noise waveform produces a sequence of random numbers between 0 and 255 at a rate dependant upon the frequency of Voice 3. This can be used as a random number generator.

Normally, when this register is used for modulation of other values in the SID registers, output from Voice 3 is not required. This is achieved by switching bit 7 of Register 24 to 1.

REGISTER 28: VOICE 3 ENVELOPE

The values produced at this location mirror the output of the envelope generator for Voice 3. Used in a similar way to the values of Register 27, the changing set of values for the amplitude of Voice 3 produced by the selected **ASDR** can be

fed back into other SID chip registers for a variety of effects.

SID SOUNDS OFF

If you've read the above description of SID's configuration, you'll probably be a bit overwhelmed by all the data. If you've skipped past it, you'll be clear-headed but ill-informed. Either way, we'll now get down to putting SID through his paces. Refer to the tables of the registers for a quick reminder of their function, or the detailed description above for explanations, if you don't understand anything that's happening in the programs. So how do we get a sound out of SID? Key in the following program and **RUN** it.

```
5 REM SIMPLEST SOUND
10 SID=54272:REM DEFINE CHIP START
20 FOR C=0 TO 24: POKE SID+C,0:NEXT:REM
CLAR CHIP
30 POKE SID+24,10: REM VOLUME
40 A=9:REM ATTACK 0-15
50 D=2:REM DECAY 0-15
60 POKE SID+5,16*A+D:REM A & D REGISTER
70 S=5: REM SUSTAIN VOLUME 0-15
80 R=0: REM RELEASE 0-15
90 POKE SID+6,16*S+R:REM S & R REGISTER
100 WF=2↑5:REM DEFINE TRIANGLE WAVEFORM
110 REM IF PULSE SELECTED,SET PULSE WIDT
H HERE
120 FLO=209:FHI=18:REM SELECT FREQUENCY
130 POKE SID,FLO:POKE SID+1,FHI:REM STOR
E FREQUENCY
140 NL=500:REM NOTE LENGTH VARIABLE
150 POKE SID+4,WF+1:REM DEFINE WAVEFORM,
GATE SOUND
160 FOR T=0 TO NL:NEXT:REM NOTE LENGTH
170 POKE SID+4,WF:REM GATE OFF
180 FOR T=0 TO 50:NEXT:REM RELEASE DELAY
```

The program merely goes through the minimum steps to define and play a note on Voice 1. Not an amazing result from all those program lines, you may think, but it serves to introduce the elements of SID in practice. Let's look at each line to get the essentials.

Line 10 sets a variable **SID** equal to **54272**, the location of Register 0. This enables us to reference all the SID registers by their number, as they have been treated in the discussion above.

Line 20 has a loop which runs through all the registers (25) which we can write values to, setting each to 0. This clears the chip, leaving us a clear field for our musicological endeavours. Since we're only dealing with Voice 1, and in a simple fashion, we won't be accessing most of these registers, but this should be standard practice before using SID.

Line 30 places the overall volume setting (10 in this case) into Register 24.

Line 40 and 50 selected values for the attack and decay rates from the range 0-15, and line 60 calculates the value to be **POKE**d into Register 5.

Lines 70 and 80 similarly select values for the sustain volume and release rate, and line 90 **POKE**s the correct value into Register 6.

Line100 defines a variable **WF** to hold the value to be **POKE**d into the Voice 1 control register (Register 4). We don't **POKE** this value in yet, but note that using the value 2 ↑ 5 is a simple method of defining the correct value needed to set bit 5 of a byte to 1, which in this case will select the triangular waveform. This is an easy way of keeping track of which bits are set to 1, avoiding the necessity of thinking in binary and then converting to decimal. The use of variables, apart from being vital to store current values, since the registers cannot be **PEEK**ed, helps clarify what is going on in SID's somewhat complex innards.

Line 110 is just a reminder that, if we select the pulse waveform, we must set the pulse width values in Registers 2 and 3. Remember we are only dealing with Voice 1 here, and any of the operations we are describing here must also be carried out on the equivalent registers for Voice 2 and Voice 3 if we want to activate those as well.

Line 120 sets the **FLO** (frequency low byte) and **FHI** (frequency high byte) values to sound the note D in octave 4 of the 64's range. (See the note table given above).

Line 130 **POKE**s the frequency registers for Voice 1 with the values stored in **FLO** and **FHI.**

Line 140 sets a variable **NL** to hold a value defining the note length. Since the sustain portion of the **ADSR** cycle can go on indefinitely, we need a timing mechanism to provide us with a finite note length and use empty **FOR. . .NEXT** loops for the purpose. A rough check will show you that a count of 1000 in such a loop takes approximately a second, so if we set **NL** to 500 and then use it in a timing loop we'll get a note that lasts 0.5 seconds, including the attack and decay portions of the cycle, but not including the release cycle, which occurs after our count has finished.

Line 150 starts the note off, by **POKE**ing the control register with the waveform value **WF** defined in line 100, plus 1 to set bit 0 to 1 and trigger the envelope cycle.

Line 160 uses **NL** to define the number of times the empty **FOR. . .NEXT** loop will execute.

Line 170 switches off the gate bit, and ends the sustain portion of the cycle. We do this by **POKE**ing Register 4 with the waveform value **WF.** If we set the whole byte to 0, we would not only switch off the envelope, but would also abort the release portion of the cycle, since SID would have no defined waveform, and the note would cease abruptly.

Line 180 uses another **FOR. . .NEXT** loop to give a delay

which allows time for the release cycle to complete before the program ends.

To illustrate the utility of storing values in variables, and the need for delays, add these lines to the program:

```
5 REM SIMSOUND
190 NL=NL-100:IF NL<100 THEN END
200 GOTO 150
```

RUN the modified program, and you'll hear a repeated note, with a shorter cycle as **NL** is decreased in line 190. The attack and decay cycle does not have time to execute with smaller values of **NL**, and the sound is cut shorter each time by the jump in line 200, which sends the program back to start the envelope cycle again. Only after the program has **END**ed is there enough time for the full release cycle to execute. Try increasing the size of the loop in line 180 to, say, 200 and running the program again. Changing the assignment statements of lines 40, 50, 70 and 80 to **INPUT** statements, as per the program below, will enable you to experiment with the **ADSR** values easily.

The other change we've introduced into this modified program is a demonstration of the capability of dynamic control of SID's sounds. The change in line 160, and the new line 165 use the value of the loop counter **T** to derive a suitably scaled value to be **POKE**d into the high byte of the frequency register at every pass through the loop. SID continually checks his registers, and updates any changes.

```
5 REM **SIMPLE SOUND MODIFIED*
10 SID=54272:REM DEFINE CHIP START
20 FOR C=0 TO 24: POKE SID+C,0:NEXT:REM
CLAR CHIP
30 POKE SID+24,10: REM VOLUME
40 INPUT"ATTACK?(0-15)";A
50 INPUT"DECAY?(0-15)";D
```

```
60 POKE SID+5,16*A+D:REM A & D REGISTER
70 INPUT"SUSTAIN?(0-15)";S
80 INPUT"RELEASE?(0-15)";R
90 POKE SID+6,16*S+R:REM S & R REGISTER
100 WF=2↑5:REM DEFINE TRIANGLE WAVEFORM
110 REM IF PULSE SELECTED,SET PULSE WIDT
H HERE
120 FLO=209:FHI=18:REM SELECT FREQUENCY
130 POKE SID,FLO:POKE SID+1,FHI:REM STOR
E FREQUENCY
140 NL=500:REM NOTE LENGTH VARIABLE
150 POKE SID+4,WF+1:REM DEFINE WAVEFORM,
GATE SOUND
160 FOR T=0 TO NL
165 POKE SID+1,T/2:NEXT
170 POKE SID+4,WF:REM GATE OFF
180 FOR T=0 TO 50:NEXT:REM RELEASE DELAY
190 NL=NL-100:IF NL<100 THEN END
200 GOTO 150
```

SID OSCILLATES A BIT

The only waveform we've used up to now is the triangular waveform (selected by setting bit 5 of the control register to 1). Setting other bits gives us different waveforms. The next program demonstrates how the same note sounds with different waveforms.

```
5 REM **WAVEFORM DEMO**
10 SID=54272:FOR C=0 TO 24:POKE SID+C,0:
NEXT
20 POKE SID,195:POKE SID+1,16:REM FREQUE
NCY
30 POKE SID+24,15:REM VOLUME
40 POKE SID+6,240:REM MAX SUSTAIN VOLUME
50 REM A,D,R VALUES ZERO
60 POKE SID+3,8:REM SQUARE WAVE PULSE WI
DTH
```

```
70 POKE SID+4,2↑7+1:REM NOISE W/FORM AND
 SOUND ON
80 PRINT"▓▚NOISE ENVELOPE"
90 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"
100 FOR C=0 TO 1500:NEXT
110 POKE SID+4,2↑4+1:REM SET TRIANGLE
120 PRINT"▓▚TRIANGULAR WAVEFORM"
125 PRINT"/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\"
130 FORC=1 TO 1500:NEXT
140 POKE SID+4,2↑5+1:REM SET SAWTOOTH
150 PRINT"▓▚SAWTOOTH WAVEFORM"
155 PRINT"◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤◤"
160 FORC=1 TO 1500:NEXT
170 POKE SID+4,2↑6+1:REM SET PULSE W/F
180 PRINT"▓▚PULSE WAVEFORM":PRINT:PRINT"
SET FOR SQUARE WAVE"
185 PRINT"_⎍⎍⎍⎍⎍⎍⎍⎍⎍⎍⎍⎍⎍⎍"
190 FORC=1 TO 1500:NEXT
200 PRINT"□□DIFFERENT PULSE WIDTHS NOW"
210 PRINT"▓▓PULSE WIDTH VALUE="
220 POKE SID+4,2↑6+1:FOR P=50 TO 4050 ST
EP 200:REM PULSE W/F ON, PW VALUES LOOP
230 PW=P:PHI=INT(PW/256):PLO=PW-PHI*256:
REM PULSE WIDTH VALUES FOR POKING
240 POKE SID+2,PLO:POKE SID+3,PHI
250 PRINT"□"SPC(18)P
260 FOR C=0 TO 500:NEXT
270 NEXT:POKE SID+4,0
```

After clearing the chip and setting a frequency and volume, line 40 sets the sustain volume to maximum. This is simple binary, so it's hardly necessary, even for a mind made arithmetically slothful by the availability of cheap computing power, to get the 64 to work out **16★15.**

Since this program will use the pulse waveform, we need to place suitable values for the pulse width into Registers 1 and 2. Initially we want a square wave, and, as given above, the value for this is 2048. Conveniently enough, this is **$800,** and so we just have to **POKE** the high byte (Register 3) with 8. Line

70 sets the noise waveform on (bit 7=1) and gates the sound on. The program displays suitable information on the screen, and the sound continues until the timing loop ends. We then go through the same process for each waveform.

At line 200, we **PRINT** some more relevant words, and then proceed to experiment with different pulse widths. Line 220 sets up a loop which runs through the range of values available for pulse width (omitting maximum and minimum values, which give a steady and rather boring pure tone) in audibly significant steps. Line 230 takes the current value of the loop variable **PW** and divides it into high and low byte values **(PHI** and **PLO),** and line 240 does the actual **POKE**ing. So that you can get some idea of the effect of different values for the pulse width, line 250 displays them, and after a half-second wait produced by line 260, the program loops back for the next pulse width value. When there are no more, we turn off both waveform and sound by zeroising the control register.

SID SHOWS HIS FORM

To get a better appreciation of the characteristics of waveforms, we can switch to Voice 3 and take advantage of Registers 27 and 28, which give us access to the Voice 3 oscillator (the waveform generator) and envelope generator respectively. Within the constraints imposed by the slowness of **BASIC,** we can display the results of **PEEK**ing these registers graphically, to get a better idea of what the output is like. Here's a program which shows the waveform output.

```
5 REM**WAVEFORM DISPLAY**
10 SID=54272:FOR C=0 TO 28:POKE SID+C,0:
NEXT
20 POKE SID+24,15:POKE SID+14,10:POKE SI
D+20,16*6:POKE SID+17,8
30 D$=">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>"
40 FOR WF=4 TO 7:POKE SID+18,2↑WF+1
50 PRINT LEFT$(D$,(PEEK(SID+27)/8))
```

```
60 IF PEEK(203)=64 THEN 50
70 POKE SID+18,0
80 FOR F=1 TO 300:NEXT
90 NEXT
```

After the usual opening clearance, and setting the volume, we get to **POKE** some locations we haven't previously had a go at. The Voice 3 frequency registers are dealt with easily, since we can only choose an extremely low frequency for the oscillator, or **BASIC** will access the changing values too slowly to create a meaningful display. We **POKE** register 14 (the low byte register) with 10. This is not an audible frequency, so we don't get a tone, although individual clicks can be heard.

SID+20 defines the sustain/release register, and sustain is set at 6. Finally, Register 17 is the pulse width high byte, and is set to 8 to give a square wave.

Line 30 defines a string for use in the display, and line 40 has a loop to cycle through the possible waveforms. The control register for Voice 3 is **POKE**d to set each waveform in turn and activate the sound.

In line 50, register 27 is **PEEK**ed and the result scaled down to give a maximum value of 32, used to define how many characters of **D$** are to be selected. Line 60 uses location 203 to determine when a key has been pressed (see below for more on the use of **PEEK(203)**). If not, the display continues, otherwise the control register is set to zero and after a pause, the next waveform is selected.

We can do the same for the envelope values available from Register 28. You should not have any problems with this program. Since we want audible output, we select a frequency, define a suitable set of **ADSR** values (suitable for the display, that is), turn the sound on and use the values appearing at Register 28 to generate a representation of the envelope form:

```
5 REM ** ENVELOPE DISPLAY **
10 SID=54272:FOR C=0 TO 28:POKE SID+C,0:
NEXT
20 POKE SID+24,15:POKE SID+14,48:POKE SI
D+15,4
30 POKE SID+19,10*16+10:REM  ATTACK,DECA
Y
40 POKE SID+20,5*16+10:REM SUSTAIN,RELEA
SE
50 D$=">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>"
60 POKE SID+18,2↑5+1:REM SAWTOOTH+GATE
70 PRINT LEFT$(D$,(PEEK(SID+28)/8))
80 IF PEEK(203)=64 THEN 70
90 POKE SID+18,0:REM UNGATE IF KEYPRESS
100 GOTO 70
```

The sound is sustained until you press a key. The release period then follows, with a gradual diminution of the amplitude of the sound. Try altering the **ADSR** parameters to see other shapes of envelope.

With both these programs, of course, we've used the scrolling screen to simplify the display and speed up the repeated accesses to the registers. If it bothers you having the time axis of the graphs pointing to the centre of the earth, you can always turn you head clockwise or your TV widdershins through 90 degrees! For the same reason, you'll have to **STOP** the envelope program yourself; too many conditional tests slow down the program.

SID PLAYS AROUND

With what we've seen thus far, you may feel SID isn't working for his keep, but we're almost there! This next program demonstrates both the simplest method of storing note values, as **DATA** statements, and changing the **ADSR** settings in a program by the same means. An array is set up to hold the note values (from Table 3) which form a D minor scale. The

DATA is **READ** into the array, after the usual SID palaver at the beginning. A set of nested loops then starts, in the outermost of which the different **ADSR** values are **READ** and **POKE**d. The value of the loop variable **I** used in the next loop provides the waveform value, and in line 100 the frequency high and low byte values stored in the array are used to set the frequency for each note of the scale in turn.

```
10 DIM S(6,1):REM ARRAY FOR NOTE VALUES
20 SID=54272:FOR C=0 TO 24:POKE SID+C,0:
NEXT:REM DEFINE & CLEAR CHIP REGISTERS
30 POKE SID+24,15:REM VOLUME
40 PW=2048:PHI=INT(PW/256):PLO=PW-PHI*25
6:REM PULSE WIDTH VALUES FOR SQR WAVE
50 POKE SID+2,PLO:POKE SID+3,PHI
60 FOR C=0 TO 6:READ FLO,FHI:S(C,0)=FLO:
S(C,1)=FHI:NEXT:REM SET ARRAY
70 FOR K=1 TO 3
80 READ A,D,S,R:POKE SID+5,16*A+D:POKE S
ID+6,16*S+R:REM EACH ADSR IN TURN
90 FOR I=4 TO 6:WF=2↑I:REM SET EACH WAVE
FORM IN TURN
100 FOR N=0 TO 6:POKE SID,S(N,0):POKE SI
D+1,S(N,1): REM SET FREQUENCY
110 POKE SID+4,WF+1:REM SET W/FORM,GATE
ON
120 FOR T=1 TO 500:NEXT:REM NOTE LENGTH
130 POKE SID+4,WF:REM GATE OFF
140 FOR T=1 TO 75:NEXT:REM RELEASE TIME
150 NEXT N
160 NEXT I
170 NEXT K
180 POKE SID+4,0
1000 DATA 142,19,243,21,54,23,19,26,68,2
9,220,32,201,34:REM D MINOR SCALE
1010 DATA 0,9,0,9: REM ADSR XYLOPHONE
1020 DATA 5,5,0,0: REM ADSR CLARINET
1030 DATA 3,0,9,1: REM ADSR FLUTE
```

The **ADSR** values given are those which provide a reasonable facsimile of the instruments given in the **REM** statements, dependant upon whether the waveform is appropriate. Versimilitude in reproducing the sound of a particular instrument is perfectly possible with SID's facilities, within the scope of the waveform, envelope, filter and resonance contributions to the sound, but it's probably more fun to invent synthetic sounds. Just because man used wood and metal for instruments before he discovered silicon, doesn't mean that Bach wouldn't have given his kids one!

Getting your 64 to play a tune is not that difficult. The program below again uses just Voice 1. The note values are stored in **DATA** statements, along with the duration of the note, i.e. the proportionate time for which it is to be played. In order to assist coding the notes from music notation, which is likely if you are going to do much of this sort of thing, we need a system of storing note length which allows us to deal with all the different note length requirements. The binary system of each size of note being half the previous one (or twice the previous one, depending on which way you're going) presents no problems, but the question of triplets (three notes of a particular type which are sounded in the time it would normally take for two of them) means that we have to be able to divide our duration by both two and three, so we set a standard crotchet at 24. The actual timing of the piece of music is achieved by scaling the coded duration to get the size of the **FOR...NEXT** loop needed. Changing the tempo of the piece then only requires the scale factor to be changed. The only other requirements for a simple tune are rests (silences), coded into the **DATA** as zero frequency, and a suitable set of **ADSR** values. We must also separate the notes, allowing for the release time of the envelope. Here's an example:

```
5 REM **TUNE1**
10 SID=54272:REM DEFINE CHIP START
```

```
20 FOR C=0 TO 24: POKE SID+C,0:NEXT:REM
CLEAR CHIP
30 POKE SID+24,15: REM VOLUME
40 A=2:D=0:S=10:R=1
50 POKE SID+5,16*A+D:REM A & D REGISTER
60 POKE SID+6,16*S+R:REM S & R REGISTER
70 WF=2↑5:REM DEFINE TRIANGLE WAVEFORM
80 SCALE%=25:REM CHANGE THIS TO ALTER TE
MPO
90 REM:READ NOTE VALUES,DURATION
100 READ FH%,FL%,DUR%:IF FH%=-1 THEN END
110 POKE SID,FL%:POKE SID+1,FH%
120 POKE SID+4,WF+1:REM DEFINE WAVEFORM,
GATE SOUND
130 FOR T=0 TO DUR%*SCALE%:NEXT
140 POKE SID+4,WF:REM GATE OFF
150 FOR T=0 TO 75:NEXT:REM RELEASE DELAY
160 GOTO 100:REM NEXT NOTE
500 REM TUNE DATA:HAUL THE BOWLINE
510 DATA69,147,36,69,147,12,65,184,24,87
,171,3,78,40,15,65,184,6,58,136,18
520 DATA65,184,6,69,147,12,58,136,12,52,
38,12,43,213,12,34,201,24,69,147,36
530 DATA69,147,12,65,184,24,52,38,18,52,
38,6,58,136,24,65,184,24,69,147,12
540 DATA-1,0,0:REM END TUNE VALUES
```

After initialising the SID chip values, the frequency high and low bytes and duration for each note in turn are read. The frequency registers are **POKE**d with the new values, and the sound turned on for the duration of the **FOR. . .NEXT** loop. The sound (but not the waveform) is turned off, and the release/note separation delay loop started. When this is completed, the program loops back to get the next note. Depending on the type of sound you want, you may wish to abort the release cycle before completion. This will occur when the next note starts, but in this case there will not be a clear gap between notes. This effect would require the sound to be turned off completely **(POKE SID+4,0)**, followed by an

additional empty loop for the note gap, before the next note is fetched from the **DATA** and the sound turned on again.

If we want to have more than one voice, the coding required is more complex, as shown in our next program. Lacking anything other than a global **RESTORE** statement, we cannot **READ** values direct from **DATA**, but must store the note values and durations in arrays. The overall timing is determined by the main program loop, which uses a **GOTO** statement to loop until the **END** condition is reached. Within this loop, separate counters must be maintained for each voice, decremented with each pass through the loop until they reach zero, when the next note value is fetched. If this is zero for the high byte, a silent period is indicated, and if negative, then no more notes are to be played by this voice. The variable **FIN** is incremented when this occurs, and **END**s the program when it equals the number of voices used for the music.

Setting up the initial values for the SID registers is also done somewhat differently. The first set of **DATA** is 24 items, giving the values to be **POKE**d into each register. Zero values ensure that any unused registers are cleared. The control registers are set using the values for the waveforms stored in the array **W(3),** for ease of switching the sound on and off, and the frequency register addresses for each voice are stored in the array **FR.**

```
5 REM *TRITUNE*
10 SID=54272:DIM FR(3)
20 DIM F(3,100,2):DIM D(3,100):DIM N(3):
DIM NN(3):REM FREQ,DURATION,NOTE ARRAYS
30 FOR C=0 TO 24:READ R:POKE SID+C,R:NEX
T:REM SET UP CHIP
40 W(1)=64:W(2)=64:W(3)=64:REM WAVEFORMS
 FOR VOICES
50 FR(1)=SID:FR(2)=SID+7:FR(3)=SID+14:RE
M LOW BYTE FREQ REGISTERS
```

```
60 REM READ NOTE DATA
70 FOR V=1 TO 3:C=0
80 READ FH:IF FH=-1 THEN N(V)=C:GOTO 100
90 C=C+1:READ FL,DUR:F(V,C,1)=FH:F(V,C,2
)=FL:D(V,C)=DUR/3:GOTO80
100 NEXT V
110 FOR V=1 TO 3
120 POKE FR(V)+4,W(V)+1:NEXT V:REM GATE
CHANNELS
130 NN(1)=1:NN(2)=1:NN(3)=1:REM NOTE COU
NTERS
140 FOR V=1TO3:REM EACH VOICE
150 POKE FR(V),F(V,NN(V),2):POKE FR(V)+1
,F(V,NN(V),1):REM SET CURRENT NOTE FREQ
160 D(V,NN(V))=D(V,NN(V))-1:REM DECREMEN
T DURATION
165 IF D(V,NN(V))<1THEN NN(V)=NN(V)+1:RE
M INCREMENT NOTE COUNT IF END NOTE
170 IF NN(V)>N(V)THEN FIN=FIN+1:POKE FR(
V)+4,W(V):IF FIN=3 THEN END
180 NEXT V
190 GOTO140
400 REM INITIAL SID CHIP VALUES
410 DATA 0,0,0,8,0,26,242:REM VOICE1
420 DATA 0,0,0,8,0,36,250:REM V2
430 DATA 0,0,0,8,0,10,96:REM V3
440 DATA 0,0,0,10:REM FILT/RES/VOL
1000 REM VOICE1 DATA
1010 DATA34,201,36,30,255,6,29,68,6,26,1
9,12,21,243,12,43,213,24
1020 DATA43,213,12,39,11,6,36,217,6,39,1
1,12,34,201,6,32,220,6,29,68,24,0,0,24
1030 DATA0,0,96
1040 DATA0,0,72,58,136,24,-1
1100 REM VOICE2 DATA
1110 DATA26,19,48,30,255,48
1120 DATA29,68,48,34,201,48
1130 DATA32,220,36,36,217,12,39,11,48
1140 DATA39,11,36,36,217,6,32,220,6,34,2
01,48,-1
```

```
1200 REM VOICE3 DATA
1210 DATA11,163,24,10,249,24,10,249,24,1
3,9,24
1220 DATA13,9,24,12,78,24,12,78,12,13,21
3,12,14,162,24
1230 DATA14,162,48,14,162,12,13,213,6,12
,78,6,13,9,24
```

SID TICKLES THE IVORIES

You can turn your 64 into a musical instrument with the next program. It produces a keyboard made up of the top row of keys on the 64 (←,1,2...to INST/DEL), each set to a different half tone, so that they represent all the notes on a piano, white and black. This fairly simple arrangement is capable of expansion in two ways, either by using more keys, say with the second row representing the white keys, and (some of) the top row the black, or by adding to the range of effects and parameters which can be specified. The first we'll leave to you, but we'll see how dynamic effects and filtering can be incorporated later. Here's the keyboard program:

```
5 REM *KEYBOARD*
10 DIM N(15,1):SID=54272:FOR C=0 TO 24:P
OKE SID+C,0:NEXT
20 FOR C=0 TO 15:READ FHI,FLO:N(C,0)=FLO
:N(C,1)=FHI:NEXT
30 PRINT"■■      KEYBOARD":PRINT"■ENTER
PARAMETERS:"
40 INPUT"VOLUME (0-15)";V:POKE SID+24,V
50 INPUT"ATTACK (0-15)";A
60 INPUT"DECAY (0-15)";D
70 POKE SID+5,16*A+D:REM A & D REGISTER
80 INPUT"SUSTAIN (0-15)";S
90 INPUT "RELEASE(0-15)";R
100 POKE SID+6,16*S+R:REM S & R REGISTER
110 PRINT"CHOOSE WAVEFORM: 1 TRIANGLE"
120 PRINT SPC(17)"2 SAWTOOTH":PRINT SPC(
17)"3 PULSE":PRINT SPC(17)"4 NOISE"
```

```
130 INPUT"ENTER 1,2,3 OR 4";WF
140 WF=WF+3
150 IF WF=6 THEN POKE SID+3,8:REM SQUARE
 WAVE
160 PRINT:PRINT"PRESS S TO END"
190 REM SCAN KEYBOARD
200 POKE 650,255:REM AUTOREPEAT ON KEYS
210 KB=PEEK(197):GET A$:IF KB=64 THEN 21
0
220 IF KB=57 THEN K=0:GOTO 250
221 IF KB=56 THEN K=1:GOTO 250
222 IF KB=59 THEN K=2:GOTO 250
223 IF KB=8 THEN K=3:GOTO 250
224 IF KB=11 THEN K=4:GOTO 250
225 IF KB=16 THEN K=5:GOTO 250
226 IF KB=19 THEN K=6:GOTO 250
227 IF KB=24 THEN K=7:GOTO 250
228 IF KB=27 THEN K=8:GOTO 250
229 IF KB=32 THEN K=9:GOTO 250
230 IF KB=35 THEN K=10:GOTO 250
231 IF KB=40 THEN K=11:GOTO 250
232 IF KB=43 THEN K=12:GOTO 250
233 IF KB=48 THEN K=13:GOTO 250
234 IF KB=51 THEN K=14:GOTO 250
235 IF KB=0 THEN K=15:GOTO 250
239 IF KB=13 THEN END :REM S FOR STOP
240 GOTO 200
250 POKE SID,N(K,1):POKE SID+1,N(K,0)
260 POKE SID+4,2↑WF+1
270 NK=PEEK(197):GET A$:IF NK=KB THEN 27
0
280 POKE SID+4,2↑WF:GOTO 200
990 REM NOTE FREQUENCY DATA
1000 DATA 109,17,108,18,142,19,175,20,24
3,21,54,23,156,24,19,26
1010 DATA 154,27,68,29,255,30,220,32,201
,34,217,36,11,39,95,41
```

The note values are held in an array **N(15.1), DIM**ensioned in line 10, into which the high and low byte frequency values are

READ in the loop in line 20. The next section sets the Volume, **ADSR** and Waveform for the notes. If the pulse waveform is selected then a square wave value is **POKE**d into Register 3.

Line 200 uses location 650 to enable autorepeat on all keys, and location 197 is used to scan the keyboard. Since we need to know not just when a key is pressed, but also when it is released (since we want to be able to play a note for as long as we want), we must continually check the key pressed. If we use **GET,** although it produces a continual sequence of characters after **POKE 650,255,** there is less time between each **GET**ting of a character than there is between each character entering, so a loop including **GET** will produce a null character when in fact the key is still pressed. Using **PEEK(197),** which holds the keyboard code number for the current key being pressed avoids this problem. The code numbers are not character codes, but the result of the way the 64 scans the keyboard, and if you want to find out what the code is for any key, just enter this:

```
10 PRINT PEEK(197):GOTO 10
```

and press the key as we said earlier, location 203 does the same thing. Note this code does not vary for **SHIFT**ed characters - it just indicates a specific key. We put a **GET A$** instruction following the **PEEK** to prevent the key pressed appearing on the screen. Each key has the note array subscript set when pressed, and the note values are **POKE**d in line 250 and the sound then gated. Line 270 checks whether the key is still pressed, passing control back to the keyboard loop when it is released.

SID THE SOUND DYNAMO

To achieve dynamic sound effects, we use the facility to change any of SID's parameters whilst the sound is playing, or introduce our own controls over the sound. An echo effect is produced by a simple set of loops, changing the volume by the calculated expression in line 60:

```
5 REM *ECHO*
10 SID=54272:FOR C=0TO24:POKESID+C,0:NEX
T
20 POKE SID+24,15:POKE SID+5,16*1+2:POKE
 SID+6,3*16
30 POKE SID+1,20
40 FOR T=1 TO 4:POKE SID+4,17
50 FOR C=1 TO 200:NEXT
60 POKE SID+4,16:POKE SID+24,15/(T*1.5)
70 FOR F=1TO30:NEXT
80 NEXT
90 GOTO20
```

You could also calculate the end value of the loop in line 50 according to the value of **T**. The effect produced can be almost infinitely varied by such manipulations of the loop values. The other sources for varying parameter values are Registers 27 and 28, where we can **PEEK** the changing values of the Voice 3 Oscillator and Envelope respectively. Here's a tremolo effect produced by scaling the value **PEEK**ed from the oscillator register to be in the range 0 - 10 and **POKE**ing it into the volume register (line 120):

```
10 REM*TREMOLO EFFECT **
20 SID=54272:FOR C=0 TO 24: POKE SID+C,0
:NEXT
30 POKE SID+5,16*1+1
40 POKE SID+6,16*10+1
50 POKE SID+18,17:REM TRIANGLE WF VOICE3
60 POKESID+14,90:REM VOICE 3 FREQ
70 FLO=209:FHI=18:REM SELECT VOICE 1 FRE
Q
80 POKE SID,FLO:POKE SID+1,FHI:REM SET F
REQUENCY
90 POKE SID+24,10
100 POKE SID+4,33
110 PRINT"■PRESS A KEY TO STOP"
120 DV=PEEK(SID+27)/26:POKE SID+24,5+DV
130 GET A$:IF A$=""THEN120
140 FOR C=0 TO 500:NEXT:POKE SID+4,0
```

The frequency of the triangular waveform of Voice 3 is set low (5 cycles per second or so) because otherwise **BASIC** is too slow to follow the waveform. Higher frequencies produce a different effect because only partial sampling of the waveform occurs. Try changing the values of the Voice 3 frequency and the waveform (remembering that if you choose pulse you also have to set the pulse width), or vary the scale factor. The total volume must always be in the range 1-15, however.

We can produce a phasing effect by reading the envelope value for Voice 3 from Register 28, and using this to affect the pulse width of the Voice 1 pulse waveform. Changing the attack rate alters the speed of the phasing:

```
10 REM *PHASING*
20 SID=54272:FOR C=0 TO 24: POKE SID+C,0
:NEXT
30 POKE SID+5,16*1+0:POKE SID+6,16*15+1:
REM VOICE 1 ADSR
40 POKE SID+19,16*9+9:REM VOICE 3 A,D
50 POKE SID+18,17:REM TRIANGLE WF VOICE3
60 POKESID+14,90:REM VOICE 3 FREQ
70 POKE SID,209:POKE SID+1,18:POKE SID+4
,64+1:POKE SID+3,8:REM V1 FREQ,PULSE WF
80 POKE SID+24,15+128:REM VOLUME,VOICE 3
 OUTPUT OFF
100 POKE SID+18,17:REM TRIANGLE WF VOICE
3
110 E%=PEEK(SID+28):POKE SID+3,E%/16
120 IF PEEK(SID+28)>16THEN 110
130 POKE SID+18,16:GOTO100
```

SID POKES

We've now seen some of the dynamic effects possible, but they've been presented as simple fixed sound demonstrations. To cover the filter and resonance features of the SID chip, we'll also use a program which allows a selected set of parameters to be changed whilst the sound plays. First the program:

```
5 REM *FILTER/RESONANCE DEMO*
10 SID=54272:FOR C=0 TO 24:POKE SID+C,0:
NEXT:POKE 650,255
20 VOL=15:A=0:D=0:S=15:R=0:REM VOLUME,AD
SR VALUES-EDIT TO CHANGE
30 PRINT"■RESONANCE/FILTER DEMO":PRINT
40 PRINT"CHOOSE WAVEFORM: 1 TRIANGLE"
50 PRINT SPC(17)"2 SAWTOOTH":PRINT SPC(1
7)"3 PULSE":PRINT SPC(17)"4 NOISE"
60 INPUT"ENTER 1,2,3 OR 4";WF:WF=WF+3
70 PW=5:POKE SID+3,8:REM PULSE WIDTH
80 PRINT"ENTER VOICE FREQUENCY:"
90 INPUT"HIGH BYTE";HF:INPUT"LOW BYTE";L
F:POKE SID,LF:POKE SID+1,HF
100 PRINT"ENTER FILTER FREQUENCY:"
110 INPUT"HIGH BYTE";FH
120 PRINT"LOW BITS NOT REQUIRED":FORC=0T
01000:NEXT
130 POKE SID+22,FH
140 R=0:F$="NONE":F=0:REM INITIALISE RES
ONANCE,FILTER
160 POKE SID+24,VOL
170 POKE SID+5,16*A+D:POKE SID+6,16*S+R:
REM SET ADSR
180 POKE SID+4,2↑WF+1:REM GATE SOUND
200 PRINT"■F1:RESONANCE  "R
210 PRINT"F3:FILTER TYPE "F$
220 PRINT"F5:FILT.HI.REG"FH
230 PRINT"F7:PULSE WIDTH"PW*10"%"
240 PRINT"F8:VOICE FREQ."INT((HF*256+LF)
*.058726)
250 PRINT:PRINT"HIT SPECIFIED FUNCTION K
EY TO CHANGE"
260 REM*SCAN KEYS*
270 GET K$:IF K$=""THEN270
280 IF K$="■" THEN GOSUB900:REM F1:RES
290 IF K$="■" THEN GOSUB500 :REM F3:FILT
ER TYPE
300 IF K$="■" THEN GOSUB400:REM F5:FILTE
R FREQUENCY
```

```
310 IF K$="■" THEN GOSUB800:REM F7:PULSE
 WIDTH
320 IF K$="■"THENGOSUB1000:REMF8:FREQ
330 GOTO 200
380 END
390 REM**********************
       FILTER FREQUENCY SUB
400 PRINT"■■"
405 PRINT"FILTER FREQ. HIGH REG:"FH"    "
410 PRINT"FILTER FREQUENCY(HZ):"FH*8*6"
     "
415 PRINT"+ AND - TO CHANGE,PLUS SHIFT F
OR FASTER RATE,RETURN FOR MENU"
420 GET K$:IF K$=""THEN420
430 IF K$="+"THEN FH=FH+1:IFFH=256THEN F
H=0
440 IF K$="-"THEN FH=FH-1:IFFH<0 THEN FH
=255
450 IF K$=CHR$(219)THEN FH=FH+8:IF FH>=2
56 THEN FH=0
460 IF K$=CHR$(221)THEN FH=FH-8:IF FH<0
THEN FH=255
470 IF K$=CHR$(13)THEN RETURN
480 POKE SID+22,FH:PRINT"■":GOTO 405
490 REM**************************
495 REM  FILTER TYPE
500 PRINT"■■"
505 PRINT"FILTER TYPE: "F$
510 PRINT"+ TO STEP THROUGH TYPES,RETURN
 FOR MENU"
520 GET K$:IFK$<>"+"AND K$<>CHR$(13)THEN
520
530 IF K$=CHR$(13)THEN RETURN
540 F=F+1:IF F=4 THEN F=0
550 ON F GOTO 570,580,590
560 F$="NONE      ":GOTO 600
570 F$="LOW PASS ":GOTO 610
580 F$="BAND PASS":GOTO 610
590 F$="HIGH PASS":GOTO 610
600 POKE SID+24,VOL:POKE SID+23,16*R:PRI
NT"■":GOTO 505
```

```
610 POKE SID+24,2↑(F+3)+VOL:POKE SID+23,
16*R+1:PRINT"■":GOTO 505
780 REM**********************
790 REM CHANGE PULSE WIDTH SUB
800 PRINT"■"
805 IF WF<>6 THEN PRINT"PULSE W/FORM NOT
 SELECTED":FORF=1TO5000:NEXT:RETURN
810 PRINT"■PULSE WIDTH(%):"STR$(PW*10)"
"
815 PRINT"+ AND - TO CHANGE,RETURN TO ME
NU"
820 GET K$:IF K$=""THEN 820
830 IF K$="+"THEN PW=PW+1:IFPW=9 THEN PW
=1
840 IF K$="-"THEN PW=PW-1:IFPW<1 THEN PW
=9
850 IF K$=CHR$(13)THEN RETURN
860 PR=PW*409.5:PHI=INT(PR/256):POKE SID
+3,PHI:PLO=PHI AND 255:POKE SID+2,PLO
870 GOTO 805
880 REM**********************
890 REM**********************
895 REM RESONANCE SUB
900 PRINT"■"
910 PRINT"■RESONANCE "STR$(R)" "
920 PRINT"+ AND - TO CHANGE,RETURN FOR M
ENU"
930 GET K$:IFK$=""THEN930
940 IF K$=CHR$(13) THEN RETURN
950 IF K$="+"THEN R=R+1:IF R=16 THEN R=0
960 IF K$="-"THEN R=R-1:IF R=-1 THENR=15
970 IF F<>0 THEN POKE SID+23,16*R+1:GOTO
910 :REM WITH FILTER
980 POKE SID+23,16*R:GOTO910:REM WITHOUT
 FILTER
990 REM**********************
995 REM NOTE FREQUENCY
1000 PRINT"■":REM CLR/HOME
1005 PRINT"NOTE FREQUENCY(HZ):"INT((256*
HF+LF)*.0587255)"          "
```

```
1010 PRINT"+ AND - CHANGE HI BYTE,@ AND
* CHANGE LO BYTE,RETURN FOR MENU"
1020 GET K$:IF K$=""THEN 1020
1030 IF K$="+"THEN HF=HF+1:IF HF=256 THE
N HF=0
1040 IF K$="@"THENLF=LF+1:IF LF=256 THEN
 LF=0:HF=HF+1
1050 IF K$="-"THEN HF=HF-1:IF HF=-1THEN
HF=255
1060 IF K$="*"THEN LF=LF-1:IF LF=-1 THEN
 LF=255:HF=HF-1
1070 IF K$=CHR$(13) THEN RETURN
1080 POKE SID,LF:POKE SID+1,HF:PRINT"■":
GOTO 1005
```

The volume and **ADSR** values are set in the program, and the user enters a choice of waveform, initial voice frequency (Voice 1), and a value for the high byte of the filter cut-off frequency (Register 22). The lower three bits of the filter frequency (Register 21) have minimal effect and are neglected. Resonance is initially set to zero, and after all the initial settings have been made, the screen display shows the current values for resonance, type of filter (initially none, as with all our previous programs), the high byte value of the filter frequency, the pulse width (in steps of 10%) and the voice frequency.

By pressing the appropriate function key a subroutine is called which enables the chosen parameter to be dynamically altered. The first of these (in the order in which they appear in the listing) is for the filter frequency. This is the frequency above (lowpass filter), around (bandpass filter) or below (highpass) which the filter starts to take effect. The high byte register value is displayed along with the actual frequency value this represents. Pressing the + and - keys increments or decrements the values by 1 (repeatedly if the key is held down) and if **SHIFT** is depressed along with either of these keys the filter frequency register value changes in steps of 8, to enable the user to hear the effect of sweeping the

frequency quickly up and down. Pressing **RETURN** passes the program back to the main menu display.

The next subroutine deals with the filter type. This is controlled by the setting of the bits 4-7 of Register 23, the low nybble of which controls the volume. The filter types are stepped through by pressing the + key, and there are two different operations for setting the filter type, and for a no-filter situation. To set a filter, we set the appropriate bit of Register 24, remembering to include the value for the volume, but we must also set the bit of Register 23 which passes the audio signal through the filter. This is bit 0 in this case (to switch the Voice 1 output through the filter), and so we **POKE** Register 23 with **16*R** (the resonance value) plus 1 (to set the filter). For all three filter choices this is done in line 610. Line 600 deals with switching off the filtering, and in order to get the sound to continue, we must not only switch out the filters by **POKE**ing Register 24 with the volume nybble, thus setting all filter bits to zero, but must also switch out the bit of Register 23 which selects filtering. If bit 1 (in the case of Voice 1) of Register 23 is set to 1, selecting filtering, whilst no filter bit(s) are selected in Register 24 the sound has no output path, and we get a resounding silence.

Note the way in which line 550 selects the appropriate line to **GOTO.** The filters are specified by a value 1-3, and no filtering by a value 0. The **ON. . .GOTO** of line 550 makes use of the fact that the value following **GOTO** must be an integer value of 1 or greater, and if it is 0, control passes to the next line, as is done here when the filter choice is 0, in which case line 560 is activated.

The pulse width subroutine should be comprehensible in terms of what we've said before. Note that the values are restricted to pulse width ratios of 10% to 90% in steps of ten, since, firstly, the effect of changes in pulse width of less than ten per cent is not very noticeable, and secondly there is little point in having values of 100% or 0%, since neither gives any audible output!

The resonance subroutine has only one noteworthy feature. As explained for the filter specification routine, we must differentiate between resonance with filtering, when (in line 970) we include the filter bit in the value **POKE**d to Register 23, and resonance when no filter is specified, when this bit is omitted (line 980).

For the note frequency subroutine, we scale the frequency register values for the display in line 1005, and use two extra keys to allow fine tuning of the frequency value if required by decrementing or incrementing the low byte values.

The program allows you to investigate the effect of various combinations of filter type, resonance and filter frequency with different note settings. Any nice effects you come up with can be included in sounds built into your own programs, since all values are either displayed or specified within the program. Sweeping the various parameters through different ranges can given you a lead into other dynamic effects.

To round off this chapter, our last program shows you how to start to build our simple keyboard program into a more complex synthesiser program, incorporating the type of technique used in the last program to allow further options for the sound. It demonstrates the last two facilities built into the SID chip - ring modulation and synchronisation. These are controlled by toggling bit 1 (synchronisation) and bit 2 (ring modulation) of the control register (Register 4). Voice 3 frequency is involved in both these processes, so we include a facility to change this. Note that ring modulation requires the triangle waveform to be selected for Voice 1, and since the facility to change waveforms and pulse width is included in the program, we must make sure that this waveform is selected when ring modulation is specified.

```
5 REM *KEYBOARD PLUS EFFECTS*
10 DIM N(15,1):SID=54272:FOR C=0 TO 24:P
OKE SID+C,0:NEXT
20 FOR C=0 TO 15:READ FHI,FLO:N(C,0)=FLO
:N(C,1)=FHI:NEXT:PRINT"■"
30 PRINT"ENHANCED KEYBOARD":PRINT"■ENTER
 INITIAL PARAMETERS:"
40 INPUT"VOLUME (0-15)";V:POKE SID+24,V+
2↑7:REM VOLUME,VOICE 3 OFF
50 INPUT"ATTACK (0-15)";A
60 INPUT"DECAY (0-15)";D
70 POKE SID+5,16*A+D:REM A & D REGISTER
80 INPUT"SUSTAIN (0-15)";S
90 INPUT "RELEASE(0-15)";R
100 POKE SID+6,16*S+R:REM S & R REGISTER
110 RM=0:PW=5:SY=0:S$="OFF":R$="OFF"
120 POKE SID+3,8:REM INITIAL PULSE WIDTH
130 WF=4:W$="TRIANGLE":W=2↑WF:PRINT"■"
140 HF=19:LF=142:POKE SID+14,LF:POKE SID
+15,HF:REM V3 FREQ.
150 PRINT"W/F ";W$;":R/M ";R$;":SYNC ";S
$
160 PRINT:PRINT"FUNCTION KEYS:":PRINT"F1
:TOGGLE SYNCHRONISATION"
170 PRINT"F3:TOGGLE RING MODULATION":PRI
NT"F5:CHANGE VOICE 3 FREQ."
180 PRINT"F7:CHANGE WAVEFORM":PRINT"↑:CH
ANGE PULSE WIDTH":PRINT:PRINT"S TO END"
190 REM SCAN KEYBOARD
200 POKE 650,255:REM AUTOREPEAT ON KEYS
210 KB=PEEK(197):GET A$:IF KB=64 THEN 21
0
220 IF KB=57 THEN K=0:GOTO 250
221 IF KB=56 THEN K=1:GOTO 250
222 IF KB=59 THEN K=2:GOTO 250
223 IF KB=8 THEN K=3:GOTO 250
224 IF KB=11 THEN K=4:GOTO 250
225 IF KB=16 THEN K=5:GOTO 250
226 IF KB=19 THEN K=6:GOTO 250
227 IF KB=24 THEN K=7:GOTO 250
228 IF KB=27 THEN K=8:GOTO 250
```

```
229 IF KB=32 THEN K=9:GOTO 250
230 IF KB=35 THEN K=10:GOTO 250
231 IF KB=40 THEN K=11:GOTO 250
232 IF KB=43 THEN K=12:GOTO 250
233 IF KB=48 THEN K=13:GOTO 250
234 IF KB=51 THEN K=14:GOTO 250
235 IF KB=0 THEN K=15:GOTO 250
239 IF KB=13 THEN END :REM S FOR STOP
240 IF KB=3 THEN GOSUB 1800:GOTO 150:REM
 F7 W/F
241 IF KB=5 THEN GOSUB 1200:GOTO 150:REM
 F3 RING MODULATION
242 IF KB=4 THEN GOSUB 1500:GOTO 150:REM
 F1 SYNCH
243 IF KB=6 THEN GOSUB 2000:GOTO 150:REM
 V3 FREQ
244 IF KB=54 THEN GOSUB 3000:GOTO 150:RE
M PULSE
250 POKE SID,N(K,1):POKE SID+1,N(K,0)
260 POKE SID+4,W+1:REM GATE SOUND
270 NK=PEEK(197):GET A$:IF NK=KB THEN 27
0
280 POKE SID+4,W:GOTO 210
990 REM NOTE FREQUENCY DATA
1000 DATA 109,17,108,18,142,19,175,20,24
3,21,54,23,156,24,19,26
1010 DATA 154,27,68,29,255,30,220,32,201
,34,217,36,11,39,95,41
1190 REM********************
1195 REM**RING MODULATION**
1200 IF RM=0 THEN RM=1:R$="ON ":WF=4:W=2
↑WF+SY*2↑1+2↑2:W$="TRIANGLE":GOTO 1220
1210 IF RM=1 THEN RM=0:R$="OFF":W=W-4
1220 PRINT"␣":RETURN
1490 REM************************
1495 REM**SYNCHRONISATION    **
1500 IF SY=0 THEN SY=1:S$="ON ":W=2↑WF+R
M*2↑2+2↑1:GOTO 1520
1510 IF SY=1 THEN SY=0:S$="OFF":W=2↑WF+R
M*2↑2
1520 PRINT"␣":RETURN
```

```
1790 REM*******************
1795 REM**WAVEFORM CHANGE**
1800 PRINT"■CHANGE WAVEFORM:":PRINT"CURR
ENT WAVEFORM IS "W$
1810 PRINT"PRESS T FOR TRIANGLE":PRINT"P
RESS S FOR SAWTOOTH"
1820 PRINT"PRESS P FOR PULSE":PRINT:PRIN
T"RETURN FOR MENU"
1830 GET A$:IF A$="T"THEN WF=4:W$="TRIAN
GLE":GOTO 1880
1840 IF A$="S"THEN WF=5:W$="SAWTOOTH":GO
TO 1880
1850 IF A$="P"THEN WF=6:W$="PULSE    ":GO
TO 1880
1860 IF A$=CHR$(13) THEN PRINT"■":RETURN
1870 GOTO 1830
1880 IF WF=4 THEN W=2↑WF+RM*2↑2+SY*2↑1:P
RINT"■":RETURN
1890 IF WF<>4 THEN RM=0:R$="OFF":W=2↑WF+
SY*2↑1:PRINT"■":RETURN
1990 REM**********************
1995 REM*VOICE 3 OSCILLATOR **
        * FREQUENCY CHANGE   **
2000 PRINT"■VOICE 3 FREQUENCY(HZ):"INT((
256*HF+LF)*.0587255)"          "
2010 PRINT"Z AND X CHANGE HI BYTE,A AND
S CHANGE LO BYTE,RETURN FOR MENU"
2020 GET K$:IF K$=""THEN 2020
2030 IF K$="X"THEN HF=HF+1:IF HF=256 THE
N HF=0
2040 IF K$="S"THENLF=LF+1:IF LF=256 THEN
 LF=0:HF=HF+1
2050 IF K$="Z"THEN HF=HF-1:IF HF=-1THEN
HF=255
2060 IF K$="A"THEN LF=LF-1:IF LF=-1 THEN
 LF=255:HF=HF-1
2070 IF K$=CHR$(13) THEN PRINT"■":RETURN
2080 POKE SID+14,LF:POKE SID+15,HF:PRINT
"■":GOTO 2000
```

```
2990 REM**********************
2995 REM CHANGE PULSE WIDTH SUB
3000 PRINT"■"
3010 IFWF<>6THENPRINT"CURRENT W/F NOT PU
LSE":FORF=1TO2000:NEXT:PRINT"■":RETURN
3020 PRINT"■PULSE WIDTH(%):"STR$(PW*10)"
 "
3030 PRINT"Z AND X TO CHANGE,RETURN TO M
ENU"
3040 GET K$:IF K$=""THEN 3040
3050 IF K$="X"THEN PW=PW+1:IFPW=9 THEN P
W=1
3060 IF K$="Z"THEN PW=PW-1:IFPW<1 THEN P
W=9
3070 IF K$=CHR$(13)THEN PRINT"■":RETURN
3080 PR=PW*409.5:PHI=INT(PR/256):POKE SI
D+3,PHI:PLO=PHI AND 255:POKE SID+2,PLO
3090 GOTO 3020
```

CHAPTER EIGHT
GRAPHICS ON THE 64

DEPRESSION AT THE KEYBOARD

Before we have an extensive look at graphics on the 64, it is worth taking an extended look at the keyboard of your 64. It is all too easy to think of the keyboard as nothing more than an electronic typewriter and thus miss out on some of the most accessible features of your machine. For instance, the Commodore range of computers have just about the easiest method of changing program lines available on home micros, using the excellent full-screen editor.

The main keyboard contains 62 full-travel keys, spaced at the normal typewriter distance, plus a group of four function keys to the right. The main keyboard does, of course, have the usual range of alpha and numerical keys plus a number of extra keys that have special uses. As we explained in the first chapter, most of the keys can be used in different combinations, let's have a quick recap of the keyboard to see what can be achieved.

The **RUN/STOP** key, when used by itself, will stop a **BASIC** program that is running and give you control of the computer again. When the **RUN/STOP** key is pressed at the same time as the **RESTORE** key then the computer will perform what is known as a warm start. Pressing both of these keys will, more often than not, stop your program and reset the screen to the normal colours whilst leaving your program in memory. Also, the values of all variables will still be in memory so that you

can examine them, when necessary. If pressing both of these keys does not stop the program then your program has probably crashed and you will have to turn the computer off and then back on again. Doing this will, of course, lose the program that was in the memory so you should always **SAVE** your program before **RUN**ning it, just in case it crashes!

The **RUN/STOP** key has one further function, when used in conjunction with either of the shift keys. Using this combination tells the computer to **LOAD** the first program that it finds on the cassette recorder and then to **RUN** that program. This is the built-in auto-run facility that you will find is used extensively by commercial software programmers and which you can use for your own programs without any special preparation.

Generally speaking, the **SHIFT** keys perform the same function as they do on a typewriter or the **SHIFT** key can be used in conjunction with the **CBM** logo key (bottom left corner) to switch between the two character sets that are available. When the computer is first switched on the so called **UPPER CASE/GRAPHICS** character set is selected. Also, whenever you use the **RUN/STOP** and **RESTORE** combination the computer resets to this character set. In this mode typing normally on the keyboard will give you the characters that are featured on the top of the keys, including uppercase letters. When you press the **SHIFT** key and any other key then you will get the graphics shape that is shown on the front of the key on the right side. Similarly, when you press the **CBM** key in conjunction with any other then you will get the graphics character that is shown on the left of the key. Why not try a few keys to see just how easy it is to generate the different characters onto the screen?

If you depress the **CBM** key and the **SHIFT** key simultaneously then the computer will switch to the **LOWER/ UPPER CASE** character set. As you will see if you try it, normal typing will now produce lower-case letters and **SHIFT**ed letters appear on the screen as **UPPER** case.

Using the **CBM** key still allows you to use the graphics on the left of the keys but you can no longer use the graphics on the right of the keys. You should also notice that this is an all or nothing process. It is not possible to display the two character sets on the screen simultaneously. Repeated presses of the **CBM/SHIFT** combination will switch (toggle) between the two character sets.

While on the topic of character sets, you can also produce any of these characters in reverse video. Pressing the **CTRL** key and the **RVS ON** keys together will allow you to produce the same characters, but with reversed colours. Pressing the **CTRL** and **RVS OFF** keys together will return the computer to the normal mode again. Notice that it is possible to mix normal and reversed characters on the screen, although you are limited to one or other of the character sets.

The printing colour can be changed very simply. If you press the **CTRL** key and one of the keys 1 to 8 then all text printed after that will appear in the colour shown on the front of those keys. A further eight colours can be obtained by using the **CBM** key in conjunction with the keys 1 to 8, although these colours are not shown on the keys themselves. They are:

KEYS	COLOUR
CBM + 1	ORANGE
+ 2	BROWN
+ 3	LIGHT RED
+ 4	LIGHT GREY
+ 5	MEDIUM GREY
+ 6	LIGHT GREEN
+ 7	LIGHT BLUE
+ 8	DARK GREY

There are also four cursor control keys. At the bottom right corner of the keyboard are two keys that allow you, between them, to move the cursor around the screen. When used

unshifted, you can move the cursor either down the screen or to the right, whilst using these two keys with the **SHIFT** key allows movement either upwards or to the left.

The final two keys on the main keyboard are at the top right, the **CLR/HOME** key and the **INST/DEL** key. Each of these keys has two functions, depending on whether they are shifted or unshifted. When you press the **CRL/HOME** key the cursor will move to its **HOME** position, the top left corner of the screen. Pressing the same key with the shift key will **CL**ea**R** the screen and move the cursor to its home position. The final key, the **INST/DEL** key is used almost exclusively for editing program lines. Pressing this key by itself will **DEL**ete the character that is under the cursor and move all of the other characters on the line to the left, to fill the gap left by the deleted character. On the other hand, pressing the **SHIFT** and **INST/DEL** keys together will **INS**er**T** a space into the line and move all of the characters one space to the right. You can then type any new characters that you require.

The final keys, to the right of the main keyboard, are the function keys. These keys have no use when in the direct mode, but you are able to use them within programs, as we shall see later.

Although you may not have realised it, there are two modes of action when you use the keyboard, the direct mode and the program mode. When you use the direct mode then your actions at the keyboard are immediately displayed upon the screen. Getting away from the fancy terms, you are using the computer just like a typewriter. As you type, so it is displayed upon the screen. This applies equally to the cursor control keys, for instance, because when you press a cursor key the cursor actually moves in the required direction around the screen.

The deferred, or program mode, is different only in the sense that you are now instructing the computer to do something that it will only perform after you press the **RETURN** key (to

show that you have finished instructing!). For instance, if you type:

PRINT "HELLO"

then, when you press the **RETURN** key the computer will print HELLO. This is, of course, fairly obvious but the main point is that any of the features mentioned earlier can be used in the deferred mode, or even within a program. This includes the cursor movements, reverse on/off, colour changes and so on. Whenever you use any of these keys within quote marks then the action will not take place but instead a symbol will be printed into the statement. For instance, if you type:

PRINT "CUR.DOWN"

where, of course, you don't type CUR.DOWN but use the cursor down key. Instead of the cursor moving down one line, you will see that a reversed Q appears. When you press the **RETURN** key the cursor down instruction will be carried out. Similarly, colour changes can be incorporated within **PRINT** statement as can reversed characters and any of the graphics characters. Combinations of these control codes and the more normal printable characters can be used very effectively to format an attractive screen display within your programs. By using the graphics characters it is simple to draw rudimentary pictures, but I'll leave it as an exercise for you to do that.

As well as using these control codes within the print statements, you can also use the equivalent character code number. For instance, the following two statements are equivalent and have the same effect:

PRINT "<SHIFT AND CLR/HOME>"
PRINT CHR$(147)

As the second statement has the same effect as the first, you may be wondering why you should ever want to use the second. Well, the main reason concerns not the computer itself but the use of a printer, especially a non-Commodore printer. Many printers are unable to reproduce the Commodore control code symbols, and so, when this is the case, you may have to convert your program listing so that it uses the **CHR$(X)** statement rather than the control code symbol within quotation marks. This process is reasonably tedious and results in a longer program so should be avoided whenever possible. Commodore printers are able to print the entire **CBM** character set and conversion would then be unnecessary.

A second reason for using **CHR$** codes instead of control code symbols is that there are a few instructions contained within the characters set that cannot easily be represented by a symbol. The codes 0 to 26 can be obtained (although there is no mention of the fact in the *Users Guide*), by using the **CTRL** key in conjunction with one of the keys @ to Z. You must, of course do this within quotation marks and then the symbol printed to the screen will be a reversed character, @=0, A=1, and so on, to Z=26. As you can see from the list of **ASCII** codes in Appendix 7, not all of the codes from 0 to 26 are actually used. In fact, only the codes 8, 9 and 14 are of particular interest to the programmer. Character code 8 disables the ability to switch between character sets, preventing the use of the **CBM/SHIFT** combination. To use, for example, type:

```
100 PRINT "<CTRL>+H"
```

or

```
100 PRINT CHR$(8)
```

Similarly, **CHR$**(9) enables the ability to switch between character sets and is obtained by pressing **CTRL** and I

together. It should be remembered that these two controls only control the ability to switch, they don't actually change the character set at all. To do that you must use **CHR$**(14), or **CTRL**+N, to switch to the lower case/upper case set or **CHR$**(142) to switch to the upper case/graphics set. There is no symbol for code 142, you must use the form **CHR$**(142).

AN OVERVIEW

The powerful graphics capabilities of the Commodore 64 comes exclusively from the Video Interface Chip, otherwise known as the **VIC-II** chip. Incorporating this chip into the hardware of the machine enables the support of a wide variety of graphics capabilities which we will look at in the next three chapters. In this chapter we concentrate on the various character modes:

1 **Standard Character Mode**
2 **Multi-coloured character mode**
3 **Extended background colour mode**

In the following chapter we take a look at the High Resolution modes:

1 **Standard Bit-Mapped mode**
2 **Multi-colour Bit-Mapped mode**

The final chapter in this section on graphics takes an in-depth look at the sprite capabilities of the 64:

1 **Standard Series**
2 **Multi-colour Sprites**

The **VIC-II** chip supports all the graphics on the machine and an understanding of these capabilities goes hand-in-hand with an understanding of the chip itself, so let's start by taking a look at the chip.

The **VIC-II** chip can access ("see") 16K of memory at a time. Since there is 64K of memory you want the **VIC-II** chip to be able to access the whole area. This is possible by having four banks, each of 16K. The bank selects bits (at location 56576) will allow you to select any one bank at a time, thus covering the whole 64K. Before changing banks, however, you must make sure that the two lowest bits of location 56578 are set to 1 to activate the bank switching process. The following statements will allow you to switch to **BANK** A.

```
POKE 56578,PEEK (56578) OR 3
POKE 56576, (PEEK 56576 AND 252) OR A
```

For the four values of A the **VIC-II** chip will access the **BANKS** starting at locations:

VALUE OF A	BANK	STARTING LOCATION	END LOCATION
0	3	49152	65535
1	2	32768	49151
2	1	16384	32767
3	0	0 (DEFAULT VALUE)	16383

Within a 16K block the **VIC** chip divides up the area for the following purposes:

Standard Screen	**1024 bytes (1K)**
Bit-map Screen	**8192 bytes (8K)**
Character definitions up to	**4096 bytes (4K)**
Sprite definitions	

Sprite definitions can take up the unused portion of the 16K

block. So, if the high resolution screen is not being used that area can be used for sprite definitions.

The standard text screen can be placed at any 1K block within the 16K **BANK**. In theory, at least, that means that the text screen can be placed anywhere within the 64K memory area. However, in practice, many locations are not available for **BASIC** programs. For instance, the **BASIC ROM**, from 40960 to 49151, cannot be used by the text screen.

Changing the location of the text screen is achieved by changing the upper four bits of location 53272:

POKE 53272,(PEEK(53272)AND 15) OR A

where A has one of the following values:

VALUE OF A	LOCATION	
0	0	
16	1024	(DEFAULT)
32	2048	
48	3072	
64	4096	
80	5120	
96	6144	
112	7168	
128	8192	
144	9216	
160	10240	
176	11264	
192	12288	

208	**13312**
224	**14336**
240	**15360**

Remember that the **BANK** start address must be added to the screen address to find the actual location of the screen. When moving the screen you must also tell the screen editor routines where the screen is. This is done by **POKE** 648,PAGE where PAGE is calculated by PAGE=ADDRESS/256.

The standard text screen is used to print characters, either **ROM** characters or user-definable characters. There is a second screen area associated with the text screen that is used to store the colour information of each character square. The **COLOUR** memory is located at 55296-56319. Only the first 1000 locations of this area are actually used and only the lowest four bits of each byte are used to store the colour data, which gives you the possibility of storing one of 16 colours per byte of colour memory.

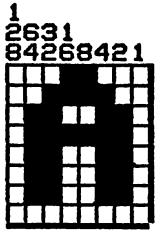
There is a direct relationship between the locations of the text screen and those of the colour memory. The first byte of colour memory stores the colour for the first byte of the text screen, the second byte stores the colour for the second byte of screen memory, and so on. The colour memory cannot be moved around within memory, but it is used in different ways by different modes. A graphic display created in one mode will look completely different when displayed in another mode.

STANDARD CHARACTER MODE

The Standard Character Mode is the mode that the computer is in when it is first turned on. It is this mode that you will generally write programs in.

The normal text screen is made up of 25 rows, each of 40 characters, a total of 1000 locations. Each location is, in fact, used to store a pointer into the character memory area. Normally this character area is in **ROM**to give you the standard character set, although this can be changed to **RAM** locations so that you would then be able to alter any of the shapes. Before looking at programmable characters, let's first of all see how a character is put onto the TV screen.

The standard character set comprises the definitions for 256 letters, numbers and various graphics shapes. This character set is built into **ROM** (Read Only Memory) normally located at location 53248 onwards for 4K. Each character on the screen is made up of a grid of 8*8 pixels, a total of 64 pixels per character.



Within the character memory area the definitions for each character are laid out in blocks of eight bytes. The first block contains the eight bytes that define the @ symbol, the next block defines the A, and so on. The screen **POKE** codes (see Appendix) show the pointer numbers for every character that can be displayed on the screen. To find the start address of any character use the following formula:

CHAR ADDRESS = 53248 + (SCREEN CODE * 8)

This equation, of course, will only be true for the standard character **ROM**, which starts at 53248. For other character locations the general formula should be used:

CHAR ADDRESS = START ADDRESS+(SCREEN CODE * 8)

The location where the **VIC-II** chip looks for the character data can be altered by:

POKE 53272,(PEEK (53272)AND 240) OR A

where A can have one of the following values:

VALUE OF	LOCATION OF CHARACTER MEMORY
0	0
2	2048
4	4096 (DEFAULT)
6	6144
8	8192
10	10240
12	12288
14	14336

To the addresses shown above, you should add the start address of the **BANK** to get the final location of the start of character memory. If you type in the following line then the **VIC-II** chip will get its character data from location 2048 onwards:

POKE 53272,(PEEK(53272)AND 240) OR 2

After you have pressed <**RETURN**> you will see that all of the characters on the screen turn to garbage.
Typing letters from the keyboard also results in nothing but garbage coming up on the screen. What is happening is quite simple. Although we have told the **VIC** chip to get its character information from the 2K block starting at 2048, there is not any character data there! Location 2048 is the start of **BASIC** program area and the **VIC** chip is trying to interpret any program that may be there as character data. (Pressing the **RUN/STOP** and **RESTORE** keys together will return you to the normal character set.)

PROGRAMMABLE CHARACTERS

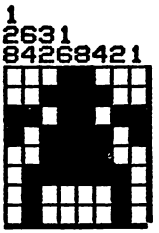
The solution to the problem shown above is to put character data into the area of memory that the **VIC-II** chip is looking for, since the area that we used is the start of **BASIC** programming area we cannot use this area any more since it would corrupt the program itself. From now on we shall use the area starting from location 14336 onwards.

The first thing that must be done is to transfer the character data that is stored in the character **ROM**. Having done this, any character can then be altered by **POKE**ing new data into the correct locations.

The following program will put the data for 256 characters into **RAM** memory starting at location 14336:-

```
10  POKE 56334,PEEK(56334) AND 254
20  POKE 1,PEEK(1) AND 251
30  FOR CHAR = 0 TO 256*8
40  POKE 14336+CHAR,PEEK(53248+CHAR)
50  NEXT CHAR
60  POKE 1,PEEK(1) OR 4
70  POKE 56334,PEEK(56334) OR 1
80  POKE 53272,(PEEK(53272)AND240)OR 14
```

The program will take about one minute to transfer the whole character set into **RAM**. When it ends you will not see any change to the characters themselves but the information is now stored in a **RAM** area and can now be altered to suit our needs. Suppose, for instance, that you wanted to redefine the "A" to be a space invader. The first thing to do is to design the shape on the grid 8*8 pixels.



The next thing to do is to work out the data numbers that define the shape. Each row of eight pixels on the grid is represented by one byte of data. One byte is the same as eight bits, so each bit represents one pixel of the shape. Adding up the values of each bit that is on will give you the eight bytes of data that define the shape. Once you have the data then all that is necessary is to **POKE** the numbers into the character memory:

```
100 FOR CHAR = 0 TO 7
110 READ X
120 POKE 14336+(8*1)+CHAR,X
130 NEXT CHAR
140 DATA 24,60,219,189,60,126,66,66
```

When you add these lines to the program above and then **RUN** it, you will see that whenever you print an A on the

screen what appears is the space invader shape. You have your first user-defined character!

Up to 256 characters can be defined in this way to give you a wide variety of shapes for the games programs that you want to write. However, the drawing of so many shapes can be very long winded if you are using a pencil and paper. The next program allows you to draw character shapes on the screen. When you are happy with the shape, press function key 5 and the computer will calculate the data and display it on the screen. You will also be given a choice of the character that you wish to define from the selection shown at the top of the screen. Having made the choice, your character shape will be shown full size at the top of the screen. When you are happy with the shape simply write down the data and you are ready to use that shape in your own programs.

```
5 REM ************************
10 REM*   CHARACTER DEFINER   *
15 REM************************
16 IFPEEK(52)=48THENPOKE53272,(PEEK(5327
2)AND240)+12:GOTO20
17 POKE52,48:POKE56,48:CLR:GOSUB7000
20 AT$=CHR$(17):FORT=1TO5:AT$=AT$+AT$:NE
XTT:AT$=CHR$(19)+AT$
25 DIMCHAR(8,8)
100 GOSUB 1000
110 GOSUB1100
1000 REM ************************
1002 REM *   DRAW THE GRID     *
1004 REM ************************
1010 PRINTCHR$(147)"   C H A R A C T E R
   D E F I N E R"
1015 PRINT"■       ■ # $ % & ' ( ) + , -
 . / "
1017 PRINT"■        # $ % & ' ( ) + , -
. / "
1020 PL$="┌┬┬┬┬┬┬┬┐ "
1030 FORT=0TO7:PRINTLEFT$(AT$,8+T)SPC(13
)T+1;PL$:NEXTT
```

```
1040 PRINTLEFT$(AT$,16)SPC(16)"━━━━━━━━"
1050 PRINTLEFT$(AT$,17)SPC(16)"76543210"
1052 PRINTLEFT$(AT$,7)SPC(25)"    CONTRO
LS"
1055 PRINTLEFT$(AT$,8)SPC(25)"F1=DRAW/UN
DRAW"
1060 PRINTLEFT$(AT$,9)SPC(25)"F3=UPDATE
DATA"
1065 PRINTLEFT$(AT$,10)SPC(25)"F5=CREATE
"
1070 PRINTLEFT$(AT$,11)SPC(25)"    CHARA
CTER"
1075 PRINTLEFT$(AT$,12)SPC(25)"F7=QUIT "
1080 PRINTLEFT$(AT$,14)SPC(25)"CURSOR KE
YS TO"
1085 PRINTLEFT$(AT$,15)SPC(25)" MOVE "
1095 GOSUB1200
1097 RETURN
1099 :
1100 REM ***********************
1102 REM *  MOVE THE CURSOR    *
1104 REM ***********************
1110 Y=8:X=16:CH$=CHR$(111)
1150 PRINTLEFT$(AT$,Y)SPC(X)"*"CHR$(157)
;:POKE204,0
1155 BIT((Y-7),(23-X))=FL
1160 GETA$:IFA$=""THEN1160
1165 POKE204,1
1170 IFA$="■"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:Y=Y-(Y<15)
1175 IFA$="□"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:Y=Y+(Y>8)
1180 IFA$="▌"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:X=X-(X<23)
1185 IFA$="▐"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:X=X+(X>16)
1190 IF FL=0THEN IFA$=CHR$(133)THENCH$="
■":FL=1:GOTO1197
1192 IF FL=1THEN IFA$=CHR$(133)THENCH$=C
HR$(111):FL=0:GOTO1197
```

```
1194 IFA$=CHR$(134)THENGOSUB1200
1195 IFA$=CHR$(135)THENGOSUB1300
1196 IFA$=CHR$(136)THENPRINTCHR$(19);:EN
D
1197 IFA$=CHR$(140) THEN RUN 17
1199 GOTO1150
1200 REM ***********************
1202 REM *     UPDATE DATA      *
1204 REM ***********************
1205 PRINTLEFT$(AT$,7)SPC(8)"DATA"
1210 FORT=1TO8
1215 DTA(T)=0
1220 FOR BIT=0TO7
1230 DTA(T)=DTA(T)+(2↑BIT)*BIT(T,BIT)
1240 NEXT BIT
1250 PRINTLEFT$(AT$,7+T)SPC(8)DTA(T)
1260 NEXTT
1270 RETURN
1300 REM ***********************
1302 REM *     CREATE A NEW     *
1304 REM *     CHARACTER        *
1306 REM ***********************
1310 GOSUB1200
1320 PRINTLEFT$(AT$,20)SPC(6)" ENTER THE
 CHARACTER YOU"
1330 PRINTLEFT$(AT$,22)SPC(6)"   WISH TO
 CHANGE:-";
1340 INPUT"      ";Q$:IFQ$=CHR$(13)
THEN1320
1342 IFQ$<CHR$(35) OR Q$>CHR$(47) THEN 1
320
1350 Q=ASC(Q$)
1355 FORT=1TO8:POKE12287+(Q*8)+T,DTA(T):
NEXTT
1399 RETURN
7000 REM ***********************
7002 REM *  MOVE THE ENTIRE     *
7004 REM *    CHARACTER SET     *
7006 REM *       INTO RAM       *
7008 REM ***********************
```

```
7010 PRINTCHR$(147);"QQQQ      RELOCATING
 THE CHARACTER SET"
7014 PRINTCHR$(17)"            PLEASE WAI
T"
7020 POKE56334,PEEK(56334)AND254
7030 POKE1,PEEK(1)AND251
7040 FORI=0TO2047:POKE12288+I,PEEK(53248
+I):NEXTI
7050 POKE1,PEEK(1)OR4:POKE56334,PEEK(563
34)OR1
7060 POKE53272,(PEEK(53272)AND240)+12
7070 RETURN
```

MULTI-COLOUR GRAPHICS MODE

Standard high-resolution graphics mode allow you to control very small dots (or pixels) on the screen. Each dot in character memory can have two possible values, either 0 or 1. When a dot is off, set to 0, then that pixel will be displayed in the screen colour. If a dot is on, set to 1, then that dot will be displayed in the character colour that has been chosen for that character square. When you are using the standard high-resolution graphics mode, all the dots within each 8*8 character square can either be displayed in the background colour or the foreground colour. In other words, each square can only contain two colours. This can be very restrictive, when drawing pictures for instance, if you want two different coloured lines to cross. In the standard high-resolution mode it is virtually impossible. Fortunately, the multi-coloured graphics mode provides a solution to this problem.

In the multi-colour character mode each dot can be one of four colours, the screen (background register 0) colour, the colour in background register 1, the colour in background register 2, or the character colour. The only sacrifice that must be made is in the horizontal resolution, because each multi-colour mode pixel is twice as wide as the standard, high-resolution pixel.

To turn on multi-colour character mode it is necessary to set bit 4 of the **VIC-II**control register at 53270 to a 1. The following **POKE** will do this:

POKE 53270,PEEK(53270) OR 16

To turn off the multi-colour mode, set the same bit of location 53270 to 0:

POKE 53270,PEEK(53270) AND 239

The **POKE**s shown above should really be thought of as enabling, or disabling, the multi-colour mode. If this mode is enabled then it is possible to set multi-colour mode on or off for each individual character square on the screen. Whether a square is in high-resolution mode or multi-colour mode is determined by bit 3 in colour memory. If you remember, the 1000 bytes of colour memory contain colour data for the character colour of the corresponding bytes of the text screen. In the high-resolution mode it is possible to have up to 16 different character colours on the screen at the same time, although each individual square can only contain one character colour.

In the multi-colour mode you can only have a maximum of 8 character colours on the screen. Choosing colours 0 to 7 will set the chosen squares in the high-resolution mode with the character colour of 0 to 7 (the colours on the front of the number keys). If you try to select any of the colours 8 to 15 then you will set the character in the multi-colour mode with a character colour of 8 less than the colour number chosen.

Once the multi-colour mode is set for a particular square, the bit-pattern of the character determines which colours are shown for the dots. The letter A, for instance, has a bit-pattern thus:

```
  **          00011000
 ****         00111100
**  **        01100110
******        01111110
**  **        01100110
**  **        01100110
**  **        01100110
              00000000
```

In the normal, high-resolution mode the screen colour is displayed wherever there is a 0 and the character colour is displayed everywhere there is a 1. In order to get a range of four colours from the bit-patterns, multi-colour mode interprets the bits in pairs. So, again looking at the letter A and grouping the bits into pairs:

```
  **          00 01 10 00
 ****         00 11 11 00
**  **        01 10 01 10
******        01 11 11 10
**  **        01 10 01 10
**  **        01 10 01 10
**  **        01 10 01 10
              00 00 00 00
```

The colours shown for pixel-pair are determined by the bit-pair of the character according to the following chart:

BIT PAIR	COLOUR REGISTER	LOCATION
00	BACKGROUND 0	53281
01	BACKGROUND 1	53282
10	BACKGROUND 2	53283
11	COLOUR MEMORY LOWER 3 BITS	COLOUR RAM

In practice, the multi-colour mode has very limited usefulness when used with the standard character set. Standard characters become almost unreadable in the multi-colour mode although, with the ability to mix high-resolution characters with multi-coloured characters it is possible to design your own characters, for games playing for example, whilst still using the standard letters and numbers to display your score at the top of the screen.

EXTENDED BACKGROUND COLOUR MODE

The Extended Background Colour Mode gives you control over the background colour of each character square as well as the normal control that you have over the character colour. This mode is really quite a versatile one and far easier to use than the multi-colour mode discussed earlier. With extended background colour mode you can put characters on the screen in any one of sixteen character colours and any one of four background colours. You can either use the standard character set or your own user-defined set. There is a limitation, however, upon the number of characters that can be used.

Colour memory is used to hold the character colour in extended background mode. It is used in exactly the same way as in the standard, high-resolution mode. Screen memory is used differently to the standard mode. The two highest bits of each character square are used to store the background colour information, whilst only the lower 6 bits are used for character data. Six bits will only hold a number between 0 and 63, and this is the limit to the number of characters that can be displayed. **POKE**ing numbers greater than 63 to the screen area will give you the same character but with a different background colour.

The background colour is determined by the following chart:

CHARACTER CODE RANGE	BIT 7	BIT 6	BACKGROUND COLOUR ADDRESS
0-63	0	0	53281
64-127	0	1	53282
128-191	1	0	53283
192-255	1	1	53284

Extended colour mode is turned on by setting bit 6 of the **VIC-II** register at location 53265 to a 1, and it is turned off by setting the same bit to 0. The following **POKE** will turn Extended Background Colour Mode on:

POKE 53265,PEEK(53265) OR 64

and to turn this mode off again use:

POKE 53265,PEEK(53265) AND 191

Also, pressing the **RUN/STOP** and **RESTORE** keys simultaneously will turn extended colour mode off.

The following short program should show the versatility of this mode:

```
10 POKE 53281,2:REM RED
20 POKE 53282,4:REM PURPLE
30 POKE 53283,6:REM BLUE
40 POKE 53284,13:REM LIGHT GREEN
50 POKE 53265,PEEK(53265) OR 64
READY.
```

When you **RUN** this routine you are simply setting up the background registers and then turning on the extended colour mode. You may notice that some of the characters on the screen change their background colour but otherwise nothing much will happen.

Now that the computer is in the extended colour mode you can see how simple it is to select each of the four background colours. The first colour is the normal background colour, in this case red. Typing on the keyboard gives you this colour. The second background colour is obtained by typing reversed characters. Normally, of course, using reversed characters (by pressing the **CTRL** and 9 keys simultaneously) will give you just that, reversed characters. When extended colour mode is on, however you get the same character but with the second background colour. Similarly, typing **SHIFT**ed characters will give you the same character with the third background colour. The fourth background colour is obtained by using reversed, **SHIFT**ed characters.

SCREEN BLANKING

The **VIC-II** chip supports the ability to turn off the screen display. This, in fact, is exactly what happens whenever the cassette recorder is in use and the screen goes blank. Under most circumstances this feature is not particularly useful, but you may find an occasion within your own programs when you may want to hide what is on the screen from prying eyes. You may, for instance, want to build up a colourful screen display before showing it to the world. On a more serious level, users of the 1540 disc drive will find that turning off the screen will allow full use of the disc drive system. With the screen display on there is a slight discrepancy between the data transmission speeds of the computer and the disc drive and occasional hang-ups and errors can occur. (This problem does not occur with the 1541 disc unit). The same problem can occur when using the 1515 or 1525 printer. If you leave the screen on then, quite frequently, the printer will hang up in the middle of printing and nothing can be done to rescue the situation. Turning the screen off before you start printing, and back on when you have finished, will solve the problem completely.

To turn the screen off, use the following **POKE**:

POKE 53265,PEEK(53265) AND 239

and to turn the screen on again use:

POKE 53265,PEEK(53265) OR 16

Turning the screen off will allow the processor to run slightly faster (approx 5.5%) and any program will also **RUN** slightly faster.

SMOOTH SCROLLING

The **VIC-II** chip supports smooth scrolling in both the horizontal and vertical directions. Smooth scrolling is a one pixel movement of the entire screen in one direction and is very useful for creating realistic background movements for games programs. It is used for moving new data smoothly onto the screen whilst smoothly removing characters from the other side.

Although the **VIC-II** chip does much of the work for you, the actual scrolling must be done by your own program. **BASIC** programs are not fast enough to carry out the scrolling of the entire screen realistically so a machine language routine has to be written. The **VIC-II** chip has the ability to place the screen in any of 8 horizontal and 8 vertical positions, controlled by the scroll registers. The **VIC-II** chip also has a 38 column mode and a 24 line mode. These smaller screen sizes are used to give you a place for your new data to scroll on from.

The following list contains the steps that should be carried out for smooth scrolling:

1) Shrink the screen in the necessary direction. For horizontal scrolling use the 38 column mode. For vertical scrolling use the 24 line mode.

2) Set the scrolling register to maximum (or minimum depending on the direction of the scroll).

3) Place the new data in the proper (covered) part of the screen.

4) Decrement (or increment) the scrolling register until it reaches a minimum (or maximum) value.

5) At this point, use your own routine to scroll the entire screen one complete character in the direction of scroll. The **BASIC** language is not fast enough to complete this task realistically; machine code should be used.

6) Go back to step 2 and repeat the process for as long as you wish scrolling to continue.

To go into the 38 column mode, bit 3 of location 53270 must be set to 0. As always, it is important to affect only this bit. The following **POKE** does this:

```
POKE 53270,PEEK(53270) AND 247
```

To return to the 40 column mode, set the same bit to 1:

```
POKE 53270,PEEK(53270) OR 8
```

To enter the 24 line mode, set bit 3 of location 53265 to 0. As always, it is important to affect only this bit. The following **POKE** does this:

```
POKE 53265,PEEK(53265) AND 247
```

To return to 25 line mode, set the same bit to 1:

```
POKE 53265,PEEK(53265) OR 8
```

When scrolling horizontally, it is necessary to place the screen in the 38 column mode. This will give a place from which new data can scroll onto the screen. If you are scrolling

from right to left then you should put your data on the right side of the screen. When scrolling left to right then the new data should be put on the left side. Remember that the screen is still 40 columns wide, but only 38 columns are visable.

When scrolling vertically, it is necessary to put the screen in the 24 line mode. To scroll upwards you must put data on the bottom line, whilst to scroll downwards you should put the new data on the top line. Unlike the 38 column mode, where there is a covered column on both sides of the screen, the 24 line mode only has one covered line. When the Y scrolling register is set to 0 then the top line is covered, ready for new data. When the Y scrolling register is set to 7 then the bottom line is covered.

To scroll horizontally, the scroll register is located in bits 2 to 0 of location 53270. As always it is important to affect only these bits. The following **POKE** does this:

POKE 53270,(PEEK(53270) AND 248)+X

where X is the horizontal position of the screen, from 0 to 7.

To scroll vertically, the scroll register is located in bits 2 to 0 of location 53265. As always it is important to affect only these bits. The following **POKE** does this:

POKE 53265,(PEEK(53265) AND 248)+Y

where Y is the vertical position of the screen, from 0 to 7.

Scrolling involves stepping through the relevant scrolling register, either from 0 to 7 or 7 to 0 depending on the direction of scroll, adding new data onto the covered part of the screen and repeating the whole process. As mentioned earlier, **BASIC** is too slow to carry out the complete scrolling process. Why not experiment for yourself? It is perfectly possible to scroll things from **BASIC**, but the scroll may be a bit jerky.

RASTER REGISTER

The raster register, the interrupt status register and the interrupt enable register are all used by machine code programmers only. Only a brief explanation of their functions is given here, and **BASIC** programmers can skip over the rest of this chapter without missing any important information.

The Raster Register is a unique register within the 64. It is the only register in the entire machine that performs two different tasks at the same time!

Reading this register will return the current raster position of the TV picture. You can use the raster register to set up display changes to eliminate screen flicker, for instance. Changes to the screen display should be timed so that the raster is not within the display area.

When the raster register is written to the number is saved for use with the raster compare function. When the actual raster value is equal to the number written into the raster register then a bit in the interrupt status register will be set to a 1. If the corresponding bit in the interrupt enable register is also set to a 1 then an interrupt request (IRQ) will be generated. If that bit in the interrupt enable register is set to a 0 then no **IRQ** will be generated. In this way, selective interrupt servicing can be easily carried out.

The raster register is a nine bit register, with the lowest eight bits at location 53266 and the most significant bit at the highest bit of location 53265. A write to this register must include the **MSB** to effectively latch the raster compare function.

The bit allocation of the Interrupt Status Register, at location 53273, is as follows:

BIT	ALLOCATION
0	CURRENT RASTER=STORED RASTER
1	SPRITE-DATA COLLISION
2	SPRITE-SPRITE COLLISION
3	LIGHT PEN (1 per TV frame)
4	UNUSED
5	UNUSED
6	UNUSED
7	ANY ENABLED INTERRUPT OCCURENCE

The Interrupt Enable Register, at location 53274, has the same bit allocation as above. If you set a bit to 1 then that will allow an interrupt to be generated when the specific event takes place. Conversely, setting a bit to 0 will prevent an interrupt request from being generated.

This is a powerful interrupt structure that is not normally seen in such a modestly priced home computer. Using its features wisely can allow you very simply to generate split-screen graphics, more than eight sprites on the screen at the same time, and so on. The secret is to use the interrupts properly. For instance, you can create a split screen, where the top half of the screen displays the text screen and the bottom half shows the high-resolution screen. Set the Raster Register to half way down the screen. When the Raster interrupt occurs you then tell the **VIC-II** chip to switch to the bit-mapped mode and reset the Raster Register to the top of the screen. When the next Raster interrupt occurs you tell the **VIC-II** chip to switch back to the standard character mode and again reset the Raster Register to the middle of the screen. In this way

the top of the screen will be displaying characters whilst the lower half will show the bit-mapped screen. It need hardly be said that these techniques are not available from **BASIC** because of the speed restrictions of the language. Interrupts are the province of the machine language programmer, and to get full use out of the interrupt structure built into the Commodore 64 you will have to be a very good programmer indeed!

CHAPTER NINE
HI-RES GRAPHICS

As you learned in the last chapter, the standard text screen on the Commodore 64 is very versatile, allowing you to mix text and graphics characters freely on the screen. The shapes that can be made using the range of graphics characters can be novel but they are also quite limited. It is, for instance, impossible to draw a square of any given size in the standard text mode simply because the resolution of the screen is not sufficiently high. Most modern home computers contain a **BASIC** language that includes a range of commands to allow you to draw shapes in a high resolution mode, to a degree of accuracy far greater than can be achieved using characters. Unfortunately, although the Commodore 64 does contain a high resolution screen, it's **BASIC** doesn't have any of the necessary commands to allow you to use the HI-RES MODE to any good purpose. In this chapter we will be looking at the HI-RES mode and how to manipulate it from within **BASIC.**

As described in the last chapter, the standard screen on your 64 contains 25 rows of 40 characters, a total of 1000 bytes of memory. The data describing the characters that can be displayed on the screen is held in the character **ROM,** with each character using 8 bytes. Each pixel in a character is represented by one bit of this group of eight bytes.

In the standard text mode the characteristics of each pixel are determined by the character **ROM** (or character **RAM** if you are using user-defined graphics). In Bit-mapped code every pixel on the screen is individually addressable, so that we are able to produce high resolution drawings. The use of bit-mapped mode is very similar to user-defined graphics in that

you can think of the screen as 1000 locations waiting to be defined. The only real difference is that the definitions are applied directly to the screen, rather than via the character memory.

STANDARD BIT-MAPPED MODE

This is the highest resolution mode available on the 64 and gives a screen resolution of 320 pixels horizontally by 200 pixels vertically. Note the relationship between the numbers here, 40 characters ★ 8 pixels = 320 pixels across and 25 rows ★ 8 pixels = 200 pixels down. Within each block of 8★8 pixels you can have only two colours, although, as you will see shortly, each square can contain a different pair of colours. Standard bit-mapped mode is turned on by setting BIT 5 of location **53265** to 1. But first we must tell the computer where in memory the hi-res screen will be:

```
5 POKE 53272,PEEK (53272) OR 8
10 POKE 53265,PEEK (53265) OR 32
```

and turned off again by resetting the same BIT to zero:

```
20 POKE 53265,PEEK (53265) AND 223
```

If you type in the lines above and run them then you will only see a momentary change to the screen display. If you now add the following delay you will be able to see the bit-mapped screen somewhat better:

```
15 FOR DELAY = 1 TO 2000:NEXT DELAY
```

Now after two seconds the normal screen will reappear. The high resolution screen will look quite a mess because we have not yet cleared it, so this is what we shall do next. Type in the following routine and **RUN** it. At the end of the routine you will be left in the hi-res mode, so press the **RUN/STOP** and **RESTORE** keys together to get back to your familiar screen.

```
10 POKE53265,PEEK(53265) OR 32
20 FOR SCREEN = 8192 TO 16192
30 POKE SCREEN,0
40 NEXT SCREEN
```

Here we are clearing the screen by setting every byte within the hi-res screen area to zero. You should notice that the screen does not actually clear completely, but leaves several blocks of garbage behind. In the bit-mapped mode the normal screen is now used, not for character data, but for colour information. To remove these blocks of colour we must also clear the standard screen area to a single colour.

COLOUR IN BIT-MAPPED MODE

The colour of the bit-mapped screen is controlled by the data in the standard screen memory area from 1024 to 2024 in the following way:

BIT	7	6	5	4	3	2	1	0
	foreground colour				background colour			

For every byte in colour memory, the lowest four bits control the background colour and the highest four bits control the foreground colour in the corresponding 8★8 pixel area of screen memory. To set a combination of colours use the simple formulae (foreground colour no.)★16 + (background colour no.). **POKE**ing the resultant number into the locations 1024-2024 will clear the entire screen to the required colour. For example, if you wanted to set the foreground to black (0) and the foreground to light red (10) you would **POKE** the colour memory with (0★16+10)=10. Add the following lines to do this:

```
50 FOR COLOUR = 1024 TO 2024
60 POKE COLOUR,10
70 NEXT COLOUR
```

Now we have a fully clear, light red screen we can get down to the harder part of actually drawing something onto it. The organisation of screen memory follows the same pattern as the nominal organisation under user-defined graphics. Looking at the first 8★8 pixel area, in the top left corner of the screen, you can visualize the area as if it were a character. You already know how a character is made up, from eight bytes of character data with the first byte defining the first row of pixels, the second byte defining the second row, and so on. The memory is used in just the same way in the bit-mapped mode. The first byte contains the pixel data for the first eight pixels across the top row of the screen, the second byte defines the first eight pixels in the second row and so on down to the eight row. The ninth byte of screen memory defines the second group of eight pixels along the top row, which if you think about it is quite logical since this is an equivalent position to the top of the second character in the standard text mode. And so the memory allocation continues, across the screen in blocks of eight bytes to cover the first character line and then down to the second line and so on down to the bottom of the screen.

Whilst this type of organisation of screen memory is very convenient for the computer, since it mimics so closely the more normal text modes, it does make the calculation of the pixel equivalent to a particular position on the screen very difficult for us poor feeble-brained humans. The solution to this problem is, luckily, quite simple. We don't actually perform any of the calculations but leave this drudgery to the computer!

If we describe the position of a pixel in terms of its **X** and **Y** co-ordinates then we can let the computer calculate the byte location and the position within that byte of the required BIT. The **ROW** number (from 0 to 24) of the dot is:

ROW = INT (Y/8)

The character position on that line (from 0 to 39) is:

CHAR = INT(X/8)

The line of that character position (from 0 to 7) is:

LINE = Y AND 7

And, finally, the bit of that byte is:

BIT = 7—(X AND 7)

We can now put all of these formulae together. The byte location of the pixel (or memory dot) **(X,Y)** is calculated by:

BYTE = BASE + ROW★320 + CHAR★8 + LINE

Finally, to turn on the pixel with co-ordinates **(X,Y)** use the following:

POKE BYTE, PEEK (BYTE) OR 2 ∧ BIT

and to turn off the same pixel use:

POKE BYTE, PEEK (BYTE) AND (255-2 ∧ BIT)

In practice it is best to put all of these calculations together in a subroutine that can then be called whenever we wish:

```
1000 REM ****************************
1001 REM *  DISPLAY A PIXEL WITH    *
1002 REM *  SCREEN CO-ORDS (X,Y)    *
1004 REM ****************************
1010 ROW = INT (Y/8)
1020 CHAR = INT(X/8)
1030 LINE = Y AND 7
1040 BIT = 7-(X AND 7)
1050 BASE = 8192
```

```
1060 BYTE =BASE+ROW*320 + CHAR*8 + LINE
1070 POKE BYTE, PEEK(BYTE) OR 2↑BIT
1080 RETURN
```

We can now use what we have learned so far to create a series of lines on the bit-mapped screen. The first program will draw a straight horizontal line across the screen. It does this by repeatedly calling the pixel plotting routine within a **FOR...NEXT** loop that steps through the **X** co-ordinate, one position at a time, from **X=0** to **X=319**

```
5 POKE 53272,PEEK(53272) OR 8
10 POKE 53265,PEEK(53265) OR 32
20 FOR SCREEN = 8192 TO 16192
30 POKE SCREEN,0
40 NEXT SCREEN
50 FOR COLOUR = 1024 TO 2024
60 POKE COLOUR,10
70 NEXT COLOUR
100 REM **********************
102 REM * DRAW A HORIZONTAL *
104 REM *        LINE        *
106 REM **********************
110 FOR X = 0 TO 319
120 Y = 128
130 GOSUB 10000
140 NEXT X
150 FOR PAUSE = 1 TO 2000:NEXT PAUSE
160 POKE 53265,PEEK(53265) AND 223
199 END
1000 REM ****************************
1001 REM *  DISPLAY A PIXEL WITH    *
1002 REM *  SCREEN CO-ORDS (X,Y)    *
1004 REM ****************************
1010 ROW = INT (Y/8)
1020 CHAR = INT(X/8)
1030 LINE = Y AND 7
1040 BIT = 7-(X AND 7)
1050 BASE = 8192
```

```
1060 BYTE =BASE+ROW*320 + CHAR*8 + LINE
1070 POKE BYTE, PEEK(BYTE) OR 2↑BIT
1080 RETURN
```

To create a program that will draw other lines and shapes is now simply a matter of replacing the control routine in lines 100-199 with the appropriate routine of your choice. As an example, to draw a circle centred on the middle of the screen insert the following routine in place of lines 100-199 above:

```
100 REM ************************
102 REM *     DRAW A CIRCLE      *
104 REM *      RADIUS = 80       *
106 REM *  CENTRE X=160 Y=100    *
108 REM ************************
110 R = 80
120 FOR COUNT = 0 TO 2*π STEP .01
130 X = R*SIN(COUNT)+160
140 Y = R*COS(COUNT)+100
150 GOSUB 10000
160 NEXT COUNT
170 FOR PAUSE = 1 TO 2000:NEXT PAUSE
180 POKE 53265,PEEK(53265) AND 223
199 END
```

This routine will draw a circle with radius **R=80,** with its centre in the middle of the screen. Line 110 sets the size of the radius, so why not try changing this to other sizes. Remember of course, to keep the whole circle within the screen!

Add the following lines to the circle routine and see the result for yourself:

```
115 FOR ELLIPSE = 1 TO 0 STEP 0.2
130 X = ELLIPSE*R*SIN(COUNT)+160
165 NEXT ELLIPSE
```

Quite a nice display, isn't it? You will notice, however, that these programs run very slowly, because the Commodore 64 does not contain any **BASIC** commands to control its high resolution screen displays. To speed up the process requires the use of machine code routines to carry out all the separate functions that are so obviously missing from the machine's **BASIC.** In the same way as we have used separate routines to clear the screen, draw a line and plot a single point from **BASIC** so you could write your own machine code routines to do the same thing and then call them (using **SYS**) whenever you wish to use them.

As mentioned earlier in this chapter, you are not restricted to only two colours across the whole screen, but you are limited to one foreground and one background colour within each 8★8 pixel block. To demonstrate this point, here is a program that will fill the screen with blocks of randomly chosen colour.

```
5 POKE 53272,PEEK(53272) OR 8
10 POKE 53265,PEEK(53265) OR 32
20 FOR SCREEN = 8192 TO 16192
30 POKE SCREEN,0
40 NEXT SCREEN
50 FOR COLOUR = 1024 TO 2024
60 POKE COLOUR,10
70 NEXT COLOUR
100 REM **********************
102 REM *  FILL THE SCREEN   *
104 REM *     WITH COLOUR    *
106 REM **********************
110 COLOUR =  INT (RND(1)*16)
120 SCREEN = 1024+INT(RND(1)*1000)
130 POKE SCREEN, COLOUR
140 GET A$:IF A$ <>"" THEN 160
150 GOTO 110
160 POKE 53265,PEEK(53265) AND 223
199 END
```

Line 110 sets the colour equal to a **RaNDomly** chosen number within the range 0 to 15. Similarly, line 120 choses a **RaNDom** screen position and line 130 **POKE**s the colour to the screen location. If you have not pressed a key then the program will loop back to line 110 again to repeat the whole process. If you have pressed a key, possibly because you are totally dazzled by the brilliant display of colours, then line 160 will return you to the text screen before **END**ing the program.

Another point to notice about this program is that the blocks of colour are drawn onto the screen much more quickly than the previous examples. This is because we are directing the programs attention to whole character sized squares instead of the individual bits required when drawing detailed shapes. So, although we can get a colourful display rather quickly like this, it is at the expense of resolution and detail. Another feature of using the standard text screen as the colour memory area for the bit-mapped screen is that you can rapidly change the colours on the screen by **PRINT**ing strings of characters onto the text screen in the normal way. Although you are **PRINT**ing characters the computer is interpreting them as colour data! This 'dodge' can be very useful for making rapid global changes to the colour scheme, but needs to be used with care if you are to achieve a reasonable screen display.

Here is a shortish program that allows you to draw on the standard bit-mapped screen using a joystick (plugged into PORT 2). All that you have to do is to point your joystick in the right direction and turn yourself into a budding Picasso.

```
5 POKE 53272,PEEK(53272) OR 8
10 POKE 53265,PEEK(53265) OR 32
20 FOR SCREEN = 8192 TO 16192
30 POKE SCREEN,0
40 NEXT SCREEN
50 FOR COLOUR = 1024 TO 2024
60 POKE COLOUR,10
70 NEXT COLOUR
```

```
100 REM ***************************
102 REM * READ THE JOYSTICK PORT *
104 REM ***************************
110 STICK = PEEK(56320)
120 FIRE = STICK AND 16
130 IF(STICK AND 1)=0 THEN Y=Y+(Y>0)
140 IF(STICK AND 2)=0 THEN Y=Y-(Y<200)
150 IF(STICK AND 4)=0 THEN X=X+(X>0)
160 IF(STICK AND 8)=0 THEN X=X-(X<320)
170 GOSUB 10000
180 GOTO 110
10000 REM ***************************
10001 REM *   DISPLAY A PIXEL       *
10002 REM *   WITH SCREEN CO-ORDS   *
10003 REM *         (X,Y)           *
10004 REM ***************************
10010 ROW = INT (Y/8)
10020 CHAR = INT(X/8)
10030 LINE = Y AND 7
10040 BIT = 7-(X AND 7)
10050 BASE = 8192
10060 BYTE = BASE+ROW*320+CHAR*8+LINE
10070 IF (FIRE=0) THEN POKE BYTE,PEEK(BY
TE) OR 2↑BIT:REM SET A PIXEL
10080 IF (FIRE<>0) THEN POKE BYTE,PEEK(B
YTE) AND (255-2↑BIT):REM UNSET
10090 RETURN
```

Because of the speed limitations mentioned above, the program runs rather slowly and further emphasises the point that without machine code (or an extension to **BASIC** cartridge such as **SIMONS BASIC**), high resolution graphics on the 64 are certainly not up to arcade action standards.

Whilst a high degree of resolution can be obtained in the standard bit-mapped graphics mode, it does not allow for very much flexibility in terms of colour since each pixel within every 8 by 8 block must be in the same colour. The Commodore allows a way round this by using a second bit-mapped mode, the multi-coloured bit-mapped mode.

Again, there is a parallel with multi-coloured graphics in that up to four colours are available for each 8 by 8 block and adjacent pixels can be in different colours. The drawback is that there is a loss of horizontal resolution with the display width being reduced to 160 pixels and each pixel now becomes twice its previous width.

MULTI-COLOURED BIT-MAPPED MODE

In order to be able to display up to four colours per 8*8 block, the computer uses the 8000 bytes of screen memory in a different way. Each pixel is now two BITs wide, which allows a total of 4 combinations of BIT pattern for each pixel, whilst the screen resolution is reduced to 160 pixels horizontally by 200 pixels vertically. The colour information for this mode is stored in four separate areas, as shown in the table below:

BITS	COLOUR INFORMATION STORED IN
00	Background colour 0 (53281)
01	Upper four bits of text screen memory
10	Lower four bits of text screen memory
11	Colour nybbles

Of the four colours that can be displayed within each square, the background colour is fixed globally across the entire screen area, whilst the other three colours can be different for each 8 by 8 square. An unfortunate side-effect of using the standard text screen to store colour information for the bit-mapped screen display is that the text screen is still able to scroll upwards whenever anything is printed onto the bottom line, whereas the hi-res screen display will not scroll. The result will be the ruination of your carefully prepared display, so be careful, if you do change colour data by printing to the screen, that you do not cause a scroll. Of course, **POKE**ing colour **DATA** onto the screen is safer from this respect, since **DATA** can be **POKE**d anywhere on the screen without fear of scrolling.

The drawing of a multi-coloured bit-mapped screen requires a lot of thought and planning. It is best to plot out the general areas of different colours on a piece of graph paper before committing the work of art to the computer and then onto the screen. As has been said before, multi-coloured screen displays can create a very dramatic background for games using sprites for all of the moving objects and it is well worth the effort needed.

CHAPTER TEN
SPRITES

Sprites are one of the most versatile and sophisticated features of the Commodore 64. When using sprites, or Moveable Object Blocks as Commodore also call them, you can create intricate and realistic shapes that can then be moved around the screen fairly easily from **BASIC.**

A sprite is a special type of user definable character which can be displayed anywhere on the screen. All the sprites; you can have up to eight on the screen at any one time; are maintained directly by the VIC-II chip, so the only thing that you have to do is to tell the sprites what shape they should be, what colour they should be, and where they should appear on the screen. The VIC chip will take care of all the rest.

Sprites can be used in combination with any of the graphics modes, Bit-mapped, Text, Multi-coloured, and so on. As has been already mentioned, ordinarily up to eight sprites can be displayed simultaneously, although by using machine code routines and raster interrupt techniques many more can be displayed. The most interesting and useful features of sprites are:

1) 24 horizontal by 21 vertical dot size - 504 pixel total area.
2) Each sprite can be any one of the 16 colours that are available on the Commodore 64.
3) Every sprite can be a different colour to every other.
4) When in multi-coloured mode, each sprite can be up to four colours, although in this mode three of these four colours are common to all sprites.

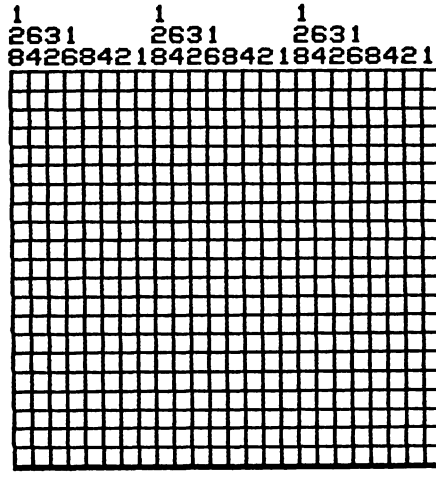
5) Each sprite can be individually magnified by 2 times in either the horizontal or vertical direction, or both at the same time.
6) Standard and magnified sprites can be freely mixed on the screen.
7) Selectable sprite to background priority.
8) Fixed sprite to sprite priorities.
9) Automatic sprite to sprite collision detection.
10) Automatic sprite to background collision detection.

These special sprite abilities make it simple to program many arcade style games. Because the sprites are maintained by hardware (the VIC chip) it is even possible to write a good quality game from **BASIC!**

There are eight sprites supported directly by the VIC chip, numbered from 0 to 7. Each of these sprites has its own definition location, position registers and colour register, and has its own bits for enable and collision detection. Let's take a further look into sprites, starting with how a sprite shape is created.

CREATING A SPRITE

Sprites are defined in the same way as user defined characters. Sprites are, of course, much larger than characters and so more bytes are needed to store all of the sprite **DATA.** The size of a sprite is 24 pixels horizontally by 21 pixels vertically, a total area of 504 pixels. Each pixel requires one bit of memory for its definition so a total of 63 bytes (504/8) to hold all of the **DATA** required to define one sprite.



In a standard (HI-RES) sprite, each bit that is set to a 1 is displayed in the sprites foreground colour. Every bit that is set to a zero is transparent and will show whatever data is displayed on the screen behind the sprite. This is in contrast, in a way, to a character where a zero bit will be shown as the background colour and nothing else. Whilst a character may be thought of as being transparent when a bit is set to zero, there can be nothing behind the character except the background screen, so we need not concern ourselves with the philosophical differences between transparency and background colours. With a sprite, however, it is important to visualize the consequences of transparency. The eight sprites have a set priority level, that is sprite 0 will always appear in front of sprite 1, sprite 1 will be in front of sprite 2 and so on. It is possible to build up intricately detailed sprite patterns by overlaying several sprites upon each other, using the transparency of the nearer shapes to allow the otherwise overshadowed sprites to show through. Using this technique it is possible to create a high resolution multi-coloured sprite combination with, if you were to use four sprites together, a

total of five colours. At a more fundamental level, the transparency of zero bits within a sprite can be used to good effect to create the illusion of a third dimension on the screen with a lower priority sprite being seen behind the transparent parts of a nearer one as the two pass over and under each other.

Although each sprite requires 63 bytes for its definition, the VIC chip, being well tutored in the pleasures of binary arithmetic, divides the sprite memory into blocks of 64 bytes. The last byte, and it is the last byte of each block, is unused and can thus be set to any value or, if you are really short of memory for some reason, used for other purposes. Each of the eight sprites has a byte associated with it called the **SPRITE POINTER.** The sprite pointers control where in memory the sprite **DATA** is stored or, to be more precise, where the VIC chip looks for the **DATA** for the associated sprite shape. These eight bytes, one per sprite, are always located at the last eight bytes of the 1K block of screen memory. Ordinarily the standard text screen starts at location 1024 and is 1000 bytes long, taking it up to 2024. The **SPRITE POINTERS** are locations 2040 to 2047 in this case. If, however, the screen is relocated then the sprite pointers will also be moved, but they will always be the last eight bytes of the 1K of screen memory.

Each sprite pointer can hold a number between 0 and 255. This number points towards the location of the sprite data. Since each sprite definition requires 64 bytes, that means that the point can point to anywhere within an area of 16K (256★64=16K), which as you have already learned is the same amount of memory that the VIC chip can 'see'.

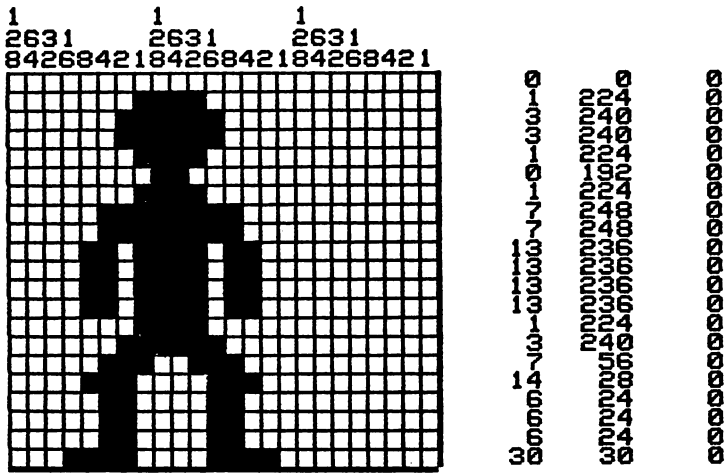
If the pointer for sprite 0, at location 2040, contains the number 13, for example, this means that sprite 0 will be displayed using the 64 bytes of data starting at location 13★64 = 832 which, incidentally, is in the cassette buffer. The following general formulae can be used:

LOCATION = (BANK★16384)+ (SPRITE POINTER★64)

Where **BANK** is the 16K segment of memory that the VIC-II chip is looking at, from bank 0 through to bank 3.

When the VIC-II chip is looking at **BANK 0** or **BANK 2** there will be an image of the character **ROM** in certain locations and these locations cannot be used to store sprite data. If for some reason you require more than 128 different sprite definitions then you should use one of the banks without the **ROM** image, 1 or 3.

So far you have seen how to create the actual sprite shape and point the VIC-II chip to the starting location of the data held in memory. Before we go any further let's create a sprite and put the data for it into memory.



The man is defined and put into memory using these lines:

```
100 GOSUB 1000
110 END
1000 REM ************************
1002 REM *    A SPRITELY MAN    *
1004 REM ************************
1010 FOR COUNT = 0 TO 63
1020 READ NUMBER
1030 POKE (13*64)+COUNT,NUMBER
1040 NEXT COUNT
1050 POKE 2040,13
1060 RETURN
1070 DATA 0,0,0
1071 DATA 1,224,0
1072 DATA 3,240,0
1073 DATA 3,240,0
1074 DATA 1,224,0
1075 DATA 0,192,0
1076 DATA 1,224,0
1077 DATA 7,248,0
1078 DATA 7,248,0
1079 DATA 13,236,0
1080 DATA 13,236,0
1081 DATA 13,236,0
1082 DATA 13,236,0
1083 DATA 1,224,0
1084 DATA 3,240,0
1085 DATA 7,56,0
1086 DATA 14,28,0
1087 DATA 6,24,0
1088 DATA 6,24,0
1089 DATA 6,24,0
1090 DATA 30,30,0
1091 DATA 0
```

On running this program nothing noticeable will happen, yet. Although we have now defined the shape of the sprite, there are still a few other items to take care of before we can see the man on the screen in all of his glory.

COLOURS

Each sprite can be any of the sixteen colours available on the Commodore computer. Each sprite has its own colour register and **POKE**ing the required colour number into a register is all that need be done to assign a colour to a sprite.

ADDRESS	COLOUR REGISTER
53287	SPRITE 0
53288	SPRITE 1
53289	SPRITE 2
53290	SPRITE 3
53291	SPRITE 4
53292	SPRITE 5
53293	SPRITE 6
53294	SPRITE 7

All data in the sprite will be displayed in the colour contained in the associated sprite colour register. The rest of the sprite will be transparent and will show whatever is behind the sprite.

TURNING SPRITES ON

Our little man may be spritely but there is only one thing that will turn him on. The register at location 53269 is known as the **SPRITE ENABLE** register. Within this register each bit is allocated to one sprite, bit 0 for sprite 0, bit 1 for sprite 1 and so on. For each sprite, if the bit is set to a zero then the sprite will be turned off and if the bit is set to a 1 then the sprite will be turned on. To turn on sprite 0, for instance, it is necessary to set bit 0 to a 1. The following **POKE** will do this:

```
POKE 53269,PEEK(53269) OR 1
```

A more general statement is:

POKE 53269,PEEK(53269) OR (2 ∧ N)

where N is the sprite number, from 0 to 7.

A sprite cannot be seen on the screen until it is turned on and, conversely, turned off will prevent it from being seen. The following **POKE** will turn sprite **N** off:

POKE 53269,PEEK(53269)—(255-2 ∧ N)

Adding the following lines to our sprite man program will give him a little colour and enable him to be seen on the screen.

```
1052 POKE 53287,1:REM COLOUR
1053 POKE 53269,PEEK(53269) OR 1
```

Now we have a sprite shape, white in colour and it has been enabled so that it should be seen on the screen. When you **RUN** the program, there is still no sprite to be seen because we have not yet told the VIC-II chip where to display the shape.

SPRITE POSITIONING

Any sprite can be positioned within a grid size of 350 pixels horizontally by 256 pixels vertically. For each sprite there are three Sprite Positioning Registers that store the current position of the sprite. Each sprite has an X-position Register, a Y-position Register and a bit in the X-most significant bit register. This last register is needed in order to store X positions greater than 255, which is the largest number that can be stored in a single byte. The X and Y registers are grouped in pairs from location 53248 to 53263.

LOCATION	DESCRIPTION
53248	SPRITE 0 X-POSITION
53249	SPRITE 0 Y-POSITION
53250	SPRITE 1 X-POSITION
53251	SPRITE 1 Y-POSITION
53252	SPRITE 2 X-POSITION
53253	SPRITE 2 Y-POSITION
53254	SPRITE 3 X-POSITION
53255	SPRITE 3 Y-POSITION
53256	SPRITE 4 X-POSITION
53257	SPRITE 4 Y-POSITION
53258	SPRITE 5 X-POSITION
53259	SPRITE 5 Y-POSITION
53260	SPRITE 6 X-POSITION
53261	SPRITE 6 Y-POSITION
53262	SPRITE 7 X-POSITION
53263	SPRITE 7 Y-POSITION
53264	X MOST SIGNIFICANT BIT REGISTER

Positioning a sprite on the screen is a simple matter of **POKE**ing the correct numbers into the relevant registers. To position sprite 0 in the middle of the screen, for instance, is achieved by:

```
POKE 53248,160:POKE 53249,100
```

and this statement can be added to our sprite man program as:

```
1056 POKE 53248,160:POKE 53249,100
```

and now, at last, when you **RUN** the program there is a visable sprite on the screen, in all his glory!

Now that we have a live sprite on our hands (well, on the screen at least) the time has come to look more closely at the positioning of a sprite in relation to the grid of 320 by 256

pixels. For simplicity let's deal first with the positioning in the Y direction.

A sprite can be placed at any position between 0, which is at the top of the screen, and 256, the bottom. With a normal size sprite placing it at position 0 will, in fact, put the sprite above the top of the screen and out of sight. This is neither a bug nor an accident but is a deliberate feature of the CBM 64. Type in the following additional lines for the sprite man program:

```
1056 POKE 53248,160
1058 FORT=0 TO 256:POKE 53249,T:NEXT
```

When you **RUN** this program you will see that the sprite slowly appears from the top of the screen and glides down to the bottom and disappears again. Although the sprite can be positioned within a 0 to 255 vertical range, positions 0 to 29 are off the top of the screen and positions 250-255 are off the screen at the bottom. This feature is quite deliberate and allows you to smooth scroll a sprite onto and off the screen, to give a more realistic effect to your games. A standard sized sprite is only fully visable on the screen within the vertical range of 54-246, outside of this range at least some of the sprite will be obscured by the screen border.

A similar facility is available when positioning sprites in the horizontal direction, but here the situation is a little bit more complicated.

A sprite can be positioned horizontally within a range of 0-511 pixels. Obviously a single byte cannot hold a number larger than 255. The X POSITION MSB REGISTER is used to store the ninth bit that is necessary to store numbers between 256 and 511. If you are still a little unsure about binary notation then it is worthwhile to stop at this point and consider for yourself the reason why a ninth bit is necessary to hold numbers larger than 256. Each bit has a decimal value of 2 ↑ N, where N is the bit number. Bit 9 will hold the value 2 ↑ 9=256

when it is set to a 1, so 256 should be added to the value stored in the other eight bits. This is the reason why there is a MSB (Most Significant Bit) register for X positions.

In a very similar manner to the Y positioning, not all of the available X positions are actually visible on the screen. For a standard sized sprite the position 0 is completely invisible to the left of the screen, whilst positions 344 to 511 are off the screen to the right. Only within the horizontal range 24 to 320 is a sprite completely visible on the screen.

The actual movement of a sprite is complicated, slightly, by having to set or unset the relevant bit in the MSB register. It is important that you only alter the related bit and leave the other bits unchanged. This is most easily achieved by using a variable to store the X position of the sprite and then using bitwise manipulation to arrive at the final numbers for **POKE**ing to the X position register and the MSB register. For instance, if the variable XO is used to store the X position of sprite 0 then the statements would be:

```
POKE 53248,(X0 AND 255)
POKE 53264,PEEK(53264) AND (255+(XO <256)) OR
ABS (XO >255)
```

This will be correct for all values of X0 between 0 and 511. For the other sprites it is also necessary to affect only the correct bit in the MSB register. The general formulae:

```
POKE 53264,PEEK(53264) AND (255-2 ↑ SN★(SX<256))
OR ABS (2 ↑ SN★(SX>255))
```

will set the correct bit for sprite SN at the position SX.

It need hardly by mentioned, after all that has gone before, that to move a sprite from one position to another is a simple matter of **POKE**ing the new co-ordinates into the relevant registers. The VIC-II chip takes care of all of the movement.

The following lines can be added to the sprite man program to allow you to move him about the screen by pressing the Keys Q-up, A-down, N-left, and M-right.

```
110 X0=160:Y0=120
120 GET A$:IF A$="" THEN GOTO 120
130 IF A$="Q" THEN Y0=Y0-1
140 IF A$="A" THEN Y0=Y0+1
150 IF A$="N" THEN X0=X0-1
160 IF A$="M" THEN X0=X0+1
170 IF Y0>255 THEN Y0=255
180 IF Y0<0 THEN Y0=0
190 IF X0>320 THEN X0=320
200 IF X0<0 THEN X0=0
210 POKE 53248,(X0 AND 255)
220 POKE 53249,Y0
230 POKE 53264,PEEK(53264) AND (255+(X0<
256)) OR ABS(X0>255)
240 GOTO 120
```

If you want to keep the man on the screen then you should alter lines 180 to 210 so that the values of X0 and Y0 cannot go outside the required range.

SPRITE PRIORITIES

It is possible to create dramatic three dimensional effects by making sprites appear to move in front of each other as they cross the screen, or even move in front of, or behind, screen background data. Each sprite has a fixed priority level in relation to the other sprites whilst the priority between each individual sprite and the background data can be user defined.

For sprite to sprite priority, the rule is first come, first served. That is to say that sprite 0 will always have priority over, and will always appear in front of, sprites 1 to 7. Sprite 1 will appear in front of sprites 2 to 7 but behind sprite 0, and so on. Sprite 7 has the lowest priority and will always appear to be behind the other sprites.

As has been mentioned earlier, this advanced feature can be used to good effect in creating realistic three dimensional displays. Sprite priorities can also be used to create a "window" effect where the gaps in a high priority sprite can allow a lower priority sprite to show through, thus you can build up a colourful and hi-res sprite shape using several sprites built on top of each other. For instance, a shadow effect can be created by using sprite 0 as the main shape and sprite 1 as its shadow. Moving both sprites in relation to each other can create a very good illusion of perspective. This is similar to the technique used by the arcade game ZAXXONS, where you judge the height of your spaceship by the size and position of its shadow on the ground.

SPRITE TO BACKGROUND PRIORITY

Sprites also have priority over the background data, the characters or shapes that are printed on the screen. This level of priority applies to any shapes on the high resolution screen as well as the standard text screen. Normally, all sprites will appear to pass in front of any data that is on the background screen. This priority can be changed for any sprite by setting the relevant bit in the **SPRITE-BACKGROUND PRIORITY REGISTER.** As usual, each sprite has its own bit within this register, bit 0 for sprite 0, bit 1 for sprite 1 and so on, and setting a bit to 1 will give the background priority over the sprite and allow the sprite to move behind any screen data. Setting a bit to zero will reverse this arrangement and give the sprite a higher level of priority than the background.

The **SPRITE-BACKGROUND PRIORITY REGISTER** is at location 53275 and the following statement will give sprite SN a lower priority than the background:

```
POKE 53275,PEEK(53275) OR 2^ SN
```

and the next statement will reverse the priority levels and

allow sprite SN to pass in front of the screen data:

POKE 53275,PEEK(53275) AND 255-2 ∧ SN)

In the following lines of the sprite man program you can see the sprite to sprite and the sprite to background priorities working. We now have two men on the screen, the yellow man is static and you can still control the white man using the same keys as before. In addition, pressing function Key 1 will set the background to have priority over the white man and pressing F2 will restore the man's priority to normal. Try moving the man around to see the full effect.

```
100 PRINT CHR$(147):GOSUB 1000
105 FOR T=1 TO 20:PRINT "DEMONSTRATION P
ROGRAM":NEXT T
110 X0=160:Y0=120
120 GET A$:IF A$="" THEN GOTO 120
130 IF A$="Q" THEN Y0=Y0-1
140 IF A$="A" THEN Y0=Y0+1
150 IF A$="N" THEN X0=X0-1
160 IF A$="M" THEN X0=X0+1
170 IF Y0>255 THEN Y0=255
180 IF Y0<0 THEN Y0=0
190 IF X0>320 THEN X0=320
200 IF X0<0 THEN X0=0
210 POKE 53248,(X0 AND 255)
220 POKE 53249,Y0
230 POKE 53264,PEEK(53264) AND (255+(X0<
256)) OR ABS(X0>255)
240 GOTO 120
1000 REM ********************
1002 REM *  A SPRITELY MAN *
1004 REM ********************
1010 FOR COUNT = 0 TO 63
1020 READ NUMBER
1030 POKE (13*64)+COUNT,NUMBER
1040 NEXT COUNT
1050 POKE 2040,13:POKE2041,13
```

```
1052 POKE 53287,1:POKE 53288,7:REM COLOU
R
1053 POKE 53269,PEEK(53269) OR 3
1056 POKE 53250,160:POKE 53251,100
1060 RETURN
1070 DATA 0,0,0
1071 DATA 1,224,0
1072 DATA 3,240,0
1073 DATA 3,240,0
1074 DATA 1,224,0
1075 DATA 0,192,0
1076 DATA 1,224,0
1077 DATA 7,248,0
1078 DATA 7,248,0
1079 DATA 13,236,0
1080 DATA 13,236,0
1081 DATA 13,236,0
1082 DATA 13,236,0
1083 DATA 1,224,0
1084 DATA 3,240,0
1085 DATA 7,56,0
1086 DATA 14,28,0
1087 DATA 6,24,0
1088 DATA 6,24,0
1089 DATA 6,24,0
1090 DATA 30,30,0
1091 DATA 0
```

COLLISION DETECTION

One of the tasks that the VIC-II chip performs is to maintain two registers that show whether or not a collision has occured. The first of these registers is at location 53278 and is used to show collisions between two sprites. The second register, at location 53279 shows collisions between sprites and the background data.

The sprite to sprite register will set a bit to 1 if a sprite has collided with another. The bit allocation is the, by now usual

one, bit 0 for sprite 0, bit 1 for sprite 1 and so on. If, for instance, a collision occurs between sprite 0 and sprite 7 then bits 0 and 7 in location 53278 will both be set to 1. **PEEK**ing this location will return the number 129 (that is, 1 + 128) and the collision detection can then be used within a program, for instance, to initiate a dramatic sound effect.

There will always be at least 2 bits that are set within this register since two sprites will collide into each other. Once a collision has occured the relevant bits will stay set to 1 until the register is read by **PEEK**ing into it. As soon as the register is read then it will be reset to zero and collision data will therefore be lost. On many occasions, however, you will want to make several uses of the same collision data. You may, for instance, want to test to see which of the sprites have collided, with branches to different parts of the program depending on the outcome. In a case such as this it is necessary to first of all set a variable to the value in the collision detection register, for example:

```
COLLIDE=PEEK(53278)
```

Any tests can now be performed upon the variable **COLLIDE:** the register has been reset, ready to detect the next collision. You can, for instance, make a test for a collision between sprite 0 and any other by using:

```
IF (COLLIDE AND 1) THEN ....
```

In general, **(COLLIDE AND 2 ↑ SN)** will give a true result **(—1)** if sprite no. **SN** has collided with any other sprite.
A second collision detection register is located at 52379. This is used to show that a collision has occured between a sprite and non zero background data. Again, bit 0 is used for sprite 0, bit 1 for sprite 1 and so on. The relevant bit will be set to a 1 if the sprite has collided with the background, otherwise it will be a zero.

This register works in exactly the same way as the sprite-sprite collision register. When a collision between a sprite and the background takes place, the relevant bit is set to one. The register will be reset to 0 whenever it is read (**PEEK**ed) so it is best to read the register value into a variable before performing any tests.

When using either of these registers within a program you may, occasionally, find that a collision is flagged that hasn't really occured. This happens because it is the hardware (the VIC-II chip) that is maintaining the collision detection, not the program itself. If a collision occurs at any time then the appropriate register will be updated to show the collision, but the register will not be reset until a **PEEK** is made to it. Previous collisions may still be flagged and a false reading will occur. The solution to this problem is to make a dummy reading of both registers so that both will be reset. A second read of either register will then give an accurate and reliable result.

MULTI-COLOURED SPRITES

Multi-coloured sprites can contain up to four colours. This is, of course, the same as the other multi-coloured modes, multi-coloured user-definable character mode, and multi-coloured bit-mapped mode. The horizontal resolution of the sprite shape is halved because each pixel is now two bits wide. The colour of each pixel is determined by the bit pair combination, according to the following table:

BIT PATTERN	COLOUR DATA FROM
00	BACKGROUND (53281)
01	MULTI 0 (53285)
10	SPRITE (53287-53294)
11	MULTI 1 (53286)

If you study the table carefully you will notice that there are three colours which are common to all eight sprites. The background colour, multi-colour 0 and multi-colour 1 are each defined by a single register to cover all sprites, whereas only the sprite colour can be different for each sprite.

Each sprite can be set, individually, to multi-coloured mode by setting the relevant bit in the Sprite Multi-colour Register, which is at location 53276. Standard and multi-colour sprites can be freely mixed on the screen, if you so wish.

MAGNIFICATION

Any of the sprites can be expanded either in the horizontal or vertical direction. A two times horizontal magnification is achieved by setting the relevant bit of register 53271. The bits in this register are allocated in the familiar way. To expand a sprite in the Y direction the same bits in register 53277 should be set to a one. Of course, expansion can be made in both directions at the same time and different sized sprites can be freely mixed on the screen. All of the collision detection and priority levels of standard sprites still work in just the same way with expanded sprites. When sprites are expanded there is no increase in the resolution of the sprite.

A standard sprite is still 24★21 pixels (12★21 for multi-colour sprites) but the overall size of the sprite is doubled in the desired direction.

Sprites are very versatile, definable objects that can be moved around the screen easily using **BASIC.** Different types of sprite can be freely mixed on the screen and you don't even have the bother of detecting collisions because the VIC-II chip takes care of this automatically. To help you create the shapes that you may need for your sprite games the last program in this section will allow you to draw the shape within a large grid on the screen. Move the cursor around using the cursor keys and press function Key 1 to toggle between

drawing and erasing. When you think you are satisfied with the shape then press function Key 5 and (after a short wait) the large shape will be turned into a real sprite. At this point you can move the sprite around the screen, again by using the cursor keys. Pressing **RETURN** will return you to the creation stage so that you can make any modifications. When you are completely happy with your masterpiece then press function Key 5 again to ensure that the data numbers have been updated. Finally, jot down these numbers on a piece of paper ready for use in your own program.

```
5 REM ************************
10 REM*     SPRITE  DEFINER    *
15 REM************************
20 AT$=CHR$(17):FORT=1TO5:AT$=AT$+AT$:NE
XTT:AT$=CHR$(19)+AT$
25 DIMBIT(21,24):DIMDTA(21,2)
30 SX=100:SY=100
100 GOSUB 1000
110 GOSUB2000
120 GOSUB1100
1000 REM ***********************
1002 REM *    DRAW THE GRID     *
1004 REM ***********************
1010 PRINTCHR$(147)"      S P R I T E
 D E F I N E R"
1020 PL$="┌┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┬┐ "
1030 FORT=0TO20:PRINTLEFT$(AT$,3+T)SPC(1
)PL$:NEXTT
1040 PRINTLEFT$(AT$,24)SPC(1)"└──────────
──────────┘"
1052 PRINTLEFT$(AT$,6)SPC(26)"   CONTROL
S"
1055 PRINTLEFT$(AT$,8)SPC(26)"F1=DRAW/UN
DRAW"
1060 PRINTLEFT$(AT$,9)SPC(26)"F3=UPDATE
DATA"
1065 PRINTLEFT$(AT$,10)SPC(26)"F5=CREATE
"
```

```
1070 PRINTLEFT$(AT$,11)SPC(26)"      SPRI
TE "
1075 PRINTLEFT$(AT$,12)SPC(26)"F7=QUIT "
1080 PRINTLEFT$(AT$,14)SPC(26)"USE THE C
ORSOR"
1085 PRINTLEFT$(AT$,15)SPC(26)" KEYS TO
MOVE"
1097 RETURN
1099 :
1100 REM ************************
1102 REM *  MOVE THE CURSOR    *
1104 REM ************************
1110 Y=3:X=1:CH$=CHR$(111)
1150 PRINTLEFT$(AT$,Y)SPC(X)"*"CHR$(157)
;:POKE204,0
1155 BIT((Y-3),(X-1))=FL
1160 GETA$:IFA$=""THEN1160
1165 POKE204,1
1170 IFA$="■"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:Y=Y-(Y<23)
1175 IFA$="□"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:Y=Y+(Y>3)
1180 IFA$="▌"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:X=X-(X<24)
1185 IFA$="▐"THENPRINTLEFT$(AT$,Y)SPC(X)
CH$;:X=X+(X>1)
1190 IF FL=0THEN IFA$=CHR$(133)THENCH$="
▚ ▞":FL=1:GOTO1197
1192 IF FL=1THEN IFA$=CHR$(133)THENCH$=C
HR$(111):FL=0:GOTO1197
1194 IFA$=CHR$(134)THENGOSUB1200
1195 IFA$=CHR$(135)THENGOSUB1300
1196 IFA$=CHR$(136)THENPRINTCHR$(19);:EN
D
1197 IFA$=CHR$(140) THEN RUN 17
1199 GOTO1150
1200 REM ************************
1202 REM *     UPDATE DATA      *
1204 REM ************************
1205 GOSUB 1400
```

```
1210 FORT=0TO20
1215 DTA(T,0)=0:DTA(T,1)=0:DTA(T,2)=0
1220 FOR BIT=0TO7
1230 DTA(T,0)=DTA(T,0)+(2↑(7-BIT))*BIT(T
,BIT)
1232 DTA(T,1)=DTA(T,1)+(2↑(7-BIT))*BIT(T
,BIT+8)
1234 DTA(T,2)=DTA(T,2)+(2↑(7-BIT))*BIT(T
,BIT+16)
1240 NEXT BIT
1245 PRINTLEFT$(AT$,T+3)SPC(30-LEN(STR$(
DTA(T,0))));DTA(T,0)
1250 PRINTLEFT$(AT$,T+3)SPC(34-LEN(STR$(
DTA(T,1))));DTA(T,1)
1255 PRINTLEFT$(AT$,T+3)SPC(38-LEN(STR$(
DTA(T,2))));DTA(T,2)
1260 NEXTT
1299 RETURN
1300 REM **********************
1302 REM * CREATE THE SPRITE *
1304 REM **********************
1310 GOSUB1200
1320 FORT=0 TO 20
1330 POKE832+T*3,DTA(T,0):POKE833+T*3,DT
A(T,1):POKE834+T*3,DTA(T,2)
1340 NEXTT
1350 POKE2040,13
1360 GOSUB2100
1399 RETURN
1400 REM **********************
1402 REM * CLEAR RIGHT PANEL *
1404 REM **********************
1410 FOR XX=2 TO 23
1420 PRINTLEFT$(AT$,XX)SPC(26);"
     "
1430 NEXT XX
1499 RETURN
2000 REM **********************
2002 REM * INITIALIZE SPRITE *
2004 REM **********************
```

```
2010 SPRITE=53248
2020 POKESPRITE,100
2030 POKESPRITE+1,100
2040 POKESPRITE+39,1
2050 POKE2040,13
2060 POKESPRITE+21,1
2099 RETURN
2100 REM **********************
2102 REM *  MOVE THE SPRITE  *
2104 REM **********************
2105 PRINTLEFT$(AT$,25)SPC(3);"■ USE CUR
SOR KEYS TO MOVE OR ■RETURN";
2107 PRINTLEFT$(AT$,25)SPC(3);" USE CURS
OR KEYS TO MOVE OR ■RETURN";
2110 GET A$:IF A$= "" THEN 2105
2120 IF A$="■"THEN SY=SY-(SY<255)
2130 IF A$="■"THEN SY=SY+(SY>1)
2140 IF A$="■"THEN SX=SX-(SX<255)
2150 IF A$="■"THEN SX=SX+(SX>1)
2160 IF A$=CHR$(13)THEN GOTO 2190
2170 POKESPRITE,SX:POKESPRITE+1,SY
2180 GOTO2105
2190 PRINTLEFT$(AT$,25)SPC(3);"■
                                  ";
2199 RETURN
```

APPENDIX

APPENDIX 1

GLOSSARY

The following glossary attempts to explain some of the computer jargon that is used in this book and elsewhere.

Abort	To abandon an activity, usually due to an error.
AC	Alternating Current.
Access	To gain control of a system or the acquisition of data from a peripheral.
Accumulator	A type of register.
ADC	An abbreviation for Analogue to Digital Converter. Converts Analogue signals into Digital signals.
Address	A memory location, used in much the same way as a house address.
AI	An abbreviation for Artifical Intelligence.
Algorithm	The set of steps or formula used to perform a set task.
Alphanumeric	Numbers, letters and all other characters that are represented by an ASCII code.
Array	A means of allocating memory space for the storage of numeric or string data.
ASCII	An acronym for American Standard Code for Information Interchange pronounced "as-key". A way of storing alphanumeric information in binary.
Assembler	A program which transfers a symbolic language program into a machine language program.
Assembly Language	One of the two languages resident in memory also known as machine code.
Back-up	A procedure or facility which allows the user to retain information in the event of a computer failure.
BASIC	An acronym for Beginners All-Purpose Symbolic Instruction code. The simplest of the two languages resident in memory.
Baud Rate	Number of bits per second that are transmitted down the wire when saving or loading a program.
BCD	An abbreviation for Binary Coded Decimal. A way of expressing decimal code in bits using four binary bits for each decimal number.
Binary	A number system (base 2) which only uses ones and zeroes.
Binary Digit	A digit on the binary scale of notation; either 1 or 0.

Bit	Short for binary digit, it is always either a 1 or 0. The basic unit for information storage.
Bleed	The spreading of ink beyond the edge of a printed character.
Block	A group of information units that are handled as a single unit. In many magnetic storage devices only complete blocks can be accessed or transferred.
Boot	Short for bootstrap. A method of inputting data prior to the loading of a program.
Branch	A point in a program where the micro must select one out of two, or more pathways.
Buffer	Where data is temporarily stored until the CPU is ready to process it.
Bug	A hardware, or more commonly, software error.
Bus	A set of connections allowing a route around the micro for signals.
Byte	Made up of 8 bits usually representing a character or number.
Cassette	A medium for storing information.
Centronics	Type of printer interface.
Character String	A sequence of characters.
Chip	A piece of silicon known as an integrated circuit, an integral part of the micro.
Compiler	Translates source code into object code or, more simply BASIC into machine code.
CPU	An abbreviation for Central Processing Unit, the computer's brain.
Cursor	A flashing box on the screen of the VDU which indicates the current PRINT position.
Data	The information required by a computer before it can act.
Database	A store of data on files which can be made accessible to a computer.
DC	An abbreviation for Direct Current.
Debug	The removal of errors.
Dedicated	A program, procedure, machine, network channel or system set apart for special use.
Disc	A medium of storing information.
DOS	An acronym for Disc Operating System. A program which controls the operation of all activities related to the use of magnetic discs in a computer system.
Diskette	Another name for a 5.25 inch floppy disc.
Edit	To change or correct a program or statement.
Editor	Software which aids the editing of a file or which edits in the course of a program.
EPROM	An acronym for Erasable, Programmable Read Only Memory.

Error Message	An indicator that informs you of an error.
Escape Code	Code used with text input to indicate that the following character(s) will represent a function.
File	A block of data that can be stored and retrieved.
File name	A series of characters used to identify a file. A file name is often composed of a code which indicates the nature, ownership and or status of the file.
Flag	A label used to indicate something about data.
Flippy	A double sided floppy disc (but sometimes used as a synonym for floppy disc).
Floppy Disc	A disc made of a flexible material, eg plastic, coated with a magnetic surface. Such discs are relatively cheap to make and easy to handle; but they only store a small amount of information from approximately 100 to 1000 kilobytes.
Flowchart	A way of representing the order of a sequence of events.
Forth	A high level programming language designed to have as many commands as possible in "plain English".
FORTRAN	An abbreviation for formula translation. A high level computer language extensively used for scientific and mathematical programming.
Function code	A code which controls an operation.
Garbage	Meaningless or unwanted data.
Gate	A single logic function.
Ghost	A shadow, or weak additional image, eg on a television screen.
GIGO	An acronym for Garbage In - Garbage Out. A computing term referring to incorrect output resulting from an incorrect input.
Glitch	A spike of electrical noise that can destroy the contents of the memory.
Graphics	The use of computers to display pictorial images. A user can generate these images using either a keyboard or some special graphic input device.
Handshaking	The exchange of alerting signals between transmitting and receiving points prior to full transmission between the two.
Hard disc	A magnetic disc used for bulk storage of computer data. Hard discs offer a much larger storage capacity than floppy discs, but have to be handled more carefully.
Hardware	The mechanical, magnetic, electronic, and electrical devices which go to make up a computer.
Hertz (Hz)	Measure of frequency meaning cycles per second.

Indexing	The process by which "labels" are produced for documents, or for information.
Information Technology	The acquistion, processing, storage and dissemination of vocal, pictorial, textural and numerical information by means of computers and telecommunications.
Input	Information put into the memory of the micro usually from the keyboard.
I/O	An abbreviation for Input/Output.
Integer	A whole number.
Interface	Software or hardware used to enable the talking between a micro and a peripheral.
Joystick	A peripheral used mainly to play games.
Kilo (K)	One thousand in normal terms, but a kilobyte is one thousand and twenty-four.
Label	A character or group of characters used to identify an item of data, a record or a file.
Line number	A number used to label a line of code that has been written in BASIC.
Load	The transferral of data from an external storage medium into memory.
Local Area Network	A system which links together computers, electronic mail, word processors and other electronic office equipment to form an inter-office, or inter-site network.
Location	Same as address.
Machine Code	One of the languages resident in memory also known as assembler. It is the language used to talk directly to the computer.
Matrix	The master from which type images are formed in phototypesetting.
Mega	A prefix signifying one million.
Memory	Storage inside the 64 for data and files. The space is measured in bytes.
Memory Map	A guide to the location of various elements within the computer's memory.
Micro	Short for microcomputer. A small computer which uses a microprocessor chip.
Microelectronics	The use of integrated circuits in electronic devices.
Microprocessor	Also referred to as the CPU. The chip that acts as your computer's brain.
Microsecond	One millionth of a second.
Milli	A prefix meaning one thousandth.
Minicomputer	A computer of indeterminate size and computing power, between a mainframe and a microcomputer.

Modem	An abbreviation of modulator-demodulator. A device for converting a digital signal into an analogue signal.
Modify	To alter the address of an operand.
Modulator	A device that impresses audio or video signals onto a carrier wave.
Monitor	A visual output device.
Mouse	A device which can be moved over the surface of a graphics tablet. Its position is recorded by the computer, and can be used in moving text and illustrations about.
Network	A system of physically dispersed computers interconnected and "talking" to each other.
Null character	A character employed either to fill unused time in a data transmission or to provide a space in a storage device.
Null string	An empty string. The string must exist and it must have nothing in it.
Number crunching	Quickly performing complex calculations.
On-line	Refers to any use of equipment that interacts directly with the control processor of a computer.
Operand	The items such as addresses or data on which the computer is operating at a time.
Output	Information transmitted to the outside world.
Pack	A way of compacting information to economise on storage space inside a computer.
Page	A block of data, as displayed by the VDU. Sometimes a page is made up with several frames, or screens of data.
Paging	Switching between blocks of computer memory.
Parallel	A data transmission system where the bits representing a character are transmitted simultaneously.
Peripherals	Devices linked to the computer to enable it to gather and display information.
Pixel	The smallest area of display that the computer can control.
Port	A socket on the computer that connects the peripherals.
Printer	A peripheral used to print out your files.
Program	A set of instructions carried out by the computer.
PROM	Programmable Read Only Memory.
RAM	Random Access Memory.
Raster graphics	A number of positions on a display screen which can be defined using its raster.
Read	To copy from one storage area to another.
Register	A special storage location in the CPU which holds data on which calculations are performed.

Reserved Word	A word that has a specific meaning to the compiler such as a BASIC keyword.
Robotics	The application of artificial intelligence techniques to the design and production of robots.
ROM	Read Only Memory.
RS232	A type of interface.
Serial	To handle items, or actions sequentially.
Software	The program or set of instructions that are used to drive a computer.
Source Code	Code which has not been written in machine language, for example BASIC.
String	A sequence of records, words, letters or numbers.
Subroutine	A section of a program which can be called upon at any given point.
Syntax	The format that a command needs to use.
Variable	An element that can be given any name and value where the value need not remain constant.

APPENDIX 2

In this section there is a complete listing of all the **BASIC** keywords that the 64 uses along with their abbreviations, when relevant.

KEYWORD	ABBREVIATION
ABS	A SHIFT B
AND	A SHIFT N
ASC	A SHIFT S
ATN	A SHIFT T
CHR$	C SHIFT H
CLOSE	CL SHIFT O
CLR	C SHIFT L
CMD	C SHIFT M
CONT	C SHIFT O
COS	n/a
DATA	D SHIFT A
DEF	D SHIFT E
DIM	D SHIFT I
END	E SHIFT N
EXP	E SHIFT X
FN	n/a
FOR	F SHIFT O
FRE	F SHIFT R
GET	G SHIFT E
GET	n/a
GOSUB	GO SHIFT S
GOTO	G SHIFT 0
IF	n/a
INPUT	n/a
INPUT	I SHIFT N
INT	n/a
LEFT$	LE SHIFT F
LEN	n/a
LET	L SHIFT E
LIST	L SHIFT I
LOAD	L SHIFT O
LOG	n/a
MID$	M SHIFT I
NEW	n/a
NEXT	N SHIFT E

NOT	N SHIFT O
ON	n/a
OPEN	O SHIFT P
OR	n/a
PEEK	P SHIFT E
POKE	P SHIFT O
POS	n/a
PRINT	?
PRINT	P SHIFT R
READ	R SHIFT E
REM	n/a
RESTORE	RE SHIFT S
RETURN	RE SHIFT T
RIGHT$	R SHIFT I
RND	R SHIFT N
RUN	R SHIFT U
SAVE	S SHIFT A
SGN	S SHIFT G
SIN	S SHIFT I
SPC	S SHIFT P
SQR	S SHIFT Q
STATUS	ST
STEP	ST SHIFT E
STOP	S SHIFT T
STR$	ST SHIFT R
SYS	S SHIFT Y
TAB	T SHIFT A
TAN	n/a
THEN	T SHIFT H
TIME	TI
TIME$	TI $
TO	n/a
USR	U SHIFT S
VAL	V SHIFT A
VERIFY	V SHIFT E
WAIT	W SHIFT A

APPENDIX 3

ERROR MESSAGES

This appendix gives a list of all the error messages that are thrown up by the 64 when it encounters a programming error. These messages are intended to help you locate the error in question indicating the type of error you are looking for, and in most cases the line number. Let's move on and take a look at all of the error messages.

BAD DATA

This message is used to indicate that the data read from a file was not the type expected, for instance string data was found when numeric data was specified.

BAD SUBSCRIPT

It is used to indicate that an array was wrongly dimensioned, for instance, if you create a multi-dimensional array when you really needed a single dimensioned array. This error message is also thrown up if you try to access an array element that is outside the dimensions of an array.

BREAK

This message is displayed when the **RUN/STOP** key has been pressed to halt a program or when the **BASIC** keyword **STOP** has been used.

CAN'T CONTINUE

This error message is displayed when you attempt to **CONT**inue a program that has had alterations made. You can never **CONT**inue a program after you have made any alteration to the listing.

DEVICE NOT PRESENT

This error message is displayed when an attempt is made to use a peripheral that you have forgotten to plug into the 64!

DIVISION BY ZERO

This message is displayed if you are attempting to divide any number by zero as it is impossible to divide by zero.

EXTRA IGNORED

This message is displayed as a response to an **INPUT** statement where too

many items have been entered, the extra data is ignored. This is one of the few error messages that does not halt a program.

FILE NOT FOUND

This message is displayed when you are trying to **LOAD** a file into the memory of the 64, when it encounters the end of file marker on a cassette system. If this message is displayed when you are using a disc system it means that the specified file does not exist on the disc.

FILE NOT OPEN

This message is displayed when a file has been specified in conjunction with a file handling command where the file in question has not yet been **OPEN**ed.

FORMULA TOO COMPLEX

Is used to tell you that a series of string handling functions are too complex to be handled at one time, so the user has to split the expression into smaller steps.

ILLEGAL DIRECT

This message is displayed to indicate that one of the following commands have been used in Immediate mode which is contrary to the laws of 64 **BASIC: DEF FN, GET,GET ,INPUT,INPUT**

ILLEGAL QUANTITY

This message is displayed along with the number of the line that the error was encountered. A value given to the parameter of a mathematical or string function that was outside the legal range.

LOAD

This error message is displayed when the 64 has attempted to **LOAD** a file from tape but been unsuccessful thus aborting the attempt.

NEXT WITHOUT FOR

This error message is self-explanatory really, it is thrown up when the 64 encounters a **NEXT** statement that does not have a corresponding **FOR** statement, or when **FOR. . .NEXT** loops have been nested incorrectly.

NOT INPUT FILE

This message is displayed when an attempt has been made to read data from a file that has been made into a write only file.

NOT OUTPUT FILE

This message is displayed when an attempt has been made to write data to a file when the file has been set up to read only.

OUT OF DATA

It occurs when a **READ** statement is executed and there is no **DATA** left to **READ.**

OUT OF MEMORY

This error message is thrown up for a variety of reasons: either the program is too large to fit in the memory, there are too many **GOSUB**s, too much space has been allocated for arrays, or there are too many **FOR. . .NEXT** loops.

OVERFLOW

This error occurs when the result of a calculation is too large for the 64.

REDIM'D ARRAY

This error is displayed if you have broken a cardinal rule and **DIM**ensioned an array more than once.

REDO FROM START

This is another of the error messages that does not halt the execution of a program, it is displayed when the program is executing the **INPUT** statement and the wrong type of **INPUT** has been entered; when numeric data is expected and string data has been entered.

RETURN WITHOUT GOSUB

This error message is fairly self-explanatory, it is displayed when the 64 encounters a **RETURN** statement which is not preceeded by a **GOSUB.**

STRING TOO LONG

This error message occurs when a string is longer than 255 characters after concatenation (several strings have been joined together).

SYNTAX

The **SYNTAX** error message can occur for a variety of reasons: either missing brackets, incorrect punctuation, missing parameters, illegal characters or mispelt (or inaccurate) **BASIC** words. The most common typing mistakes tend to be the confusion of the following characters: I and 1, l(L) and 1, 0(zero) and O, :(colon) and ;(semi-colon).

TYPE MISMATCH

This error occurs when a variable has been assigned to the wrong type, that is, a numeric variable has been assigned to a string of characters or vice versa. It can also occur when a numeric argument was passed to a function that required a string argument.

UNDEF'D FUNCTION

This error message occurs when you have made a reference to an undefined user-defined function.

UNDEF'D STATEMENT

This error occurs if you try to **RUN, GOTO, GOSUB,** or **THEN** a non-existant statement or line number.

VERIFY

The **VERIFY** error message is displayed if there is a difference between the program on tape and the one in memory.

APPENDIX 4

CASSETTE HANDLING

When you key in a program it is stored in the memory of the 64, however, as switching off the machine erases the memory we need a means of storing the program. In order to do this we need to store the file on some medium that is external to the 64 itself.

There are two ways of doing this: on a disc or cassette. Discs are the most efficient way of storing your files, they are reliable, fast and easy to handle, but they are very expensive.

The Commodore cassette recorder is the only one that is compatible with the 64 and, fortunately, it is very reliable. This said, it is possible to buy an interface that will make most cassette players compatible with the 64. However, this is not recommended as, on the whole they do not prove to be at all reliable and in the long run it would probably work out a lot cheaper and less effort to buy the standard Commodore cassette recorder.

To connect it up to the 64, plug it in to the socket second to the left on the back of the micro. You will see that there is no power connector for this piece of equipment, this is because it is powered by the micro itself.

When you tape a record the microphone picks up the music and stores it on the magnetic tape used by the cassette and when you want to listen to it on your hi-fi the music is sent down the wire to the speaker where it is turned into music that we can appreciate. Storing a program on cassette works on much the same principle; the file is first "recorded" onto the tape (this process is referred to as **SAVE**ing a program), when we want to put the file back into memory we press the play button on the cassette player and the file is sent down the wire connected to the 64 where it is turned back into a program (this process is called **LOAD**ing a program).

Before we move on to look at how programs are **SAVE**d and **LOAD**ed a few words need to be said about the cassettes that should be used. You should never try to save a few pennies when you go out to buy a cassette, as this small saving could cost you dearly. All micros are very fussy when it comes to **LOAD**ing a program and if there are any faults with the cassette itself they will refuse to accept a file, so if you use cheap cassettes you are very likely to lose some of your much loved programs.

It is also a false economy to buy C120's, C90's or C60's as by filling up 1/2 such large cassettes with files you are endangering the lives of numerous programs should anything go wrong. The cassettes to use are the small data cassettes such as C5's, C12's, C15's or C20's and if you only put one

program on each cassette you will only lose one program if the tape proves to be faulty or if the tape corrupts.

This said, it is always wise to have one large cassette on hand (such as a C60) when you are **SAVE**ing a very long program just in case it is too large to fit onto a smaller cassette.

The final word that needs to be said when using cassettes is that it is always advisable to make a back up copy of your program, that way it is very unlikely you will lose a program because you will always have two copies of it.

The 64 uses three commands **LOAD, SAVE** and **VERIFY.** Let's run through them one by one.

LOAD

This command is used to **LOAD** programs into the memory of the 64 and uses the syntax:

LOAD

or

LOAD "filename"

The first of these two formats is used when either you don't know the exact name of the program or if you want to **LOAD** the first program on the tape. It is also wise to use this format if you are not sure of the name of the program in question, if you try to **LOAD** a program using an inaccurate filename the 64, quite simply, won't **LOAD** anything.

The second format is used when you know the name of the program that you want to **LOAD.** It is useful to use this format when you have a number of programs on one cassette as by using this format you are asking the 64 to **LOAD** a specific file. It will run through the tape looking for the specified filename and when it is found will **LOAD** the program into memory.

If you want to **LOAD** the first program on tape there is an alternative method of entering the **LOAD** command by pressing the **RUN/STOP** and **SHIFT** keys simultaneously using this method the program, once **LOAD**ed, will **RUN** automatically.

Before you press the <**RETURN**> key make sure that the tape has been fully rewound, now press <**RETURN**> and the message "PRESS PLAY ON TAPE" is displayed on the screen. Do as instructed, the screen is cleared and when it finds a program the following message is displayed;

SEARCHING FOR "filename"
FOUND "filename"

or

SEARCHING
FOUND "filename"

The first message appears when you specified the filename and is displayed everytime the 64 encounters a program until it finds the desired file which is then **LOAD**ed into memory. The second message appears when **LOAD** was used without a filename.

The screen clears once again as the program is being **LOAD**ed into the memory. When this process is completed the familiar **"READY"** prompt reappears and the motor in the cassette recorder stops automatically. However, if you used the **RUN/STOP-SHIFT** syntax the program will **RUN** as soon as it has **LOAD**ed.

If a program won't **LOAD** for some reason or another, one of two things happen; either an error message is displayed or the screen remains blank indefinitely. Should this happen press the **RUN/STOP** key and you will break into the **LOAD**ing process.

SAVE

This command is used to store or **SAVE** the contents of the memory onto cassette using the syntax:

SAVE "filename"

Before you **SAVE** a file it is important to make a routine check, firstly that the cassette has been fully rewound (presuming that it is the first program to be **SAVE**d onto the cassette), once you have rewound the cassette you must ensure that the header or leader of the tape is past the heads. The header is usually transparent and non-magnetic so if you try to **SAVE** a program onto it you will not succeed.

You will notice that we have only given one syntax for this command –**SAVE "filename"** – there are in fact two (as for **LOAD**). You can give the instruction **SAVE** on its own but this is not to be recommended. It is very important that every program has a name, that the tape is clearly labelled and, if possible, that you keep an index of your tapes. It is also wise when labelling your tapes to make a note of their start and end point indicated by the counter on the top of the cassette recorder.

Suppose you had broken the cardinal rule and **SAVE**d numerous programs on one tape without using a filename, wanted to **LOAD** a particular program and weren't sure of its position on the tape. It would drive you mad having to type in the **LOAD** command, starting at the beginning of the tape, then, once the first program had **LOAD**ed you would have to **RUN** it to see if it was the one you wanted (it wouldn't be of course) so you would have to repeat this process until you found the desired program, which inevitably would be the last one on the tape. However, if you had given all the progams a filename you could have found the program in question by using the **LOAD** "filename" syntax.

Once the tape is in the correct position press <**RETURN**> and the message "PRESS RECORD & PLAY ON TAPE" is displayed, once again as directed. The screen is then cleared and the motor in the cassette recorder stops as soon as the **SAVE**ing process has been completed.

It is always wise to make a back up of each program (preferably on a different tape) to ensure its safety.

Right so you've **SAVE**d your program but how do you know that the **SAVE**ing process was a success? Well you don't, but you can check by using the **VERIFY** command.

VERIFY

This command compares the program that has been **SAVE**d on cassette to the program in the memory using the syntax:

VERIFY

or

VERIFY "filename"

It is always sensible to **VERIFY** a program after it has been **SAVE**d, you will probably find it a bit of a chore, but it is a chore that's worth the effort: taking five minutes of your time to **VERIFY** a program is considerably less effort than having to write the program again.

The first syntax is used when you know where your program began and the second if the filename was ommitted from the **VERIFY** command. If all goes well "OK" is printed to the screen followed by the **"READY"** prompt, however, should the **VERIFY**ing process fail the message **"VERIFYING ERRORS"** will appear. Should this happen re-**SAVE** the program and try to **VERIFY** again.

The last thing that remains to be said on the subject of cassette storage is about the cassettes themselves. We stated earlier in the chapter that the 64 was very fussy about **LOAD**ing programs, because of this it is very important that you adhere to the following guidelines when handling cassettes and the recorder.

a) Never touch the magnetic tape itself as this could destroy or corrupt the program.
b) Ensure that the heads on the recorder are kept clean. If they are allowed to get a build up of magnetic dust and other grime it will gradually become impossible to **LOAD** programs successfully.
c) If you want to protect a program and ensure that the cassette is not accidentally recorded on, remove the two tabs on the top of the cassette. However, if at a later date you decide to use the cassette again put some sticky tape over the holes left by the tabs.
d) Always put a cassette back in its box after use.

APPENDIX 5

SOUND CHIP REGISTERS

BYTE	DESCRIPTION
00	Low Frequency value of note for voice 1
01	High Frequency value of note for voice 1
02	Low Pulse Rate for voice 1
03	High Pulse Rate for voice 1
04	Waveform for voice 1
05	Attack/Decay for voice 1
06	Sustain/Release for voice 1
07	Low Frequency value of note for voice 2
08	High Frequency value of note for voice 2
09	Low Pulse Rate for voice 2
10	High Pulse Rate for voice 2
11	Waveform for voice 2
12	Attack/Decay for voice 2
13	Sustain/Release for voice 2
14	Low Frequency value of note for voice 3
15	High Frequency value of note for voice 3
16	Low Pulse Rate for voice 3
17	High Pulse Rate for voice 3
18	Waveform for voice 3
19	Attack/Decay for voice 3
20	Sustain/Release for voice 3
21	High Frequency Cut-Off
22	Low Frequency Cut-Off
23	Turn on filtering
24	Set volume for all three voices plus select filter type
25	Access to Output of envelope generator of voice 3
26	Digitised output from voice 3
27	Digitised output from envelope generator 3

APPENDIX 6

SPRITE MEMORY DIAGRAM

ADDRESS	DESCRIPTION	(53248+1)
00	X position of sprite 0	
01	Y position of sprite 0	
02-15	Ditto for sprites 1 through to 7	
16	Most Significant Bit of X position	
17	Most Significant Bit of Y position	
18	Raster Register	
19	X position of Light Pen	
20	Y position of Light Pen	
21	Turn sprite on	
22	Selects Multicolour Mode & Scrolls Screen	
23	Expand Sprite in Y direction	
24	Memory Pointers	
25	Interrupt Register	
26	Enable Interrupt	
27	Sprite Data Priority	
28	Multi-colour Sprites!	
29	Expand Sprite in X direction	
30	Sprite to Sprite collision	
31	Sprite Data collision	
32	Exterior Colour	
33	Background Colour 0	
34	Background Colour 1	
35	Background Colour 2	
36	Background Colour 3	
37	Sprite Multi-Colour 0	
38	Sprite Multi-Colour 1	
39	Colour for Sprite 0	
40-46	Ditto for sprites 1 through 7	

APPENDIX 7
ASCII AND CHR$ CODES

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0		17	"	34	3	51
	1		18	#	35	4	52
	2		19	$	36	5	53
	3		20	%	37	6	54
	4		21	&	38	7	55
	5		22	·	39	8	56
	6		23	(	40	9	57
	7		24	)	41	:	58
DISABLES 	8		25	*	42	;	59
ENABLES 	9		26	+	43		60
	10		27	,	44	=	61
	11		28	–	45		62
	12		29	.	46	?	63
RETURN	13		30	/	47	@	64
SWITCH TO LOWER CASE	14		31	0	48	A	65
	15	SPACE	32	1	49	B	66
	16	!	33	2	50	C	67

D	68		97		126	Grey 3	155
E	69		98		127		156
F	70		99		128		157
G	71		100	Orange	129		158
H	72		101		130		159
I	73		102		131	SPACE	160
J	74		103		132		161
K	75		104	f1	133		162
L	76		105	f3	134		163

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
M	77		106	f5	135		164
N	78		107	f7	136		165
O	79		108	f2	137		166
P	80		109	f4	138		167
Q	81		110	f6	139		168
R	82		111	f8	140		169
S	83		112		141		170
T	84		113		142		171
U	85		114		143		172
V	86		115		144		173
W	87		116		145		174
X	88		117		146		175
Y	89		118		147		176
Z	90		119		148		177
[	91		120	Brown	149		178
£	92		121	Lt. Red	150		179
]	93		122	Grey 1	151		180
↑	94		123	Grey 2	152		181
←	95		124	Lt. Green	153		182
	96		125	Lt. Blue	154		183
	184		186		188		190
	185		187		189		191

CODES	**192-223**	**SAME AS**	**96-127**
CODES	**224-254**	**SAME AS**	**160-190**
CODE	**255**	**SAME AS**	**126**

APPENDIX 8
SELECTED MEMORY LOCATIONS

ADDRESS	DESCRIPTION
0 & 1	—6510 Registers.
2 up to: 1023	—Start of memory. —Memory used by the operating system.
1024 up to: 2039	—Screen memory.
2040 up to: 2047	—SPRITE pointers.
2048 up to: 40959	—This is YOUR memory. This is where your BASIC or machine language programs, or both, are stored.
40960 up to: 49151	—8K CBM BASIC Interpreter.
49152 up to: 53247	—Special programs RAM area.
53248 up to: 53294	—VIC-II.
54272 up to: 55295	—SID Registers.
55296 up to: 56296	—Color RAM.
56320 up to: 57343	—I/O Registers. (6526's)
57344 up to: 65535	—8K CBM KERNAL Operating System.

INDEX

WHETHER YOU ARE A COMPLETE NEWCOMER TO COMMODORE COMPUTING, OR AN EXPERIENCED USER, YOU WILL FIND THAT THIS BOOK CONTAINS ALL YOU NEED TO KNOW TO MAKE FULL USE OF YOUR COMMODORE 64

FULL COVERAGE OF COMMODORE BASIC
COMPREHENSIVE KEY WORK SECTION
DATA STORAGE
SOUND and GRAPHICS
HIGH RESOLUTION SCREENS AND SPRITES
COMPLETE ERROR MESSAGE SECTION
FULL INDEX

THIS IS

THE COMPLETE GUIDE

FOR THE COMMODORE 64

PHOENIX
PUBLISHING
ASSOCIATES

£9.95

ISBN 0-9465-7621-1

9 780946 576210 90000