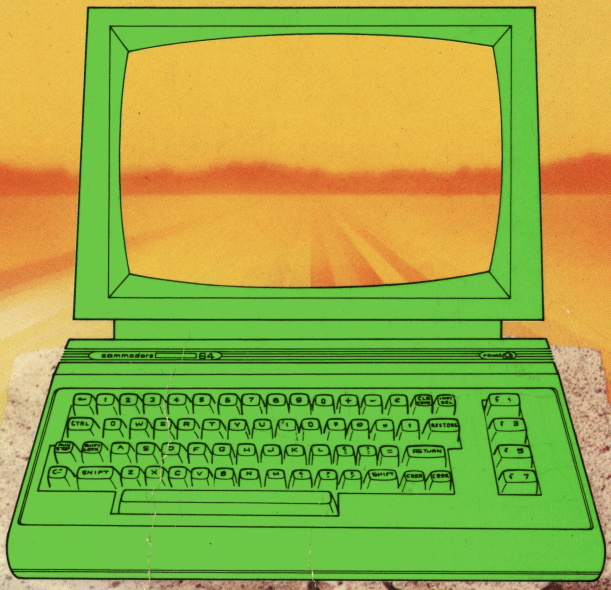
USING THE 64



Peter Gerrard

Using the 64

USING THE 64

Peter Gerrard



Duckworth

Second impression November 1983
First published in September 1983
Gerald Duckworth & Co. Ltd.
The Old Piano Factory
43 Gloucester Crescent, London NW1

© 1983 by Peter Gerrard

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publisher.

ISBN 0 7156 1777 X

British Library Cataloguing in Publication Data
Gerrard, Pete
Using the 64.
1. Commodore 64 (Computer)
I. Title
001.64'04 QA76.8.C/
ISBN 0-7156-1777-X

Typeset by The Electronic Village, Richmond
Printed in Great Britain by
Redwood Burn Ltd., Trowbridge
and bound by Pegasus Bookbinding, Melksham

Contents

1. An Introduction to Basic Programming 1
2. More Basic Programming 41
3. An Introduction to Machine Code Programming 69
4. Colour on the Commodore 64 103
5. High Resolution and Sprite Graphics 108
6. Sound on the Commodore 64 148
7. Peripheral Support 175
8. 6500 Chips in General and the 6566 in Particular 201
9. The 6581 Sound Interface Device 229
10. The 6510 and 6526 Input/Output Chips 250

Appendixes 273

Index 327

Preface

This book is designed for those who would like to do a little more with their Commodore 64 than draw pretty patterns on the screen, or make it speak to them in a collection of weird and wonderful bleeps and burbles.

This book will show you how to do those things, but I hope that the information contained here, along with the many example programs, will help and encourage you to go further in your exploration of the Commodore 64.

We commence with a swift gallop through Basic and Machine Code programming, before looking at colour, graphics and sound in some detail.

After considering the more common peripherals that can be linked up to a 64, we then move on to the four major chips within the computer, namely the 6566 Video Interface Chip, the 6581 Sound Interface Device, the 6510 processor itself, and the 6526 Complex Interface Adaptor.

Finally, the comprehensive appendixes should tell you all you need to know about Basic, Machine Code, and the inner workings of your machine. They also include a number of useful charts and a complete listing for an assembler/disassembler for the 64 (thanks Jim!).

P.G.

I'd like to dedicate this book to my gran! She may be a good deal more than 64, and she definitely won't be persuading any one to buy a copy of this book, but she's a good 'un.

Long may she remain so.

1
An Introduction to Basic Programming

Introduction

Learning to program a computer using the language known as Basic is not as daunting a task as you might at first imagine.

Most home computers nowadays come equipped with a version of Basic built into them, and there are many reasons why this particular language has been chosen.

First and foremost, it is easy to learn, and you'll be writing programs in Basic long before the end of even this first chapter.

Basic, after all, stands for Beginners All-purpose Symbolic Instruction Code. As a beginner, it is probably the easiest of all languages to use and understand.

If it helps, treat the learning of this, or indeed any other, computer language as akin to learning a language other than your native tongue. First of all, you will have to learn the words that you can use when talking to your computer, and later you'll begin to start stringing these words together to form sentences, or lines in a computer program.

This analogy mustn't be taken too far. Those of you who had the misfortune to study Latin or Greek at school will be relieved to know that there are no past or future tenses to grapple with here. Computers aren't that intelligent yet!

Basic on the Commodore 64

The version, or dialect, of Basic used on the Commodore 64 is

Commodore's own implementation of another version called Microsoft Basic.

This is very similar to that used on other computers, and so one valuable source of program listings that you could type in and use on your machine is those published in the popular computing press.

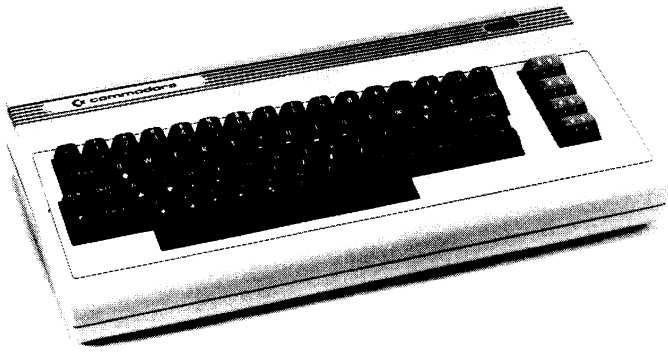
Of course, programs for different makes of computers will not use any of the features of the 64 such as sound and sprite graphics.

However, you've got to learn to walk before you can run, and so we too will steer clear of the more advanced features for a few pages yet.

The Keyboard

Programs are typed in, naturally enough, via the keyboard of the 64. This keyboard has a number of interesting features, as well as being one on which typing comes remarkably easy.

So different from the flat, membrane keyboards found on other machines!



The standard alphabet is there, in the usual "QWERTY" format, with a row of numeric keys along the top. To the right are the four function keys, and later we'll give you a number of sample programs for using those. For now, they'll have to remain a mysterious set of four keys!

There are four keys dedicated to moving the cursor around, the cursor

being the word used for the flashing little blob that greets you when you turn the machine on.

The two keys at the bottom right, the ones with the arrows on, are used for moving the cursor about the screen. Normally they would move the cursor down and to the right respectively, but with the shift key pressed down they will then move up and left.

The two keys at the top right of the keyboard have two very different functions.

The one marked CLR/HOME normally just moves the cursor to the top left hand corner of the screen, but if you press the shift key down as well, it will also wipe out any text that happened to be on the screen at the time. To be treated with caution, until you get used to using it.

The one marked INST/DEL is used for INSerTing characters or DELeting them. Used normally, it will move the cursor to the left, wiping out any characters it meets en route. If you press the shift key down as well, the cursor will stay where it is, and all text after it will move along to the right, leaving blank spaces between the cursor and it.

You can only insert up to a maximum of 80 (eighty) characters. This is because, as we will explain more fully later, the Commodore 64 can only accept program lines up to that same maximum length (the equivalent of two full screen lines, the screen being 40 characters across).

You may, if you've inserted a few characters, notice something strange happening if you then try to move the cursor about. Instead of the cursor going where you want it to go, odd characters start appearing on the screen.

This is because you've entered 'cursor control' mode. This is usually used in conjunction with the PRINT statement, and as we will see can be a useful tool when printing information out onto the screen. For now, pressing SHIFT and RETURN at the same time will get you out of this, and move the cursor down to the start of the next line on the screen.

The RETURN Key

This is our most powerful weapon, and is located to the right of the keyboard. Without this key the computer would be something which

could display information on the screen, and nothing more.

The Return key is the one used to tell the computer that we want it to accept (or listen to, in our language analogy) what we've just typed onto the screen.

Of course, getting it to accept it is one thing, but getting it to understand it is something else entirely!

For instance, if you typed HELLO MY NAME IS JIM onto the screen, and then pressed Return, the Commodore 64 would reply instantly with the words ?SYNTAX ERROR

This is one of a list of 28 error messages (they're all listed in appendix B for handy reference) designed to make it easier for you when writing programs.

Put simply, it means that you've made an error in your syntax, and the 64 doesn't understand what you mean.

However, we'll have you both conversing properly in time!

The RESTORE key

The RESTORE key, located just above Return, will do nothing when used on its own, but does have one function when used in conjunction with the RUN/STOP key over to the left of the keyboard.

When pressed together these two keys have the effect of switching the machine off and on again, and save wear and tear on the on/off switch.

More commonly, they're used when the program you're running gets hopelessly lost and jammed up, and the 64 refuses to co-operate.

Computer people will tell you that this is called 'crashing the machine', and it is just as disastrous as crashing anything else. Talking cars there may be, but they haven't given them Restore keys yet, so take care!

The RUN/STOP key

The Run/Stop key, like most of the others, has a dual purpose. If pressed normally, it will halt the execution of any program that is

running in the computer at the time, and the computer will obligingly respond by telling you 'break in xxxx', where xxxx was the line number the computer was working on when you pressed the key.

There are ways around this, which will stop other people interfering with your programs, but they'll have to wait till chapter 2.

Run/Stop has another function as well. Pressing shift and run/stop at the same time will produce the message 'Press Play on Tape' on the screen. This is a shorthand way to load a program that has been previously stored on tape.

When the program has finished loading, it will automatically run. It saves you time, if nothing else.

The Commodore Logo Key

The Commodore logo key, down at the bottom left of the keyboard, acts almost like a second shift key when used in conjunction with the alphabetic keys on the keyboard.

You'll have noticed (haven't you ?!) that all the alphabetic keys, the mathematical operators (plus, minus, divide and multiply), as well as the pound sign and the 'at' sign, have two sets of symbols in front of them.

If you press the shift key and one of the above keys at the same time you'll get the symbol display on the left hand side of the appropriate key.

Press the logo key and the same key again, and you'll get the symbol shown on the right hand side of that key.

Using these symbols a wide variety of graphical displays can be made up.

As these symbols are identical to the ones found on all the earlier Commodore computers (collectively referred to as the 'PETs'), many of the early programs written for those machines will also work on your Commodore 64.

Pressing shift and logo at the same time will send you into what is known as Upper/Lower case mode. Now, if you type normally, you'll see the letters coming out in lower case, and you'll only get capitals

by pressing the shift key.

Press shift and logo again to get back to normal graphics mode.

The CONTROL Key

The logo key has another purpose in life, and for this we come to the CONTROL key, situated just above it.

The numeric keys 0 through 9 all have words written on the front of them, and for the numbers 1 to 8 these are abbreviations for eight of the sixteen different colours that can be achieved on your Commodore 64.

Pressing Control and 1, for instance, will change the text colour to black, Control and 2 will change it to white, and so on.

If you press the logo key and 1, you now get a totally different colour. Actually, it's a fairly reasonable orange shade.

This works for all of the numbers from 1 to 8. Control and the number produces the first eight colours, whilst logo and the number produces the next eight.

You will see that the numbers 9 and 0 have the words 'RVS ON' and 'RVS OFF' written on them. RVS is an abbreviation for ReVerSe, and pressing Control and 9 turns reverse mode on.

In other words, if you were previously typing in black on a white background, you'll now be typing in white on a black background.

Pressing Control and 0 gets us back to normal again.

The SHIFT/LOCK key

Finally, SHIFT/LOCK operates exactly as on a conventional typewriter: it permanently shifts everything until you press the key again and thus release it.

Okay, now you know your way around the keyboard, let's start using that knowledge to begin writing some programs!

Line Numbers

Computers, like good computer programmers, are logical in their ways and thoughts, and so things have to be presented to them in a similarly logical manner.

Unfortunately, computers are also inherently stupid, and won't do anything until you tell them what to do and how to do it.

This is where line numbers come in.

If you glance at some of the programs later on in this book you will see that most of them always have a number at the left hand side of the page, and these numbers increase as you go down the page.

These numbers are in fact line numbers, and are used to tell the computer what is happening, and more importantly, where it is happening.

On every line you will see a collection of words, letters and numbers, and as we will see it is these that tell the computer what we want it to do.

Some of these words are real English words (PRINT, for instance, when you want to PRINT something on the screen), while others must remain a mystery for a little while yet.

If you glance back at those listings again, you will see that, although the line numbers are always increasing, they do not always do so in an orderly progression.

This is because computers don't particularly care what the numbers are as long as they're going up.

A good practice, as a programmer, is to have these numbers increasing in steps of ten. In other words, the first line in our program becomes line 10, the second one line 20, and so on. Not only does it look neater, but it allows you to correct any mistakes with a lot more ease than if you'd started off at line number 1, and gone up in steps of one.

To see why this is so, we'll start writing our first program.

Printing, and Some Simple Calculations

Not everything that you type into your computer has to be done as a program.

It can, for example, be used as a very powerful (and very expensive) calculator.

If we want to work out a simple calculation, such as 5 plus 5, you might try and type in :=

5 + 5 =

Well unfortunately this doesn't work, and we have to tell the 64 that we want it to print out the result.

So, how about this :=

PRINT 5 + 5 =

Try it and see, but we're still not getting the right answer!

The correct syntax to use for this, and all other calculations, is as follows :=

PRINT 5 + 5

The answer of course is 10, and this is displayed on the next screen line. As ever, the word READY appears (the computer is ready for us to do something else), and the little cursor re-appears.

A useful shorthand for PRINT is the question mark symbol '?'. Thus the above calculation could have been written as :=

? 5 + 5

Remembering that the symbols to use are + for plus, - for minus, * for multiply and / for divide, we can get a little more complicated and try things like :=

? 5 * 6 + 4

The answer is 34, but it could have been 50: 5 multiplied by 6 + 4,

in other words 5 times 10.

How do we decide in what order a calculation will be carried out ?

The Commodore 64 always does things in the same way : first, evaluate any minus signs, second, deal with any exponentiation (the up-arrow key), third, multiply and divide from left to right, and finally, add and subtract from left to right.

If you want to change this, we have to start using brackets to break up the order of calculation. For instance, if we really had wanted the answer to the above sum to be 50, we could have typed : =

? 5 * (6 + 4)

Calculations in brackets are always carried out first.

We can of course print words onto the screen as well, but words have to be enclosed in quotation marks (") to distinguish them from anything else. Thus, we might have : =

? "MY NAME IS JIM"

And pressing Return will cause the computer to print MY NAME IS JIM on the screen, and then wait for you to do something else.

If you want to combine the two types of print statements, we might have : =

? "6 times 5 equals ";6*5

Everything in quotes is printed exactly 'as is', and then the expression evaluated. The semi-colon signifies that no gap is to be left between the message and the calculation. If we'd used a comma (the only other separator) instead of a semi-colon, we'd have skipped across ten columns instead of none.

We'll now print out a few things from a program.

Program Editing

As we've seen, programs consist of line numbers, followed by a lot of strange statements.

Sticking to our simple PRINT command, try typing in the following program exactly as shown, remembering to hit Return at the end of each line.

```
10 PRINT "HELLO THERE."
20 PRINT "WATCH THIS ..."
30 PRINT "9 TIMES 9 = ";9*9
40 PINT "4 TIMES 4 = ";4*4
50 PRINT "GOOD STUFF, EH ?"
```

A fairly inane example, but it will show the concept of program editing.

Try running this program, using the command RUN (followed by Return), and you'll find that the first three lines are printed out correctly, but the program then stops with the message ?syntax error in 40.

So you see, it wasn't a deliberate mis-print! How do we get the program to work correctly ? The command LIST (Return) displays all our program on the screen again.

Now use the cursor keys to move the cursor about until it's sitting over the letter I in PINT. Hold the Shift key down and press the Inst/Del key once, then release both keys. A gap of one character has opened up, and we can now type in the missing letter R. Do that, then press Return again. This tells the computer that really we want the program line to look like this.

Return does not have to be pressed at the end of a line, it can be pressed anywhere on that line, to enter it into the machine.

Using the cursor keys to get the cursor down below the program again, if we type LIST we will see that the program has indeed been corrected, and now runs properly.

If we'd typed PRIINT in, instead of PRINT, moving the cursor until it was over the N and pressing Inst/Del once (no Shift key this time, we're deleting characters) would have removed the offending extra I.

The Commodore 64, in common with other Commodore machines, has probably got the finest program editing facilities of any computer currently on the market, and it's worth making a few mistakes to find out how good it really is.

Adding and Deleting Lines

You'll have noticed that our line numbers went up in steps of ten. If we want to add a new line, say line 15, all we do is type in : =

```
15 PRINT "HELLO I'M A NEW LINE"
```

and hit Return, and the 64 slots it into its correct place in the program. Try LISTing the program again, and you'll see what I mean.

If you want to get rid of a line from your program, all you need to do is type the line number, and Return, and it's gone. So, if we type in : =

40

and Return, LISTing the program will reveal that there is no longer a line 40 anywhere.

If you want to duplicate a line of a program, move the cursor to the line to be duplicated, say line 50 in our little example, and type the number 60 over the number 50.

Listing the program again will show you that we now have a new line 60, which is identical to line 50.

Easy, isn't it?

Now we'll carry on and learn some more words that the 64 can understand. Each one will have a sample program with it for you to type in and run, which will give you a better understanding of how that command works.

These are not presented in alphabetical order, but instead in the way that most newcomers to programming would find easiest to understand.

If you're after a specific command, the index should enable you to find the one you want quickly enough.

Input

This is simply a way of typing in information, from a program, that

the program will understand and then use in the rest of the program.

However, before we can start using Input, we need to talk about a couple of other things: cursor control mode, and the concept of variables.

Cursor Control

By now you'll have been happily moving the cursor about by pressing the appropriate keys, but the cursor can also be made to move about from within a program.

If you type in PRINT ", and then start trying to move the cursor, you'll see that the cursor just moves to the left, leaving behind a trail of strange characters. These are known as the cursor control characters, and you'll soon get to know which one corresponds to which cursor key.

Pressing Shift and Return will always get you out of this mode, but for now we'll stay in it.

To make it easier for you to type in listings, the following symbols will be used to represent one key press of the corresponding key :=

```
[CU]    : CURSOR UP (USE SHIFT)
[CD]    : CURSOR DOWN
[CL]    : CURSOR LEFT (USE SHIFT)
[CR]    : CURSOR RIGHT
[CLR]   : CLEAR SCREEN (USE SHIFT)
[HOME]  : CURSOR HOME
[RVS]   : REVERSE ON (USE CONTROL)
[OFF]   : REVERSE OFF (USE CONTROL)
```

If ever we want you to press a key more than once, the number of times that the key must be pressed will be shown. Thus:

[5CR,CD], for example, means press the cursor right key five times, and the cursor down key once.

For instance, the following program will print HELLO in the top left hand corner of the screen, as well as clearing the screen :=

```
10 PRINT "[CLR]HELLO"
```

Variables

A variable is simply a term used to describe a number, or some text, that can be stored in the computer.

There are three different types of variables: real, string, and integer.

Real variables are just numbers, and any of the following is a legal syntax for a variable NAME :=

```
A, A5, AZ, BANANA, JAWS, etc.
```

That is, the name must be at least one letter long, and must start with a letter, and anything after that can either be a letter or a number. However, the computer only recognises the first two letters of any variable name, so that (to the computer) PLOD and PLANK mean the same thing.

If you're using longer names to make your program more readable, remember the above rule.

String variables can be numbers, letters, or a mixture of both, and the restrictions on string variable names are the same as for real variables, with one addition.

String variable names must end with a dollar '$' sign. Thus, all of the following are legal string variable names :=

```
A$, A1$, ZZ$, FRED$, etc.
```

Integer variables are again numbers, but this time without the decimal part attached.

Thus an example of a real variable might be A = 12.5, but the integer counterpart would be A% = 12. All integer variable names must end with the '%' sign.

The following would all be acceptable variables of their own individual types :=

```
A=26.45, A%=54, A$="I'M A STRING VARIABLE", A1$=
"I AM 1 2!", etc.
```

There are three variable names, known as reserved variables, that cannot be used on your Commodore 64. These are ST$ (used in input/output operations), ST (which changes during these operations), and TI and TI$, which both refer to the 64's internal clock (chapter 2). Thus you can't use names like : =

```
TIMES$, STATE, CAST, BUTI, etc.
```

as they all contain one or other of those reserved variable names. Also, logically enough, you can't use any of the reserved Basic words (see appendix B for a full list).

Back to Input

Input allows you to type some information into the computer from a program, and that information is stored as a variable. For example : =

```
10 PRINT "[CLR]HELLO, WHAT'S YOUR NAME "
20 INPUT A$
30 PRINT "HELLO ";A$
```

would allow you to enter your name, and then say hello to you.

If you tried typing in 1.45 as your name, you'd be referred to as 1.45! That's because we specified that we wanted a string to be input (A$). Try the following : =

```
10 PRINT "[CLR]HELLO, HOW OLD ARE YOU "
20 INPUT A
30 PRINT "[CD]THAT MEANS YOU'RE ";A*365;" DAYS OLD!"
```

If you'd typed in your age as FRED, the computer would have responded with ?REDO FROM START. It was expecting a number, not a string, and you'll have to type something sensible in.

If you press Return, and nothing else, the program will continue, but it will treat the string as a null one, i.e. one that contains nothing.

Input can also contain some text as well, as in the next example : =

```
10 INPUT "[CLR]HELLO, WHAT'S YOUR NAME";A$
20 PRINT "HI THERE, ";A$
```

Not only does it make the program shorter, it also makes it neater.

Stop, End and Continue

When running through a program, it is often desirable to STOP it at some point, or break into it to examine it and have a look at any 'active' variables, i.e. ones that have been defined in the program.

Unless you decide to change anything, or the program has halted because of some error in it (in which case you'll HAVE to change something!), it is possible to start the program running again from the point at which it left off.

Three commands enable you to do this: STOP, END and CONT.

STOP can be used as in the following example :=

```
10 PRINT "[CLR]"
20 A=1:B=10:C$="FRED"
30 STOP
40 A=A+1:B=B+10:C$=C$+" AND BERT"
50 END
```

When you run this program, the screen will clear, and almost immediately the program will stop with the message ?BREAK IN 30 : this is the line with the word STOP in it.

If you now PRINT A,B,C$, you will see that they have the values as listed in the program.

If you now type CONT (Return), the program will start running again, and stop almost immediately when it reaches the end. This time it won't tell you where it stopped, just that it has!

If we now print out the three variables again, we'll see that A is now equal to 2, B is equal to 20, and C$ is now equal to "FRED AND BERT".

Strings can be added just like anything else, but instead of getting bigger they just get longer!

If you keep adding strings together, you will eventually receive the error message STRING TOO LONG ERROR IN xxxx where xxxx is the line number where the string expired. The upper limit is 255 characters for any string.

In this simple example it wasn't really worth stopping the program, but in a much longer one you'll soon get to realise the value of STOP, as it tells you where the program halted execution, and CONT to get you going again.

If you do change anything after a stop, and then try to continue, the error message CAN'T CONTINUE ERROR will appear on the screen. The poor old computer is confused, and can't carry on: you'll just have to start again from the beginning.

Stops can also be used in a longer program that isn't working quite as it should do. It allows you to examine all the 'working parts' of the program, and systematically track down the error.

Data, and the Reading of it

We saw earlier that we can input information, or data, into a program by using the input statement, and of course a lot of information could be typed in just by using a lot of input statements.

However, this could get exceedingly tedious if you were using the same information over and over again, hence the need for data statements.

Here the data is typed in as part of a program, read off from within the program, and then acted upon.

Not only does it save you typing in vast amounts of data each time you run the program, but it also allows you to change just one data item, and see how that affects the rest of the program.

In this short example we'll read ten numbers, add them up and then take an average of the whole lot.

```
10 PRINT "[CLR]"
20 READ A
25 IF A=0 THEN 40
30 B=B+A
40 C=B/10
45 PRINT "[CD]THE TOTAL IS ";B
50 PRINT "[CD]AND THE AVERAGE NUMBER READ WAS ";C
60 END
70 DATA 1,2,4,5,6,7,3,35,80,43,0
```

The IF ... THEN branching statement in line 25 will be explained more fully later, but here it allows us to stop adding up numbers when we've read ten of them, and reached a number of 0 : the last data statement.

Data statements can be anywhere in a program, and if you're reading real numbers, that's what the data statements must contain. If you're reading strings, again they must contain strings. Otherwise you'll get a BAD DATA error message flung at you, and quite right too!

What you must remember is that data is read as it is encountered, so wherever it does happen to be in the program, make sure that it corresponds to what you want to read.

Also, make sure that you don't try to read more data than you've actually typed in, otherwise an OUT OF DATA error will occur.

If you try to read the same data again, another OUT OF DATA error will take place, unless you use the ....

RESTORE command

This allows you to re-read data, and takes the following syntax : =

```
55 RESTORE
56 GOTO 20
```

There are two other concepts to explain here. GOTO, which transfers program execution from one part of a program to another, will be examined in more detail later.

The other concept involves the line numbers we've used, namely 55 and 56.

You see now how, by using steps of ten in the first place, we can easily add new lines to our existing program, thus expanding it, and the 64 will happily slot them into the correct place in the program.

To finish with data for a while, here's another example that mixes strings and numeric data : =

```
10 PRINT "[CLR]"
20 READ A$,B,C,D
30 PRINT "[CD]";A$;" IS ";B;" YEARS, ";C;" MONTHS
AND ";D;" DAYS OLD!"
```

```
40 GOTO20
50 DATA PETE,25,7,3
60 DATA BERYL,24,10,0
```

When run, this will generate an OUT OF DATA error, as we send it back to line 20 to read more data that isn't there, but the concept is, none the less, a sound one.

GET, IF and THEN

The GET command can be used in many ways, and in chapter 2 we'll explore its use in filing information onto tape or disk.

For now, we'll confine ourselves to using the keyboard, where we find that GET allows us to input one character at a time, without the need to keep pressing the Return key.

The following program will illustrate this point : =

```
10 PRINT "[CLR]PRESS ANY KEY"
20 GET A$:IF A$=""THEN20
30 PRINT "[CD]YOU PRESSED ";A$;"!"
40 GOTO 20
```

A number of new ideas here.

In line 20, the line is executed as follows : =

Step 1) See if a key has been pressed on the keyboard.
Step 2) If it hasn't (i.e. if a null string, "", has been detected) then go back to the start of line 20 again.
Step 3) It has, so we fall through to line 30, which prints out which key was actually pressed.

Line 40 just sends us back to line 20 again, and waits for another key to be pressed.

The only way to stop this program is by pressing the Run/Stop key, otherwise it will loop around for ever!

We can be selective in which key we press, by only moving on if the correct one is depressed. For instance, suppose we want to halt a program until the space bar is pressed. This part of our program might look something like : =

```
100 GET A$:IFA$<>" "THEN 100
110 carry on ....
```

Here, if A$ is not equal to (the < and > keys together) a space i.e. the space bar has not been pressed, then go back to line 100 and wait until it has.

This can be extended further, for example if we want someone to make a Yes or No decision, and only want him to press the Y or N keys. There are a number of ways of doing this (although we don't recommend typing in those parts of the program that are in lower case!): =

```
100 GET A$:IFA$=""THEN100
110 IFA$="Y"THEN goto another bit of the program.
120 IFA$="N"THEN go somewhere else.
130 GOTO 100
```

So, if you press Y we go to one part of the program, N and we go to another, but if neither are pressed then we wait until one of them is.

Or how about this : =

```
100 GET A$:IFA$<>"Y" AND A$<>"N"THEN100
110 IFA$="Y"THEN goto one part of the program
120 this is what happens if A$ is equal to N
```

Here, we sit and wait till either Y or N is pressed. It takes up less program space, and is just another way of doing the same thing.

This kind of selective key pressing is one of the principal uses of the IF ... THEN statement.

Its other main role is in decision making according to the value of string or numerical variables.

Strings or numbers can be compared using the greater than '>' and less than '<' operators, which have the following connotations : =

A > B : A greater than B
A >= B : A greater than or equal to B
A = B : A equal to B
A <= B : A less than or equal to B
A < B : A less than B
A <> B : A not equal to B

Thus our program might contain a line something like :=

```
100 IF A <= B THEN 200
```

Thus, if A is less than or equal to B then we go to line 200. If A is greater than B we simply slip through to the next line of the program.

Strings are compared alphabetically. Thus "AAAA" is reckoned to be less than "ABAA", and so on, and these can also be used in IF ... THEN statements as above.

Making REMARKS

As your programs build up, you will find it more and more difficult to keep track of what's going on, however careful you are with your variable names and so on.

One extremely useful feature of Commodore 64 Basic is the ability to add remarks to program listings, and thus make them more legible to others.

Not only that, if you write a fairly lengthy program and then leave it for a few months, coming back to it after that space of time will, at best, have you struggling to remember what you were doing at the time, and at worst make it totally incomprehensible.

Hence the REMARK statement, which can be used like this :=

```
10 REM THIS IS THE FIRST LINE OF THE PROGRAM
20 REM DEFINE ALL VARIABLES HERE
30 A=1:B=12:C=50:A$="HELLO THERE"
40 REM ADD NUMBERS TOGETHER
50 D=A+B+C
60 REM AND PRINT OUT RESULT
70 PRINT "THE TOTAL IS ";D
80 REM END OF PROGRAM
90 END
```

REM can be made to stand out a lot more than this, with just a little bit of time and care :≐

```
10 REM ****************************
20 REM * THIS PROGRAM WAS WRITTEN *
30 REM *  ON 14TH SEPTEMBER 1983  *
40 REM *  IT MAY NOT DO VERY MUCH *
50 REM * BUT AT LEAST IT LOOKS OK *
60 REM ****************************
70 REM START OF PROGRAM PROPER
```

Thus important sections of a program can be highlighted and made a lot more noticeable.

REMs can also come at the end of a program line :=

```
10 PRINT "[CLR]":REM CLEAR THE SCREEN
```

as long as you remember to separate the REM from the rest of the line with a colon.

This can be used to 'temporarily' hide experimental pieces of code within a program, for bringing out later :=

```
10 PRINT "THE RESULT IS ";D:REM DUMMY RESULT,
SHOULD BE C=A*B-E
```

SUBROUTINES

Some sections of a program have to be performed time and time again, and it would become very tedious, as well as wasting a lot of memory, if you had to keep typing out the following lines every time you wanted the program to execute them.

```
10 A=B+C
20 D=E+F
30 H=A+D
40 PRINT H
50 REM GET ON WITH PROGRAM AGAIN.
```

Of course, if our program segments were only this long there wouldn't be too much trouble, but as we learn more and more commands the complexity of our programs will grow, and the need to perform repetitive calculations will grow with it.

Thus we have subroutines, lines which are used a lot within a main program, and which generally just perform one specific function.

We'll see how to 'call up' subroutines in the next couple of pages, but the point to be made here is that they too, like the rest of the program, should be REMmed.

For instance : =

```
5000 REM ************************************
5010 REM * START OF BORDER DRAWING SUBROUTINE *
5020 REM ************************************
5030 PRINT "[CLR] ++++++++++++++++++++++++++++++++
++++++++ ";
5040 A=A+1:IFA=24THEN5060
5050 PRINT"+
+";
5060 PRINT " ++++++++++++++++++++++++++++++++++++
+++";
5070 REM ************************************
5080 REM * END OF BORDER DRAWING SUBROUTINE *
5090 REM ************************************
```

I don't pretend for a minute that this is the most elegant way of drawing a border around the screen, but at least it works, using only the commands we've so far come across.

By structuring programs in this way, the REM statement becomes a powerful ally in keeping your programs neat, tidy and intelligible.

More String Commands : LEN

LEN, as you might reasonably guess, is associated with the LENgth of a string.

For instance, if we assign a string A$ to be equal to "A LONG STRING", the command : =

```
PRINT LEN(A$)
```

would return a value of 13, this being the number of characters (including spaces), contained within the string A$.

We can also assign another variable to be equal to the length of a string thus : =

```
10 A$="ANOTHER STRING"
20 B=LEN(A$)
30 PRINTB
```

Running this would give us the result 14, this being the value of the variable B, or in other words the number of characters in the string A$.

LEN comes into its own when taken in conjunction with the next set of string commands.

MID$

This is the most flexible of all the string handling commands, and is taken first because it's probably the one that you'll use most often.

Strings can be manipulated in many ways. As we've seen, they can be added up (more correctly termed 'concatenated'), and they can be compared with each other, but MID$ opens up a whole new field.

The command takes the following syntax : =

```
MID$(A$,I,J)
```

Let us take a typical example.

We'll assign the string A$ to be equal to the name of my home county, Lancashire. So, if we say A$ = "LANCASHIRE", A$ becomes a string of length 10 characters.

The command MID$(A$,I,J) takes the string A$, starts at the Ith character in that string, and takes J characters out of it.

To give a programming example.

```
10 A$="LANCASHIRE"
20 PRINT MID$(A$,4,4)
```

When run, this would print out the new string CASH : A$ is unaffected.

As with LEN, this can also be assigned to another variable. For instance: =

```
10 A$="LANCASHIRE"
20 B$=MID$(A$,7,4)
30 PRINT B$
```

would result in the string HIRE being printed out, this being the value now stored in B$.

There is one other way in which MID$ can be used, and this is to take all the characters in a string, starting from a specified point. That is, MID$(A$,I), would start at the Ith character, and take all the remaining ones.

Thus, with our string A$ = "LANCASHIRE", the command : =

```
PRINT MID$(A$,6)
```

would print out the word SHIRE.

LEFT$

Not as flexible as MID$, but none the less a command with its uses when handling strings, is LEFT$.

It is a fairly safe bet to assume that this has something to do with the left-hand side of a string, and indeed it does.

Sticking with counties, we'll assign the string A$ to equal "DEVON".

When we issue the following command : =

```
PRINT LEFT$(A$,4)
```

the result is printed to the screen as DEVO. Thus, with LEFT$ we always start at the first character in the string, and take as many characters as indicated in the argument.

So, in the following program : =

```
10 A$="DEVON"
20 B$=LEFT$(A$,3)
30 PRINT B$
```

we would get the rather strange word DEV being printed out.

As you can see, not as powerful as MID$, but not without use.

RIGHT$

Well, you'd never guess would you ?

RIGHT$ is concerned with the right-hand side of a string, and works in pretty much the same way as LEFT$.

Thus, if we assign the string A$ = "CORNWALL", the command : =

```
PRINT RIGHT$(A$,4)
```

would print out the word WALL.

As before, other variables can be assigned using this same command.

For example, the following program will define the variable B$: =

```
10 A$="CORNWALL"
20 B$=RIGHT$(A$,7)
30 PRINT B$
```

and print it out as ORNWALL.

Of course, all of these commands can be combined in many ways, to make manipulation of strings very easy.

Take the following short program : =

```
10 A$="PETER GERRARD"
20 B$=LEFT$(A$,6)
30 C$=MID$(A$,4,5)
40 D$=RIGHT$(A$,7)
50 PRINT B$;C$;D$
```

When run, this would print out : =

PETER ER GE GERRARD

To illustrate further: how about this program to reverse the direction of a word.

```
10 A$="SURFING"
20 B$=MID$(A$,7,1)
30 C$=MID$(A$,6,1)
40 D$=MID$(A$,5,1)
50 E$=MID$(A$,4,1)
60 F$=MID$(A$,3,1)
70 G$=MID$(A$,2,1)
80 H$=MID$(A$,1,1)
90 I$=B$+C$+D$+E$+F$+G$+H$
100 PRINTI$
```

When run, the word GNIFRUS is printed out.

There are much more elegant ways of doing this kind of thing, as we'll see when we encounter FOR ... NEXT loops shortly.

STR$ and VAL

Two functions which are essentially the inverse of each other, and both of which are concerned with string and numeric manipulation.

Take a number A, equal to (say) 12.123.

The command : =

```
PRINT STR$(A)
```

will print out the string 12.123, although the number A has remained the same.

This command is more useful when assigning variables, so the following program shows this in action : =

```
10 A=24.232425
20 A$=STR$(A)
30 PRINT A$
40 PRINT LEN(A$)
50 PRINT MID$(A$,1,2)
60 PRINT MID$(A$,4)
```

When run, this program will print out the following : =

```
24.232425
9
24
232425
```

So you can see, by finding the position of the decimal point, we can split a number up into its two components.

How do we do this ?

Well, one way would be to use the inverse function, VAL.

VAL takes a string, and converts it into a number. Thus, if the string A$ was equal to "10", the command : =

```
PRINT VAL(A$)
```

would print out the number 10.

If A$ = "12.123", VAL(A$) would also equal 12.123, but of course this time it would be in numerical format.

VAL comes to a halt when it comes across something that is not a number.

Thus, if A$ = "88A88B", VAL(A$) would return just 88.

We can also print out straightforward variables. That is, in the following program, we are defining the variable A to be equal to the VALue of various strings : =

```
10 A=VAL("23.23")
20 PRINTA
30 A=VAL("A")
40 PRINTA
50 A=VAL("-100.9")
60 PRINTA
```

When run, the results on the screen would be : =

```
23.23
0
-100.9
```

So, to split a number up into its component parts, we must find the decimal point by turning the number into a string, taking each number at a time until we find the decimal point, and so on. Thus : =

```
10 A=345.678
15 B=B+1
20 A$=MID$(STR$(A),B,1)
30 IF VAL(A$)=0THEN100:REM THE DECIMAL POINT
40 B$=B$+A$:GOTO15:REM KEEP ADDING NUMBERS UNTIL
WE REACH DECIMAL POINT
100 C$=MID$(STR$(A),B+1)
110 PRINT B$,C$
```

Just to explain a little :=

Line 10 : define the number
Line 15 : increment our counter
Line 20 : turn A into a string, and take one character at a time
Line 30 : is that character a decimal point (VAL(A$) equal to zero)
: if yes go to line 100
: otherwise, add the number to our string B$ (line 40)
Line 100: C$ is made equal to the string equivalent of the number, starting at the character after the decimal point.
Line 110: print out the numbers

The resulting display would read :=

345 678

Two powerful functions!

CHR$ and ASC

Another two analogous functions, again concerned with string handling, but ASC in particular assumes great importance when talking about communicating from one microcomputer to another.

ASC is short for ASCII, the American Standard Code for the Interchange of Information, although when used on the Commodore 64 it would probably be more correct to call it ComSCII, as Commodore seem to enter a world of their own when designing character sets.

A complete set of these ASCII codes is given in Appendix D, but to generate them on the screen the following syntax is used :=

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

which would return a value of 65, or:=

```
PRINT ASC(A$)
```

which would return the Ascii value of the first character contained in the string A$.

CHR$ is the opposite of this, as it returns a character whose Ascii value

is as given in the table in appendix D

For instance, we've mentioned earlier that to switch graphics modes you type the Shift and Logo keys simultaneously.

This can also be achieved with the following statement : =

```
PRINT CHR$(14)
```

Upper case is reached with CHR$(142), to save you looking it up!

Everything else, like printing in black text, printing letters, etc., can be done with the CHR$ command.

Both of these commands can again be used to define other variables. For example : =

A = ASC("A")
will put the value of 65 into the variable A, and

A$ = CHR$(13)
will put the character string 13 (in fact, a carriage return) into the string A$.

FOR ... NEXT

Where would we be without FOR ... NEXT loops ?

Although we've been instructing the computer to do the same thing a number of times over, by use of a simple incrementing variable, the 'loop' approach is far better, and far easier to operate.

For instance : =

```
10 PRINT "[CLR]"
20 FOR I = 1 TO 100
30 PRINT I
40 NEXT
```

This will just print out the numbers from 1 to 100 in rapid succession, but illustrates the point.

Line 20 is the start of our loop, and tells the computer that we want to do something 100 times. In fact, we want to print out the numbers

from 1 to 100, and as the value of I increases, it is printed out in line 30. Line 40 then tells the computer NEXT, i.e. there's more to come, and the program branches back to line 20.

It keeps on doing this until I has reached the value of 100, at which point it stops and our short program ceases execution.

Actually, I reaches the value of 101. Why ? Well, when it has the value of 100, it prints it out as in line 30, sees the NEXT statement in line 40, and increases the value of I to 101. However, on branching back the computer finds that the limit of the loop is when I is equal to 100, so it stops!

The correct syntax in line 40 should have been : =

```
40 NEXT I
```

as we can have more than one loop active at a time. Like this : =

```
10 PRINT "[CLR]"
20 FOR I = 1 TO 20
30 FOR J = 1 TO 3
40 PRINT J,I
50 NEXT J
60 NEXT I
```

The first time around, I is set to 1, and J counts through from 1 to 3. Thus the display goes something like : =

```
1        1
2        1
3        1
```

Then J has finished, so we go on to line 60, where I is incremented again, so it's back through the loop once more, for : =

```
1        2
2        2
3        2
```

and so on, until we finally reach : =

```
1        20
2        20
3        20
```

at which point everything stops.

Lines 50 and 60 could have been abbreviated to the rather more straightforward :=

```
50  NEXT J,I
```

Just make sure you keep everything in the right order, and don't have more than 26 loops in action at the same time, otherwise the computer will blow its stack (computing joke).

Loops can be made to count in steps as well, for instance :=

```
20 FORI=1TO100STEP2
30 PRINTI
40 NEXTI
```

when run, will print out the numbers 2,4,6, ... 100. We can also go backwards :=

```
20 FORI=100TO1STEP-2
30 PRINTI
40 NEXTI
```

when run, will print out the numbers 100,98,96 ... 2.

For an interesting application, using only commands we've seen so far, can you work out what this program is doing (type it in and see, if you can't!) ?

```
10 A$="ABCDEFG"
20 GETB$:IFB$=""THEN20
30 FORI=1TOLEN(A$)
40 IFB$=MID$(A$,I,1)THENPRINTB$;:GOTO20
50 NEXTI: GOTO20
```

GOTO somewhere

We've already encountered this one. Basically it sends command of a program to somewhere else within the program, or back to the same line as in the example on the previous page in line 20.

The syntax used is GOTO xxx, where xxx is an existing line number.

If it isn't, you'll get the UNIDENTIFIED STATEMENT error being thrown in your face!

As a short example :=

```
10 PRINT "[CLR]";
20 PRINT "HELLO!"
30 GOTO 20
```

When run, this just prints up hundreds of HELLO!s, until you hit the Run/Stop key.

Changing line 30 to read GOTO 10, produces a slightly flickering display.

GOSUB and RETURNing

Subroutines have been met before, as small, or maybe even large, segments of programs that have to be repeated many times.

Performing the same function over and over again is a repetitive task, and having to type the code in each time you wanted it actioned would take a lot of time, and a lot of memory.

Thus subroutines were born, and the command used to send program control to them is GOSUB xxx, where xxx is the line number at the start of the subroutine.

Once actioned, the command to send control back to the main program again is RETURN.

Great care must be taken in matching up GOSUBs with RETURNs, otherwise a RETURN WITHOUT GOSUB error will take place sooner rather than later.

As with FOR ... NEXT loops you can have up to 26 subroutines in action at the same time, but no more.

Thus you can jump about from one subroutine to another, and quite often it is necessary to do this, but it isn't really very good programming practice.

A few examples :=

```
10 PRINT "[CLR]"
20 A=5:B=10
30 GOSUB 100
40 GOSUB 200
50 GOSUB 100
60 PRINT A,B
70 END: REM IMPORTANT, OTHERWISE PROGRAM FALLS
THROUGH!
100 A=A*A
110 A=A+5
120 RETURN
200 B=B-1
210 RETURN
```

When run, the first subroutine is encountered twice, the second once only, and the resultant printout is : =

90 59

Of course, one can get a lot more complicated than this!

```
10 PRINT"[CLR]"
20 A=1:B=2:C=3
30 GOSUB100
40 GOSUB200
50 GOSUB300
60 PRINTA,B,C
70 END
100 A=A+B+C
110 GOSUB200
120 RETURN
200 GOSUB300
210 A=A+B+C
220 GOSUB300
230 RETURN
300 A=A+1
310 RETURN
```

What value will A have when this program is run ? Try it and see!

What's GOing ON

Quite often within a program, the subroutine or line number you want to go to will depend on the value of a particular variable.

This could be achieved in the following way : =

```
10 IF A = 1 THEN 100
20 IF A = 2 THEN 200
30 IF A = 3 THEN 300
40 IF A = 4 THEN 400
50 IF A = 5 THEN 500
60 etc.
```

Although this works, it could hardly be described as an elegant way of programming.

In its place we can use the ON ... GOTO command, and the similar ON ... GOSUB. As both work in the same way we'll take the former as an example, although with the latter you do have to take care over matching up RETURNs with GOSUBs.

```
10 ON A GOTO 100,200,300,400,500
```

Here, if A has the value 0, the program continues execution at line 100 onwards, a value of 1 and it goes to line 200, and so on up to a value of 4, when it goes to line 500.

A can be varied, to make it match our earlier IF ... THEN example as follows : =

```
10 ON A-1 GOTO 100,200,300,400,500
```

Thus we now have an exact match of the original program, but in four fewer lines! Now, if A equals 1 program execution continues at line 100, and so on.

This could be of great value in a menu driven program, where the user has to input any one of (say) five keys. For example : =

```
10 K$="ABCDE"
20 PRINT"ACTIVITY 'A' : PRESS A
30 PRINT"ACTIVITY 'B' : PRESS B
40 PRINT"ACTIVITY 'C' : PRESS C
50 PRINT"ACTIVITY 'D' : PRESS D
60 PRINT"ACTIVITY 'E' : PRESS E
70 GETA$
80 FORI=1TOLEN(K$)
90 IFA$=MID$(K$,I,1)THEN1000
100 NEXT I
110 GOTO70
1000 ON (ASC(A$)-65) GOTO 1100,1200,1300,1400,1500
1100 rest of program.
```

This short but effective menu option program can be incorporated in all kinds of different areas.

Positioning the Cursor

There are a number of ways of doing this, and the three commands under consideration here are TAB, SPC and POS.

Before examining each of those, there is a fourth way!

To position anything on the screen, define two strings as follows : =

```
10 A$="[39CL]":B$="[24CD]"
```

Using the LEFT$ command, we can now position anything on the screen using those two strings. 10 columns across, 12 columns down ? Easy!

```
20 PRINT "[CLR]";LEFT$(A$,9);LEFT$(B$,11);"*"
```

TAB

This operates in exactly the same way as on an ordinary typewriter. i.e. you TAB across X number of columns and then print something.

For example, TAB(10) will move you across ten columns.

So, in the following program : =

```
10 PRINT "[CLR]"TAB(10)"*"
```

the asterisk will appear in column 10. However, TAB always operates with the left hand column as its home base, so : =

```
10 PRINT"[CLR]"TAB(10)"*"TAB(20)"*"
```

will print an asterisk in column 10, and another one in column 20.

If we want to skip from one thing to another, we must use : =

SPC

This does leave a space from one printing position to the next, so that: =

```
10 PRINT "[CLR]"TAB(10)"*"SPC(20)"*"
```

will print our first asterisk in column 10, and our second in column 30.

Both of these can use variables as their argument. For instance, TAB(A), SPC(B), and so on.

POS

One of those strange commands that don't appear to do very much, POS simply returns the number of the column at which the next PRINT statement will print something. Thus : =

```
10 PRINT "[CLR]"TAB(10)"*"SPC(20)"*"POS(0)
```

will print our two asterisks, and then the value 32, as that is the next column at which anything can be printed. POS can use any number or variable as its argument: it doesn't matter what you use.

None of these work particularly well on a printer, and we'll be showing you in chapter 7 how to get around this!

RaNDom INTegers

Like most of the home computers currently available, the Commodore 64 is not without a random number generator.

Alas, like most of them it isn't particularly random, and so a few setting up operations have to be done before we can begin setting up 'genuinely' random numbers.

The syntax to be observed is RND (A), which will give a number in the range 0 to 1.

To start things off differently every time, we'll need to use the 64's internal clock (TI, remember?). Thus, our first number should be made using RND (-TI).

After that, A should remain as a positive number, otherwise a zero will return the same random number as the last one, or a negative number will start everything off again. Using the same negative number will always give us the same sequence of not very random numbers.

The INT command comes in useful here, as elsewhere. It chops off the numbers after the decimal point, basically, so INT(2.24) becomes 2, as does INT (2.89).

INT of a negative number returns the next lower number. Thus, INT (-2.24) becomes -3!

So, to generate an integer random number, we could use : =

```
10 PRINT INT(RND(-TI))
```

However, this will not be very satisfactory for generating future numbers, since RND always returns a number between 0 and 1. So, we need to scale things up a little : =

```
10 PRINT INT(RND(.5)*10+1)
```

which will produce a number in the range 1 to 10.

To generate numbers between a given range, where X is the top limit and Y the lower limit, we must use the formula : =

```
10 PRINT INT((X-Y+1)*RND(.5)+Y)
```

Card games can usefully be set up using this random number feature, as long as we check that the same cards are not dealt twice!

A New DIMension

We've already seen how numbers and strings can be stored as variables like A, A$, and so on. However, this gets a mite restrictive after a while, and we need to resort to other things. After all, there are only so many numbers in the alphabet!

Let's say that we're generating ten random numbers, and we want to store them all as variables.

We could have a very lengthy program to do this : =

```
10 A=INT(RND(.5)*10+1)
20 B= ..... etc.
```

but this is extremely space consuming, and there are better ways.

This is where arrays, otherwise called subscripted variables, come in.

The syntax for referring to these is A(0), A(1), etc., up to a normal limit of A(10), and these subscripted variables could be assigned numbers something like this : =

```
10 FORI=0TO10
20 A(I)=INT(RND(.5)*10+1)
30 NEXT I
```

Now we have the eleven different numbers stored in A(0), A(1) etc. up to A(11).

These numbers can then be selected at will. For example, PRINT (A(4)) will print the fifth number, or element, in our array A : remember that the first element is referenced as number 0.

To prove it, we could print them all out by adding to our program : =

```
40 FORI=0TO10
50 PRINTA(I)
60 NEXT I
```

The numbers in an array can be assigned to other variables (e.g. A = A(3)), or even calculated dynamically by using another variable (e.g.PRINT A(B*2)).

However, more often than not we'll be wanting to use a lot more than eleven elements in an array, and this is where the DIM statement comes in.

The syntax for this is DIM A(199), or whatever, which sets aside a certain amount of room in the computer's memory for storing all the numbers that you might be wanting to save. Whether you use them all or not, that memory is reserved, so use arrays selectively.

A useful trick, if running low on memory space, is to use something like DIM A(2), if we're only going to need a maximum of three numbers storing in the array A. Any array you refer to in your program is automatically set aside 11 elements of memory space, the equivalent

of typing in DIM ARRAY(10), and the few bytes saved might mean all the difference to the amount of program you can cram into your machine!

Arrays are not limited to one dimension either. You can dimension something as A(7,7) if you like, for instance in a chess game, where you have a board 8 squares by 8.

The elements in that array are referred to as A(1,5), A(6,3), and so on. It is helpful to think of these values as being stored in rows and columns, where the first number refers to the row and the second to the column. Thus A(5,7) is the seventh column of the fifth row. Thinking of it all as boxes of numbers, or strings, stored in rows and columns will always help when you want to reference a particular one within a program.

One of our later games (Minefield) uses this two-dimensional feature extensively, and you might care to take a look at that to see how it all works in practice. The REMs should see you through, and it's a fun game if nothing else!

Fans of Einstein will be pleased to know that you can have multi-dimensional arrays as well.

POKEing and PEEKing around

Two of the most important commands on the Commodore 64, these are used to control displays on the screen, colour, high resolution graphics, sound, and a whole host of other things.

But, as we'll be encountering them so often in the not too distant future, we won't duplicate any material here, and instead will give you just straightforward, formal definitions.

POKE takes the syntax POKE A,B, where A refers to one of the memory locations within the computer, and B is a number between 0 and 255.

Some memory locations cannot be altered (they're built into the machine, and it won't operate without them).

Random POKEing around affects computers as much as humans, and is to be avoided. You can't do any permanent harm to your computer, but you'll end up having to turn it off and on again more frequently

than you or it will like, so tread carefully!

PEEK is the opposite of this, and instead of altering a memory location, allows you to look at the contents of one.

The syntax for using this command is PRINT PEEK (A), where A is the memory location to be looked at.

As Commodore, like everyone else, has a number of secrets to protect, it won't allow you to look at certain sensitive locations just by using a simple PEEK, so don't be disappointed if you simply get a lot of zeroes, or totally random numbers.

By the end of this book, you'll be sick of the sight of PEEK and POKE!

2

More Basic Programming

Introduction

So far we've been fairly gently going through the more commonly used Basic commands on the Commodore 64, and trying as far as possible to illustrate their use with the aid of program listings.

Even a quick glance at the simple user guide that comes with the computer will have told you that there are many more commands yet to be explored, and the purpose of this chapter is to complete our round up of the Basic commands available.

At the end of the chapter, we'll present you with three complete program listings, using only commands that we've encountered in Basic, that perform a wide variety of tasks :

1) A completely general purpose input routine, that you can't drop through by pressing Return, that will only allow a pre-defined input, and that will not allow you to insert or delete characters beyond the input prompt. As most programs rely on the input of data, you should find this one of use.

2) A program that will allow you to list a program in both directions. As you know, the normal Commodore list command will only let you look at a listing in one direction: it all disappears off the top of the screen. With this routine, you can get it back again!

3) And finally a game, called Minefield. Home computers should, after all, be fun, and in this game you have to steer a passage through the (invisible) mines to reach the top left hand corner of the screen. No sound, since we haven't looked at that yet, no sprites for the same reason, and only limited colour from the use of the Control key. I don't intend to present you with anything that you won't understand! Still, it's fun, and after you've finished reading the book I'm sure you'll be able to improve upon it.

The next section in this chapter deals with the WAIT command, often mentioned but not often used or understood. It can perform a wide variety of functions.

WAITing Around

The WAIT command performs exactly the sort of function that you'd think it would : it waits for something to happen.

Whether that something is a key being pressed, or some external device being operated, is immaterial to the 64, provided that we've used the command properly.

The syntax to follow is : =

WAIT A,B,C

where A lies in the range 0 to 65535 (you may recognise this number as being the largest line number you can have in a program!), and B and C must lie between 1 and 255 (another number that will become all too familiar - you'll see why in the next chapter).

If necessary, you don't have to use the final (,C) argument, in which case it defaults to zero.

So, what are A, B and C?

Later on in this chapter we'll come to the logical operators AND, OR and NOT, and it is these which give the clue to the operation of the WAIT command. If you know how those work, then read on. If not, turn to the page headed AND, OR and NOT!

Technically, WAIT looks at the content of memory location A, exclusively ORs it with C, and then ANDs the result with B, and carries on doing this until it gets an answer of zero. With me ? I thought not. Let's take an example : =

```
10 PRINT "[CLR]PRESS FIRE BUTTON ON JOYSTICK"
20 WAIT 56464,16
30 WAIT 56464,16,16
40 PRINT "[CD]WELL DONE!"
50 REM CARRY ON WITH PROGRAM
```

After printing a message on the screen, line 20 looks at memory location

56464, and the result of ANDing this with 32 is to clear any previous fire button presses.

We'll see later, when a program is waiting to detect a key being pressed on the keyboard, that something called the 'keyboard buffer', an area of computer memory that remembers which keys have been pressed (and in what order!), also has to be cleared.

Keyboards, fire buttons or whatever, anything that requires something to be pressed, they all have what is termed a buffer. This is an area that can store up keystrokes (say), and one fault of the GET command looked at earlier is that, if you've been a mite over-enthusiastic whilst pressing the keys, you may think you've only pressed 'Y' once, but as far as the 64 is concerned you've hit it about three times, and those extra two are stored in the keyboard buffer. Thus, when it comes to getting you to press Y again, it looks at the keyboard buffer and thinks 'Aha, there are two Ys already in here, he/she must already have pressed it', and moves on through the program: usually the last thing you want to happen.

So, fire buttons, like keyboards, must be cleared, and line 20 does just that.

Line 30 then WAITs for the fire button either to be pressed, or released again, and when this is done the program carries on again.

You must have your joystick in port 2 for this to happen. The equivalent result in port 1 is achieved by substituting 145 for 56464.

Other movements of the joystick can be detected as well, so to assist in your writing of games that require one (or two) joysticks, here's a complete list of the values to be used in the above program to detect whether the joystick is moving up, down, left or right. Fire is given again as a reference.

	PORT 1	PORT 2
FIRE	145,16,16	56464,16,16
UP	145,4,4	56464,4,4
DOWN	145,2,2	56464,2,2
LEFT	145,1,1	56464,1,1
RIGHT	145,8,8	56464,8,8
ANYTHING!	145,31,31	56464,31,31

As an example, to modify our program to detect when the joystick

in port 1 is pushed to the left, we now have :=

```
10 PRINT "[CLR]MOVE JOYSTICK TO THE LEFT"
20 WAIT 145,1
30 WAIT 145,1,1
40 PRINT "[CD]WELL DONE!"
50 REM CARRY ON WITH PROGRAM
```

WAIT will only function with memory locations whose contents can change independently of a program running, otherwise there's no way of regaining control of the program once the WAIT command has been set up.

Few memory locations meet with this criterion, so for those of you who want to play around, here's a list of just some of those that do :=

Location 162	:	increments every 1/60 second (1 jiffy)
Location 161	:	increments every 256 jiffies (4.2 seconds)
Location 160	:	increments every 65536 jiffies (18.2 minutes)
Location 197	:	returns a unique value for the key being pressed
Location 653	:	status of shift key (0 = up, 1 = down)
Location 198	:	number of characters in the keyboard buffer (0 to 9)

There are many others, but before experimenting make sure that you save any program that has extraneous WAITs in it : even run/stop and restore, the accepted method for re-setting the machine, might not work!

There are some very satisfying things about using Wait. First and foremost is that it shows your friends you know a little bit about programming, but there are other reasons!

Most important, the STOP key will not break into a Wait command, so, if, for example, you're waiting for the shift key to be pressed and you want to halt the program, you'll have to press shift first and then the stop key immediately afterwards.

Of almost equal importance is the fact that the internal clock in the machine (the TI$ reserved word mentioned earlier) continues during a Wait, which it doesn't if you've disabled the run/stop key. We'll be looking at time in more detail shortly.

Two final, useful, examples of Wait should convince you that it's a command worth learning about.

They can be used in FOR ... NEXT loops, in either program or immediate mode. The following short program allows you to examine Basic ROM memory : =

```
FOR X=10*4096 TO X + 8191:PRINTX,PEEK(X):WAIT
197,64:NEXT
```

A list of memory addresses and contents will appear on the screen, and to stop them printing just press any key (other than shift, control, logo or restore). Release the key to get it going again.

If you want one key only to halt everything, change the Wait command to a WAIT 653,1,1: thus pressing shift/lock allows it to trundle on quite happily.

Of course, putting these Wait commands into a program will allow you to scroll through it line by line: instead of the one Wait, we just have WAIT 197,64:WAIT 197,64,64 for any key control, or WAIT 653,1,1:WAIT 653,1 for shift key control.

Tape Control

The humble cassette deck is the storage device that most people will be using with their Commodore 64, so let's take a look at how it operates.

You will, I trust, be familiar with the Load, Save and Verify commands, but here's a quick resume just in case : =

1) To save a program onto tape, you must insert the cassette in the cassette deck, and type one of the following commands : =

a) SAVE - which saves the program without a name.
b) SAVE "FRED" - which saves the program, and calls it FRED.
c) SAVE A$ - which saves the program and gives it the name in A$
d) SAVE "FRED",1,1 - which saves the program, calls it FRED, and gives it an End Of Tape (EOT) marker.

2) To load a program back from tape, after inserting the relevant cassette, type one of the following commands : =

a) LOAD - just loads the first program it comes to.
b) LOAD "FRED" - looks for the program called FRED and, if

found, loads it.

c) LOAD A$ - looks for the program whose name is in the variable A$, and, if found, loads it.

d) LOAD "FRED",1,1 - looks for the machine code program called FRED and, if found, loads it into the 64 without relocating it. It also, like all other load commands, wipes out the program currently in memory.

You may have wondered why the screen goes blank when loading a program from tape, other than when it informs you that the program's been found, and is waiting for you to press the Logo key (you don't have to, it will load it anyway if you're patient).

Well, the poor old 6526 interface chip, which looks after cassette operations, is not the fastest chip in the world, and loading a program whilst still displaying and refreshing the screen image at the same time is a bit too much for it. So it switches the screen off until the program's finished loading. Quite sensible really.

3) To verify a program, you use the same syntax as save. Verifying is a good idea, as tape decks have a habit of becoming unreliable after prolonged use.

Saving Data

Saving data to tape is, obviously, done on a sequential basis, and uses the commands PRINT£, INPUT£ and GET£, variations on old friends.

In chapter 7 we'll take a closer look at file handling.

Tape Odds and Ends

You can sense from within a program whether or not the various keys on the cassette deck are depressed, and this short program from Kevin Bergin illustrates the method employed : =

```
10 A=PEEK(1)OR32:B=PEEK(1)AND16
20 POKE 192,A:POKE1,A:REM STOPS TAPE MOTOR
30 PRINT"[CLR]TAPE MOTOR STOPPED!"
40 IFB<>0THEN 60: REM IF NO SWITCHES PRESSED
THEN 60
50 PRINT"[CD]PRESS STOP ON TAPE"
60 IF (PEEK(1)AND16)=0THEN60
70 PRINT"[CD]ALL SWITCHES OFF"
80 END
```

This, as you will have seen from the Peek statements, revolves around the contents of memory location 1 : an interesting location, to say the least, but one which is best attacked from machine code rather than Basic, as it has the most annoying habit of crashing the machine every five seconds.

Disk Usage

As with cassettes, we'll go into much more detail on disk drives in chapter 7, where we take a look at all sorts of peripherals, how they work and how they store information.

For now, a short summing up : =

1) Programs are loaded and saved in exactly the same way as with cassettes, but with the device number (8) tagged on to the end. Thus a typical load command would look like this : =

LOAD "ALBERT",8

2) There's no need to verify programs that have been stored on disk : the disk unit, being an intelligent device, does its own verifying.

3) Disk drives can be chained together, using the standard cables supplied, but as they all come with a device number of 8 it makes little sense to do so. Unless, that is, you can change the device number, and the following program shows you how to do just that, although this program uses some commands that you haven't yet met in this book (OPEN, PRINT£, M-W), all will be revealed in the section on using the disk drive in chapter 7.

For now, you'll just have to accept the fact that it works!

```
10 OPEN 15,8,15
20 PRINT£15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$(32+
ND)CHR$(64+ND)
30 CLOSE15
```

When run, this program will change the disk drive's device number from 8 to whatever value you give ND. Thus, to change it to device number 9 : =

```
10 OPEN 15,8,15
20 PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2)CHR$(41)
CHR$(73)
30 CLOSE15
```

To change it back again, use 40 and 72 in place of 41 and 73 respectively.

Disk drive device numbers can also be changed by hardware (the above method resets itself when you turn the drive off and on again), but I for one don't recommend anyone fiddling about inside a disk drive: they cost a lot of money, and an invalidated warranty can be an expensive thing.

The only other disk drive that can be used directly with the Commodore 64, other than the 1541, is its predecessor the 1540, although it is fair to say that a number of interfaces are currently available that allow you to link up to just about anything, hard disks included.

Still, back to the 1540.

When you're trying to use this direct (i.e. just plugged in at the back of the 64), nothing very much happens, since we encounter the same problem as when trying to load from tape : the 6526 chip is trying to do too much, and cannot handle the data.

Thus, if we turn the screen off before doing anything we can then load or save programs or data, and then turn the screen back on again.

This is all very well from within a program, but just loading new programs can be tedious if you can't see what you're typing. So the following procedure is recommended : =

1) Move the cursor to the top of the screen, type POKE 53265,11: don't hit Return, or you'll blank the screen!

2) Press shift/Return instead, and move the cursor down two additional lines.

3) Type in LOAD "PROG NAME",8 - or whatever - then press shift/Return again, move the cursor down an additional four lines, and ...

4) Type POKE 53265,27.

5) Finally, move the cursor back to the top of the screen again, and hit Return three times. The 64 will do all the work, and your program will load in quite happily without any problems!

AND, OR and NOT

We've already encountered these in a variety of situations, so it's about time we explained how they work.

They essentially follow their English meaning when used in a straightforward program.

Take the following examples :=

```
10 IF A>10 AND B<5 THEN 200
```

This means that if the variable A has a value greater than 10, and the variable B has a value less than 5, then branch to the program line 200. Otherwise, just carry on to the next line of the program.

And again :=

```
10 IF A>10 OR B<5 THEN 200
```

Almost the same line of code, but not quite.

In this case, the program will branch to line 200 if A is greater than 10, OR if B is less than 5.

As you can see, quite complex decision making can be achieved using these operators.

NOT is a peculiar one, and doesn't seem to be used very much. Still, just about every decision-making process you will require can be achieved with AND and OR, as the following example should serve to show.

```
10 IF A<10 OR B>5 AND A$="Y" THEN 200
```

Here, the program will branch off to line 200 if A is less than 10 or B is greater than 5, but only if A$ is equal to "Y" as well.

If all of these conditions are not met, then the program just falls through to the next line.

What we're doing here is testing to see whether various statements are true or false. Try the following short example : =

```
10 A=20
20 PRINT (A>15)
```

When run, this program will print out the value -1, because the statement A > 15 is a true one : we've just defined A to be equal to 20.

So, if something is true, the computer prints out a -1, and if it is false it prints out a zero. For instance : =

```
10 A=20
20 PRINT (A>25)
```

will result in 0 being printed, as the statement is patently not true.

All of this is based on what are called TRUTH TABLES, and these tables for AND and OR work as follows : =

Truth Tables

A	B	C
0	0	0
0	-1	0
-1	0	0
-1	-1	-1

The tables work in the following way: if the first statement A is false (zero) and the second one B is false (zero), then the result C is also false.

It's reassuring to note that even with computers two wrongs don't make a right.

And so it goes on. If the first statement A is true (-1) and the second one B is false, the result C is also false.

That was the truth table for AND. For OR, the results, as you might expect, are slightly different : =

```
 A     B     C

 0     0     0
 0    -1    -1
-1     0    -1
-1    -1    -1
```

Thus if either A or B is true, then so is the result. Only if both statements are false will the result also be false.

Using the above two tables, our complex example above becomes much more straightforward.

```
5 A=12:B=6:A$="Y"
10 IF A<10 OR B>5 AND A$="Y" THEN 200
```

A is less than ten, so we have a false (0). B is certainly greater than 5, so we have a true (-1). A false ORed with a true gives a true.

A$ is equal to "Y", another true, so the result is a true ANDed with a true, which is again true. Thus the program branches off to line 200.

To finish off this discussion, you may have seen in program listings statements like :=

A=B AND C
A=B OR C

and wondered how they work. Knowing what we do now, the statements become much easier to understand.

Let's take some concrete examples. Let's have A equal to 27, and B equal to 128. To calculate the results, we need to break these down into their binary representations.

Binary, as you know, is a system of representing numbers as a series of 0s or 1s, rather than using the digits 0 to 9 as we do. This is because computers can only understand two states (an electronic circuit can only be on or off, it can't be anything in between!), and the binary notation follows logically from that.

So, in binary, 128 becomes 10000000

Remembering the rules for AND and OR, 27 AND 128 now becomes 00000000 (at no point do we have two 1s together), and 27 OR 128 becomes 10011011 (if a 1 occurs in either number, the result is a 1).

In plain English then, 27 AND 128 equals 0, and 27 OR 128 equals 155.

One final example :=

53 in binary equals 00000110101
1111 in binary equals 10001010111

53 AND 1111 equals 00000010101 = 21
53 OR 1111 equals 10001110111 = 1143

We'll return to ANDs and ORs in our section on machine code programming in chapter 3.

Other Logical Operators

These have been mentioned earlier, in statements like :=

IF A>5

and so on.

Knowing how truth tables work, it now becomes a simple matter to understand all these logical operators, and calculate the results that they will give.

To sum up, the remaining operators are :=

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

A thorough knowledge of these, along with AND and OR and the way that they work, will make structuring your programs a lot easier!

Defining Functions

Many times within a program you'll have cause to perform a number of mathematical calculations.

These calculations being what they are, the odds are fairly reasonable that you'll be performing the same function with a number of different variables.

For instance, the calculation : =

```
PRINT 10*(56-1.2*Z)
```

where Z is a variable, will obviously vary according to Z.

Now, if this calculation is performed many times, your program will contain a lot of statements of the form : =

```
PRINT 10*(56-1.2*17)
PRINT 10*(56-1.2*5.9)
PRINT 10*(56-1.2*127)
```

Not only space consuming (in terms of computer memory) but also time consuming, in terms of typing it all in.

Fortunately we have the ability in Commodore 64 Basic to define a function.

This is usually done at the start of a program, and then referred to thereafter, although this is not absolutely necessary.

However, as it is rather easy to produce an UNDEFINED FUNCTION error by referring to the wrong one, it's probably best to keep them all grouped together at the beginning of a program.

The syntax for this is : =

```
DEF FNA(Z)=10*(56-1.2*Z)
```

Then, whenever we wish to evaluate the expression, we substitute whatever value Z has at the time into the expression.

Thus we might have something like : =

```
PRINT FNA(10)
```

where, in our example, the result would be 440 (56 - 1.2*10, all multiplied by 10)

The name of the function can be any legal two letter variable name. Thus, the following are all legal function names : =

```
DEF FNAA
DEF FNA1
DEF FNAB
```

As long as you have the DEF FN part, your choice is fairly wide.

Playing for Time

The reserved variable TI, and its string counterpart TI$, have appeared before, but with little explanation of what they did.

TI and TI$ both relate to the real-time clock which is built into the Commodore 64.

TI is updated every 1/60 of a second, starting from when you turn the computer on, or whenever you resest TI$.

Thus it can be an extremely accurate measure of time, and can be used in this way in many programs.

Note, however, that certain POKE commands can cause this internal clock temporarily to suspend operations, so be careful when accuracy is of the essence.

TI$, the associated string, is also updated as the internal clock ticks over.

However, it can also be altered by the user, in the following manner : =

```
TI$="123045"
```

which sets the time at 12 hours, 30 minutes, and 45 seconds.

TI$ can be split up into its component parts in the following manner : =

```
10 PRINT"[CLR]THE TIME IS NOW :"
20 TI$="101112"
30 A$=LEFT$(TI$,2)
40 B$=MID$(TI$,3,2)
50 C$=RIGHT$(TI$,2)
60 PRINTA$;"HOURS ";B$;" MINUTES AND ";C$;" SECO
NDS":PRINT"PRECISELY[2CU]"
70 GOTO 30
```

This will just display the time at the top of the screen.

SIN, COS and TAN

We mentioned near the beginning of the first chapter that your Commodore 64 could be used as an extremely expensive calculator, and showed you how to perform various mathematical calculations on it.

Built into the 64's Basic language are a number of other numerical functions that increase this calculator capability, and which can also be used in direct program mode.

The trigonometric functions SIN, COS and TAN are three of these, although alas they are the only trigonometric functions that are actually readily available, apart from ATN (see next section).

Appendix D contains a full list of all the other hyperbolic functions which can be built up using these three stepping stones, and which all use the function defining capability of the machine mentioned in the last section.

The syntax for using these three is as follows : =

```
PRINT SIN(X)
```

will print the sine of X, where X is an angle expressed in radians.

```
PRINT COS(X)
```

will print the cosine of X, X again being in radians.

```
PRINT TAN(X)
```

will print the tangent of X, X in radians again.

Those of you who left school longer ago than you may care to remember, or even those who are still there and get hopelessly confused by people dropping perpendiculars, may wish to be reminded what these actually mean :=



The sine of the angle at B is equal to the length of the OPPosite side AC divided by the HYPotenuse (the side opposite the right angle) AB.

The cosine of that angle is equal to the ADJacent side (BC) divided by the HYPotenuse (AB), and the tangent is equal to the OPPosite (AC) divided by the ADJacent (BC).

$$\text{SIN} = \frac{\text{OPP}}{\text{HYP}} \; ; \; \text{COS} = \frac{\text{ADJ}}{\text{HYP}} \; ; \; \text{TAN} = \frac{\text{OPP}}{\text{ADJ}}$$

However, all this will give is our angles in degrees, and we want them in radians. So, to convert from degrees to radians, use the following formula :=

A degrees = (A * PI)/180 radians
B radians = (B * 180)/PI degrees

PI is the well known mathematical symbol (accessible from the keyboard), and is equal to the circumference of a circle divided by its diameter. This is approximately equal to 22/7, but not quite! So, don't use it, use PI from the keyboard instead.

More Operations

A quick round-up now of some of the other Basic commands available from the keyboard of the Commodore 64.

SQR

This returns the square root of a number, and the syntax for using this is as follows : =

```
PRINT SQR(A)
```

where A is any number greater than or equal to zero.

As you know, negative numbers don't have a square root (multiply two negative numbers together and you'll get a positive one), so you'll only get an ILLEGAL QUANTITY ERROR if you try it.

LOG

Takes you back to school, doesn't it ? Seems strange, a modern computer using old logarithms, but never mind. The syntax for using this is : =

```
PRINT LOG (A)
```

This will give you the natural logarithm, to the base E. To convert to logs base 10, which most of us are used to, divide the result by the LOG (10) to the base E.

We'll take a look at exponentials in the next section.

SGN

Another of those great commands that don't do very much. SGN simply returns the sign of a variable or a number, whether it be positive, negative or zero, and the syntax to use is : =

```
PRINT SGN(A)
```

which will give you a 1 if positive, 0 if zero, and -1 if negative.

ATN

Really this belongs in the last section, as it takes us back to geometry again.

It returns the angle, in radians of course, whose tangent is equal to a variable or number.

The syntax for doing this is :=

```
PRINT ATN(A)
```

ABS

One of the simplest commands of all, this just gives you the absolute value of a number. In other words, it removes the positive or negative sign from in front of the number.

The syntax for using this is :=

```
PRINT ABS(A)
```

ABS is, of course, always positive.

FRE

Really free, but unfortunately not really accurate, unless the size of your program begins to exceed about 6K.

The reason for this is that the 64 is a humble 8 bit computer, and all you need to know about that at the moment is that 8 bit computers can only 'see' a maximum of 32K at a time.

Thus, when the amount of available memory becomes less than 32K, it performs the job admirably.

Until it does, you'll get some weird and wonderful results, but there is a way around it.

The syntax, when you've got less than 32K left, is :=

```
PRINT FRE(0)
```

You can use anything as the argument. Here we just used zero, but anything will do. Performing the function takes up two bytes, so always remember to add them on again.

When there's more than 32K, you have to make a slight amendment

thus : =

```
PRINT FRE(0)+256*256
```

A short program will show you where the transition point occurs. Type this in : =

```
10 DIM A(1224)
```

and RUN it. The FRE(0) on its own will return a value of 32764 bytes free.

Now, amend the array to be of size 1223, and run the program again. FRE(0) will now give you the value of -32767!

Adding 256*256 will give you the correct value of 32769 bytes free.

Incidentally, you may wonder why it's called a 64K machine, when you only get 38911 bytes on power up.

Well, it HAS got 64K altogether, including everything that makes it tick, and you CAN (we'll show you in chapter 3) switch things about inside the machine (temporarily!) to give you more than 38911 bytes to play with.

However, having done that there is very little else that you can do!

Exponentialism

No, not a mysterious oriental religion, but it is related to a well known mathematical constant, which is available on the 64.

The mathematical constant e, equal (approximately) to 2.71828, can be found using the syntax : =

```
PRINT EXP(1)
```

This function can be used to calculate e raised to any power (i.e. multiplied by itself a certain number of times). For instance, e to the power 5 is the same as saying 'e multiplied by itself 4 times'. To calculate this, use : =

```
PRINT EXP(5)
```

To save you trying it, as far as mathematicians are concerned any number raised to the power zero is equal to 1. Oh go on then, try it!

One of the reasons why e is so popular is that it can be used to calculate the number of terms you will need in a mathematical series in order to reach a given sum.

Take the harmonic series :=

1 + 1/2 + 1/3 + 1/4 + 1/5 + ... 1/N

To find out how many terms are required to reach a given sum A, you take the constant e, multiply it by itself A times, and divide the result by 1.781.

The number you get is approximately the number of terms needed to reach the sum A. As you can imagine, this increases rapidly as the number increases. In fact, it increases exponentially, which brings us back to EXP again!

SYS and USR

These refer more to the next chapter, where we start getting involved with programming the Commodore 64 in machine code, so for now we'll content ourselves with a brief definition of both, and come back to them in a few pages time.

SYS

The syntax for using this SYStem call is :=

SYS A

where A lies between 0 and 65535. Control of the program will then pass to the machine code program that commences at the memory location as defined by A. Thus if we'd said SYS 64738, program control would have gone to memory location 64738 and executed the code that it found there.

Incidentally, 64738 is a useful number to remember: that SYS call resets the computer, without turning it off and on again!

SYS does not allow you to pass parameters from Basic to machine

code, unlike ...

USR

which does!

The syntax for using this is : =

```
A=USR(B)
```

This passes the variable B into a machine code program, which resides at a starting location indicated by the contents of memory locations 784 and 785 (you guessed it, more in the next chapter!).

When the program has finished executing, it returns a value back to our Basic program and stores it in the variable A. This can now be printed out using the standard PRINT statement.

Odds and Ends

A collection of little oddities, which should enable you to have some fun on the 64.

To understand them all would take more room than is available in this book, but at least you'll know that they're all tried and tested 'facts and figures', and that they all work.

To get you started, one of the best one line programs I've ever seen, which certainly bears repetition. It's called Burrow, and looks something like this : =

```
1 A$="[CU,CD,CL,CR]:PRINTMID$(A$,RND(.5)*4+1,1)"
*[CL]";:FORI=1TO30:NEXT:PRINT"[RVS] [CL]";:GOTO1
```

Well, it took us more than one line to enter it, because we had to use symbols to represent the cursor control characters, but by using those instead, and using the shorthand form of PRINT, namely ?, you shouldn't have any problems.

A fun graphics display!

Stopping Program Examination

There are a number of handy little POKEs which, when executed, will stop people listing your program, saving it, or even breaking into it by pressing the Stop key, or Run/Stop and Restore simultaneously.

Normally you'd place these at the start of a program so that they'd be the first statements encountered.

Just ensure that you've got a proper, debugged, working version of your program before you go putting these in, otherwise you won't be able to look at it!

Prevent Listing

To stop people getting a sensible listing of a program, even if it does look rather pretty as a lot of garbage appears on the screen, have this as one of the lines of your program :=

```
POKE 775,200
```

And to get back to normal again, you'll need :=

```
POKE 755,167
```

These, and the others to follow, can all be typed in direct mode if required.

Stopping the STOP key

To prevent people breaking into your program by using the Run/Stop key, use this :=

```
POKE 808,239
```

And this to get back to normal again :=

```
POKE 808,237
```

And Even the Restore Key

This has a side effect of making the program listing look a little strange,

although it doesn't effect running the program.

You'll need : =

```
POKE 808,255
```

And to get back again : =

```
POKE 808,237
```

A No-Key Keyboard

To completely disable the keyboard, although I can't see any reason why you'd want to, use : =

```
POKE 649,0
```

And use this to get back to normal again : =

```
POKE 649,10
```

Prevent Save & List

Two POKEs this time, to prevent the above two commands working properly : =

```
POKE 808,255:POKE 818,32
```

And back again with : =

```
POKE 808,237:POKE 818,237
```

Unfortunately, POKEing 808 with 255 has a side effect - it stops the internal clock working. Oh well, you can't win 'em all!

Repeating Keys

As you know, all the cursor keys repeat, along with the space bar, but to make them all repeat type : =

```
POKE 650,255
```

And to stop them all, type : =

```
POKE 650,0
```

There are others of course, but that should be enough to be going on with!

Some Program Listings

Three programs, as we said at the start of the chapter, commencing with a bi-directional listing program.

This was originally in a German book of software listings. To those unknown people who originally wrote it, thank you for a very useful utility.

```
60000 REM BI-DIRECTIONAL LISTING
60010 AD=2048
60020 LI=PEEK(AD+3)+PEEK(AD+4)*256
60030 PRINT"[CLR,BLU]GOTO60050":PRINT"LIST";LI;
60040 POKE631,19:POKE632,17:POKE633,159:POKE634,13
:POKE635,19:POKE636,13:POKE198,6:END
60050 IFPEEK(197)=40 THEN 60080
60060 IFPEEK(197)=43 THEN 60100
60070 GOTO60050
60080 IFPEEK(AD+5)<>0THENAD=AD+1:GOTO60080
60090 AD=AD+5:GOTO60020
60100 AD=AD-1
60110 IFPEEK(AD)=0ANDPEEK(AD-4)<>0ANDPEEK(AD-3)<>0
THEN60020
60120 GOTO60100
```

The program is intended to be used as a subroutine, and thus the line numbers extend from 60000 onwards. You'll have to insert a check for the key being pressed (lines 60050 - 60070) to return from the subroutine at the press of an appropriate key. Perhaps a line like this : =

```
60065 IF PEEK(197)=4 THEN RETURN : REM FUNCTION
KEY F1 PRESSED
```

The program initially displays in the background colour (so that you can't see it!) the words GOTO 60050, and LIST LI, where LI is the next line number to be listed (calculated in line 60020.)

Line 60040 then puts into the keyboard buffer the characters HOME, Cursor Down, Character Colour, Carriage Return, Cursor Down, Carriage Return, and then the number of characters that are going in there.

Lines 60050 and 60060 check to see whether you've pressed the '+' or '-' keys to increase or decrease the line number, and if so goes off to the appropriate part of the program.

If you've pressed neither it loops back and waits until you do.

Then, the line is displayed on the screen (one line only at a time), and this is updated depending on which key you press.

Universal Input Subroutine

Not many notes for this one, as the program is heavily REMmed and you can see for yourself what's going on.

```
100 PRINT"[BLK]WHAT NOW ? ";
120 GOSUB 60000
140 PRINTCM$
160 END
60000 CM$="":
60001 REM NULLIFY STRING
60002 PRINT"[RVS]*[OFF,CL] ";
60003 REM PUT UP PROMPT
60005 GETZ$:IFZ$=""THEN60005
60006 REM GET A KEY
60010 Z=ASC(Z$):IFZ>95THEN60005
60011 REM CHECK ASCII VALUE - REJECT GREATER THAN
95 (SHIFT KEY USED)
60080 ZL=LEN(CM$):IFZL>27THEN60110
60081 REM MAXIMUM INPUT LENGTH 27 CHARACTERS
60100 IFZ>31THENCM$=CM$+Z$:PRINTZ$;:GOTO60002
60101 REM REJECT CURSOR KEYS AND OTHER STRANGERS!
60110 IFZ=13ANDZLTHENPRINT" ":RETURN
60111 REM IF RETURN PRESSED, AND STRING HAS A CHAR
ACTER, THEN RETURN
60120 IFZ=20ANDZLTHENCM$=LEFT$(CM$,ZL-1):PRINTZ$;
60121 REM CHECK FOR DELETE, THEN DELETE LAST CHARA
CTER
60140 GOTO60002
60141 REM BACK FOR NEXT CHARACTER
```

The only thing that will break out of the program as it stands is pressing the Run/Stop, or Run/Stop and Restore keys, but as we saw in the last section there are a number of ways of getting around this.

The program at present just checks for ASCII values, and doesn't differentiate between numerical input and alphabetic.

You could easily insert a check on upper and lower limits of ASCII values: say between 65 and 90 if you only wanted alphabetic information to be typed in.

The table in Appendix D will help you decide on that one.

Minefield

Well it's about time you had some fun, it's been learning all the way so far!

Again, no notes, as the program is REMmed to let you know what's going on.

A simple game: all you have to do is tread through the Minefield from the bottom left of the screen to the top right, avoiding the mines as you go.

Your rusty radar will give you a limited warning, but it becomes a test of mathematical logic to get from beginning to end, especially as you step up a level or two.

If you walk into a mine, the display will change to show you where they all were.

Have fun!

```
1 PRINT"[CLR]"
5 POKE 53281,4
8 GOSUB 2000:REM RULES
10 DIMM(15,11):REM ARRAY FOR GRID
20 L=L+1:J=885
27 PRINT"[CLR,BLK,CD]"
28 FORI=1TO11:REM PLOT GRID
30 PRINT"[5SP,SHIFTED C/SPACE/SHIFTED C/SPACE ETC
15 TIMES]
```

```
40 PRINT"[4SP,SHIFTED B/SPACE/SHIFTED B/SPACE ETC
16 TIMES]
50 NEXT
53 PRINT"[5SP,SHIFTED C/SPACE/SHIFTED C/SPACE ETC
15 TIMES]
55 IFEOG=1THEN 3000:REM END OF GAME
56 POKE 1024+J,102:POKE 1024+113,42:X=2:Y=2:REM PU
T YOU ON GRID
57 FORI=1TO4+L*6:A=INT(RND(.5)*14):B=INT(RND(.5)*1
0):M(A,B)=1:REM GENERATE MINES
58 IF(A=0ANDB=0)OR(A=14ANDB=10)OR(A=1ANDB=0)THEN M
(A,B)=0:NEXT
59 NEXT
60 IFX=16ANDY=12THEN5000
61 GOSUB 1500
62 Q=J
64 GETP$:IFP$=""THEN 60:REM MOVEMENT
70 IFP$="I"THEN1000:REM UP
75 IFP$="M"THEN1200:REM DOWN
80 IFP$="A"THEN1400:REM LEFT
85 IFP$="D"THEN1600:REM RIGHT
90 GOTO 64
1000 IFJ<120THEN60
1020 J=J-80:POKE1024+Q,46:POKE1024+J,102
1022 Y=Y+1
1025 GOSUB 1500
1040 GOTO60
1200 IFJ>865THEN60
1220 J=J+80:POKE1024+Q,46:POKE1024+J,102
1222 Y=Y-1
1225 GOSUB 1500
1240 GOTO60
1400 IF INT((J-5)/40)=(J-5)/40THEN60
1420 J=J-2:POKE 1024+Q,46:POKE 1024+J,102
1422 X=X-1
1425 GOSUB 1500
1440 GOTO 60
1500 REM CHECK FOR BOMBS
1502 IFM(X-2,Y-2)=1 THEN3000
1510 X1=X+1:
1511 IFM(X1-2,Y-1)=1THENB1=B1+1
1512 IFM(X1-2,Y-2)=1THENB1=B1+1
1513 IFY<3THEN1515
1514 IFM(X1-2,Y-3)=1THENB1=B1+1
1515 X1=X
1516 IFM(X1-2,Y-1)=1THENB1=B1+1
1517 IFM(X1-2,Y-2)=1THENB1=B1+1
1518 IFY<3THEN1520
1519 IFM(X1-2,Y-3)=1THENB1=B1+1
1520 X1=X-1:IFX<3THEN1570
1521 IFM(X1-2,Y-1)=1THENB1=B1+1
```

```
1522 IFM(X1-2,Y-2)=1THENB1=B1+1
1523 IFY<3THEN1570
1524 IFM(X1-2,Y-3)=1THENB1=B1+1
1570 PRINT"[HOME]THERE'S ";B1;" BOMB(S) ONE SQUARE
 AWAY"
1575 B1=0
1599 RETURN
1600 IF INT((J+7)/40)=(J+7)/40THEN60
1620 J=J+2:POKE 1024+0,46:POKE 1024+J,102
1622 X=X+1
1625 GOSUB 1500
1640 GOTO 60
1998 GETZ$:IFZ$=""THEN1998
1999 END
2000 PRINT"[CLR,BLK]WELCOME TO MINEFIELD!"
2001 PRINT"[CD]YOU HAVE TO FIND YOUR WAY ACROSS A
MINE-"
2002 PRINT"[CU]FIELD, TO REACH SAFETY IN THE TOP R
IGHT"
2003 PRINT"CORNER OF THE SCREEN"
2004 PRINT"[CD]YOUR RUSTY RADAR ONLY SHOWS MINES T
HAT"
2005 PRINT"ARE ONE SQUARE AWAY : TREAD CAREFULLY!"
2006 PRINT"[CD]USE 'A' TO MOVE LEFT, 'D' RIGHT, 'I
' UP"
2007 PRINT"[CD]AND 'M' DOWN'"
2008 PRINT"[CD]PRESS 'SPACE' WHEN READY TO START"
2009 GETSW$:IFSW$=" "THENPRINT"":RETURN
2010 GOTO 2009
3000 PRINT"[CLR]DESTROYED!"
3002 PRINT"[CD]PRESS 'SPACE' TO SEE THE BOMBS"
3004 GETSP$:IFSP$=" "THEN3008
3006 GOTO 3004
3008 EOG=1:GOSUB 27
3009 POKE 1024+J,102
3010 FORA=0TO15:FORB=0TO11
3015 IFM(A,B)=1THENSS=A*2+5+(10-B)*80+80:POKE 1024
+SS,224:POKE 55296+SS,7
3020 NEXT B,A
3040 PRINT"[HOME]ANOTHER GAME (Y OR N)"
3045 GETG$:IFG$="Y"THENRUN
3050 IFG$="N"THENPRINT"[CLR]BYE!":END
3055 GOTO 3045
5000 REM SURVIVED A LEVEL! ONTO NEXT ONE
5002 PRINT"[CLR]YOU WERE LUCKY ON LEVEL ";L
5005 FORI=0TO15:FORK=0TO11:M(I,K)=0:NEXTK,I
5010 PRINT"[2CD]BUT JUST PRESS 'SPACE' AND I'LL PU
T YOU"
5020 PRINT"[CD]ON LEVEL ";L+1
5030 GETSP$:IFSP$=" "THENPRINT"[CLR]":GOTO20
5040 GOTO5030
```

3

An Introduction to Machine Code Programming

Introduction

Machine Code Programming is one of the more difficult things to learn to do on the Commodore 64.

The major reason for this is probably the fact that, as soon as you switch the machine on, you're put into an environment that forces you to program in Basic.

After all, Basic is fairly easy to learn, you can do an awful lot of work in Basic, and most important of all, how do I program in machine code on this machine anyway?

It is certainly true that the Commodore 64 has no built-in monitor to assist you in the transition from Basic to machine code programming.

Unlike all but the very earliest Commodore PETs, all of which had an easily accessible monitor in them, the Commodore 64 is totally devoid of this link into the world of assembly language.

You may be thinking 'monitor'? What's he on about?

A monitor, as talked of here, is nothing to do with a television set, or anything that you plug into your computer.

A monitor (or MLM, for machine language monitor, as we shall refer to it from now on), is nothing more than a programmer's aid, rather like the many that you see advertised in the popular computing press.

Commodore, as we've seen, used to give you an MLM free, but not with the 64.

Using an MLM you are able to examine the internal RAMifications of the computer, as well as take a look at the workings of the Read Only Memory inside it, and alter the areas of Random Access Memory as you see fit.

The simplest of monitors will display nothing more than a curious mixture of letters and numbers, looking something like this perhaps : =

```
C000 : A2 01 8E 02 04 A9 01 6D
C008 : 02 04 8D 00 04 AA AA AA
```

To the newcomer to machine code programming, this looks incomprehensible.

However, if you put this in the form of a Basic program, you'll see what it's doing immediately : =

```
10 A=1
20 B=1
30 C = A + B
40 PRINT C
```

All we're doing is adding 1 and 1 together, and printing out the result. Somewhat easier to understand!

Many people have, over the years, added extensions to that rather simple monitor that existed on those early PETs, and one of those people was Jim Butterfield: a legend in his own spare time.

Jim has put together, with outside help (just to spare his blushes) a program called Extramon: an Extra Monitor. This has long been a public domain program, and as such anyone is free to have a copy of it.

The version published in Appendix D is the latest one, and works on the Commodore 64.

Together with full instructions on how to use it, as well as how to type it in, this monitor makes programming in machine code an awful lot easier. It takes away about 2 1/2K from your computer, thus leaving you about 36K to play with : ample room for some programming work.

Using Extramon, we can make use of a feature known as a Disassembly, which takes the sequence of numbers and letters shown above and transforms it into something much more intelligible. Like this : =

```
D C000 C010
.; C000  A2 01    LDX £$01
.; C002  8E 02 04 STX $0402
.; C005  A9 01    LDA £$01
.; C007  6D 02 04 ADC $0402
.; C00A  8D 00 04 STA $0400
```

Now this is starting to make a bit more sense. With a running commentary as well, we'd be almost there. So :=

```
D C000 C010
.; C000  A2 01    LDX £$01  Load X register
                            with a 1
.; C002  8E 02 04 STX $0402 Store the content
                            of X in 0402
.; C005  A9 01    LDA £$01  Load the Accumulator
                            with a 1
.; C007  6D 02 04 ADC $0402 Add the contents of
                            the Accumulator
                            To those of memory
                            location 0402
.; C00A  8D 00 04 STA $0400 And print the result
                            at memory
                            location 0400 (i.e.on
                            the screen)
```

Comparing this with the earlier Basic listing, you'll see that instead of letting something equal something else, we're storing a number and putting it somewhere. Instead of printing on the screen, we're storing the result in a memory location.

However, that location happens to be 0400, which in decimal form is 1024, and that is the location of the top left hand corner of the screen. Thus the result appears there.

If you try this program, you'll see that a letter A appears offset from the top, and a letter B in the corner. This is because screen display code number 1 is an 'A', number 2 is a 'B', and so on: see Appendix D.

So you see, learning machine code is no more difficult than learning Basic really, all you have to remember is a different set of symbols.

But Why Learn It ?

The principal reason must be speed. Machine Code is so much faster than Basic (later examples will seem unbelievably so), that a lot more work can be done in a shorter space of time.

That's why some of the fabulous arcade games that you see on the market look so good : they're written totally in machine code for speed of action.

Packages like the word processor used for writing this book are also written only in machine code. If you attempted to do in Basic what can be done in machine code, the result would be so slow as to be pedestrian. More importantly, you wouldn't sell any products!

So, bearing in mind that it isn't really that difficult, and that with the listing of Extramon given at the back of the book it isn't going to be too unintelligible, let's take a look at the chip behind the computer, and start learning a few fundamentals.

Using the 6510

And just what is a 6510 ?

It is the heart of the Commodore 64, the microchip inside that does all the work when running a program and looking after the machine generally.

This is the latest version of the grand old 6502 chip, one of the ones to revolutionise the world of microcomputing when it appeared in the original Commodore PETs.

If you want to get extremely technical, turn to chapter 10 and read all about it there. For now, it's good enough to know that it's the son of the 6502.

Let's define a few terms first of all, starting with the oft-heard word Hexadecimal.

Hexadecimal

Humans have long since counted in decimal, or units of ten.

Thus 1234 is the shorthand way of writing 4 + 3*10 + 2*10*10 + 1*10*10*10.

Computers, as we have seen, can count in units of 2 (zero and one). To make life easier for both of us, and because 16 is a nice number for computers to deal with, someone invented the hexadecimal system of counting, using 16 as a base instead of 10.

Thus, in hexadecimal, our original number 1234 becomes 4 + 3*16 + 2*16*16 + 1*16*16*16, or, in decimal, 4650. To avoid confusion, we'll give hexadecimal numbers the symbol $. Thus $1234 is the hexadecimal number which represents 4650 in decimal.

But if we've only got the digits 0 to 9, how can we count in the base 16? The answer is that we use letters as well. Counting now goes from 0 to 9, and carries on through A,B,C,D,E and F : thus $F represents 15 in decimal, as we've seen $10 would represent 16, so $FF would be equal to 15 + 15*16, or 255.

There is a hex to dec (shorthand terms) convertor to be found in Appendix D. This also converts to binary as well.

Op-Codes and Operands

Op-codes is the term used to describe machine code instructions. These instructions are understood directly by the computer (Basic has to be converted first, which is one of the reasons why it's slower), and Appendix C contains a full list of all the instructions available with the 6510 (exactly the same as the 6502, as it happens), together with the syntax for using them.

An operand is simply an item of data, or what the op-codes operate on.

The Extramon program (hereafter referred to, perhaps incorrectly, some might argue, as the assembler) doesn't understand op-codes direct, but instead takes its instructions as mnemonics. For example, LoaD the X register becomes LDX, and so on.

A program written in this way is referred to as an assembly language program : hence we call Extramon the assembler, since we're programming in assembly language.

Bits and Bytes

The last bit of definition, before getting down to the nitty-gritty, but if you don't understand this we'll never get anywhere, so be patient!

As you know, computer memory is counted in bytes: one kilobyte is equal to 1024 ordinary bytes, and so on.

Each byte can be further sub-divided into 8 bits: the Commodore 64 is called an 8-bit computer precisely because each byte contains 8 bits. If it only contained 2 I suppose we'd call it a 2-bit computer.

When we looked at the POKE instruction for altering memory, we stated that you could POKE any memory location with a number from 0 to 255. It is important to realise why this is so.

A typical diagram of a byte, and its 8 bits :=

7	6	5	4	3	2	1	0
128	64	32	16	8	4	2	1

The numbers below relate back to our earlier discussion on binary numbers, where everything was treated as ones or zeros. By doubling the number each time, we arrive at bit 7 and the value of 128.

If you look at those numbers and add them all up, you reach a total of 255. That is why we are limited to POKEing values between 0 and 255: the byte can't hold any more information.

Finally (honest), when you POKE a memory location with a number, you're altering various bits in it, not the whole byte.

For instance, supposing we POKEd 1024 with a 65. The total 65 comes from bits 6 and 0 of the byte (64 + 1 = 65). Every number between 0 and 255 POKEd into a memory location is a unique combination of 1 or more bits in that location.

If we POKE 1024 with 128, we're simply altering bit 7, and so on.

That's enough to get us going now. Have a good, stiff drink after all that, and we'll come back and take a look at what actually makes up the 6510 microprocessor.

A Model of the 6510

The 6510 chip contains 6 different registers that you, the programmer, can alter. These are as follows :=

The Accumulator	:	A
X Index Register	:	X
Y Index Register	:	Y
Program Counter	:	PC
Stack Pointer	:	SP
Status Register	:	P

If you load and run the assembler whilst reading this, which would be a good idea, then we can work hand in hand between the book and the computer. The above should explain some of the strange symbols that appear on the screen when you first run it. Basically it's the state of play as you enter the assembler.

The Accumulator

The key to the whole affair, the accumulator is the most important register at our disposal.

It is the only register in which data can be placed, and that data subsequently altered by arithmetical and logical (re-read all about AND and OR!) operations.

It can also be used as a storage location whilst shuffling data around from one register to another.

Just to give a couple of examples of using the accumulator (the mnemonics can be found in the Appendix) :=

```
LDA £$01   : Put a 1 (one) in the accumulator.
ADC $0402  : Add the contents of memory location
             0402
             to the contents of the accumulator,
             as in ADd with Carry
STA $0400  : Store the contents of the accumulator
             in memory location 0400.
TXA        : Transfer the contents of the X
             register into the accumulator.
```

As you can see, these expressions are not exactly complex. However, the 6510 can perform these at an incredibly high space, and so this apparent limitation of having to use a lot of commands to perform a relatively simple task doesn't really hinder

More Registers, and a Program Counter

The X and Y registers are the real work horses of the 6510, and one of their principal functions is to transfer data from one register to another (quite often each other!), and from one memory location to another.

They can be used in a similar way to FOR ... NEXT loops in Basic, in that they can be incremented or decremented, but only in steps of 1 at a time.

Finally they can be used to perform a number of mathematical operations on the accumulator.

Some examples : =

```
LDX £$01   : Load the X register with 1.
STX $0402  : Store the contents of the X register
             in memory location 0402.
DEY        : Decrease the content of the Y
             register by one.
CPY £$28   : Compare the contents of memory
             location 28 with the Y register.
```

The Program Counter

This keeps track of what is going on, and looking at the contents of the program counter will reveal the location of the next machine code instruction to be executed.

This is similar to the behaviour of the Basic memory locations 57 and 58, which tell you which line of Basic is currently being executed.

It is often called a 16 bit register, although in reality it is only two bytes shoved together.

This is all very well if a program is stepped through one instruction at a time, but what about subroutines? How does it remember where the program has come from, where it's going back to, and how many subroutines deep we are?

The answer is that it doesn't, but the Stack Pointer does. This, along with the Status register, allows us to perform subroutines and make decisions.

The Stack Pointer

The Stack is a 256 byte (I bet you knew that number was coming up) block of memory, that fills memory locations 256 to 511, but it fills them in a rather strange fashion.

First, what does it fill them with?

Well, one function of the Stack is to hold all the addresses during subroutine jumps, and this it does automatically.

Another purpose is to transfer data rapidly from register to register, memory location to memory location, and whether it is storing data or jump addresses, anything that goes in there is recorded from memory location 511 downwards.

In other words, location 256 is the last one to be filled.

When pulling data back off the stack, it is retrieved on a 'last-in, first-out' basis, usually abbreviated to LIFO. Thus, the last item of information that went in there is the first one to come back out again.

What keeps track of where the next empty byte of stack space is ? Well, just as the machine code program itself has a program counter, so the stack has a stack pointer, which stores where the next block of information can go.

To illustrate this properly, let's look at a concrete example.

The following (short) machine code program illustrates all the necessary points : =

```
C000 : LDA £$01   : Load the Accumulator with 1.
C002 : JSR $F6ED : Jump to Internal Subroutine
                   to check stop key.
C005 : TAX        : Transfer Contents of Acc
                   into X register.
```

Step by step, here's what happens : =

1) Find address of next instruction, i.e. C002, and put this in the program counter.
2) Execute instruction, and get back address for next one from program counter (i.e.C002)
3) Fetch next instruction, i.e.JSR $F6ED
4) Find address for next instruction (C005), and put this onto stack.
5) Put next vacant position (509, as C005 occupies two bytes) into stack pointer.
6) Put F6ED into program counter, and jump to subroutine at $F6ED.
7) Come back and look at Stack Pointer to find where last data stored: Stack Pointer says 509, so last data stored in 510 and 511.
8) Get that (i.e. C005) and put that into program counter.
9) Go and execute instruction at C005.

Now aren't you glad that all this is done automatically ?

A program may well have to find its way back through several subroutines, as programs grow in complexity. Thankfully, the stack, stack pointer and program counter take care of all this for you.

The contents of the stack can be altered by four statements, two of which concern the Accumulator. These are : =

PHA : PusH contents of Accumulator onto stack.
PLA : PuLl top of stack into Accumulator.

The status register and stack pointer can also be altered, although with care, and the commands to do this are : =

PHP : PusH status register onto stack.
PLP : PuLl status register from stack.
TSX : Transfer Stack pointer to X register.
TXS : Transfer X register to Stack pointer.

Status Register

We briefly mentioned the Status Register in the last section, and it is this which gives us the ability to make decisions based upon calculations. To a large extent, mathematical calculations would be impossible without it.

It is an 8 bit register consisting of 7 flags (the 6th one isn't used), and

we'll look at the first and arguably most important one, the carry flag.

The Carry Flag

This particular bit of the Status Register comes into operation when we want to carry out work on numbers larger than 255.

As you may recall, we outlined earlier the reasons why a single byte can only hold a number up to and including 255. This is all very well when only simple mathematics are involved (e.g. adding 1 and 1), but if that was all the 6510 was capable of doing it would be a very poor chip indeed.

Consequently, we have the ability to link two bytes together to form a single number, which allows us to count up to 65535: another number you'll be getting increasingly familiar with.

Naturally one could beyond even this, and use three or more bytes, but for now we'll stick to using just two. This is known as Double Precision.

To link two bytes together, there must be something that joins the first one to the second. This is the function of the carry flag : you can think of it as a number carrying over from one byte to another.

There are three important instructions for using the carry flag : =

BCC : Branch on Carry Clear.

This branches to another part of our machine code program if the carry flag is clear, i.e. set equal to 0. In order to ensure that this doesn't happen accidentally, and only occurs when we want it to, there is a second instruction to clear the carry flag before we perform any numerical calculations, and this is : =

CLC : CLear the Carry flag.

The third command is the opposite of the first one. That is, it branches off to another part of the program if the carry flag is set, i.e. equal to 1.

The syntax for this is : =

BCS : Branch on Carry Set.

The following short program will illustrate the point, using BCS.

```
.,  C000 18       CLC
.,  C001 A9 01    LDA £$01
.,  C003 69 01    ADC £$01
.,  C005 B0 03    BCS C00A
.,  C007 4C 03 C0 JMP C003
.,  C00A 8D 00 04 STA $0400
```

This clears the carry flag, loads the accumulator with one, adds one to it, and then checks to see if a carry has been performed (i.e. is the total in the accumulator equal to 255). If it hasn't, we JuMP back and add another 1 to the accumulator, and so on.

However, if we have reached the magic total, all the bits in the accumulator that had 1s in them (255 being equal, in 8 bit binary, to 11111111) are changed to 0s, and the carry flag is set.

Thus we branch to the part of the program that stores the contents of the accumulator on the screen (memory location $0400, or 1024 in decimal), and the result is a screen code zero displayed in the top left hand corner of the screen: the '' symbol.

Now, we'll take a look at the other flags that are affected in addition and subtraction operations.

Additional Flags

Three other flags come into play in certain addition and subtraction operations, and these are all part of the above mentioned status register.

We've already covered the carry flag. Briefly, the other three are :=

The Zero Flag (Z) : the 2nd bit of the status register, this is set if the addition of two numbers is equal to zero, otherwise it is reset.

The Overflow Flag (V) : the 7th bit of the status register, this is set if two positive or two negative numbers are added together, and the result exceeds +$7F or -$80, otherwise it is reset.

The Negative Flag (N) : the 8th bit of the status register, this is set if two signed numbers, when added together, give a negative result, otherwise it is reset.

The last two are only of importance if the numbers we're dealing with are actually signed, otherwise we can forget them and only concern ourselves with the carry flag (discussed) or the zero flag.

Zero Flag

This is set if the result of a mathematical operation is zero. For instance :=

```
LDX £$FF
DEX
```

This loads the X register with $FF, and decrements it (DEX) by one at a time. When X becomes zero the zero flag is set, and we can then jump to another part of the program.

Of course, before we can actually use this we need to know a few commands that can test to see if a result is zero or otherwise.

Two commands come into play here, and they are :=

BEQ : Branch if the result is EQual to zero.

BNE : Branch if the result is Not Equal to zero.

To illustrate this, here's a short example.

```
LDY £$FF   (load Y register with $FF).
DEY        (decrement Y register by one).
BEQ xxxx   (branch to STY command if result equal
           to zero).
JMP yyyy   (otherwise jump back to DEY command)
STY $0400  (store result at memory location $0400
           i.e. on screen).
```

It's about time we used the Y register for something! In most instances, it is exactly analogous to the X register.

Addition and Subtraction

Using the knowledge that we now have, it is a simple matter to add or subtract two numbers together, provided that the result or the numbers are not greater than 255, or less than 0.

To add numbers together that are greater than this, the following example program should help to make things clearer.

What we have to do is store each number as two bytes: the double precision mentioned earlier.

For instance, the number 1926 in decimal is equivalent to $0786. To use this number, we split it up into two parts, namely $07 and $86. These are referred to as the Most Significant Byte (MSB) and Least Significant Byte (LSB).

First of all, we need to add the two LSBs, and see if there is a carry. Well, $86 plus $86 is equal to $16,12 or carry +0C.

```
86
86
--
16,12 = carry plus 0C
```

The two MSBs, $07, added together give 14, or $0E. With the carry from the addition of the two LSBs this becomes $0F, and so the final total is $0F0C, or decimal 3852, which is indeed correct.

Phew! Let's put this into a program.

Sample Program

Our code could look something like this :=

```
.,  C000 18       CLC
.,  C001 D8       CLD
.,  C002 A9 86    LDA £$86
.,  C004 69 86    ADC £$86
.,  C006 8D 02 04 STA $0402
.,  C009 A9 07    LDA £$07
.,  C00B 69 07    ADC £$07
.,  C00D 8D 00 04 STA $0400
.,  C010 60       RTS
```

The RTS at the end of the program indicates ReTurn from Subroutine. This is there so that, when we run the program with a SYS 49152, (since this is the decimal equivalent of C000), the start of the program, we don't end up back in the monitor again, but remain in Basic.

Line by line then : =

Clear the carry flag.
Clear the decimal flag (4th bit of the status register, this indicates whether or not arithmetic is to be performed in decimal or binary : the 6510 is happier in decimal, but it doesn't really matter in a program such as this. Why put it in ? It's one way of introducing a new command.)
Load the accumulator with $86.
Add with carry $86.
Store the result in memory location $0402.
Load the accumulator with $07.
Add with carry $07.
Store the result in memory location $0400.

The result of running this program is that an O appears in the corner of the screen, with an L close by. O is screen character code 15, or $0F, and L is character code 12, or $0C.

Thus our answer is $0F0C.

You may think that we haven't added anything up at all, but just displayed things on the screen.

The answer is put on the screen so that it can be seen, and could just as well be stored in any other two memory locations in the usual MSB - LSB format, where it could have been used as part of a future calculation.

It all depends on how you look at it.

Modes of Addressing

So far, when putting numbers into the X and Y registers, or the accumulator, we have used commands of the form : =

```
LDA £$01
ADC $0402
```

and so on.

This is known as Immediate addressing, i.e. we are loading something either as a pure number, or from a memory location, and doing it directly.

However, there are many other modes of addressing, and we'll start with Implied Addressing.

Implied Addressing

You may not know it, but we've already seen this one in operation.

It is probably the easiest of all the modes to use, as the 6510 does all the work for you. Commands in this group fall into two separate categories, those where the whole work is done within the 6510 itself, and those where some external reference is necessary.

Of the first group, commands include TAX, TAY, DEX, DEY, and others like TXA, INY etc.

In the second group, the only one seen so far is RTS, return from subroutine.

Absolute Addressing

Another fairly simple one to understand, because, as the name implies, we are using absolute references.

As an example, the program for adding two numbers together stored the result in memory location $0400: we named it directly, or absolutely.

Other commands that do this include ADC, JMP, JSR (which jumps to a specified address, like JMP, but in this case an RTS will send program execution back to the statement after the JSR: we saw this when discussing the stack), and so on.

Zero Page Addressing

Really this is beyond the scope of this introduction, but briefly it involves using zero page (see memory maps in Appendix A), and most of the absolute commands can be used with it.

It is used because it executes faster than the rest of the computer's memory (since everything is limited to numbers between 0 and 255: zero page only has 255 bytes in it), but because of that the computer likes to use it as well as you!

It's best, at the beginning, to leave this one well alone.

Indexed Addressing

We'll see more of this one later, when we start performing some simple screen animation, but briefly we are calculating an address (memory location) using the contents of one register added to a specified address.

One such command is : =

STA ($0400),X

which stores the contents of the accumulator at memory location $0400 plus X. Thus if X was equal to 5, say, the contents of the accumulator would be stored at location $0405.

More of this one later.

Relative Addressing

Again, we've already seen this one, and all the branching instructions met so far use it.

Indeed, it is ONLY branching instructions that use relative addressing. For example, the program illustrating BCS (Branch with carry set) contained the instruction B0 03 : branch if carry set forward by three bytes, and execute the next instruction.

Similarly, branching on equal or not equal, plus or minus, all use this mode of addressing.

Indirect Addressing

The most complicated, but arguably the most powerful and versatile of all addressing modes, indirect addressing is none the less restricted to using numbers from 0 to 255.

As an example, we'll take the X register, and the command LDA (aa,X). This loads the accumulator with either a number (aa) or the contents of a memory location, and adds the contents of the X register to the accumulator.

Thus we can use this to retrieve strings of data occupying a whole row of memory locations, and switch them about, perform mathematical calculations on them, and generally make machine code do the sort of thing it should be doing.

Indirect Absolute Addressing

Used by one command only, the JMP command, in the form JMP (aaaa).

What happens when a program executes this command is that it jumps to the memory location (aaaa) specified, finds the number stored there, and in the next byte puts the number into the program counter, and the program then goes to the address thus stored.

Together with its ordinary mode of operation, and the JSR command, subroutines can be built up and executed.

Making Comparisons

The ability to make decisions in a machine code program relies a great deal on the ability to make comparisons between numbers, registers, and memory locations.

There are a number of commands which allow us to do this, and these are :=

CPX : ComPare the contents of a memory location with the contents of the X register.

Syntax := CPX $0404

CPY : ComPare the contents of a memory location with the contents of the Y register.

Syntax := CPY $0402

CMP : CoMPare the contents of a memory location with the contents of the accumulator.

Syntax := CMP $0400

When encountering the compare command, for example CPX $0404,

the program reads the contents of memory location $0400, subtracts that from the contents of the X register, and sets various flags depending on the result.

This can be used in various ways, and the following program illustrates just one example :=

```
.,  C000 A9 53    LDA £$53
.,  C002 8D 04 04 STA $0404
.,  C005 A2 01    LDX £$01
.,  C007 E8       INX
.,  C008 EC 04 04 CPX $0404
.,  C00B F0 03    BEQ $C010
.,  C00D 4C 07 C0 JMP $C007
.,  C010 8E 00 04 STX $0400
.,  C013 60       RTS
```

This program loads a 'heart' symbol into the accumulator, then stores it at memory location $0404 (the screen). X register is then loaded with a 1, and incremented.

Next, we compare location $0404 with the contents of the X register, and subtracting that (53) from the X register value (currently 29) does not give us a value of zero.

Hence the Branch if EQual instruction is not obeyed, and the program jumps back to increment X again.

Finally, when X equals 53, the BEQ is obeyed and the result, another heart, is printed out onto the screen.

Simple Animation

One of the major uses so far (or at least that's the way the market looks in these early days) for machine code programming on the 64 is to produce some amazing games.

The use of graphics depends to a large extent on various items that we haven't yet met, like sprite handling, bit mapping of the screen, and so on, although we will come to those in due course.

For the present, a couple of simple illustrations will serve to show the power of machine code, and the speed with which animated displays can be moved.

This first program simply prints a row of 255 hearts onto the screen: about 6 1/2 lines worth.

You may have to change the background colour (see chapter 4) to be able to see them, but the point to note is the sheer speed with which things happen.

```
.,  C000 A9 53    LDA £$53
.,  C002 A2 01    LDX £$01
.,  C004 E8       INX
.,  C005 F0 06    BEQ $C00D
.,  C007 9D 00 04 STA $0400,X
.,  C00A 40 04 C0 JMP $C004
.,  C00D 60       RTS
```

Quite simply, the heart character code is loaded into the accumulator, and a 1 is loaded into the X register, which is then incremented.

A test is made for X being equal to zero, which it will be when it is incremented up from being equal to 255: it flips back to 0 again.

However, until then it isn't zero, and so the contents of the accumulator are stored at location $0400, offset with X, and we jump back to increase X again.

You may have been puzzled by the different natures involved in jumping and branching commands: well, jumping usually uses direct addresses (you tell it which location to go to), and branching uses relative ones, as we've seen.

Thus, to calculate a branch jump, you just have to work out how many bytes the program needs to go forward or backwards. Jumps forward are indicated by using the numbers 0 to 127 (e.g. 50 means 'jump forward 50 bytes and execute the next instruction found'), and backward jumps by the numbers 128 to 255 (e.g. 220 means 'jump backwards 256 - 220, or 36 bytes').

Some More Animation

A well known technique when moving things about on the screen is to print something, and then fill the space behind it with a space character, thus obliterating the previous image. Then, move the character on one stage and obliterate the image in the place just moved from, and so on.

The following program accomplishes this in machine code, but is so fast that you won't be able to see what's happening!

In the next chapter we'll look at program timing, and having said very early on that one of the chief advantages of machine code is in its speed of operation, we'll also look at ways of slowing programs down!

```
.,  C000 A9 53    LDA £$53
.,  C002 A2 00    LDX £$00
.,  C004 A0 20    LDY £$20
.,  C006 8C 25 C0 STY $C025
.,  C009 8D 26 C0 STA $C026
.,  C00C 9D 00 04 STA $0400,X
.,  C00F 98       TYA
.,  C010 9D FF 03 STA $03FF,X
.,  C013 E8       INX
.,  C014 EA
.,  C015 EA
.,  C016 EA
.,  C017 EA
.,  C018 EA
.,  C019 EA
.,  C020 D0 ED    BNE $C00C
.,  C022 60       RTS
```

This program puts a heart into the accumulator, a zero into X, and the code for a space into Y. Y is then stored in memory location $C025, and the contents of the accumulator in $C026: both safely out of the way.

The accumulator contents are then stored at location $0400, offset with X, the contents of the Y register transferred to the accumulator and then stored at $03FF (i.e. a space is stored one memory location, or one screen 'square' behind the heart each time), and the X register incremented.

When X tops 255 the BNE instruction fails as X is flipped back to zero, and the proram exits.

Until then, we loop back and print out more hearts and spaces.

This technique is the basis for almost all the screen animation displays that you see in the popular arcade games.

And the NOPs ? They just tell the computer to do nothing for 2 cycles.

They're there so that, after reading the next section on timing, you can come back to this program and alter it so that you can actually see what's going on!

Timing

Appendix C has all the mnemonics used in machine code listed out for you, and with each one you'll see that we've included the cycle times. In other words, you know how long each instruction will take to execute.

That is why the previous program is impossible to watch: it takes some 30 cycles, or micro-seconds, to print and then overwrite a heart shape - that's pretty fast, and much too fast for the eye to see.

So, the use of NOPs, as mentioned in the last program, can slow us down a little, but 2 micro-seconds isn't exactly a long time. Thus we have to build up various delay programs, and the simplest one must surely be :=

```
LDX £$01 - 2 cycles
INX      - 2 cycles
BNE back to LDX again. - 3 cycles
```

This just loads a 1 into the X register, checks to see if X is zero, which it will be after flipping over from 255 to zero again, and if it isn't going back and incrementing X again.

However, this takes up a huge 1277 cycles, which still isn't very long. Thus, we have to extend the program a little, rather like this :=

```
LDY £$01 - 2 cycles
LDX £$01   2 cycles
INX      - 2 cycles
BNE      - 3 cycles (go back and increase X again
           until it equals zero
INY      - 2 cycles
BNE      - 3 cycles, go and increment Y again.
```

Thus we run through our original delay loop 255 times, which gives us a realistic delay of slightly over a quarter of a second. A little better!

There are other methods of course, but if we're going to use the 6510 for precise timing of instruments, program control or whatever, it is obviously better to use this timer than the jiffy clock. Also, every micro-

second counts, so use the table in Appendix C: some operations take longer under different circumstances.

Charget and the Interrupt

Sounding rather like a bizarre American comedy film, these are the names given to two of the most important functions within the computer.

Charget

Charget, short for CHARacter GET, is a machine code program that resides in page zero of the Commodore 64 : in fact it sits in locations 115 through 138. Thus, it is quite a short routine. However, it is also a very useful one, as it provides the link from Basic to the Interpreter.

Charget acts as follows : =

When a Basic program is running, each program line is copied from the RAM area into which you typed it, into the Basic input buffer. There the charget routine scans through it (ignoring spaces, in which case you could modify charget to remove the code that checks for spaces, which will make Basic run a bit faster, but does mean that you can't type in spaces any more!), until it finds a byte that it knows.

This is then pushed to the accumulator, program execution returns to the interpreter, where the byte is dealt with.

It is by modifying this charget routine that new commands can be added to Basic.

Using the Assembler, and disassembling the code from locations 115 to 138, allows you to do this, but be careful : charget is operating all the time, so any changes will have to make sense to it.

And the Interrupt

An Interrupt is precisely what it says it is: an interruption. Computers, like humans, don't like being interrupted on occasions, and so a number of commands in the 6510 instruction set allow you to switch off, and switch back on again, any interrupts.

These commands are :=

SEI : SEt Interrupt Disable. This stops all interruptions (i.e. stop keys, external devices and the like, although it can't cover everything).

CLI CLear Interrupt flag. This resets everything.

RTI : ReTurn from Interrupt.

Now, just to complicate things a little further, the 6510 has two different input pins. The NMI, or Non Maskable Input pin cannot be blocked, but the IRQ, or Interrupt ReQuest pin can : it is this one that is set or cleared with SEI and CLI, which are themselves only operating on the Interrupt flag, the third bit of the status register.

The interrupt procedure, from which RTI returns you, is only instigated in the case of an IRQ interrupt, whereupon the 64 makes a half-hearted attempt at keeping everything like it was before the interrupt happened.

BRK: BReaK.

This starts up another interrupt sequence, and when used in machine code programs that are running on a machine with a monitor (e.g. a 64 with the assembler working), BRK will halt program operation and drop you into the monitor, whereupon you will get the usual display of PC, IRQ, XR, YR and so on, with PC as usual displaying the address of the current statement.

Making Room

When running machine code programs on the 64, using the assembler, we are restricted to a machine with a little more than 36K (actually 36044 bytes) of program space. Ample, for most purposes.

The top of Basic memory, as far as the 64 is concerned, can be checked by coming out of the monitor, and looking at memory locations 53 and 54, which hold the LSB and MSB respectively of the top of memory.

PEEKing those two locations should give you the values of 0 and 160. 160 in hexadecimal is A0, and the number A000 in decimal is 40960: indeed, the top of Basic memory.

If you want to get a little bit more memory for your money, the

following short program will switch in another 8K of usable memory : =

```
7000 LDA $01
7002 AND £$FE
7004 STA $01
7006 RTS
```

To get back to normal, we need to reverse this process, and type : =

```
70A0 LDA $01
70A2 ORA £$01
70A4 STA $01
70A6 RTS
```

Why do this in machine code rather than Basic? If you attempt this in Basic the machine will crash.

Thanks for that one, Kevin!

As you start writing longer and longer machine code programs, it will be necessary to reserve some space for them at the top of the computer's memory, in order that they may have somewhere to live whilst being executed.

You could (by studying the memory maps) insert your code into any one of the many alternate work areas, and the chapter on the 6510 goes into this in a little more detail.

For now, assuming that you have a normal machine and a piece of code that needs to sit at the top of normal Basic memory, all you have to do is fool the machine into thinking that it's got less memory than it really has by altering the values in locations 53 and 54.

Remember that the place you want the top of memory to sit (say 32768) must be converted into hex: in this case it becomes $8000, and in its two components $80 and $00.

Back to decimal now, and these become 128 and 0, so 128 must be POKEd into location 54, and 0 into location 53.

Just by altering these numbers we can put the top of memory virtually anywhere.

Multiplication and Division

We've already seen how addition of large numbers works, and subtraction is achieved using the opposite command to ADC, namely SBC : subtract with carry.

Multiplication and division, however, are not quite so easy, as we unfortunately don't have MTC and DVC commands to multiply and divide.

So, a long-winded approach must be adopted, but considering the inherent speed of operating in machine code this is not a major problem.

But first, a lesson in binary multiplication. This will enable us to see more clearly how the 6510 handles this sort of calculation, and also lets us introduce a few more commands from the 6510 instruction set.

In decimal terms, the calculation 123 multiplied by 89 would be worked out as follows : =

```
   123
 *  89
 ------
  1107  Partial  Product 1
 0984      ,,       ,,    2
 ------
 10947
```

In other words, as we multiply by each number, we move along one row of digits.

Binary multiplication follows the same rule, and using the same numbers we get : =

```
    01111011    123
    01011001     89
-----------------
    01111011
   00000000
  00000000
 01111011
01111011
00000000
 01111011
00000000
-----------------
0101010110000011    10947
```

So, even though we've occasionally produced a result of 0, each partial product keeps moving along to the left each time we perform a multiplication.

There are two important differences in the way the computer performs this operation and the way we do.

1) It keeps a running total as it goes along: this is updated after every partial product. We wait until the end and then add the whole lot up.

2) When we multiply, each digit is examined from right to left, and each line is moved along one (i.e. is ten times more significant in decimal, or two times in binary). In other words, we move everything along one when we multiply. On the 6510 (and indeed most other microprocessors) it is far easier to rotate the current partial product along one whole row to the left (as we would) or to the right (known as high order to low order multiplication).

Two commands enable us to do this :=

LSR = Logical Shift Right.

ROR : ROtate Right one place.

Both these shift every bit along one, keeping track of carries etc. as they go.

To use signed numbers, the same kind of principles apply, and when dividing we again do the same sort of thing. Perhaps the easiest way to understand division is to perform the old fashioned, longhand

division method using decimal numbers, and then the same thing in binary.

Back to multiplication :=

The following program is a simplified version of a machine code multiplication using unsigned numbers: nevertheless it shows the principles involved.

Our two numbers are stored in locations $C050 and $C052. The result will be stored in locations $C054 and $C055 in LSB, MSB fashion.

```
LDA £$00   (clear everything)
LDX £$08   (8 bit multiplication)
LSR $50C0  (get next multiplier bit)
BCC xxxx   (does it equal one)
CLC        (it does, so add it up!)
ADC $52C0
RORA       (BCC would have come here)
ROR $54C0
DEX        (decrease bit count)
BNE yyyy   (back to LSR until all 8 bits are done)
STA $55C0  (store MSB in $C055)
RTS        (that's it!)
```

Linking to Basic

There are two ways of accessing and using machine code from Basic, rather than simple PEEKs and POKEs, which allow us to look at and alter the contents of various memory locations.

The first of these that we'll look at is USR. As we saw earlier, this sends a variable (A=USR(B)) B to the floating point accumulator.

Control then jumps to the machine code routine whose address is found from locations 784 and 785. Thus by altering those two locations prior to the USR call, we can pass variables to and from machine code subroutines, and hence mix Basic and machine code within the same program.

The following short example should serve to demonstrate this :=

```
10 PRINT "[CLR]"
20 POKE 784,0 : REM LSB
30 POKE 785,192 : REM MSB (=$C0)
40 B=123
50 A=USR(B)
60 PRINT "[CD]A =";A
```

Provided that we have earlier put an RTS into memory location $C000 (this can be achieved by using the statement POKE 49152,96), control will go to that memory location, encounter the RTS and come back, and our Basic program will then print out the content of the floating point accumulator.

SYStem Calling

As you will see from the memory maps in Appendix A, the Commodore 64 is equipped with a wealth of internal machine code subroutines that can be accommodated within your own programs.

I am not going to go into any great detail about how to use these : this is beyond the scope of the present book.

However, a few words of advice : =

1) It may seem obvious, but do study the memory maps to see where these routines start.

2) Use the assembler before attempting to use any of them. Disassemble the listing for the various routines, and see where they start and end, and where they go to whilst in operation. Some of them will have additional subroutines built into them, which cause program control to skip about all over the place.

3) Find out where these routines pick up their data from and where they subsequently store the result.

4) Bear in mind that they will be using the accumulator as much as you want to, and programs will need careful thought if they are to work correctly.

With the disassembler option in the assembler, and the complete memory maps to guide you around, using built-in routines is not as difficult as it might first appear.

Some of the programs that follow this section use the Commodore 64's own routines.

By following these through you'll get a clearer idea of what's happening.

Some Sample Programs

Three machine code programs in all, including one to define the function keys on the 64: an extremely useful aid!

Hard Copy

Another one of German origin, this one is located at $9000 onwards, and occupies just 94 bytes. Even short machine code routines are capable of doing quite a lot!

The first few lines open the printer for output (the recommended Commodore printer), and then at $9014 we jump to the Kernal routine (the Kernal is explained in much more detail later) for opening the print file.

All the JSR instructions are using built-in routines in the 64, so to go through them one by one :=

```
JSR $FFC9 : OPEN a channel for output.
JSR $FFD2 : Output a character to the channel.
JSR $FFE1 : Check for stop key.
JSR $FFD2 : Output another character.
JSR $FFCC : Close input and output channels.
JMP $FFC3 : Close specified logical file.
```

And the routine :=

```
., 9000 A9 04    LDA £$04
., 9002 85 BA    STA $BA
., 9004 A9 7E    LDA £$7E
., 9006 85 B8    STA $B8
., 9008 A9 00    LDA £$00
., 900A A0 04    LDY £$04
```

```
., 900C 85 71    STA $71
., 900E 84 72    STY $72
., 9010 85 B7    STA $B7
., 9012 85 B9    STA $B9
., 9014 20 C0 FF JSR $FFC0
., 9017 A6 B8    LDX $B8    (zero page)
., 9019 20 C9 FF JSR $FFC9
., 901C A2 19    LDX £$19
., 901E A9 0D    LDA £$0D
., 9020 20 D2 FF JSR $FFD2
., 9023 20 E1 FF JSR $FFE1
., 9026 F0 2E    BEQ $9056
., 9028 A0 00    LDY £$00
., 902A B1 71    LDA ($71),Y
., 902C 85 67    STA $67
., 902E 29 3F    AND £$3F
., 9030 06 67    ASL $67
., 9032 24 67    BIT $67
., 9034 10 02    BPL $9038
., 9036 09 80    ORA £$80
., 9038 70 02    BVS $903C
., 903A 09 40    ORA £$40
., 903C 20 D2 FF JSR $FFD2
., 903F C8       INY
., 9040 C0 28    CPY £$28
., 9042 D0 E6    BNE $902A
., 9044 98       TYA
., 9045 18       CLC
., 9046 65 71    ADC $71
.. 9048 85 71    STA $71
., 904A 90 02    BCC $904E
., 904C E6 72    INC $72
., 904E CA       DEX
., 904F D0 CD    BNE $901E
., 9051 A9 0D    LDA £$0D
., 9053 20 D2 FF JSR $FFD2
., 9056 20 CC FF JSR $FFCC
., 9059 A2 7E    LDX £$7E
., 905B 4C C3 FF JMP $FFC3
```

Defining the Function Keys

The four grey function keys can ordinarily be detected in a program with either a GET command, PEEK (197), or testing for the CHR$ of the key pressed. These three short one liners illustrate all these points.

```
10 GET A$:IFA$<>"[F1]"THEN10
```

where [F1] represents the graphical symbol that appears when F1 is pressed inside quotes.

```
10 PRINT PEEK(197):GOTO 10
```

Just running this will allow you to note the values associated with the function keys (F1 = 4, for instance), and then insert appropriate checks within your programs.

```
10 GET A$:IFA$<>CHR$(133)THEN 10
```

where CHR$(133) is F1. The others can be found in Appendix D.

However, we can do a bit more than this, as the following program shows.

It allows you to define the four function keys (eight, with shift), to be Basic keywords. i.e. F1 might become LIST + Carriage Return, for instance.

Written in Basic, it POKEs the machine code contained in the data statements in lines 100-420 into variables in lines 20-40. Line 50 tells you how to access the routine, by typing in SYS 49152 (so in other words the code resides at $C000 onwards).

The keys are defined by typing in whatever value or string you want when the prompt F1 = ? appears, and the program then flicks through the rest of the function keys.

If a Return is required (e.g. you might want F1 to equal LIST plus a Carriage Return), then type in LIST, followed by the left arrow key.

Hitting Run/Stop and Restore resets everything, and allows you to go through the program again and change the values stored in the keys.

If this is done accidentally, and you don't want to go through the program again, typing POKE 788,144:POKE 789,192 (followed by Return) will set everything back to normal again.

And the program :=

```
10 PRINT"[CLR,11CD,6CR]PROCESSING 'FUNCTION KEYS'"
20 FORX=49152TO49415
30 READ A : POKE X,A
40 NEXT X
50 PRINT"[CLR,11CD,6CR]USE SYS 49152 FOR ACCESS"
100 DATA 169,0,170,157,0,194,157,0
110 DATA 195,157,0,196,232,208,244,133
120 DATA 251,169,194,133,252,169,49,133
130 DATA 253,169,133,133,254,169,13,32
140 DATA 210,255,169,70,32,210,255,165
150 DATA 253,32,210,255,169,61,32,210
160 DATA 255,169,63,32,210,255,32,207
170 DATA 255,72,160,0,165,254,145,251
180 DATA 104,32,133,192,201,13,240,17
190 DATA 201,95,208,2,169,13,145,251
200 DATA 32,133,192,32,207,255,76,68
210 DATA 192,230,253,165,253,41,1,208
220 DATA 10,24,165,254,105,4,133,254
230 DATA 76,114,192,56,165,254,233,3
240 DATA 133,254,165,253,201,57,48,165
250 DATA 120,169,144,141,20,3,169,192
260 DATA 141,21,3,88,96,166,251,224
270 DATA 255,208,2,230,252,230,251,96
280 DATA 165,197,197,254,240,58,201,3
290 DATA 48,54,201,7,16,50,133,254
300 DATA 201,3,208,3,24,105,4,24
310 DATA 105,129,174,141,2,240,3,24
320 DATA 105,4,133,253,160,0,169,194
330 DATA 133,252,132,251,177,251,197,253
340 DATA 240,19,200,208,247,230,252,165
350 DATA 252,201,197,208,239,76,49,234
360 DATA 133,254,76,49,234,200,208,8
370 DATA 230,252,165,252,201,197,240,242
380 DATA 177,251,201,13,208,10,230,198
390 DATA 166,198,157,119,2,76,213,192
400 DATA 201,0,240,222,201,133,48,7
410 DATA 201,141,16,3,76,49,234,32
420 DATA 210,255,76,213,192,0,0,0
```

Old for New

Ever typed in NEW by mistake, and wished you could retrieve the program you've just wiped out ? This short (62) byte program allows you to do just that, by entering SYS CF00 after typing the dreaded NEW!

```
., CF00 A5 2B    LDA $2B (look for startof Basic)
., CF02 A4 2C    LDY $2C
., CF04 85 22    STA $22
., CF06 84 23    STY $23
., CF08 A0 03    LDY £$03
., CF0A C8       INY
., CF0B B1 22    LDA ($22),Y
., CF0D D0 FB    BNE $CF0A (check for zero byte)
., CF0F C8       INY
., CF10 98       TYA
., CF11 18       CLC
., CF12 65 22    ADC $22
., CF14 A0 00    LDY £$00
., CF16 91 2B    STA ($2B),Y
., CF18 A5 23    LDA $23
., CF1A 69 00    ADC $00
., CF1C C8       INY
., CF1D 91 2B    STA ($2B),Y
., CF1F 88       DEY
., CF20 A2 03    LDX £$03
., CF22 E6 22    INC $22
., CF24 D0 02    BNE $CF28
., CF26 E6 23    INC $23
., CF28 B1 22    LDA ($22),Y
., CF2A D0 F4    BNE $CF20
., CF2C CA       DEX
., CF2D D0 F3    BNE $CF22
., CF2F A5 22    LDA $22
., CF31 69 02    ADC £$02
., CF33 85 2D    STA $2D
., CF35 A5 23    LDA $23
., CF37 69 00    ADC £$00
., CF39 85 2E    STA $2E
., CF3B 4C 63 A6 JMP $A663.
```

If our German friend who developed the original version of this program would come forward, I could thank him properly.

Kevin Bergin provided some important alterations to this last one.

4

Colour on the Commodore 64

What, only five pages for one of the most important features on the computer ?

The reason for this is that colour is very strongly related to graphical work, and sprite handling and manipulation, and as such most of the material that could have been covered here has been relegated to chapter 5 : High Resolution and Sprite Graphics.

This particular section is basically a round-up of colour in general, plus a few simple ground rules to help us out when the going begins to get a bit tough in chapter 5.

One of the first things that we'll want to do is to check that all the colours are actually being displayed correctly, and that your television or colour monitor is properly tuned in.

You'll have seen the shorthand descriptions of the colours underneath the numeric keys at the top of the keyboard.

These colours are activated with the control key, or the Commodore logo key.

If we turn reverse mode on by pressing control and 9, pressing the space bar just produces a slightly murky blue line on the screen.

Now, going through each colour in turn, press control 1, press the space bar for a while, control 2, space bar, and so on up to control 8, and then move on to pressing the logo key and 1, then space bar, logo key and 2, space bar, and so on up to logo and 8.

The following short program does all this work for you, as long as you keep the space bar pressed : =

```
10 A$="[CTRL1,CTRL2 ... CTRL8,LOGO1,LOGO2 ...
LOGO8]"
20 GET B$:IFB$=""THEN20
30 PRINTMID$(A$,(INT(RND(.5)*LEN(A$))+1)),1)"[
RVS] ";:GOTO20
```

Or, if you can't be bothered to press keys, remove line 20 and change the end of line 30 to read GOTO 30.

The string A$ is defined to hold all the colours, then in line 30 we just pick one of them out at random, and display it as a reverse field space.

Colour Registers

As we've seen, the text colour can be changed by using the control or logo keys, and this same effect can also be gained by using the CHR$ command.

Appendix D lists the CHR$ numbers associated with the various colours.

However, this isn't much good if we can't change anything else, like the background colour and the border colour.

Two memory locations allow access to this, and these are locations 53281 and 53280 respectively.

There are other registers for colour alteration, but we'll see more of those in chapter 5.

For now, back to 53281 and 53280.

POKEing one or other of these locations with a number between 0 and 15 will change either the background or the border colour, and the numbers to use are :=

0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	LIGHT RED
3	CYAN	11	FIRST GREY COLOUR
4	PURPLE	12	SECOND GREY COLOUR
5	GREEN	13	LIGHT GREEN
6	BLUE	14	LIGHT BLUE
7	YELLOW	15	THIRD GREY COLOUR

The following short program allows you to scan through all the border/background combinations.

```
10 FORI = 0 TO 15
20 FORJ = 0 TO 15
30 POKE 53280,I
40 POKE 53281,J
50 FORK=1TO1000:NEXTK
60 NEXTJ
70 NEXTI
```

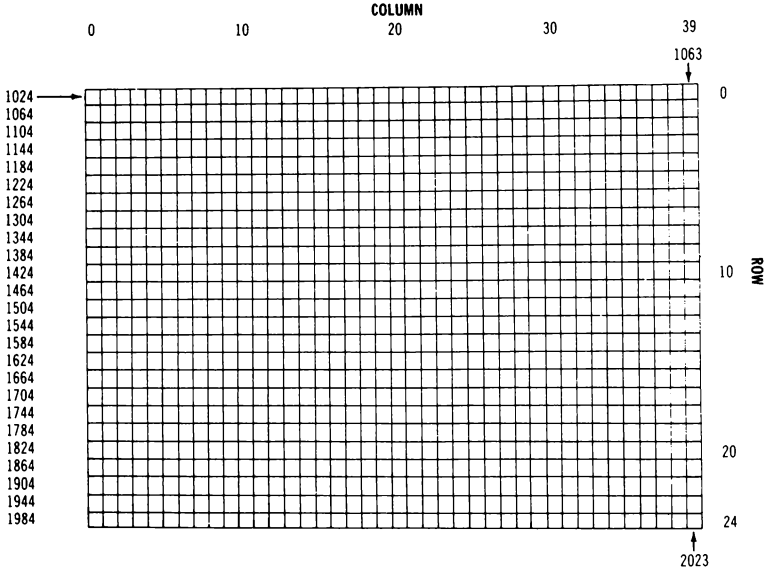
The two outer loops change border (I) and background (J), whilst the inner (K) loop is just a display to let you see what's happening. Without that it all looks a little sickly!

Screen and Colour Memory Maps

As we've already seen, the first screen memory location is number 1024, and since the screen is 40 columns across by 25 rows down, this means that we have 1,000 locations on the screen (40 times 25).

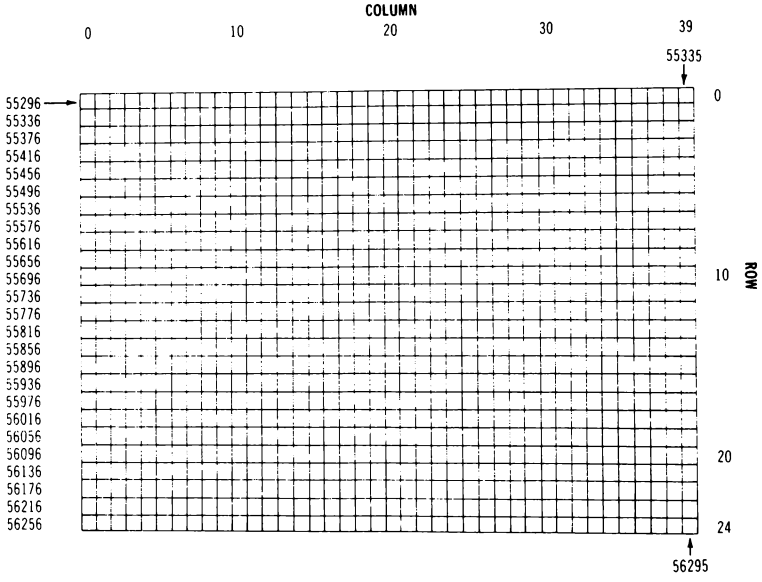
Thus the last screen location is number 2023, one thousand on from 1024.

The screen memory map looks like this : =



The colour screen memory map lives totally independently of the ordinary screen one, and starts at memory location 55796. It finishes on 56295, a thousand locations further on.

The whole memory map looks like this : =



Putting Coloured Characters on the Screen

To make a character appear on the screen, we can either print it there, or POKE it there.

Screen memory is exactly like any other. POKEing to that memory simply alters the contents of the relevant location, and thus something different appears there. In this case it also appears on the screen, because that is where the change is taking place.

To POKE something (a letter A) into the top left-hand corner of the screen, we must : =

```
POKE 1024,A
```

To put something anywhere on the screen, we need to know the location of that point on the screen. This can be found from the

memory map, or from the simple formula : =

```
POKE 1024 + X + 40*Y
```

where X is the X co-ordinate (number of columns across), and Y is the Y co-ordinate (numbers of rows down) of the point we want to put something at.

To change the colour of a character on the screen, we also need to change the relevant colour screen memory location.

The same formula, slightly modified, will allow us to do this : =

```
POKE 55296 + X + 40*Y
```

We use the same colour numbers as were given for the background and border colours.

Thus, to put a letter A 21 columns across and 12 rows down we must: =

```
POKE 1024 + 21 + 40*10,65 or POKE 1445,65.
```

To change it to a red letter A, we must then POKE 55296 + 21 + 40*10,2 or POKE 55717,2.

You may find it easier to remember the colour locations as being (55296 - 1024), or 54272 locations further on. So, our earlier POKEs now become : =

```
POKE 1445,65 : POKE 1445 + 54272,2.
```

5

High Resolution and Sprite Graphics

The Commodore 64 is an extremely powerful graphical computer, with a maximum resolution of 320 by 200 pixels, capable of displaying 16 different colours at once, and utilising graphical images known as Sprites for ease of producing high resolution animation.

In this chapter we're going to look at all of this in turn, with a number of sample programs to get you started and to display just some of the capabilities of this computer.

Few of these programs are intended to be the finished article. However, all should give you some idea of just what can be achieved.

However, as with all things, there is a drawback to all these wonderful capabilities.

You're going to have to go through an awful lot of theory before we can start putting things into practice!

So, just to let you relax for a moment we've included a games listing, called Trap!

You control a little blob roaming around the screen, which leaves a trail behind it as it goes.

The computer is controlling another, different coloured, one.

The object of the game is to trap your opponent, in this case the computer.

All instructions are included in the program, and notes on the more relevant parts follow the listing.

Perhaps, when you've finished reading this chapter, you'd like to convert it into moving sprites around the screen, rather than just simple blobs ?

And after chapter 6, you could add some sound as well.

Trap! A game for the Commodore 64

Explanation

Line 10 : Subroutine to start game and explain rules.

Lines 9-12 : Place blobs on screen!

Line 14 : Generate computer move up, down, left or right.

Lines 20 - 45 : Scan keyboard for your movement.

Lines 50 - 65 : going down!

Lines 75 - 90 : and left

Lines 100 - 120 : and right

Lines 125 - 140 : and up

Line 2000 : end of game because you've crashed.

Lines 3000 - 3010 : move computer blob.

Line 3012 : computer crashed, grr!

Lines 3500 - 3507 : panic, computer getting boxed in!

Lines 4000 - 4060 : end of game, do you want another?

And now, the listing : =

```
8 GOSUB5000
9 POKE53280,1:POKE 53281,14
10 I=500:POKE 1024+I,214:POKE55296+I,3
12 J=250:POKE 1024+J,42:POKE 55296+J,7
14 Q$="UDLR":W$=MID$(Q$,RND(.4)*3+1,1):GOTO50
20 A=PEEK(203)
25 IF A=36THEN50
30 IF A=10THEN75
35 IF A=18THEN100
40 IF A=33THEN125
45 GOTO 20
50 IFI>960THENI=I-1000
52 FORD=1TOL:NEXT
54 IF PEEK(1024+I+40)<>46 THEN 2000
55 I=I+40:POKE 1024+I,214:POKE 55296+I,1
56 GOSUB 3001
60 A=PEEK(203):IFA=64THEN50
65 GOTO25
75 IF INT(I/40)=I/40THENI=I+40
76 IF PEEK(1024+I-1)<>46 THEN 2000
78 FORD=1TOL:NEXT
80 I=I-1:POKE 1024+I,214:POKE 55296+I,1
81 GOSUB 3001
85 A=PEEK(203):IFA=64THEN75
90 GOTO25
100 IF INT((I-39)/40)=(I-39)/40THENI=I-40
102 FORD=1TOL:NEXT
104 IF PEEK(1024+I+1)<>46 THEN 2000
105 I=I+1:POKE 1024+I,214:POKE55296+I,1
106 GOSUB 3001
110 A=PEEK(203):IFA=64THEN100
120 A=PEEK(203):GOTO25
125 IFI<40THENI=I+1000
126 IF PEEK(1024+I-40)<>46 THEN 2000
127 FORD=1TOL:NEXT
130 I=I-40:POKE 1024+I,214:POKE55296+I,1
131 GOSUB 3001
135 A=PEEK(203):IFA=64THEN125
140 GOTO25
2000 PRINT"[CLR,WHT]TOUGH!":C=C+1:GOTO 4000
3000 Q$="UDLR":W$=MID$(Q$,RND(.4)*3+1,1):
3001 K=INT(RND(.5)*10):IFK>8THEN3000
3002 Q=J
3003 IFW$="U"THENJ=J-40:IFJ<0THENJ=J+1000
3004 IFW$="D"THENJ=J+40:IFJ>1000THENJ=J-1000
3006 IF W$="L"THENJ=J-1:IF INT((J+1)/40)=(J+1)/40
THEN J=J+40
3008 IF W$="R"THENJ=J+1:IF INT((J-40)/40)=(J-40)/4
0 THEN J=J-40
3009 IF PEEK(1024+J)=46 THEN POKE 1024+J,42:POKE 5
5296+J,7:S=0:RETURN
```

```
3010 GOTO 3500
3012 PRINT"[CLR,WHT]GRRR...":H=H+1:GOTO 4000
3500 P$=W$
3501 IFP$="U"THENW$="D":J=0:GOTO3002
3502 IFP$="D"THENW$="L":J=0:GOTO3002
3503 IFP$="L"THENW$="R":J=0:GOTO3002
3504 IFP$="R"THENS=S+1
3506 IFS=3THENS=0:GOTO3012
3507 W$="U":J=0:GOTO3002
4000 POKE 53280,14:POKE 53281,6
4005 FORI=1TO10:GETF$:NEXT
4010 PRINT"[2CD]SCORE NOW STANDS AT YOU ";H:PRINT"
AND THE COMPUTER ";C"
4020 PRINT"[CD]ANOTHER GAME (Y OR N)"
4030 GET F$:IFF$="" THEN 4030
4040 IF F$="Y"THEN8
4050 IFF$="N" THEN PRINT"[CLR]BYE":END
4060 GOTO 4020
5000 PRINT"[CLR,YEL]WELCOME TO THE GAME OF TRAP!"
5010 PRINT"[CD]THE OBJECT OF THE GAME IS TO TRAP T
HE "
5020 PRINT"COMPUTER SO THAT IT CAN'T MOVE"
5030 PRINT"[CD]OF COURSE, IT IS TRYING TO DO THE S
AME"
5040 PRINT"TO YOU!!"
5050 PRINT"[CD]PRESS M TO MOVE DOWN, A LEFT, D RIG
HT, "
5060 PRINT"AND I UP"
5070 PRINT"[CD]DO YOU WANT A FAST, MEDIUM OR SLOW
GAME"
5080 PRINT"PRESS F, M, OR S"
5090 GETD$:IFD$=""THEN 5090
5100 IF D$="F"THENPRINT"FAST!":L=0:GOTO5120
5101 IF D$="M"THENPRINT"MEDIUM!":L=125:GOTO5120
5102 IF D$="S"THENPRINT"SLOW!":L=250:GOTO 5120
5110 GOTO 5090
5120 PRINT"[CD]PRESS SPACE BAR TO START"
5130 GETSD$:IFSD$<>" "THEN 5130
5131 PRINT"[CLR.BLK]";
5132 FORI=0TO998:PRINT".";
5134 NEXT:POKE2023,46:POKE 56295,0
5140 RETURN
```

Right, enough of fun, down to some theory.

Different Graphics Modes

There are a number of different display modes on the Commodore 64, and later on in this chapter we'll be telling you how to set and utilise each one.

For now, some definitions, starting with :=

Standard Character Mode

This is the mode taken by the 64 whenever you turn it on. This allows you to display all the 62 graphical symbols which are accessed from the keyboard: the information for these 8 pixel by 8 pixel characters is stored in the computer's ROM.

There is another 'standard' character mode, which involves switching out the computer's ROM, and getting the character information from RAM. This allows you to build up your own character sets (e.g. for writing in foreign languages).

In addition, we can program standard characters in multi-colour mode, or in extended background colour mode. More of this later.

High Resolution Graphics

Sometimes known as BIT mapping, since we can alter the value of any pixel, or bit, on the screen, this uses the maximum graphics capability of the 64, i.e. 320 by 200 pixels.

Multi-Colour High-Res Mode

This is closely connected with high-res graphics, and allows you to display multiple colours in high resolution, but with a corresponding loss in resolution.

In fact, in multi-colour mode the maximum resolution available is just 160 by 200, still quite powerful, and we do have the extra ability to address each pixel in one of four different colours, as opposed to two in ordinary high resolution.

This forms the bulk of this chapter, and as with everything else we can have 'normal' sprites, which are moveable characters 24 pixels wide by 21 pixels deep, or multi-colour sprites, which are 12 pixels by 21.

Sprites are sometimes called MOBs, for Moveable Object Blocks, and we'll see why shortly.

Colour and Character Memory

Before we can start playing with anything, we need to know where the computer stores all its information.

The memory maps in appendix A will show you most things, but colour memory commences at memory location 55296, with the corresponding screen memory starting at location 1024.

However, as we've seen there are various different character sets and graphics modes, and screen and colour memory are used differently in each mode.

Normally, the 64 gets its character information (i.e. the shape used to make up each character) from the character generator ROM, which starts at location 53248 and continues up to 57344, in 8 blocks of 512 bytes each.

In order, these are stored as follows : =

Block	Address		Contents
	Decimal	Hex	
0	53248	D000-D1FF	Upper Case Characters
0	53760	D200-D3FF	Graphics Characters
0	54272	D400-D5FF	Reverse Case Upper Case Characters
0	54784	D600-D7FF	Reversed Graphics Characters
1	55296	D800-D9FF	Lower Case Characters
1	55808	DA00-DBFF	Upper Case & Graphics
1	56320	DC00-DDFF	Reverse Case Lower Case Characters
1	56832	DE00-DFFF	Reversed Upper Case & Graphics Characters.

To use this information, we'll need to know how to turn character sets on and off, so ...

Choosing your Characters

The VIC Chip which controls all of the graphics on the computer, like any other 8 bit chip, can only 'see' 16K of memory at a time, so it has to be told which block of 16K you want it to look at.

There are four blocks of memory in the 64 (as you might imagine!), and to swap from one to the other we need to do something like : =

```
POKE 56578,PEEK(56578)OR3: REM SETTING BITS 0 AND
1 OF PORT A OF 6526 CHIP2 TO ZERO


POKE 56576,(PEEK(56576)AND252)ORA: REM SWOP FROM
BLOCK TO BLOCK
```

Where, if A = 0, we're looking at locations $C000-$FFFF
(starting at 49152)
if A = 1, we're looking at locations $8000-$BFFF
(starting at 32768)
if A = 2, we're looking at locations $4000-$7FFF
(starting at 16384)
if A = 3, we're looking at locations $0000-$3FFF
(starting at 0)

On power up, we're always looking at locations $0000 to $3FFF, or block 0.

To swop character sets in and out, a few more things have to be done. You may have noticed, if you look at the memory maps at the back, that the locations occupied by the character ROM are the same as those occupied by the VIC chip control registers. There's nothing to worry about, because of the block switching procedure they're never in the same place at the same time.

To change the location of character memory, the following syntax is used : =

```
POKE 53272,(PEEK(53272)AND240)ORA
```

where the value of A obviously determines where character memory will now sit. A value of 0 starts it at zero, 2 at 2048, and going up in

blocks of 2048 bytes A succesively takes the values 4,6,8,10,12 and 14, whereupon character memory will start at $3800, or decimal 14336.

When swopping character sets around, we need to block off all interrupts to the computer (remember interrupts!). This is done by : =

```
POKE 56333,127
```

Since the character ROM is sandwiched between the Input/Output ROM on top of it and User RAM underneath it, we must also switch out the I/O ROM, and this is done by : =

```
POKE 1,51
```

Now we can read our character information from ROM and store it in RAM: a convenient place to house it.

The following line copies 128 characters from ROM (in fact, the first 128 characters, from location 53248 upwards) and put them into RAM: =

```
FORX=0TO1023:POKE 53248+X,PEEK(53248+X):NEXT
```

Now then, this means that the first 128 characters of character ROM are sitting in locations 53258 to 54271 (1024 bytes further on, as each character information occupies 8 bytes, and we've just moved 128 of them), so locations 54272 and upwards are now available for our own use.

Thus we now need to know how to define our own characters, so let's take a look at a typical character, the letter A : =

Defining Characters

Every character on the 64 is designed as an 8 by 8 pixel grid, like this : =

00011000
00111100
01100110
01111110
01100110
01100110
01100110
00000000

where the zeros tell the computer that that pixel is to be turned off, and the ones that it is to be turned on. Thus, the letter A is built up.

To define our own character, it too must be created on an 8 by 8 grid. Let's define a little alien : =

00011000
00111100
01011010
11111111
01100110
01100110
11000011
11000011

Not particularly awe-inspiring, but it will serve to illustrate the point!

We need to tell the computer which pixels to turn on, and which to turn off, and this is acheived by assigning a number to each column of our grid, like this : =

```
ABCDEFGH
00011000
00111100
01011010
11111111
01100110
01100110
11000011
11000011
```

where A = 128, B = 64, C = 32, D = 16, E = 8, F = 4, G = 2 and H = 1

Now we look at each row of the grid, and wherever we see a number 1 we add the appropriate value for that column. Thus our little alien now becomes a series of numbers for each row : =

24, 60, 90, 255, 102, 102, 195, 195.

Row 1, for instance, was made up of 0 + 0 + 0 + 16 + 8 + 0 + 0 + 0, equals 24, and so on.

Going back to our earlier program, we can now add the following lines to read the data for our new character, and put it immediately after the first 128 characters read from ROM. This is done by : =

```
FOR X=0TO7
READ A:POKE 54272+X,A
NEXT X

DATA 24,60,90,255,102,102,195,195
```

Obviously, you'd be making up more than one new character, and so the range of the FOR ... NEXT loop would be increased, and the data statements would be extended.

All we have to do now is turn I/O back on again, switch interrupts back on again, and tell the VIC chip where its video memory has gone to!

This is done with : =

```
POKE 1,55:POKE56333,129
POKE 648,196:POKE 56576,4:POKE 53272,21
```

So, video RAM now starts at 50176, character ROM (or more accurately RAM) now starts at 53248, but colour memory stays where it is.

There are a couple of side effects to doing all of this : sprites now become 24 x 24 pixel characters, their data pointers (see later) now go from locations 51192 to 51199, and to find where you must now store your sprite data, use the formula (49012 + (74*A)), where A is the data block you want to point the sprite at.

Remember also that sprites are now 72 bytes, not the usual 63.

Thanks to D. Bowler for a lot of the information contained here.

Multi-Colour Mode Graphics

Multi-colour mode can be used with either high resolution standard mode graphics, or ordinary standard graphics, and it is a way of getting more than just two colours into each character square.

Normally, when displaying characters on the screen, they can take on one of two colours. Either the screen background colour, or the character colour reserved for that screen position.

Thus we can display many colours on the screen at once, but our

control of colour within each character space is limited.

Multi-colour mode allows us to get around this, but with a lowering of the maximum resolution available on the screen. We are limited to a maximum screen resolution of 160 pixels by 200 pixels, or in other words half the normal horizontal resolution.

This is more than compensated for by the additional display facilities gained, and high-res multi-colour mode can produce some staggering results.

For now we'll stick to ordinary characters, and look at the registers that have to be altered to get us into this mode.

To turn on multi-colour mode, you'll need to address memory location 53270 thus : =

```
POKE 53270,PEEK(53270)OR16
```

and to turn it off again,

```
POKE 53270,PEEK(53270)AND239
```

Multi-colour mode is actually set on or off for each character space on the screen, so that it is possible to mix multi-colour and hi-res graphics on the same screen, if required.

This is done by choosing your colours carefully. If the number in colour memory for that space is less than 8, then that space is produced in ordinary hi-res mode. If the number lies between 8 and 15 then we're into multi-colour mode for that particular space. This is simply altering bit 3 of the colour memory (beginning at decimal 55296, and never moving, unlike just about everything else on the 64.

The colours that can be used in multi-colour mode are dictated as follows : =

Each dot in a multi-colour space (8 pixels by 8 pixels, but remembering that each horizontal pixel is twice as big as a normal one) can take on one of four colours : =

The screen colour, from background register number 0
The colour in background register number 1
The colour in background register number 2
The character colour

These memory locations are respectively : =

53281 ($D021) : bit pair 00
53282 ($D022) : bit pair 01
53283 ($D023) : bit pair 10
Colour specified by colour memory, starting at 55296: bit pair 11

It is the bit pairing that determines what colour will be displayed in each pixel.

Take, for example, our old friend the letter A, made up in normal high res graphics like this : =

00011000
00111100
01100110
01111110
01100110
01100110
01100110
00000000

In normal mode, wherever there's a 1 the pixel is turned on (character colour), and where there's a zero it's turned off (screen background colour).

In multi-colour mode we use the bits in pairs (hence the halving of horizontal resolution) like this : =

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

By reference to the above table we can see which colour each one part of the letter A will come out as.

By bearing all this in mind, and utilising some of the character definition techniques as described earlier, some quite stunning displays can be made without resorting to sprites.

To give you just a short programming example, which prints a number of different coloured characters onto the screen :=

```
10 POKE 53281,7 : REM YELLOW
20 POKE 53282,1 : REM WHITE
30 POKE 53283,8 : REM ORANGE
40 POKE 53270,PEEK(53270)OR16 : REM TURN ON MULT
I-COLOUR
50 PRINT"[CLR]ABCDEFGIHJ"
60 FORI=0TO9:POKE 55296+I,8:NEXT
```

which produces a nicely horrible display on the screen, illustrating the variety of colours that can be achieved.

To finish with multi-colour mode (for the time being), it is possible to change instantly the colour of every dot drawn in a particular colour. Thus everything drawn in the screen and/or background colours can be changed immediately.

All you have to alter is either the first or the second background colour!

Finally, a quick way of getting into multi-colour mode is to turn it on in the usual way with the POKE 53270 etc., and then change the colour of everything using the logo key. Thus, logo and 5 will give you a grey colour, or multi-colour purple.

Extended Background Colour Mode

This is a further extension to multi-colour mode, and gives you the ability to choose both background and foreground colours for each character. You lose a few colours, like all the ones normally accessed with the logo key, but you do get a much greater variety of choice.

For example, you could display a white character with a blue background on a black screen, or something like that.

It is invoked with the following command :=

```
POKE 53265,PEEK(53265)OR64
```

and turned off with :=

```
POKE 53265,PEEK(53265)AND191
```

As usual, there is a trade-off for this, and we are now restricted to using just the first 64 characters of character ROM, or of course the first 64 characters of your own user defined character set.

This is because we need to know which background colour to display everything in, and this is determined by two bits of the character code, like this :=

For 0 to 63 : select background colour with location 53281
For 64 to 127 : select background colour with location 53282
For 128 to 191 : select background colour with location 53283
For 192 to 255 : select background colour with location 53284

A look at the list of screen codes in appendix D will show you which characters you can now use.

Choosing screen values greater than 63 will simply reflect back onto the screen the character corresponding to itself in the first 64 screen codes, but in a different background colour.

Thus POKEing a 72 to the screen will still produce a letter H on the screen, but in a different background colour than if you'd just POKEd an 8 onto the screen.

Bit Map Graphics

Down to the nitty-gritty now, with a look at some of the amazing features of the 6566 video chip.

As you know, the screen display on your Commodore 64 is 40 columns wide, and 25 rows deep. Since each character space is made up of an 8 pixel by 8 pixel grid, this gives us our maximum resolution of (40 * 8) wide and (25 * 8) deep, or 320 by 200.

To use a high resolution screen, obviously we are going to have to look after every single pixel on that screen, to determine whether that pixel is to be turned on, or off, and on this depends the colour that will be displayed in that particular pixel location.

To keep track of all this requires a lot of memory: 8000 bytes per screen display in fact, as we have to control 320 * 200 pixels, or 64000 of them! Since there are 8 bits in a byte, this gives us our 8000 bytes per screen.

Due to this massive consumption of memory, high resolution bit

mapping isn't invoked as often as you might imagine. Two screens full is the sensible limit when using the 64. On mainframes with this kind of capability memory doesn't really come into it, but we are somewhat limited on the humble home computer.

Again, another drawback of using bit mapping from Basic is that everything is so slow. With just short of 8K to control, this is hardly suprising, and most of the work in bit mapping is done from machine code.

However, delving into machine code straight away would be a little terrifying, so we'll stick to Basic.

What we're doing in this section is basically transferring an 8K section of your computer's memory directly onto the screen, and thus determining whether each bit on the screen is on or off.

As the only two Basic commands which allow us to transfer memory from one place and store it in another are PEEK and POKE, you begin to get some idea of why all this is so slow!

Standard Bit Mapping

As with normal character displays on the screen, standard mode gives you fewer colours, but greater resolution: the full 320 by 200, in fact, but only two colours per 8 pixel by 8 pixel grid.

Bit map mode is turned on with the following command : =

```
POKE 53265,PEEK(53265) OR32
```

and turned off with

```
POKE 53265,PEEK(53265)AND223
```

Obviously we're going to have to get our information from somewhere, and for this we'll have to clear out a section of memory. The usual area used is from memory location 8192 upwards, so to clear out our 8K we must : =

```
FORI=8192TO8192+7999:POKEI,0:NEXT
```

which takes a long time.

Now we've got to select a few colours, and the colour displayed on the screen in this mode is determined not by the colour memory, but by the actual content of each screen memory location. The value POKEd into a screen location produces the background colour from the lower 4 bits of that value, and the pixel colour from the upper four bits. Thus each screen character space can have two colours in it, and throughout the screen we can use any of the 16 colours.

Phew! Let's take a look at a few examples.

Before printing anything on the screen, we need to tell the screen where our bit map is stored, and this is done with : =

```
POKE 53272,PEEK(53272) OR8
```

which puts the bit map at locations 8192 through to 16191.

If we don't tell the screen where to go, interesting things happen! For instance, POKE 53265,59 will show the top half of the screen containing the bit map for the first 4096 memory locations, and the bottom half from the character generator area. You can actually watch it all change up at the top, as page zero continues to monitor what's happening.

The following line : =

```
FORI=8192TO8511STEP8:POKEI,255:NEXTI
```

will now produce a hi-res line across the top of the screen, given that we're still looking at locations 8192 upwards for our hi-res area.

Or again : =

```
FORI=1024TO2023:POKEI,4:NEXTI
```

will produce a purple hi-res screen.

To determine whether any particular pixel is to be on or off, we'll need to find it on the screen, and the following formula will show you where any pixel is calculated as follows, assuming we want it to be at X location horizontally, and Y location vertically : =

R =	INT(Y/8)	:	find the row
C =	INT(X/8)	:	find the character position
L =	Y AND 7	:	the line of that character position
B =	7-(XAND7)	:	the bit of that byte

Putting them all together gives us the byte where any pixel with the co-ordinates X,Y is situated : =

```
BYTE = 8192 + R*320 + C*8 + L
```

and to turn any X,Y co-ordinate bit on in that 8 by 8 space we : =

```
POKE BYTE,PEEK(BYTE) OR (2 to the power B)
```

We'll use some of the theory that we've learnt to get the 64 to draw a hi-res cosine wave.

```
5 POKE 53272,PEEK(53272)OR8
7 POKE 53265,PEEK(53265)OR32
10 FORI=8192TO8192+7999:POKEI,0:NEXT
12 FORI=1024TO2023:POKEI,1:NEXTI
15 FORX=0TO319
20 Y=INT(100+80*COS(X/10))
25 C=INT(X/8)
30 R=INT(Y/8)
35 L=YAND7
40 BYTE=8192+R*320+8*C+L
45 B=7-(XAND7)
50 POKE BYTE,PEEK(BYTE)OR(2^B)
55 NEXTX
60 POKE 1024,16
70 END
```

Multi-Colour Bit Mapping

This is similar to multi-colour mode in ordinary graphics, as we are again allowed to have up to four colours per 8 pixel by 8 pixel grid, but we have to suffer a halving of the horizontal resolution available, down to 160 by 200 pixels.

Again, we are using an 8K section of memory, and our four colours are chosen from : =

Screen Background Colour, register 53281.
Character Screen Position, where the upper four bits give us one colour, the lower four another.

Colour Memory

To turn multi-colour bit mapping on, we must : =

```
POKE 53265,PEEK(53265)OR32:POKE 53270,PEEK(53270)
OR16
```

and to turn it off again : =

```
POKE 53265,PEEK(53265)AND223:POKE 53270,PEEK(5327
0)AND239
```

Referring back to the section on ordinary multi-colour mode, the bit pairs are now read as follows : =

00 takes on the screen background colour.
01 comes from the upper four bits of screen memory.
10 from the lower four bits.
11 from the colour memory.

Moving the Screen about

With all these capabilities at your disposal, it is not suprising to learn that the Commodore 64 can graphically turn its hand to almost anything.

It is possible, for instance, to move the screen either horizontally or vertically in either direction, one pixel at a time!

This kind of high resolution movement makes a lot of things possible, and it is achieved in the following manner.

The 64 normally displays a screen that is 40 columns across and 25 rows down, but in order to scroll in either direction we can change this into a 38 by 24 display, in order to give the screen information somewhere to go to, and come from.

To go into a 38 column screen display, we must enter : =

```
POKE 53270,PEEK(53270)AND247.
```

and to get back again we must : =

```
POKE 53270,PEEK(53270)OR8
```

To get to a 24 row screen display, we must enter : =

```
POKE 53265,PEEK(53265)AND 247
```

and to go back to 25 rows again :=

```
POKE 53265,PEEK(53265)OR8
```

Whilst all this is going on, the screen border will expand and shrink again in size accordingly, in order to accommodate the screen manipulation.

To scroll horizontally, we must :=

```
POKE 53270,(PEEK(53270)AND248)+X
```

where X is the screen position from 0 to 7, and to scroll vertically :=

```
POKE 53265,(PEEK(53265)AND248)+Y
```

where Y is the Y position of the screen from 0 to 7.

To see all of this in action, a few examples :=

Values in the range 24 to 31 actually control the vertical position of the characters on the screen, so :=

```
FORJ=24TO31:POKE53265,J:NEXTJ
```

will set the screen moving downwards, leaving an empty space near the top. POKE53265,27 to get back to normal.

To illustrate 24 column mode, type POKE 53265,19, which cuts the top and bottom lines in half, and this is the basis of all these scrolling operations.

Switch to a 24 character screen, move everything up slowly, then jump back to a 25 character screen again, and so on.

Finally, to turn the screen off completely, type POKE 53265,11.

POKEing 53265 with 27 always sets everything back to normal again.

There's a lot of material to follow in all of that, and the only way to understand it all is just to play around, taking notes of everything you do and the results that follow.

You can't harm your computer, and Run/Stop and Restore will usually get you out of trouble without too much bother.

Sprites, or Moveable Object Blocks

Although we've already seen that the Commodore 64 is capable of some extremely good graphical displays, there's one aspect of its capabilities that we've yet to look at, namely sprites, otherwise known as Moveable Object Blocks (MOBs).

Commodore calls them sprites, everyone else calls them MOBs, so I guess we'd better stay with sprites.

Sprites are just another way of displaying objects on the screen, albeit a very powerful one.

With sprites, we can move them about all over the screen, make them pass in front of or behind other things on the screen, detect when a couple of sprites have collided, expand them in size horizontally and vertically, and generally create some amazing displays.

Your Commodore 64 is capable of defining up to 255 sprites, although for screen display purposes a practical limit is 8. This is usually enough for most applications.

So what, precisely, is a sprite?

It is a 24 pixel wide by 21 pixel deep object, which is defined in a very similar way to the programmable character exercise earlier.

Being 24 pixels by 21, the data for a sprite is stored in 63 memory locations, and when using individual sprites there are a number of memory locations to keep track of, not least telling the computer where to get the data for each sprite from!

The 47 registers that look after sprites start at memory location 53248, and they are defined as follows : =

Address (53248 +)	Description
00	X position of sprite 0
01	Y position of sprite 0
02-15	Ditto for sprites 1 through 7
16	Most Significant Bit of X position More of this one later
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
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	,, ,, 1
35	,, ,, 2
36	,, ,, 3
37	Sprite multi-colour 0
38	Sprite multi-colour 1
39	Colour for sprite 0
40-46	Ditto for sprites 1 through 7

Armed with the above knowledge, let's start creating a few sprites.

Later on we'll give you a sprite creation program, to make life a lot easier, but for now we're going to do it all the long way: it gives you a much better idea of how sprites work.

Building up a Sprite

Remembering the way in which we defined a programmable character, we need to create a sprite on a grid that is 24 by 21 pixels in area, and each pixel can, as before, be either on or off.

In standard sprite mode, if it's on we get that bit displayed in the sprite's foreground colour, and if it's off it displays whatever data happens to be behind it. Thus sprites travel over normal screen data.

In multi-colour sprite mode, we can have four colours defined for each sprite, but a sprite becomes only 12 characters across. Thus, as usual, multi-colour mode means a loss of horizontal resolution.

The data for each sprite is stored in a 64 byte block of memory, for

although each sprite takes up 63 bytes, a further byte is reserved to make life nice and simple for the computer. Commodore call this last byte a place holder : it's another way of saying that 64 bytes is a much easier number for the computer to deal with than 63!

Where are all these bytes stored? Normally, they are stored all over the place, wherever there happens to be room, and the memory maps at the back will show you where some of them normally live.

However, as we'll see later we can put sprites virtually where we want to, as long as we make appropriate adjustments.

And how does the computer know where they are stored? Because we tell it, using memory locations 2040 to 2047, the last 8 bytes of the 1K reserved for screen memory, with each byte controlling one of the 8 sprites on the screen.

If you move the screen around, these sprite pointers move also, but at least they always stay together.

To show where the data for each sprite is stored, an appropriate value must be POKEd into one of these 8 locations.

For instance, to tell the computer that sprite 1 is stored in the 13th block of data, we POKE 2040,13 where the 13th block starts at memory location (13*64), or location 832. Since these 8 bytes are capable of holding any number up to 255, we can store sprites anywhere up to a maximum starting location of (255*64), or 16320, the end of memory as far as the VIC chip is concerned.

We can go further, and bank in another block of memory and put sprites in there, as long as we don't place them anywhere near the ROM image of the character set.

For now, we'll stick fairly low down in memory, and design our first sprite.

Drawing a Sprite

Draw up a grid 24 characters by 21, like this :=

```
ABCDEFGHABCDEFGHABCDEFGH
........********........
.......***....***.......
......***......***......
.....***.*....*.***.....
.....***...**...***.....
.....***........***.....
.....***..*..*..***.....
.....***...**...***.....
.....***........***.....
....***..........***....
...***.*........*.***...
..***...********....***..
..***......**......***..
..***.....*..*.....***..
.***.....*....*.....***.
.***....*......*....***.
.***..**........**..***.
.***..**........**..***.
.********......********.
.********......********.
.********......********.
```

A smiling rocket! The letters A through H represent the same numbers as before, i.e. A = 128, B = 64, and so on, down to H equal to 1.

To get the bytes of data for the sprite we must, as before, add up the values for each row of the sprite, so that the first row (for instance) becomes equal to 0,255,0 (the three groups of letters).

We then carry on for each row, adding up all the numbers, until we get to the last row, in this case equal to 127,128,127.

Having now got all our 63 bytes of data, they must be POKEd into the relevant bit of memory, and the following program will do just that, for a much uglier sprite : =

```
10 V=53248 : REM START OF VIDEO CHIP
20 FORI=0TO62:READA:POKE832+I,A:NEXT
200 DATA 0,0,0,0,0,0,1,129,128,2,66,64,4,66,32,8
210 DATA 126,16,16,102,8,32,102,4,32,102,4,48,255
,12,48,85
220 DATA 12,0,170,0,0,255,0,0,255,0,1,126,128,3,
60,192
230 DATA 7,0,224,15,129,240,25,195,152,0,0,0,0,0,0
```

Okay, we've now got all our data stored, how do we start using it?

Sprite Data, and How it's Stored

The last program POKEd 63 bytes of data into memory locations 832 to 894, and this data will be used to define our sprite, but first we need to tell the computer that that's where the data is.

Memory locations 2040 to 2047, for sprites 0 to 8, tell the computer where to go, and if we treat this sprite as sprite number 0, we need to alter memory location 2040. Sprite number 1 would look for location 2041, and so on.

A look at the memory maps tells us that locations 832 onwards contain the 13th block of sprite data, so we need to add the following line of code : =

```
13 POKE 2040,13
```

Now all we need to do is to turn the sprite on. This is operated on a binary basis, using memory location 53248 + 21.

Remembering how binary numbers operate from chapter 3, POKEing a 1 into this location will turn on sprite 0, a 2 will turn on sprite 1, a 4 for sprite 2 (or a 3 for sprites 0 and 1), and so on until we reach 255, which turns every sprite under the sun on.

So the next line to add becomes : =

```
12 POKE V+21,1
```

All we need to do now is to move the sprite about the screen, so that you can see your creation in action, and to do this we'll just update the X and Y co-ordinates (V + 0 for X co-ord, V + 1 for Y co-ord for sprite 0), like this : =

```
30 FORI=0TO200
35 POKE V+0,I
40 POKE V+1,I
50 NEXT I
55 GOTO 30
```

This will now send your monster scurrying about the screen in a fit of pique, moving one pixel position in either direction at a time.

```
5 POKE 53281,1:POKE 53280,1
10 PRINT"[CLR]":T=1:GOSUB 62000
15 POKE V+21,1
20 FORI=50TO250:POKEV+1+2*0,I:POKE V+2*0,I
22 FOR J=1TO500:NEXTJ:NEXTI
25 END
62000 V=53248
62005 B(0)=248:B(1)=249:B(2)=250:B(3)=251:B(4)=252
:B(5)=253:B(6)=254:B(7)=255
62010 NS=T:IFNS=0THENRETURN
62015 FORA=1TONS
62020 READSK,M1,M2:IFSK=0THENPOKEV+28,PEEK(V+28)AN
D255-2^(A-1):GOTO62030
62025 POKEV+28,PEEK(V+28)OR2^(A-1):POKEV+37,M1:POK
EV+38,M2
62030 READCO:POKEV+38+A,CO:POKE2039+A,B(A-1)
62035 FORC=B(A-1)*64TOB(A-1)*64+63:READQ:POKEC,Q:N
EXT:NEXT:RETURN
63000 DATA 0 , 0 , 0 , 2
63005 DATA0,0,0,0,0,0,1,129,128,2,66,64,4,66,32,8
63010 DATA126,16,16,102,8,32,102,4,32,102,4,48,255
,12,48,85
63015 DATA12,0,170,0,0,255,0,0,255,0,1,126,128,3,6
0,192
63020 DATA7,0,224,15,129,240,25,195,152,0,0,0,0,0,
0,0
```

Multiple Sprites

The following listing expands on our Basic theme by putting two sprites on the screen at the same time.

In this listing we've made a number of important changes, and we'll detail those after the listing :=

```
5 POKE 53281,1:POKE 53280,1
10 PRINT"[CLR]":T=2:GOSUB 62000
15 POKE V+21,3
20 FORI=50TO250:POKEV+1+2*0,I:POKE V+2*0,I
21 POKEV+1+2*1,I-20:POKE V+2*1,I-20
22 FOR J=1TO500:NEXTJ:NEXTI
25 END
62000 V=53248
62005 B(0)=248:B(1)=249:B(2)=250:B(3)=251:B(4)=252
:B(5)=253:B(6)=254:B(7)=255
62010 NS=T:IFNS=0THENRETURN
62015 FORA=1TONS
```

```
62020 READSK.M1,M2:IFSK=0THENPOKEV+28,PEEK(V+28)AN
D255-2^(A-1):GOTO62030
62025 POKEV+28,PEEK(V+28)OR2^(A-1):POKEV+37,M1:POK
EV+38,M2
62030 READCO:POKEV+38+A,CO:POKE2039+A,B(A-1)
62035 FORC=B(A-1)*64TOB(A-1)*64+63:READQ:POKEC,Q:N
EXT:RESTORE:NEXT:RETURN
63000 DATA 0 , 0 , 0 , 2
63005 DATA0,0,0,0,0,0,1,129,128,2,66,64,4,66,32,8
63010 DATA126,16,16,102,8,32,102,4,32,102,4,48,255
,12,48,85
63015 DATA12,0,170,0,0,255,0,0,255,0,1,126,128,3,6
0,192
63020 DATA7,0,224,15,129,240,25,195,152,0,0,0,0,0,
0,0
```

Lines 63005 through 63020 contain the 64 bytes of data for our sprite. However, this time we're having two sprites on the screen, albeit both looking the same. So, we need to turn on two sprites, and this is achieved in line 15.

Line 5 just sets the background and border colours to be the same (i.e. white).

Line 20 moves sprite 0 in both X and Y directions, and line 21 moves sprite 1 in both directions.

Line 22 is a simple delay loop, just to let you see what's happening.

Line 62000 sets up the start of the VIC chip.

Line 62005 defines the starting blocks for up to eight sprites, being stored at locations 15872 onwards.

Line 62010 finds out how many sprites (NS) we've got: in this case 2.

Line 62020 reads the first 3 bytes of data in line 63000. Since the first byte is a 0, we're not in multi-colour mode, but if it wasn't, we'd jump to line 62025 and POKE in the values for selecting a multi-colour sprite, the multi-colour 0 and multi-colour 1 for the right sprite number.

Line 63030 reads the next byte and selects the actual sprite colour, and points the computer to where the data will be stored.

Finally, in line 62035, we read all the data into the correct memory locations.

This program is worth going through in some detail, as it features a number of useful points for reading in data for multiple sprites.

Multi-Colour and Expanded Sprites

As always, it's swings and roundabouts time, as our sprite now becomes a mere 12 by 21 pixels, but capable of displaying up to four different colours per sprite.

The colours are defined in the usual bit pair sequence, with each pair taking on the following values : =

00 : becomes transparent, and displays the screen colour.
01 : sprite multi-colour register 0 (53285)
10 : sprite ordinary colour register (53287-)
11 : sprite multi-colour register 1 (53286)

The previous listing will have shown you which locations to POKE the necessary values into, and the next two listings are illustrations of multiple multi-colour sprite manipulation, as well as showing you an expanded sprite in the Monster! listing.

Sprites are expanded in the X direction with the following command : =

```
POKE 53277,PEEK(53277)OR(2 to the power SN)
```

where SN is the sprite number from 0 to 7.

and in the Y direction with : =

```
POKE 53271,PEEK(53271)OR(2 to the power SN)
```

To get life back to normal again, in the X direction : =

```
POKE 53277,PEEK(53277)AND(255-(2 to the power SN))
```

and in the Y direction : =

```
POKE 53271,PEEK(53271)AND(255-(2 to the power SN))
```

And onto the listings : =

```
1 POKE 53280,1:POKE 53281,1
10 PRINT"[CLR]":T=3:GOSUB62000
15 POKE V+21,7
20 REMPOKE V+21,2
25 REMPOKE V+21,4
28 FORJ=40TO200
50 FORI=0TO2:POKE V+23,PEEK(V+23) OR 2^I:POKE V+29
,PEEK(V+29) OR 2^I:NEXT
52 POKE V+1+2*0,150:POKE V+2*0,J-40
55 POKE V+1+2*1,150:POKE V+2*1,J+40
60 POKE V+1+2*2,150:POKE V+2*2,J
65 NEXT J
61999 END
62000 V=53248
62005 B(0)=248:B(1)=249:B(2)=250:B(3)=251:B(4)=252
:B(5)=253:B(6)=254:B(7)=255
62010 NS=T:IFNS=0THENRETURN
62015 FORA=1TONS
62020 READSK,M1,M2:IFSK=0THENPOKEV+28,PEEK(V+28)AN
D255-2^(A-1):GOTO62030
62025 POKEV+28,PEEK(V+28)OR2^(A-1):POKEV+37,M1:POK
EV+38,M2
62030 READCO:POKEV+38+A,CO:POKE2039+A,B(A-1)
62035 FORC=B(A-1)*64TOB(A-1)*64+63:READ0:POKEC,0:N
EXT:NEXT:RETURN
63000 DATA 0 , 8 , 9 , 9
63005 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,15,0
63010 DATA0,31,0,0,127,0,1,252,0,7,248,0,31,240,0,
63
63015 DATA224,0,127,192,0,255,128,0,170,128,0,170,
128,0,170,128
63020 DATA0,170,0,0,168,0,0,0,0,0,0,0,0,0,0,0
63025 DATA 0 , 8 , 9 , 9
63030 DATA0,0,0,0,0,0,0,0,0,0,0,0,128,0,0,224
63035 DATA0,0,248,0,0,126,0,0,63,128,0,31,224,0,15
,248
63040 DATA0,7,252,0,3,254,0,3,254,0,2,170,0,2,170,
0
63045 DATA2,170,0,0,170,0,0,42,0,0,0,0,0,0,0,0
63050 DATA 0 , 8 , 9 , 9
63055 DATA0,255,0,1,0,128,1,36,128,1,0,128,254,24,
127,252
63060 DATA0,63,252,0,63,28,0,56,14,0,112,7,0,224,7
,255
63065 DATA192,3,255,128,1,255,0,1,255,0,1,199,0,1,
199,0
63070 DATA1,199,0,1,199,0,3,131,128,7,131,192,7,13
1,192,0
```

```
5 POKE 53280,1:POKE 53281,1
10 PRINT"[CLR]":T=5:GOSUB 62000
15 PRINT"[HOME]           [RED]COSMIC       INVASION
"
20 POKE V+21,31
24 POKE V+1+2*4,20:POKE V+2*4,165
25 POKE V+1+2*2,55:POKE V+2*2,115
26 POKE V+1+2*3,55:POKE V+2*3,215
27 XL=255:YL=255
28 FORI=1TO100
29 XL=XL-2:YL=YL-2
30 POKE V+1+2*0,YL:POKE V+2*0,XL
33 POKE V+1+2*1,YL-15:POKE V+2*1,XL-10
34 REM POKE 53281,INT(RND(0)*16)
36 POKE V+1+2*2,55+(I*2):POKE V+2*2,115
37 POKE V+1+2*3,55+(I*2):POKE V+2*3,215
38 POKE V+1+2*4,20+(I*2):POKE V+2*4,165
40 NEXT
61999 END
62000 V=53248
62005 B(0)=248:B(1)=249:B(2)=250:B(3)=251:B(4)=252
:B(5)=253:B(6)=254:B(7)=255
62010 NS=T:IFNS=OTHENRETURN
62015 FORA=1TONS
62020 READSK,M1,M2:IFSK=OTHENPOKEV+28,PEEK(V+28)AN
D255-2^(A-1):GOTO62030
62025 POKEV+28,PEEK(V+28)OR2^(A-1):POKEV+37,M1:POK
EV+38,M2
62030 READCO:POKEV+38+A,CO:POKE2039+A,B(A-1)
62035 FORC=B(A-1)*64TOB(A-1)*64+63:READQ:POKEC,Q:N
EXT:NEXT:RETURN
63000 DATA 1 , 2 , 3 , 9
63005 DATA0,36,0,0,36,0,0,255,0,0,255,0,0,231,0,24
63010 DATA102,24,60,126,60,34,60,68,1,60,128,0,255
,0,0,255
63015 DATA0,0,255,0,1,128,128,3,255,192,7,255,224,
15,255,240
63020 DATA14,0,112,14,0,112,14,0,112,30,0,120,30,0
,120,0
63030 DATA 1 , 2 , 3 , 7
63035 DATA0,36,0,0,36,0,0,255,0,0,255,0,0,231,0,24
63040 DATA102,24,60,126,60,34,60,68,1,60,128,0,255
,0,0,255
63045 DATA0,0,255,0,1,128,128,3,255,192,7,255,224,
15,255,240
63050 DATA14,0,112,14,0,112,14,0,112,30,0,120,30,0
,120,0
63060 DATA 0 , 0 , 0 , 0
63065 DATA0,36,0,0,36,0,0,255,0,0,255,0,0,231,0,24
63070 DATA102,24,60,126,60,34,60,68,1,60,128,0,255
,0,0,255
```

```
63075 DATA0,0,255,0,1,128,128,3,255,192,7,255,224,
15,255,240
63080 DATA14,0,112,14,0,112,14,0,112,30,0,120,30,0
,120,0
63090 DATA 0 , 0 , 0 , 0
63095 DATA0,36,0,0,36,0,0,255,0,0,255,0,0,231,0,24
63100 DATA102,24,60,126,60,34,60,68,1,60,128,0,255
,0,0,255
63105 DATA0,0,255,0,1,128,128,3,255,192,7,255,224,
15,255,240
63110 DATA14,0,112,14,0,112,14,0,112,30,0,120,30,0
,120,0
63120 DATA 0 , 0 , 0 , 0
63125 DATA0,36,0,0,36,0,0,255,0,0,255,0,0,231,0,24
63130 DATA102,24,60,126,60,34,60,68,1,60,128,0,255
,0,0,255
63135 DATA0,0,255,0,1,128,128,3,255,192,7,255,224,
15,255,240
63140 DATA14,0,112,14,0,112,14,0,112,30,0,120,30,0
,120,0
```

Sprite Positioning and Collision Detection

So far we've never moved sprites beyond an X co-ordinate of 255, simply because memory locations can't hold values greater than this.

However, memory location 53264 allows us to move all the way to the edge, in the following manner.

When the X co-ordinate becomes equal to 255, POKE 53264 (or V + 16), with a 1, and then reset the X values to zero again. Now we're only moving from 256 to 320, or a total of 64 positions, so X ranges from 0 to 63. When we've finished, reset V + 16 back to a zero again, to let us move from the left-hand edge of the screen again.

Sprite Priority and Collision

We'll leave you to experiment with the program listings given earlier to see precisely how this works, but in brief the priority of each sprite can be controlled from register 53275 (53248 + 27).

This register works in exactly the same way as all the others, with sprite 0 being controlled from bit 0, sprite 1 from bit 1, and so on. If the bit is set to zero, then the sprite will be displayed instead of anything else: the sprite is in the foreground, in other words.

To get the relevant sprite into the background, the bit must be set to 1.

Collision

This is controlled from memory location 53278, or 53248 + 30.

Again, this works in the same way as all the other locations, and is used to detect collisions between sprites.

If the register is showing zero, then nothing has happened; a 3 indicates a collision between sprites 0 and 1; a 6 for sprites 1 and 2, and so on.

This is based on the usual manner of selecting sprites from the appropriate bits of a particular byte.

i.e. Value	128	64	43	16	8	4	2	1
Bit	7	6	5	4	3	2	1	0
Sprite No.	7	6	5	4	3	2	1	0

Thus sprites 2 and 3 are controlled from bits 2 and 3, which respectively give the values 0f 4 and 8, and therefore a value of 12 (4 + 8) must be POKEd into that byte, or indeed read from it, and the relevant action will ensue.

Multiple sprite collision is also possible from this.

For instance, if register 53278 returns a value of 82, it means that bits 6, 4 and 1 have been affected, or in other words sprites 6,4 and 1 are involved in a pile-up.

A most useful location!

Turning Sprites Off

Well, it would help! The quick and easy way to turn them all off is to type POKE V + 21,0, but for selective sprites you must use : =

```
POKE V+21,PEEK(V+21)AND(255 - 2 to the power of
SN)
```

where SN is the sprite number from 0 to 7.

Sprite Movement

We've already shown you sprites moving across the screen, and you should now be in a position to amend the listing given at the start of this section to incorporate your own sprites.

Just define the sprites first, and then POKE the appropriate values into the X and Y co-ordinate locations as the nature of the game dictates.

Of course, this game could also be adapted for control by a joystick, since we've learnt already which locations to test for movement of the joystick in a particular direction.

For locations which return actual values, to make life easier, the following lines will do the trick :

For joystick 1 : =

```
S1=PEEK(56321)
```

-((S1AND16) = 0) gives a 1 if the fire button is pressed, and a 0 if it's not.

((S1AND15) = 4)-((S1AND15) = 8) gives a 1 for moving left, a -1 for moving right, and a 0 if nothing's doing.

((S1AND15) = 1)-((S1AND15) = 2) gives a 1 for moving down, a -1 for moving up, and a 0 if nothing's doing.

To read joystick 2, let S2 = 56320, and substitute S2 for S1 in all of the above expressions.

Having done that, it would be a relatively simple matter to have a sprite controlled joystick game, written entirely in Basic!

Have fun!

Sprite Generator

We'll take the slog out of generating sprites now, and give you a program that does it all for you.

How it Works

As you can see it's quite a long listing, so you'll need to take care when typing it in.

You'll get a fair idea of how it works just by running it, as full instructions are included in the program, but for a brief resume to encourage you to tackle the typing job :=

The program allows you to define sprite 0, using block 13 (at 832) for your storage area. Of course, having defined the sprite to go there you can always play around with it afterwards and store it somewhere else.

Having defined your sprite on the large grid using a series of spaces and reversed spaces to represent bits that are meant to be on or off, watching the miniature version of the sprite until you're satisfied that you've got it right, you can then manoeuvre it around the screen using the keyboard, expand it and contract it again, and finally save the sprite data off as a series of data statements, generated as line numbers 60000 onwards.

It is this that we're really after, and these can then readily be merged into your own sprite manipulation programs by leaving the lines on the screen, loading the new program, and then just cursoring back and hitting return over 60000 onwards to 'merge' them onto the end of the loaded program.

Even if you don't type it all in, there are some good ideas in there, and it's well worth studying the listing in detail.

```
5 REM SPRITE GENERATOR
10 REM ORIGINAL IDEA BY RICHARD FRANKLIN
12 MO$="HOME,22CD,10CR]
15 POKE 829,223
25 REM
35 REM IF ANY SPRITE DATA,SET UP SPRITE
45 REM IT LOOKS AN AWFUL LOT BETTER UNEXPANDED
55 POKE 828,0
65 READ SS
75 IF SS>OTHEN 975
85 REM
95 REM NO MORE SPRITE DATA
105 REM
115 GOSUB 1055:POKE53281,8:POKE53280,7:PRINT"[BLK]
125 DEFFNA(AA)=1064+R*40+C
135 V=53248:NO=PEEK(829)
145 XM=0:YM=1:XG=16:SE=21:XY=23:XC=29
155 SR=39:PRINT""
165 POKE 2040,NO:POKE V+SE,1:POKE V+XY,1
175 POKE V+XC,1:POKE V+XM,255:POKE V+YM,190
185 POKE V+XG,0:POKE V+SR,0
195 X=255:Y=190
205 REM
215 REM SET UP DISPLAY
225 REM
235 PRINT"[HOME,BLK]^                                    ^"
245 SAR=64*NO:PRINT"[HOME]"
255 FORI=SARTOSAR+62STEP3
265 FORJ=0TO2
275 AA=PEEK(I+J)
285 FORK=7TOOSTEP-1
295 A=INT((AAANDA%(K))/A%(K))
305 IFA=1THENPRINT"[BLK]*";:GOTO325
315 PRINT"[BLK].";
325 NEXTK
335 NEXTJ
345 PRINT
355 NEXTI
365 GOSUB1975
375 REM
385 REM SPRITE SET UP ON THE SCREEN
395 REM INPUT CHANGES
405 REM
415 R=0:C=0
425 Z=FNA(0)
435 POKEZ+54272,1
445 GETX$:IFX$=""THEN445
455 POKEZ+54272,0
465 IFX$="Q"THENPRINT"[CLR]BYE!":POKE V+21,0:END
475 GOSUB2200
515 IFX$="[HOME]"THENR=0:C=0:GOTO425
525 IFX$="[CLR]"THENGOSUB1285:GOTO425
535 IFX$="+"THEN655
```

```
545 IFX$="-"THEN865
555 IFX$="M"THEN1375
565 IFX$="B"THEN1685
575 IFX$="C" THEN 1595
585 IFX$="X"THEN1125
595 IFX$="N"ANDNO-223<31THENNO=NO+1:GOTO165
605 IFX$="E"THEN765
615 GOTO 425
625 REM
635 REM ADD POINT
645 REM
655 Z=FNA(0)
665 Z1=PEEK(Z)
675 IFZ1=81THEN425
685 POKEZ,42
695 BYTE=INT(C/8)+R*3
705 BIT=7-(C-INT(C/8)*8)
715 POKEBYTE+NO*64,PEEK(BYTE+NO*64)ORA%(BIT)
725 GOTO 425
735 REM
745 REM INPUT SPRITE £ TO EDIT
755 REM
765 PRINTMO$"WHICH SPRITE NO."
767 GETS$:IFS$=""THEN767
770 S=VAL(S$)
775 IFS<0ORS>31THEN765
785 IF NO=223+STHENAA=1:GOTO805
795 NO=223+S
805 PRINTMO$"                                        [HOME]";
815 IFAA=1THENAA=0:GOTO425
825 GOTO 165
835 REM
845 REM DELETE POINT
855 REM
865 Z=FNA(0)
875 Z1=PEEK(Z)
885 IFZ1=46THEN 425
895 POKE Z,46
905 BYTE=INT(C/8)+R*3
915 BIT=7-(C-INT(C/8)*8)
925 POKE BYTE+NO*64,PEEK(BYTE+NO*64)AND(255-A%(BIT
))
935 GOTO 425
945 REM
955 REM IF ANY DATA, SET SPRITES UP
965 REM
975 SAR=SS*64
985 FOR I=SAR TO SAR+62
995 READ A:POKE I,A
1005 NEXT I
1015 GOTO 65
1025 REM
1035 REM SET ARRAY WITH POWERS OF TWO
```

```
1045 REM
1055 FOR I=0 TO 7
1065 A%(I)=2^I
1075 NEXT I
1085 RETURN
1095 REM
1105 REM INPUT FOR EXPAND
1115 REM
1125 PRINTMO$"ENTER X OR Y"
1135 GETX$:IFX$<>"X"ANDX$<>"Y"THEN1125
1145 IFX$="X"THEN1185
1155 IFPEEK(V+XY)=1THENPOKEV+XY,0:GOTO1205
1165 POKEV+XY,1
1175 GOTO1205
1185 IFPEEK(V+XC)=1THENPOKEV+XC,0:GOTO1205
1195 POKEV+XC,1
1205 PRINTMO$"                "
1215 GOTO 425
1225 REM
1235 REM DISPAY CONTROL OPTIONS
1245 REM
1255 REM
1265 REM CLEAR PRESENT SPRITE
1275 REM
1285 FORI=0TO62:POKENO*64+I,0:NEXTI
1295 FORI=0TO20
1305 FORJ=0TO23
1315 POKE1064+I*40+J,46
1325 NEXTJ,I:R=0:C=0
1335 RETURN
1345 REM
1355 REM MOVE SPRITE AROUND SCREEN
1365 REM
1375 PRINTMO$"[RVS,5CL]USE CURSOR KEYS/RETURN TO E
XIT"
1395 GETX$:IFX$=""THEN1395
1405 IFX$="[CR]"ANDX<319THENX=X+2
1415 IFX$="[CL]"ANDX>1THENX=X-2
1425 IFX$="[CD]"ANDY<254THENY=Y+2
1435 IFX$="[CU]"ANDY>1THENY=Y-2
1445 POKE V+YM,Y
1455 POKE V+XG,INT(X/255)
1465 POKE V+XM,X-INT(X/255)*255
1475 IF X$=CHR$(13)THEN1495
1485 GOTO1375
1495 POKE V+XM,255
1505 POKE V+YM,190
1515 POKE V+XG,0
1525 X=255:Y=190
1535 PRINTMO$"[6CL]
 [HOME]";
1555 GOTO 425
1565 REM
```

```
1575 REM CHANGE SPRITE COLOUR
1585 REM
1595 PRINTMO$"WHICH COLOUR (0-15)?";
1600 INPUTCL$
1602 CL=VAL(CL$)
1605 IF CL<0ORCL>15THEN1595
1615 POKE V+SR,CL
1625 PRINTMO$"                                    [HOME]";
1635 GOTO 425
1645 REM
1655 REM CREATE DATA STATEMENTS FOR
1665 REM PRESENT SPRITE
1675 REM
1685 PRINT"[CLR,3CD]";PEEK(828)+60000;"DATA"RIGHT$
(STR$(NO),LEN(STR$(NO))-1)
1695 POKE828,PEEK(828)+1:FORI=0TO8
1705 PRINTPEEK(828)+60000"DATA";
1715 FORJ=0TO6
1725 BB=PEEK(NO*64+I*7+J)
1735 BB$=RIGHT$(STR$(BB),LEN(STR$(BB))-1)
1745 PRINTBB$;",";
1755 NEXT J
1765 PRINT"[CL] ":POKE828,PEEK(828)+1
1775 NEXT I
1785 PRINTPEEK(828)+60000;"DATA-1"
1795 PRINT"RUN[HOME]"
1805 POKE 198,12
1815 FORI=0TO11:POKE631+I,13:NEXT I
1825 POKE829,NO:END
1835 REM
1845 REM SPRITE DATA STORED FROM HERE
1855 REM
1865 DATA223
1875 DATA0,0,0,3,128,0
1885 DATA4,64,0,9,112,0,8
1895 DATA56,0,8,112,0,8,128
1905 DATA78,9,0,177,249,0,100
1915 DATA80,128,170,44,64,101,152
1925 DATA32,19,0,32,8,0,64
1935 DATA7,255,128,0,108,0,0
1945 DATA108,0,0,111,0,0,110
1955 DATA0,0,120,0,0,112,0
1965 DATA-1
1975 PRINT"[HOME]"SPC(26)"[RVS]COMMANDS"
1985 PRINTSPC(25)"SPRITE £ "NO-223
1995 PRINT:PRINTSPC(25)"[WHT,RVS]E[OFF]DIT SPRITE
£"
2005 PRINTSPC(25)"[RVS]N[OFF]EXT SPRITE £"
2015 PRINTSPC(25)"[RVS]M[OFF]OVE SPRITE"
2025 PRINTSPC(25)"[RVS]C[OFF]HANGE COLOUR"
2035 PRINTSPC(25)"[RVS]X[OFF]PAND"
2045 PRINTSPC(25)"[RVS]+[OFF] TURN ON DOT"
2055 PRINTSPC(25)"[RVS]-[OFF] TURN OFF DOT"
```

```
2065 PRINTSPC(25)"[RVS]B[OFF]ASIC DATA"
2075 PRINTSPC(25)"[RVS]Q[OFF]UIT"
2085 PRINT:PRINTSPC(25)"USE CURSOR"
2095 PRINTSPC(25)"CONTROL KEYS"
2105 PRINTSPC(25)"TO MOVE CURSOR"
2125 RETURN
2200 IFX$="[CR]"THENC=C+1:IFC=24THENC=0:GOTO425
2205 IFX$="[CL]"THENC=C-1:IFC=-1THENC=23:GOTO425
2210 IFX$="[CD]"THENR=R+1:IFR=21THENR=0:GOTO425
2215 IFX$="[CU]"THENR=R-1:IFR=-1THENR=20:GOTO425
2220 RETURN
60000 DATA223
60001 DATA0,0,0,0,3,128,0
60002 DATA4,64,0,9,112,0,8
60003 DATA56,0,8,112,0,8,128
60004 DATA78,9,0,177,249,0,100
60005 DATA80,128,170,44,64,101,152
60006 DATA32,19,0,32,8,0,64
60007 DATA7,255,128,0,108,0,0
60008 DATA108,0,0,111,0,0,110
60009 DATA0,0,120,0,0,112,0
60010 DATA-1
```

And Another Game

Well, I think you deserve it, you've had an awful lot of theory lately, and we're about to get a lot more in the next chapter, so time to relax once more, and play a game called Arrow.

This is an adaptation of a game by Jim Butterfield, and as usual it contains no sound or sprites.

Why not ? Well, there are a number of programmers around who fool most of the people most of the time by remembering a few PEEKs and POKEs parrot fashion, and who don't really know too much about the actual workings of the machine.

Until you've done something for yourself, you won't really understand it.

So, if you want to convert this program, it's up to you! It'll probably teach you more about sprite manipulation than re-reading this chapter half a dozen times and committing it all to memory.

You have to control a snake, whose job it is to roam around the screen bumping into boxes (and thus gaining points), whilst at the same time trying to avoid running into the walls and yourself: you get longer as the game progresses.

Good fun, and infuriating.

```
100 PRINT"[CLR,CD,RVS] ARROW  "
110 INPUT"[CD]INSTRUCTIONS ?";Z$
112 IFZ$=""THEN110
114 IFASC(Z$)=78THEN190
115 PRINT"[CD]OKAY, THEN "
120 PRINT"[CD]GUIDE THE MOVING 'SNAKE' WITH KEYS:"
130 PRINT" 2(DOWN), 4(LEFT), 6(RIGHT), 8(UP)"
140 PRINT"[CD]DON'T HIT THE BOUNDARY (OR YOURSELF)
;"
150 PRINT"..TRY TO HIT THE BOXES FOR POINTS."
160 PRINT"[CD]YOU HAVE 60 SECONDS OF PLAY. GOOD LU
CK!"
170 PRINT"[CD]      [RVS]HIT ANY KEY TO START"
180 GETZ$:IFZ$=""GOTO180
190 DIMP(255),D(3),V(8),H(8),T(8),R(8):K=.1
200 D(0)=22:D(1)=60:D(2)=62:D(3)=30
210 T9=1024:T6=3599
220 REMSET SCREEN UP
230 PRINT"[CLR,WHT]   SCORE: 0":PRINT"A"
240 FORJ=0TO81:IFPEEK(T9+J)<>1THENNEXTJ
250 L=J:FORJ=T9+LTOT9+2*L-1:POKEJ,81:POKEJ+23*L,81
:NEXTJ
260 FORJ=T9+2*LTOT9+24*LSTEPL:POKEJ,81:POKEJ+L-1,8
1:NEXT
265 POKE 53281,5
270 V=5:H=5:V1=0:H1=1:P2=10:D1=2
280 TI$="000000"
290 PRINT"[HOME]";RIGHT$(TI$,2):IFTI>T6GOTO620
300 GETZ$:IFZ$=""GOTO330
310 Z=(ASC(Z$)-50)/2:IFZ<>INT(Z)ORZ<0ORZ>3GOTO330
320 D1=Z:D=Z-1.5:V1=INT(ABS(D))*SGN(D):H1=SGN(D)-V
1
330 V=V-V1:H=H+H1:P=T9+V*L+H
350 P9=PEEK(P):
360 R6=R7:R7=R7+1:IFR7>P2THENR7=0
370 P1=P(R7):P(R7)=P:IFP1<>0THENPOKEP1,32
380 POKEP,D(D1):P1=P(R6):IFP1<>0THENPOKEP1,81
390 IFP9<>32GOTO540
400 IFRND(1)>KGOTO290
410 V%=RND(1)*L/10:P9=86+V%:V9=V(V%):IFV9>0GOTO591
470 V2=INT(RND(1)*20)+3:H2=INT(RND(1)*(L-4))+2
480 FORV3=V2-1TOV2+1:P3=V3*L+T9:FORH3=H2-1TOH2+1:I
FPEEK(P3+H3)<>32GOTO470
490 NEXTH3,V3:V(V%)=V2:H(V%)=H2
500 FORV3=V2-1TOV2+1:P3=V3*L+T9:FORH3=H2-1TOH2+1
510 REM
520 POKEP3+H3,P9:
```

```
530 NEXTH3,V3:T=9*RND(1):P8=V2*L+H2+T9:POKEP8,49+T
:T(V%)=T:R(V%)=P8:GOTO290
540 V%=P9-86:IFV%<0GOTO600
550 P8=R(V%):T=T(V%):P2=P2+T:T$=TI$
560 T=T-1:S=S+1:POKEP8,T+49
570 PRINT"[HOME,9CR]";S
580 FORJ=100TO30STEP-1:NEXT:IFT>=0GOTO560
590 P2=P2+1:TI$=T$:V9=V(V%)
591 FORV3=V9-1TOV9+1:P3=V3*L+T9:H9=H(V%)+P3:FORH3=
H9-1TOH9+1
594 POKEH3,32:NEXTH3,V3:V(V%)=0:POKER(V%),32:GOTO2
90
600 FORJ=1TO1000:NEXT
620 PRINT"[HOME,CD,RVS]ANOTHER GAME?[OFF]   [3CL]"
;
625 POKE 53281,6
630 GETZ$:IFZ$=""GOTO630
640 IFZ$="Y"THENCLR:GOTO190
650 IFZ$<>"N"GOTO630
660 PRINT"[CLR]BYE![CD]";
```

6
Sound on the Commodore 64

The Commodore 64 has a remarkably gifted sound capability, courtesy of the 6581 SID chip. We'll be exploring SID in much more detail in chapter 9, but for now we're going to concentrate purely on practical matters, and show you how to get the best out of the sound capabilities of the 64.

You don't need to be a musician to use this machine, although if you are, the keyboard and other musical programs will probably help you begin to understand how it all works, and certainly the table of musical notes later on in this section will be of great use when transposing music.

What you will need to know are the ways in which the Commodore 64 uses its powerful SID chip, the memory registers that are affected, and the power at your disposal.

SID can control three voices, each one having a practical octave range of 8 octaves. Unfortunately we don't have separate volume controls over each voice, but have to change them all at the same time.

For each voice we have control over four waveforms, namely triangle, sawtooth, variable pulse, and noise.

Our three envelope generators, combined with ring modulation, programmable filters and the rest give SID the same sort of capabilities as many a more expensive dedicated synthesiser.

It is a relatively easy matter to produce successful impersonations of many musical instruments, and later on we'll be doing just that, as well as giving you a number of sample tunes and programs to type in.

But for now, let's find out which sections of memory control all this.

Finding Your Way Around

A look at the memory maps at the back of the book will tell you that the SID chip occupies memory locations 54272 to 54300. Obviously, it takes up a bit more room than that, but those are the locations that we are concerned with.

As with sprites and graphics, we'll adopt the technique of setting a variable equal to the value of the base location (54272), and work our way up from there.

The following table shows what each of the 28 usable bytes does.

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/Relese 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
27 Digitised output from voice 3
28 Digitised output from envelope generator 3

More than One Voice

The SID chip comes equipped with three voices, and these can all be independently controlled to produce a variety of effects.

The waveform for each of them can be changed, using the appropriate register, and each voice can independently mimic a wide variety of musical instruments.

In order to do that, we have to adjust a variety of settings, and we'll start by looking at Attack and Decay, Sustain and Release, collectively known as ADSR.

These measure the length of time it takes a note to come to its maximum volume, and the time taken to go to total silence again, and then the length of time for which it will maintain its maximum volume before letting go again.

The following tables show the various settings for ADSR :=

ATTACK/DECAY RATE SETTINGS

ATTACK/DECAY SETTING	HIGH ATTACK	MEDIUM ATTACK	LOW ATTACK	LOWEST ATTACK	HIGH DECAY	MED. DECAY	LOW DECAY	LOWEST DECAY
VOICE 1	54277	128	64	32	16	8	4	2	1
VOICE 2	54284	128	64	32	16	8	4	2	1
VOICE 3	54291	128	64	32	16	8	4	2	1

SUSTAIN/RELEASE RATE SETTINGS

SUSTAIN/ CONTROL	RELEASE SETTING	HIGH SUSTAIN	MEDIUM SUSTAIN	LOW SUSTAIN	LOWEST SUSTAIN	HIGH RELEASE	MED. RELEASE	LOW RELEASE	LOWEST RELEASE
VOICE 1	54278	128	64	32	16	8	4	2	1
VOICE 2	54285	128	64	32	16	8	4	2	1
VOICE 3	54292	128	64	32	16	8	4	2	1

These values are combined in the following way. If, for voice one, we POKE 54277 with 16, we'd have the lowest attack rate, and no decay. POKEing it with 20 would give us the same attack rate, but this time a medium decay, as 20 is a combination of the settings for 16 and 4.

POKEing 54272 with 72 would give us a medium attack and a high decay, and so on.

Sustain/Release works in exactly the same way. POKEing 54278 with 40 would give us a low sustain and a high release, as 40 is a combination of 32 (low sustain) and 8 (high release).

Before even playing a note, we've got to know how to turn the voices on, and a look at the table on the previous page will show us that to set the volume (always wise!) we need to POKE 54296, and we can use any number from 0 (silence) through to 15 (maximum volume).

Then we must select the waveform we require.

For voice one, this is achieved by altering location 54276, and there are four values we can put in there : =

17 : gives us a triangle waveform.
33 : gives us a sawtooth waveform.
65 : gives us a pulse waveform.
129 : generates white noise.

We'll go into waveforms in more detail later, but all we need to know now is the actual note that we want to play, and this is got by POKEing locations 54273 and 54272 for voice one with the high frequency of the note to be played and the low frequency respectively.

A table of high and low frequencies is given later in this chapter, but for now we'll just play a single note, and take a detailed look at the program.

Playing a Note

If, by the way, you think that we're going to a lot of trouble to play a single note, you're absolutely right, but when it comes to composing music, most of these registers only have to be altered once, and only a couple of them will need constant changing.

So, here goes :=

```
10 S=54272
20 POKE S+24,15 : REM SET VOLUME TO HIGHEST LEVEL
30 POKE S+5,34 : REM LOW ATTACK, LOW DECAY
40 POKE S+6,130 : REM HIGH SUSTAIN, MEDIUM RELEASE
50 POKE S+1,45:POKE S,198
60 REM HI-FREQ AND LO-FREQ FOR NOTE 'F' FROM
FIFTH OCTAVE
70 POKE S+4,33 :REM A SAWTOOTH WAVEFORM
80 FORI=1TO500:NEXT: REM WAIT A BIT
90 FORI=0TO24:POKES+I,0:NEXT:REM TURN IT ALL OFF
```

A masterpiece, eh? But it's not too difficult to extend all of this to start producing simple tunes. The next piece of music might be recognised by some of you :=

```
5 S=54272
10 POKE S+14,0:POKE S+4,0:POKES+5,0:POKES+6,0
20 POKE S+5,64
30 POKE S+6,128
40 POKE S+24,15
50 READA,B
52 FORI=1TO500:NEXT
55 IFA=0THEN100
58 POKE S+4,33
60 POKE S+1,A:POKE S,B
65 GOTO50
100 POKE S+14,0:POKE S+4,0:POKES+5,0:POKES+6,0
120 DATA19,63,21,154,17,37,8,147,12,216,0,0
```

Close Encounters of the 64 kind! We'll use this tune just to give you some idea of how a few simple alterations can change the whole sound of things.

The following two programs are simple variations on a theme, all just using one voice, and they show how various different sounds can be achieved, just by changing the ADSR settings, the waveform, and (in one instance) the octave level.

```
5 S=54272
10 POKE S+14,0:POKE S+4,0:POKES+5,0:POKES+6,0
20 POKE S+5,72
30 POKE S+6,128
40 POKE S+24,15
50 READA,B
52 FORI=1TO500:NEXT
55 IFA=0THEN100
58 POKE S+4,65
59 POKE S+3,1:POKE S+2,1
60 POKE S+1,A:POKE S,B
65 GOTO50
100 POKE S+14,0:POKE S+4,0:POKES+5,0:POKES+6,0
120 DATA19,63,21,154,17,37,8,147,12,216,0,0
```

```
5 S=54272
10 POKE S+14,0:POKE S+4,0:POKES+5,0:POKES+6,0
20 POKE S+5,190
30 POKE S+6,0
40 POKE S+24,15
50 READA,B
52 FORI=1TO500:NEXT
55 IFA=0THEN100
58 POKE S+4,33
59 POKE S+3,1:POKE S+2,1
60 POKE S+1,A:POKE S,B
65 GOTO50
100 POKE S+14,0:POKE S+4,0:POKES+5,0:POKES+6,0
120 DATA4,208,5,103,4,73,2,6,3,54,0,0
```

Get Down to the Boogie

The following piece of code is a nice example of using sound on the 64, and comes courtesy of Jim Butterfield, as do most Commodore snippets of information!

```
100 PRINT"MUSIC, BY JIM BUTTERFIELD"
110 I1=54272:I2=54279:I3=54286
120 H1=I1+1:H2=I1+1:H3=I3+1
130 V1=I1+4:V2=I2+4:V3=I3+4
140 POKE 54296,15
150 POKEV1+1,9:POKEV1+2,0
160 POKEV2+1,36:POKEV2+2,36
```

```
170 POKEV3+1,18:POKEV3+2,250
180 T=TI
200 POKEV1,32:POKEV2,128:POKEV3,16
210 READS:IFS=0THEN290
220 READ X1,Y1,X2,Y2,X3,Y3
230 IFX1THENPOKEH1,X1:POKEI1,Y1:POKEV1,33
240 IFX2THENPOKEH2,X2:POKEI2,Y2:POKEV2,129
250 IFX3THENPOKEH3,X3:POKEI3,Y3:POKEV3,17
260 T=T+S
270 IFT>TITHEN270
280 GOTO200
290 FORJ=I1TO54296:POKEJ,0:NEXT
300 DATA 15,17,37,68,149,4,73
310 DATA 15,21,154,0,0,0,0
320 DATA 15,25,177,0,0,6,108
330 DATA 15,28,214,0,0,0,0
340 DATA 15,30,141,115,88,7,163
350 DATA 15,28,214,0,0,0,0
360 DATA 15,25,177,0,0,6,108
370 DATA 15,21,154,0,0,0,0
380 DATA 15,17,37,68,149,4,73
390 DATA 15,21,154,0,0,0,0
400 DATA 15,25,177,0,0,6,108
410 DATA 15,28,214,0,0,0,0
420 DATA 15,30,141,115,68,7,163
430 DATA 15,28,214,0,0,0,0
440 DATA 15,25,177,0,0,6,108
450 DATA 15,21,154,0,0,0,0
460 DATA 15,22,227,91,140,5,185
470 DATA 15,28,214,0,0,0,0
480 DATA 15,34,75,0,0,8,147
490 DATA 15,38,126,0,0,0,0
500 DATA 15,40,200,163,31,10,60
510 DATA 15,38,126,0,0,0,0
520 DATA 15,34,75,0,0,8,147
530 DATA 15,28,214,0,0,0,0
540 DATA 15,17,37,68,149,4,73
550 DATA 15,21,154,0,0,0,0
560 DATA 15,25,177,0,0,6,108
570 DATA 15,28,214,0,0,0,0
580 DATA 15,30,141,122,52,7,163
590 DATA 15,28,214,0,0,0,0
600 DATA 15,25,177,0,0,6,108
610 DATA 15,21,154,0,0,0,0
620 DATA 25,17,37,68,149,4,73
999 DATA 20,0,0,0,0,0,0
1000 DATA 0
```

And now, this is how it all works!

Multiple Voices

As you've heard, this program is using all three voices, and some of them are now beginning to sound like recognisable instruments.

By playing with the ADSR settings, and the waveforms, in the earlier, shorter listings, you'll probably have been able to work out one or two instruments for yourself, and the 64 can manage to produce impersonations of such instruments as banjos, pianos, harpsichords, accordians, and many more.

Here, we're setting up voice 1 to be something approaching the sound of a banjo being played, so in line 150, when we're defining the ADSR, we want a fast attack, a slight decay, and no sustain: the sound just dies away almost immediately.

Voice 2 is more interesting, and sounds almost like water dripping: it still manages to keep time with the music. By the way, I'm told my ear for music is about as good as Van Gogh's after his accident, so apologies for the noise that all this produces. They do get better later, thanks to Mr Butterfield!

For this, we've let the sound take a while to build up, a while to decay, and to hold on to the note for quite a while, so both AD and SR are set to 36.

Voice 3 plays the electric bass, and so in line 170 we want a fast attack with a medium decay, and we also want to hold the note for a while as well.

Playing about with all of these values will give you a good feeling for how the 64 handles its sound.

In line 200 we continue our definition of the instruments, and give sawtooth waveform to voice 1, turn voice 2 into generating white noise, and give voice 3 a triangle waveform.

Interesting results can be gained from voice 1 by altering the ADSR in line 150 to 9 and 0, and continuing up to 144 and 17 produces some quite good effects: electronic music of the bizarre kind!

By using an even number here we're basically telling the voices to play nothing, and by using an odd number later on, in lines 230 to 250, we telling them to play again.

Line 210 reads the first bit of data in each data statement, and halts everything if it finds a zero. If not, the value read into S is used as a timer in lines 260 to 270, and is there as a further delay on note playing in addition to ADSR.

Lines 230 to 250 then play the note, with the data coming in the format high frequency/low frequency of each note for each of the three voices in succession.

Line 290 sets everything back to zero again, and then comes the data.

Later on in this section we'll be giving you some more tunes to type in, and these use the same skeleton program as this one. i.e. retain lines 100 to 290, and just the different sets of data each time you come to them.

For help in typing them all in, remember one of the neater features of the Commodore 64's line editing system. To repeat a line, just move the cursor up to the line you want to repeat, and type in the new number followed by Return. Voila, two identical line numbers.

So far we haven't really mentioned the third waveform produced by POKEing S+4 with 65. This is because this pulse wave requires a couple more parameters than the other three, known as the pulsewidth.

You can see the memory locations that need to be altered by looking at the chart at the start of this section, and the values that go into the locations behind on the sort of pulse wave you want to create: more in a minute.

Calculating Musical Notes

In order to POKE the correct values into the various memory locations, we obviously need to know what those values should be.

The table on the next two pages gives you the values of the low and high frequency parts of the notes through a range of 8 octaves, but some of these may not sound quite 'right' to you, and so there are a few formulae available for more exact calculation of these frequencies.

Incidentally, you're not confined to using only the values given: they're just presented as a guideline.

To calculate these values, a little bit of Physics first.

The Physics of Sound

Sound is essentially a series of waves, commonly compared to the waves you get on the beach, or those created by throwing a pebble into a pond: the ripples spreading out can be thought of as similar to waves of sound.

The distance between the successive peaks of the waves can be measured in terms of time: i.e. how long an interval it has taken between one wave passing a spot and the next one. If we call this X seconds, then the frequency of the waves is equal to (1/X).

In other words, the number of waves passing a fixed point in one second. This is measured in cycles per second, or Hertz.

For instance, the pitch for the note A above middle C is 440 Hertz. Anything above about 3000 Hertz begins to get a bit painful after a while!

To get from a value for the frequency to two numbers which we can POKE into the computer, we need to split this up into the low order frequency, and high order frequency.

If we call the frequency FQ, the first stage is to :=

```
F=INT(FQ/0.05961)
```

Then, taking our new value for F we now find the high order frequency, HF, from the equation :=

```
HF=F/256
```

Now this will usually produce a number with a decimal point in it. If it doesn't, all well and good, and the value for the low order frequency (LF) will be zero. If it does, well, let's take a practical example. Let's say that HF works out to be 17.1234.

Then, to find LF we must :=

```
LF=256-256*0.1234
```

In other words, we multiply by the part after the decimal point, and let the integer value, in this case 17, be HF.

So, we can now find the high and low order frequencies for any musical, or not so musical, note.

The next section details the HFs and LFs for an 8 octave range on the 64, with middle C being indicated as C-4.

Note	Note–Octave	Hi Freq	Low Freq
0	C–0	1	18
1	C#–0	1	35
2	D–0	1	52
3	D#–0	1	70
4	E–0	1	90
5	F–0	1	110
6	F#–0	1	132
7	G–0	1	155
8	G#–0	1	179
9	A–0	1	205
10	A#–0	1	233
11	B–0	2	6
12	C–1	2	37
13	C#–1	2	69
14	D–1	2	104
15	D#–1	2	140
16	E–1	2	179
17	F–1	2	220
18	F#–1	3	8
19	G–1	3	54
20	G#–1	3	103
21	A–1	3	155
22	A#–1	3	210
23	B–1	4	12
24	C–2	4	73
25	C#–2	4	139
26	D–2	4	208
27	D#–2	5	25
28	E–2	5	103
29	F–2	5	185
30	F#–2	6	16
31	G–2	6	108
32	G#–2	6	206
33	A–2	7	53
34	A#–2	7	163
35	B–2	8	23
36	C–3	8	147
37	C#–3	9	21
38	D–3	9	159
39	D#–3	10	60
40	E–3	10	205
41	F–3	11	114
42	F#–3	12	32
43	G–3	12	216

Note	Note–Octave	Hi Freq	Low Freq
44	G#–3	13	156
45	A–3	14	107
46	A#–3	15	70
47	B–3	16	47
48	C–4	17	37
49	C#–4	18	42
50	D–4	19	63
51	D#–4	20	100
52	E–4	21	154
53	F–4	22	227
54	F#–4	24	63
55	G–4	25	177
56	G#–4	27	56
57	A–4	28	214
58	A#–4	30	141
59	B–4	32	94
60	C–5	34	75
61	C#–5	36	85
62	D–5	38	126
63	D#–5	40	200
64	E–5	43	52
65	F–5	45	198
66	F#–5	48	127
67	G–5	51	97
68	G#–5	54	111
69	A–5	57	172
70	A#–5	61	126
71	B–5	64	188
72	C–6	68	149
73	C#–6	72	169
74	D–6	76	252
75	D#–6	81	161
76	E–6	86	105
77	F6	91	140
78	F#–6	96	254
79	G–6	102	194
80	G#–6	108	223
81	A–6	115	88
82	A#–6	122	52
83	B–6	129	120
84	C–7	137	43
85	C#–7	145	83
86	D–7	153	247
87	D#–7	163	31
88	E–7	172	210
89	F–7	183	25
90	F#–7	193	252
91	G–7	205	133
92	G#–7	217	189
93	A–7	230	176
94	A#–7	244	103

And Now Down to Business

To use music, or noise, properly on the 64 you need to know a little about the theory behind it all.

We'll go into all that in a minute.

That is not to say that, if you don't know the first thing about sound waves, what the difference between a sawtooth and a triangle is, if you think envelope generating is something to do with a trip to the post office, you won't get a lot of enjoyment out of the 64's SID chip, and using just what we've learnt in the first section of this chapter should keep you busy impersonating various instruments, generating wonderful explosions, or whatever, for a long time to come.

However, the SID chip is capable of a lot more than the little bit we've looked at so far.

The rest of this section is devoted to the more esoteric aspects of using SID, together with a brief discussion of how they all work, and how to use them in practice.

To go into detail on all this would fill up the rest of this book, and is worthy of a book in itself, so we'll have to content ourselves with a mere outline.

But first, we'll consider waveforms, and why different sorts of waveforms produce different effects.

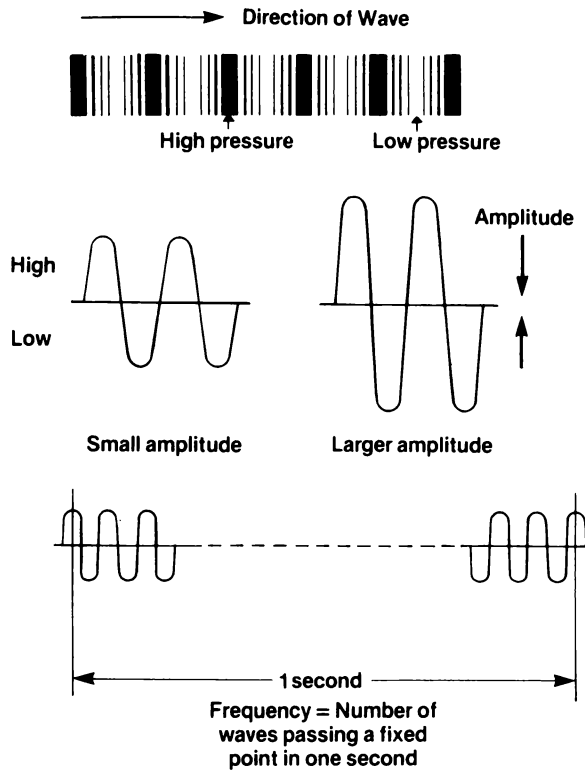
Different Types of Waveforms

As we've seen, frequency is a measure of the number of waves passing a fixed point in one second.

There are other features associated with every form of wave, and the principal of these is amplitude, simply a measure of how large or small the wave is.

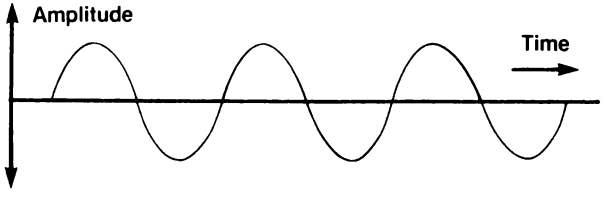
Thus, waves produced by the dropping of a stone into a pond will have a small amplitude, whereas waves down on the local beach will generally have a large amplitude.

The following diagram should help to illustrate this : =



Another term you might come across is pitch. This is just another way of reckoning up the frequency of a note: notes with a high frequency are said to have a high pitch, and ditto for a low frequency.

However, notes of the same pitch played by different instruments can sound remarkably different. This is due to the shape of the waveform that the instrument is generating, and all waveforms are made up of a variety of different sine waves.



This is the waveform on which all others are based, and any other waveform can always be reduced to a mixture of sine waves of different frequencies.

The main wave is called the fundamental one, and determines the pitch of the note, or frequency, and hence this is often called the fundamental frequency.

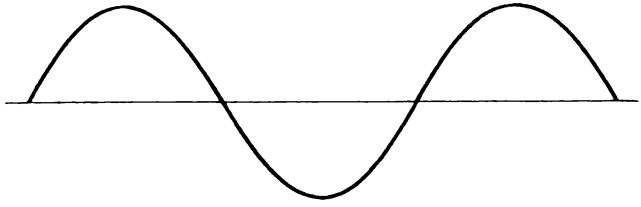
The rest of the waveform consists of a variety of variations on this fundamental frequency, and these variations are called harmonics.

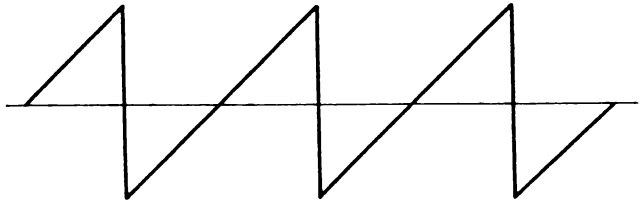
Thus a note consists of its fundamental frequency, and all the harmonics of that frequency. The frequencies of the harmonics are calculated from the fundamental, and that of the second harmonic is twice that of the fundamental (or first harmonic), the third harmonic has three times the frequency, and so on.

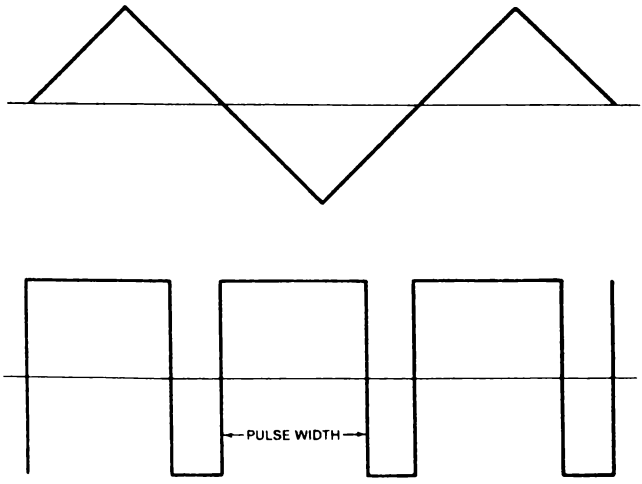
The number and mixture of harmonics go together to make up the overall quality of the note, or timbre.

The amount of each harmonic present is related to the square of the harmonic number, so that the second harmonic is 1/(2*2), or one quarter, as loud as the first one, the third is 1/(3*3), or one ninth as loud as the first one, and so on.

Now that we know all that, let's have a look at the shape of the waveforms generated by the Commodore 64.







The first one is a sine wave, reproduced here for reference.

The second is the sawtooth wave, and a look at the diagram will readily show you how it gets its name.

Sawtooths contain all the harmonics, and can be tremendously difficult to calculate, which is why precise musical impersonation can be rather difficult, and certainly beyond the scope of this current book.

The third diagram shows the triangular wave, and these are much easier to calculate, as they contain only the odd harmonics (i.e. the first, third, fifth, and so on).

The fourth diagram is that of the pulse waveform, and when using this on the 64 we have to know a little bit more about it.

On the diagram we've indicated the pulse width, or the width of each pulse of the waveform. These must be calculated for use on the 64, and the formulae go as follows :=

```
P=PW/40.95 %
```

where PW is the actual pulse width. If this is equal to 2048 for instance, we have a pure square wave.

The pulse width is made up of two components as far as the 64 is concerned, a high and a low width, and these values for voice 1 are

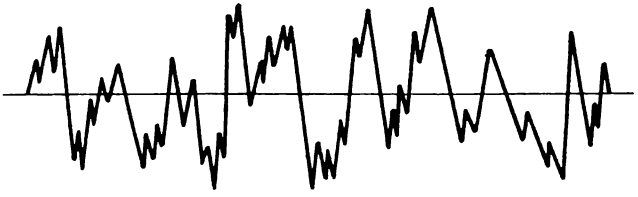
POKEd into locations S+2 and S+3. The high width lies between 0 and 15, and the low width between 0 and 255.

Looking at our earlier square wave, to program this into the machine we'd have to :=

```
POKE S+3,8:POKE S+2,0
```

in the traditional high-low form, as $0800 is equal to 2048 in decimal terms.

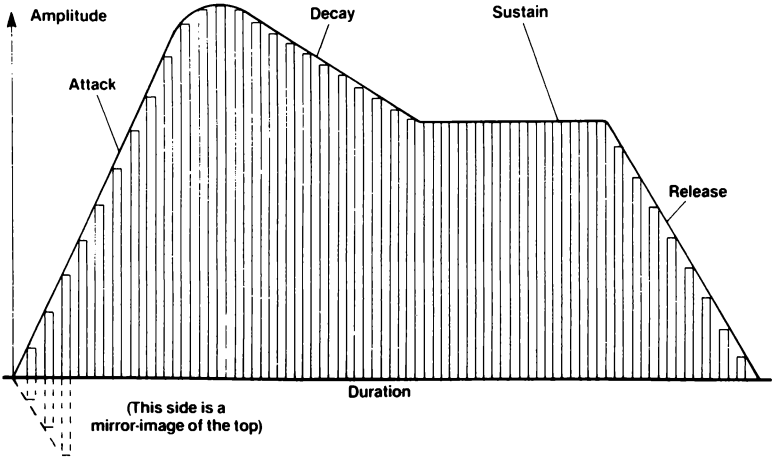
The final waveform is the noise one, which looks something like this :=



A totally random collection of waves, and hence used mainly for sound effects and the like.

To look at each individual note, as we've seen, we need to study the attck/decay and sustain/release of the note.

In graphical form, a typical note might look something like this :=



where we've plotted amplitude against time, and showed the relevant four sections of the note.

Knowing all this, perhaps you might like to turn to some of our Close Encounter programs and amend them to make them sound rather more interesting!

Envelope Generation

We have in fact already covered this, so it can't be as frightening as it sounds!

Envelope generation is just the term used to describe making up the attack/decay and sustain/release parts of a note.

Just by altering these four parameters, as we've already seen, we can produce some exciting effects.

Sample Tune Number 1

This is to be played using the Jim Butterfield tune described earlier, by putting in the data statements, and goes something like this :=

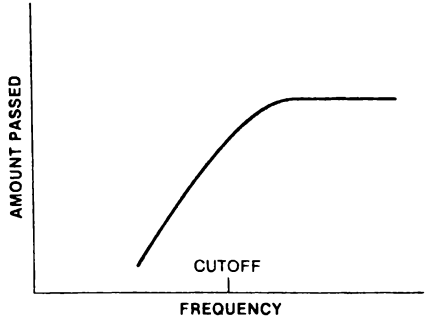
```
500 DATA 10,51,97,0,0,0,0
505 DATA 10,43,52,0,0,0,0
510 DATA 20,34,75,21,154,8,147
515 DATA 20,34,75,25,177,0,0
520 DATA 10,34,75,17,37,6,108
525 DATA 10,38,126,0,0,0,0
530 DATA 10,43,52,25,177,0,0
535 DATA 10,45,198,0,0,0,0
540 DATA 20,51,97,21,154,8,147
545 DATA 20,51,97,25,177,0,0
550 DATA 20,51,97,17,37,9,159
555 DATA 20,43,52,25,177,10,205
560 DATA 20,57,172,22,227,11,114
565 DATA 20,57,172,34,75,0,0
570 DATA 20,57,172,28,214,8,147
575 DATA 10,0,0,34,75,0,0
580 DATA 10,51,97,0,0,0,0
585 DATA 20,57,172,22,227,11,114
590 DATA 10,0,0,34,75,0,0
595 DATA 10,51,97,0,0,0,0
600 DATA 10,57,172,28,214,10,205
605 DATA 10,64,188,0,0,0,0
610 DATA 10,68,149,34,75,9,159
615 DATA 10,76,252,0,0,0,0
```

```
620 DATA 20,86,105,21,154,8,147
625 DATA 20,0,0,25,177,0,0
630 DATA 20,0,0,17,37,6,108
635 DATA 10,68,149,25,177,0,0
640 DATA 10,51,97,0,0,0,0
645 DATA 20,68,149,21,154,8,147
650 DATA 20,0,0,25,177,0,0
655 DATA 20,0,0,17,37,8,23
660 DATA 10,51,97,25,177,7,53
665 DATA 10,43,52,0,0,0,0
670 DATA 20,51,97,22,227,6,108
675 DATA 20,0,0,25,177,0,0
680 DATA 20,0,0,19,63,9,159
685 DATA 10,38,126,25,177,0,0
690 DATA 10,43,100,0,0,0,0
695 DATA 20,34,75,21,154,8,147
700 DATA 20,0,0,25,177,6,108
705 DATA 20,0,0,21,154,4,73
710 DATA 20,0,0,0,0,0,0
715 DATA 0
```

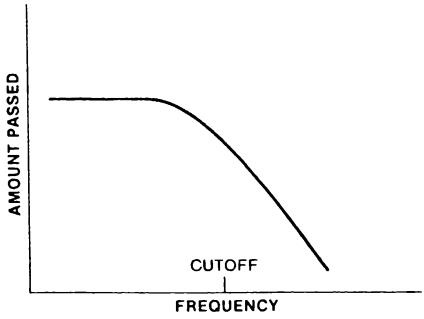
Filtering

There are three different types of filter on the 64 (see earlier for the values of the relevant memory locations), and these three filters affect the shape of the harmonic content of a waveform.

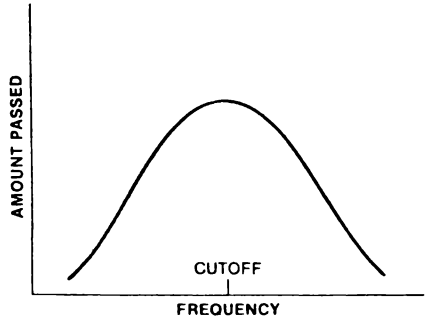
The three filters are a high pass one, which passes all the frequencies above a POKEd cut-off point :=



a low-pass filter, which, as you might guess, passes frequencies below a specified cut-off point :=



and a band-pass filter, which passes everything with a specified narrow band, and attenuates everything else :=



These can be combined in a variety of ways, and again it would be worth going back to the close encounter programs to play around with them and see what sort of effects can be generated.

Sample Tune Number 2

This is to be played using the Jim Butterfield tune described earlier, by putting in the data statements, and goes something like this :=

```
500 DATA 20,34,75,21,154,8,147
505 DATA 20,34,75,25,177,0,0
510 DATA 20,38,126,28,214,6,108
515 DATA 20,43,52,25,177,0,0
520 DATA 20,34,75,21,154,8,147
525 DATA 20,43,180,25,177,0,0
530 DATA 20,38,126,22,227,8,23
535 DATA 20,0,0,25,177,0,0
540 DATA 20,34,75,21,154,8,147
545 DATA 20,34,75,25,177,0,0
550 DATA 20,38,126,28,214,6,108
555 DATA 20,43,52,25,177,0,0
560 DATA 20,34,75,21,154,8,147
565 DATA 20,0,0,25,177,0,0
570 DATA 20,32,94,22,227,8,23
575 DATA 20,0,0,19,63,6,108
580 DATA 20,34,75,21,154,8,147
585 DATA 20,34,75,25,177,0,0
590 DATA 20,38,126,21,154,7,163
595 DATA 20,43,52,17,37,0,0
600 DATA 20,45,198,28,214,7,53
605 DATA 20,43,52,34,75,0,0
610 DATA 20,38,116,28,214,0,206
615 DATA 20,34,75,22,227,0,0
620 DATA 20,32,94,25,177,6,108
625 DATA 20,25,177,22,227,0,0
630 DATA 20,28,214,21,154,7,53
635 DATA 20,32,94,19,63,8,23
640 DATA 20,34,75,21,154,8,147
645 DATA 20,0,0,25,177,6,108
650 DATA 20,34,75,21,154,4,73
655 DATA 20,0,0,0,0,0,0
660 DATA 0
```

To Finish ...

The SID chip is capable of much more than we've discussed in these few pages, but to go into it all in detail would, as we've said, require a complete book in itself, and so we'll bring this chapter to a gentle close now.

Synchronisation and ring modulation are just two of the topics not looked at here, but briefly they involve mixing waveforms to produce some interesting results.

The demonstration programs at the end of this chapter use this technique in some cases, and these, like all the other programs, are worth playing about with.

Various parameters can be changed whilst a note is actually playing, using registers 25, 27 and 28 of the SID chip, and this too can produce some fabulous sounds.

However, you've got more than enough to be going on with, so to finish, another Butterfield tune, a program that turns the 64 into a musical keyboard, allowing you to create many musical instruments (and learn a little bit more about synchronisation), and finally a few demonstration noises, varying from a car engine to a sound that might be produced by an impolite extra-terrestrial!

Sample Tune Number 3

This is to be played using the Jim Butterfield tune described earlier, by putting in the data statements, and goes something like this :=

```
100 PRINT"MUSIC. BY JIM BUTTERFIELD"
110 I1=54272:I2=54279:I3=54286
120 H1=I1+1:H2=I1+1:H3=I3+1
130 V1=I1+4:V2=I2+4:V3=I3+4
140 POKE 54296,15
150 POKEV1+1,9:POKEV1+2,0
160 POKEV2+1,36:POKEV2+2,36
170 POKEV3+1,18:POKEV3+2,250
180 T=TI
200 POKEV1,32:POKEV2,128:POKEV3,16
210 READS:IFS=0THEN290
220 READ X1,Y1,X2,Y2,X3,Y3
230 IFX1THENPOKEH1,X1:POKEI1,Y1:POKEV1,33
240 IFX2THENPOKEH2,X2:POKEI2,Y2:POKEV2,129
250 IFX3THENPOKEH3,X3:POKEI3,Y3:POKEV3,17
260 T=T+S
270 IFT>TITHEN270
280 GOTO200
290 FORJ=I1TO54296:POKEJ,0:NEXT
300 DATA 10,50,60,0,0,0,0
310 DATA 20,67,15,42,62,8,97
320 DATA 10,67,15,0,0,0,0
330 DATA 10,67,15,44,193,6,71
340 DATA 10,63,75,0,0,0,0
350 DATA 10,56,99,0,0,0,0
360 DATA 20,50,60,42,62,8,97
370 DATA 10,44,193,37,162,0,0
380 DATA 20,42,62,33,135,6,108
390 DATA 10,50,60,0,0,0,0
400 DATA 20,67,15,42,62,8,97
```

```
410 DATA 10,67,15,0,0,0,0
420 DATA 20,75,69,44,193,9,104
430 DATA 10,63,75,37,162,0,0
440 DATA 50,67,15,42,62,8,97
450 DATA 10,50,60,0,0,0,0
460 DATA 20,67,15,42,62,8,97
470 DATA 10,75,69,0,0,0,0
480 DATA 20,84,125,44,198,6,71
490 DATA 10,89,131,0,0,0,0
500 DATA 20,100,121,42,62,8,97
510 DATA 10,85,125,37,162,0,0
520 DATA 20,67,15,33,135,10,143
530 DATA 10,75,69,0,0,0,0
540 DATA 20,84,125,50,60,8,97
550 DATA 10,84,125,0,0,0,0
560 DATA 10,84,125,50,60,7,12
570 DATA 10,75,69,44,193,0,0
580 DATA 10,67,15,42,62,0,0
590 DATA 50,75,69,44,198,6,71
600 DATA 10,50,60,0,0,0,0
610 DATA 20,67,15,42,62,8,97
620 DATA 10,75,69,0,0,0,0
630 DATA 20,84,125,44,198,6,71
640 DATA 10,89,131,0,0,0,0
650 DATA 20,100,121,42,62,8,97
660 DATA 10,85,125,37,162,0,0
670 DATA 20,67,15,33,135,10,143
680 DATA 10,75,69,0,0,0,0
690 DATA 20,84,125,50,60,8,97
700 DATA 10,84,125,0,0,0,0
710 DATA 10,84,125,50,60,7,12
720 DATA 10,75,69,44,193,0,0
730 DATA 10,67,15,42,62,0,0
740 DATA 30,75,69,44,193,6,71
750 DATA 20,84,125,50,60,0,0
760 DATA 10,75,69,44,193,0,0
770 DATA 20,67,15,42,62,8,97
780 DATA 10,67,15,0,0,0,0
790 DATA 10,67,15,44,193,6,71
800 DATA 10,63,75,0,0,0,0
810 DATA 10,56,99,0,0,0,0
820 DATA 20,50,60,42,62,8,97
830 DATA 10,44,193,37,162,0,0
840 DATA 20,42,62,33,135,6,108
850 DATA 10,50,60,0,0,0,0
860 DATA 20,67,15,42,62,8,97
870 DATA 10,67,15,0,0,0,0
880 DATA 20,75,69,44,193,9,104
890 DATA 10,63,75,37,162,0,0
900 DATA 50,67,15,42,62,8,97
910 DATA 0,0,0,0,0
920 DATA 0
```

```
5 VO(0)=1
6 V=54272:POKEV+24,15
7 POKE 53272,23
20 GOSUB6000
30 GOSUB5000
1000 K=PEEK(197):PS=PEEK(653)
1020 F=N(K)
1025 IFK=48ANDDL=0THENDL=100:GOTO 1000
1026 IFK=48ANDDL=100THENDL=0:GOTO 1000
1030 IF F=0 THEN 2000
1040 IF (F>0ANDF<9)THEN 3000
1045 IFK=1THEN4000
1050 IF PS=1 THEN F=INT(F*2^(1/12))
1060 IF PS=2 THEN F=INT(F/2^(1/12))
1070 F1=INT(F/256)
1080 F2=F-F1*256
1085 FOR I=0 TO 2
1086 IF VO(I)=0 THEN 1125
1090 POKE V+I*7+4,0
1100 POKE V+I*7+4,W(I)*16+RM(I)+SY(I)+1
1110 POKE V+I*7,F2
1120 POKE V+I*7+1,F1
1122 FORP=1TODL:NEXTP
1125 NEXT I
1130 GOTO 1000
2000 FOR I=0 TO 2
2010 POKE V+I*7,0
2020 POKE V+I*7+1,0
2030 POKE V+I*7+4,W(I)*16
2040 NEXT I
2050 GOTO 1000
3000 F=F-1
3010 FOR I=0 TO 2
3020 VO(I)=(FAND2^I)/2^I
3030 NEXT I
3040 GOTO 1000
4000 PRINT"[CLR]              VOICE 1   VOICE 2  VOIC
E 3"
4001 FORI=1TO10:GETKY$:NEXT
4002 PRINT"[CD]WAVEFORM";TAB(12);W$(0);TAB(20);W$(
1);TAB(30);W$(2)
4004 PRINT"ATT/DEC";TAB(13);AD(0);TAB(23);AD(1);TA
B(32);AD(2)
4006 PRINT"SUS/REL";TAB(13);SR(0);TAB(23);SR(1);TA
B(32);SR(2)
```

```
4008 PRINT"PULSE HI";TAB(13);PH(0);TAB(23);PH(1);T
AB(32);PH(2)
4010 PRINT"PULSE LO";TAB(13);PL(0);TAB(23);PL(1);T
AB(32);PL(2)
4012 PRINT"RING MOD";TAB(13);RM(0);TAB(23);RM(1);T
AB(32);RM(2)
4014 PRINT"SYNC     ";TAB(13);SY(0);TAB(23);SY(1);T
AB(32);SY(2)
4016 PRINT"[CD]DO YOU WANT TO CHANGE ANY VALUES (Y
/N)?"
4018 GETCH$:IFCH$="N"THEN30
4020 IFCH$<>"Y"THEN4018
4022 PRINT"[CD]WHICH VOICE (1, 2 OR 3)?"
4024 GETVC$:IFVC$=""THEN4024
4026 IFVC$="1"THENPRINT"VOICE1":VC=0:GOTO4030
4027 IFVC$="2"THENPRINT"VOICE2":VC=1:GOTO4030
4028 IFVC$="3"THENPRINT"VOICE3":VC=2:GOTO4030
4029 GOTO 4024
4030 PRINT"WAVEFORM (T,S,P,N)?"
4031 GETWF$:IFWF$=""THEN4031
4032 IFWF$="T"THENPRINT"TRIANGLE":W(VC)=1:W$(VC)="
TRIANGLE":GOTO4037
4033 IFWF$="S"THENPRINT"SAWTOOTH":W(VC)=2:W$(VC)="
SAWTOOTH":GOTO4037
4034 IFWF$="P"THENPRINT"PULSE":W(VC)=4:W$(VC)="PUL
SE":GOTO4037
4035 IFWF$="N"THENPRINT"NOISE":W(VC)=8:W$(VC)="NOI
SE":GOTO4037
4036 GOTO4031
4037 INPUT"ATTACK/DECAY ";AD(VC):IFAD(VC)<00RAD(VC
)>255THENPRINT"[2CU]":GOTO4037
4039 INPUT"SUSTAIN/RELEASE ";SR(VC):IFSR(VC)<00RSR
(VC)>255THENPRINT"[2CU]":GOTO4039
4041 INPUT"PULSE HI ";PH(VC):IFPH(VC)<00RPH(VC)>25
5THENPRINT"[2CU]":GOTO4041
4043 INPUT"PULSE LO ";PL(VC):IFPL(VC)<00RPL(VC)>25
5THENPRINT"[2CU]":GOTO4043
4045 INPUT"RING MOD ";RM(VC):IFRM(VC)<00RRM(VC)>25
5THENPRINT"[2CU]":GOTO4045
4047 INPUT"SYNC ";SY(VC):IFSY(VC)<00RSY(VC)>255THE
NPRINT"[2CU]":GOTO4047
4049 GOTO4000
4250 RETURN
5000 POKE 53280,6:POKE 53281,7
5005 PRINT"[BLK,CLR,RVS]*********   SING-A-LONG-A-
64   *********[OFF]"
5008 PRINT"[RVS]**********       BY SID        ***
******"
5010 PRINT" PLAY USING THE KEYS  [RVS]Q W E R T Y
U I[OFF]"
```

```
5020 PRINT"[CD]                              [RVS]A S D F
G H J K[OFF]"
5030 PRINT"[CD]                              [RVS]Z X C V
B N M ,[OFF]"
5040 PRINT"[CD] TO COVER THREE OCTAVES."
5050 PRINT"[CD] USE THE [RVS]SHIFT[OFF] KEY FOR A
SHARP"
5055 PRINT"          [RVS]\[OFF] TO CHANGE SPEED
5060 PRINT"          [RVS]CBM[OFF]   KEY FOR A FLAT
"
5070 PRINT"[CD]USE THE KEYS [RVS]0 1 2 3 4 5 6 7[O
FF] TO"
5080 PRINT"CHOOSE ANY COMBINATION OF THE THREE"
5090 PRINT"VOICES. THEY ARE SET UP USING BINARY"
5100 PRINT"ARITHMETIC. THUS, VOICE 1 IS TURNED ";
5110 PRINT"    ON USING KEY 1, VOICE 2 BY KEY 2, V
OICE3";
5120 PRINT"BY KEY 4, AND VOICE 7 TURNS THE LOT ON!
"
5130 PRINT"[CD]USE THE [RVS]RETURN[OFF] KEY TO CHA
NGE THE VALUES"
5140 PRINT"OF THE VOICES.[HOME]"
5150 RETURN
6000 DIM N(64)
6005 FOR I=0 TO 64
6010 READ A
6015 N(I)=A
6020 NEXT I
6025 DATA 0,-1,0,0,0,0,0,0
6030 DATA 4,9854,4389,5,2195,4927
6035 DATA 11060,0,6,11718,5530,7,2765,5859
6040 DATA 13153,2463,8,14764,6577,0,3288,7382
6045 DATA 16572,2930,0,17557,8286,1,4143,8779
6050 DATA 0,3691,0,0,0,0,0,0,0,4389,0,0,0
6055 DATA 0,0,0,0,0,2,0,0,3,0
6060 DATA 0,8779,0,0
6070 FORI=0TO2:READW(I),AD(I),SR(I),PH(I),PL(I),W$
(I),RM(I),SY(I):NEXTI
6100 DATA4,9,0,0,255,"PULSE",0,0
6110 DATA2,96,0,0,0,"SAWTOOTH",0,0
6120 DATA1,102,0,0,0,"TRIANGLE",0,0
6122 FORI=0TO2
6124 POKEV+7*I+5,AD(I):POKEV+7*I+6,SR(I):POKEV+7*I
+3,PH(I):POKEV+7*I+2,PL(I)
6126 NEXTI
6130 RETURN
```

Some Sample Noises

Three short programs to show you a little bit about the wonderful noises the 64 can produce.

```
10 S=54272
20 FORL=0TO24:POKE S+L,0:NEXT
30 POKE S+3,8
40 POKE S+5,32:POKE S+6,72
70 POKE S+24,15
80 POKE S+4,17
85 FORK=1TO12
90 FORI=1TO33:POKES+1,I:POKES,I:NEXT
95 NEXTK
100 FORL=0TO24:POKES+L,0:NEXT
```

```
10 S=54272
20 FORL=0TO24:POKE S+L,0:NEXT
30 POKE S+3,8
40 POKE S+5,32:POKE S+6,72
70 POKE S+24,15
80 POKE S+4,129
85 FORK=1TO12
90 FORI=33TO1STEP-1:POKES+1,I:POKES,I:NEXT
92 FORM=1TO1000:NEXT
95 NEXTK
100 FORL=0TO24:POKES+L,0:NEXT
```

```
10 S=54272:I=1
20 FORL=0TO24:POKE S+L,0:NEXT
30 POKE S+3,12
40 POKE S+5,40:POKE S+6,146
70 POKE S+24,15
80 POKE S+4,65
85 FORK=1TO250
90 POKES+1,I:POKES,I
95 NEXTK
100 FORL=0TO24:POKES+L,0:NEXT
```

7

Peripheral Support

In this chapter we'll be taking a look at some of the more common peripherals that people attach to their Commodore 64s, including printers, disk drives, and cassette decks, but not such esoteric devices as Z80 cards, modems or getting it running in CP/M language mode.

The reason for this is purely a commercial one, in that not many people will have the latter three facilities for their 64, and those that have will want a lot more detail than can be given in a book like this!

However, most people will have a cassette deck or a disk drive, and quite a few will also have a printer, so we intend to take a brief look at all three, as well as giving you some useful subroutines for use in your own programs.

We'll be looking at file handling on disk and tape, dumping the contents of the screen to a printer (or anywhere else for that matter!), reading program files from disk to screen without disturbing the program currently sitting in memory, and so on.

Simple Commands

In order to save your programs, or load commercially available tapes or disks, you'll obviously need to know the commands available for loading and saving of software.

The first file type that we'll consider is the simple program, something like this :=

```
10 PRINT "[CLR]HELLO THERE, I'M A COMMODORE 64."
```

If we give the program the name HELLO, then the syntax to save it on tape would be as follows :=

```
SAVE "HELLO"
```

which would prompt the computer to tell you to press the play and record buttons on your casstte deck.

To save it on disk, we'd need : =

```
SAVE "0:HELLO",8
```

where the zero indicates drive 0 (not strictly necessary on a single drive system, but you may be using a double disk drive via some commercially available interface), and the 8 indicates the device number of the disk drive.

This can be changed by software, as we've seen.

To save machine code programs from the assembler requires a totally different syntax, and you'll find that, together with the instructions for operating the assembler, in Appendix D.

To load programs, just use the word LOAD in place of the word SAVE, and keep everything else the same.

If you want to save a program over the top of another one, the command for tapes is exctly the same, but for disks you'll need to add something : =

```
SAVE "@0:HELLO",8
```

where the @ symbol means replace the old program called HELLO with this new version.

People have in the past reported strange errors occuring when using this command, but I've certainly never experienced any problems with it. It probably belongs to the 'well my friend said a friend of his told him that someone he knew ...' school of thought.

To make sure you've copied a proper working version of the program onto tape or disk, you must use the Verify command. It's not really worth bothering with this for disks, but for tapes it is virtually essential. The syntax is : =

```
VERIFY "HELLO"
```

with the appropriate alterations for disk usage.

Other Types of Files

All we've looked at so far are simple program files. In other words, just collections of line numbers that are stored as a block onto tape or disk.

However, there are a number of other file types which are used for handling data, and the two that we'll look at here are SEQuential and RELative.

Sequential files are the only other type of file that can be stored on tape apart from programs.

They are essentially collections of data, stored sequentially, i.e. one bit of data right after another, and when these files are read back they are read in the same order that they were stored in.

Thus if we stored the numbers 1 2 3 onto tape, when we came to read the data back later we'd find that the first number read in was a 1, then the 2, and finally the 3.

The size of a sequential file is simply determined by how much data you're actually going to be storing onto the tape or disk.

Disks handle sequential files in exactly the same manner, but they can also store data in a RELative file.

Relative files allow direct access to the information stored in them, rather than wading through a whole lot of other data before you get to the particular bit that you want.

The usual analogy here is with a record player and a cassette deck. On a cassette deck you have to wind through all the rest of the records to get to the fifth track, say. With a record player you simply lift up the arm of the stylus and plonk it down wherever you want it to go.

Thus, it is a lot faster finding a certain track, and so relative files are a lot faster than sequential: you just tell the disk drive where the information is, and off it goes and gets it.

Sequential Files

When operating with sequential files either on tape or on disk, it is important to remember a few rules.

Principal amongst these must be storing the data in the correct order, and reading it back in the correct order as well!

It is also far too easy to make a simple mistake when filing operations are underway, so we'll take a look at a few programs, and work out how to avoid the problems.

In a sequential file, every item of data must be separated by a carriage return, in order that the computer can tell each piece of data apart when it comes to reading it all back in again.

So, a program to file the numbers 1 to 15 onto tape might look something like this :=

```
5 CR$=CHR$(13) : REM CR$ NOW EQUALS A CARRIAGE
RETURN
10 PRINT"[CLR]SEQUENTIAL FILES ONTO TAPE"
20 OPEN 1,1,1,"DATA"
25 REM OPEN A FILE TO THE TAPE FOR WRITING
30 REM AND CALL IT DATA
40 FORI=1TO15 : REM DO THIS FIFTEEN TIMES
50 PRINT£1,I;CR$;
55 REM PRINT THE NUMBER, AND SEPARATE IT FROM THE
NEXT ONE
56 REM WITH A CARRIAGE RETURN
60 NEXT I : REM GO BACK AND DO IT AGAIN.
70 CLOSE 1 : REM CLOSE THE FILE
80 END : REM THAT'S IT!
```

Please note: here, and in the rest of the book, for the pound sign, read 'hash'.

When RUN, this program prompts you to press play and record on the cassette deck, and stores the file away.

So now we have a file called data stored onto tape, how do we read that data back again?

The answer is charmingly simple, and we have to make very little alteration to the program.

The finished result might look something like this :=

```
5 DIM A(15) : REM DIMENSION AN ARRAY FOR READING
DATA
10 PRINT"[CLR]SEQUENTIAL FILES FROM TAPE"
20 OPEN 1,1,0,"DATA"
25 REM OPEN A FILE TO THE TAPE FOR READING DATA
30 REM AND LOOK FOR A FILE CALLED DATA
40 FORI=1TO15 : REM DO THIS FIFTEEN TIMES
50 INPUT£1,I
55 REM READ IN EACH ITEM OF DATA, AND STORE IT
56 REM IN THE ARRAY A(
57 A(I)=I
60 NEXT I : REM GO BACK AND DO IT AGAIN.
70 CLOSE 1 : REM CLOSE THE FILE
80 FORI=1TO15
90 PRINTA(I) : REM PROVE WE'VE DONE IT!
100 NEXT I
110 END : REM EASY WASN'T IT ?
```

Of course, strings can be stored as well as numbers, and the syntax is exactly the same.

Another command instead of INPUT would be GET, for getting single characters or strings at a time.

However, beware of one thing. If you have a string A$ = "this is a string, with a comma", when that string is stored onto tape the part after the comma will become the next bit of data, and the whole string gets split up. This is likely to cause problems when reading data back later, so you've either got to get rid of the comma, or take great care when reading back that sort of data.

Merging Programs from Tape

There is a nice simple way of merging program files on tape with a program already in memory. This was first described by Jim Butterfield many years ago for the Commodore PET, but the technique remains the same.

First of all, prepare the lines you want to merge by loading them into memory, and then changing them to a program file in the following way :=

```
OPEN 1,1,1,"PROGNAME":CMD1:LIST
```

When this has finished, and the word READY re-appears, type

```
PRINT£1:CLOSE1
```

To merge this with a program already in memory, obviously a different program, put the prepared tape in the cassette deck, and type :=

```
POKE 19,1:OPEN1
```

When READY reappears again, clear the screen and press exactly three cursor downs. Then type :=

```
PRINTCHR$(19):POKE 198,1:POKE 631,13:POKE 153,1
```

When the tape stops you'll get an error message, but ignore that. Just type CLOSE1, and you'll find that the program lines are now merged together.

And onto Disk

When using disk drives, the same sort of syntax is required, and the following program, heavily REMmed for ease of following, is a good demonstration of how to write a sequential file onto disk, and then read all the data back again.

```
10 REM **********************
11 REM *       EXAMPLE        *
15 REM *  READ AND WRITE A    *
20 REM *  SEQUENTIAL DATA     *
25 REM * FILE ON 1541 DRIVE   *
30 REM **********************
35 PRINT"[CLR,BLK]INITIALISE DISK"
40 DIM A$(25),B(25)
45 OPEN15,8,15:REM OPEN COMMAND CHANNEL
50 PRINT£15,"I0":REM INITIALISE DRIVE
55 GOSUB245:REM CHECK ERROR CHANNEL
60 CR$=CHR$(13):REM SET CR$ EQUAL TO A CARRIAGE RE
TURN
65 PRINT "[RVS]WRITE TEST FILE"
70 REM **********************
75 REM *                      *
80 REM *   WRITE TEST FILE    *
85 REM *                      *
90 REM **********************
95 OPEN2,8,2,"@0:TEST FILE.S,W":REM OPEN FILE ON C
HANNEL 2 TO DEVICE 8
100 REM AND REPLACE THE FILE ON DRIVE ZERO CALLED
TEST FILE WITH ANOTHER
105 GOSUB 245:REM READ ERROR CHANNEL
110 INPUT"A$,B";A$,B:REM INPUT DATA FROM USER
115 IFA$="END"THEN135:REM IF USER TYPES END, THEN
STOP INPUT OF DATA
120 PRINT£2,A$","STR$(B)CR$;:REM WRITE DATA
125 GOSUB 245:REM READ ERROR CHANNEL
130 GOTO 110: REM GET MORE DATA
135 CLOSE 2:REM CLOSE CHANNEL
140 REM **********************
145 REM *                      *
150 REM *   READ TEST FILE     *
155 REM *                      *
160 REM **********************
165 PRINT"[RVS]READ TEST FILE"
```

```
170 OPEN2,8,2,"0:TEST FILE.S,R":REM READ FILE THIS
 TIME
175 GOSUB 245
180 INPUT£2,A$(I),B(I):REM INPUT DATA AGAIN
185 RS=ST:REM STORE THE DISK STATUS
190 GOSUB 245:REM READ ERROR CHANNEL
195 PRINTA$(I),B(I):REM PRINT WHAT WAS READ
200 IFRS=64THEN220:REM CHECK FOR END OF FILE STATU
S
205 IFRS<>0THEN230:REM CHECK FOR ERROR IN FILE STA
TUS
210 I=I+1:REM INCREMENT COUNTER
215 GOTO 180:REM GET MORE DATA
220 CLOSE2:REM CLOSE FILE
225 END
230 PRINT"[RVS]BAD DISK STATUS IS "RS
235 CLOSE2:REM CLOSE FILE
240 END
245 REM **********************
250 REM *   READ THE ERROR   *
255 REM *      CHANNEL       *
260 REM **********************
265 INPUT£15,EN$,EM$,ET$,ES$:REM READ ERROR
270 REM EN$ IS ERROR NUMBER
275 REM EM$ IS ERROR MESSAGE
280 REM ES$ IS ERROR SECTOR
285 IFEN$="00"THENRETURN : REM RETURN TO PROGRAM I
F NO ERRORS
290 PRINT"[RVS]ERROR ON DISK ":EM$:REM PRINT ERROR
 MESSAGE
295 CLOSE15:CLOSE2:REM CLOSE CHANNELS
300 END
```

Relative Files, and How to Write One

This will obviously only apply to disks.
Commodore's disk drives have one great advantage over a lot of others in that they are intelligent, and require no memory from the computer itself: they have their own memory, and their own chips inside to handle a lot of tasks.

So, in this section we'll utilise all this, and introduce you to a few new disk commands on the way.

It is a lot easier to write a sequential file than it is to write a relative one, as we shall see. However, all the commands that allow us to write relative files are built into the computer, and provided you know what they are and how they work, it becomes a relatively (sorry!) easy task to write a direct access system.

The terms relative files, random files and direct access files are all interchangeable, as they all mean precisely the same thing.

Once you know what's going on, you'll then be able to build up your own powerful programs, as well as possibly understanding the one given away with the free disk when you buy a Commodore disk drive!

How Information is Stored

Information is stored on the disk in a series of tracks, and each track is divided up into a series of little boxes called sectors. There is a total of 35 tracks on a 1541 disk drive, and the number of sectors depends on the location of the track. Tracks near the middle only have 17 sectors in them, and those at the outside have up 21 sectors in them.

The total number of sectors on the disk is 690, and the disk itself tells you how many of these there are left vacant on the disk. A look at any disk directory will reveal how many sectors, or blocks, free there are. Since the disk drive itself cobbles a few sectors for its own use, like linking programs together on disk, reminding itself where they all are, what they're called, how long they are and so on, we get left with 664 for our own use.

Each sector can hold up to 255 charcacters, and information is talked of as being in track 5, sector 7, or whatever.

A collection of items of data or information is called a file, and let's say we want to store the surname, first name and telephone number of all your friends in a file, held in alphabetical order. The information held for each person in the file is called a record, and the bits of information in each record are called items. So in the example we have three items of data per record.

This could of course all be held as a sequential file but if you've got a lot of friends, it would take a long time to find the telephone number of someone whose name begins with W, for instance.

If you then wanted another bit of information, you'd have to go through the whole file again. A tedious procedure.

Direct Access

Direct access allows us to specify which sector in which track and starting at which character a particular item of information is to go.

And provided you keep track of where it's all stored, we can directly access it at some later date, without having to go through all the other records.

Since we can read and write to specific places on the disk, it means

that we can also amend any items of information without having to update the whole file, which is certainly not the case with sequential files.

The main precaution that we must take is that we don't write our data into sectors which are used by other programs on the disk, otherwise those programs will become corrupted. Thus it makes sense to have one program disk and one data disk for direct access, and not to mix the two.

When using direct access we normally specify a fixed length for each item of information in the record. Thus our surname item could be 20 characters long, the Christian name 10, and the telephone number 20. Whatever length a name actually is, it will still occupy that much space.

It makes it easier to keep track of where all the data is, but it unfortunately takes up more room on the disk. A small price to pay for having a direct access file.

The Components of the System

Since the total number of characters used in this example is 50, it would be possible to maintain 5 records per sector. However, we'll keep things simple at this stage and stick to one record per block.

The seven stages in writing a direct access file are as follows : =

1) Open up a route from the 64 to a buffer in the disk unit.
2) Copy the record data into that buffer, starting at the first character position.
3) Find the next available block on the disk.
4) Tell the Disk Operating system that you want that block.
5) Put all the data from the buffer into it.
6) Make an entry in an index array relating the block to the record key, which is the index word you use to look up the record details. In our case, the surnames of the various people.

7) At some later stage, save the index
array as a
sequential file.

Once you've saved the index, that's it, and you can get on with something else.

To read a record back is a five-stage process :=

1) Read the index array back into a Basic
array.
2) Open up a route from the disk buffer
to the 64.
3) Search the index for the keyword of a
required record
and note the track and sector
numbers associated
with it.
4) Read the whole of that sector
specified from the disk
into the buffer.
5) Transfer the contents of the buffer
into a Basic
variable.

Finally, amending a direct access record is a four-stage process :=

1) Read the whole block into the disk
buffer as in
1) to 4) above.
2) Point at the part of the buffer to be
overwritten.
3) Copy the new information from Basic
into the buffer,
overwriting the specified portion of
the information
in the buffer.
4) Write the contents of the buffer back
to the block it came from.

Writing a Record

First of all, open up a command route, using whatever disk drive device number is currently active: usually 8.

We then have to specify a command channel, which can be any number from 2 to 14, since these are the routes used to transfer information to and from disk and 64, and channel 15 is the route for commands to the ROM in the disk and messages coming back from it.

We also must specify a logical file number, which can be any number from 1 to 255, and which acts as a key to other details without you having to keep typing them in.

Normally people choose the same file number as channel number, so OPEN 15,8,15 would be the syntax we want, and records can now be got at using PRINT#

The Data Route

Five routes, as well as the command route, can be open at the same time, and because of the time it takes to close a file (up to a few seconds, whilst the disk drive tidies everything up) it's as well to keep them all open whilst everything's going on, rather than opening them and closing them all the time.

We also need to reserve a buffer to hold the information going to or coming from the disk drive, and the syntax here is OPENfile number,8,channel,"#", where the # reserves the next available buffer, and associates it with that channel.

Thus if we opened channel 2 we'd use :=

```
OPEN 2,8,2,"£"
```

Copying the Data

This is done using the PRINT# command, and sending data down whatever channel we've opened into the correct buffer.

Thus if our first record was :=

TIMM.....DENIS.....01 959 1618.....

and the part to go to the buffer, as the keyword need not be part of the direct access record, was stored in the variable A$ thus :=

A$="DENIS.....01 959 1618"

it could be transferred to the buffer by PRINT#2,A$.

However, before we can do this we need to set the block pointer to a free block, and this is done using the Block Point command, or B-P.

The syntax for this would be :=

```
PRINT£15,"B-P";C;P
```

where C is the data channel number and P is the pointer position required.

So our command would be :=

```
PRINT£15,"B-P";2;1
PRINT£2,A$
```

Finding a Free Block

A simple way of finding free blocks is to put the direct access files onto an otherwise empty diskette. Then, as long as you don't use track 18 sectors 0 to 2, which are reserved for the disk directory, you control completely the placing of information on the disk.

However, if there are going to be other things there we must find out which blocks are free and reserve them. This is done using the Block Allocate command.

When you use this command to tell the disk that a particular block is to be allocated, and read the error channel, one of two messages will occur.

Either you'll be told OK, and the block will be allocated, or you'll get error message number 65, BAD BLOCK, followed by two numbers, which are the track and sector of the next free block down the disk.

So, if you always attempt to allocate track 1, sector 0, the first time you'll be okay, and ever after you'll be told BAD BLOCK, and the location of the next free track and sector. Then, if we allocate that, it will be reserved for our use and the disk drive won't use it when storing programs or sequential files.

So, the complete syntax would look like this :=

```
PRINT£15,"B-A";0;1;0
```

which attempts to allocate drive 0, track 1, sector 0

```
INPUT£15,EN,EM$,ET,ES
```

which reads the error message from the error channel. If EN = 65, EM$ will equal BAD BLOCK, and we can then : =

```
PRINT£15,"B-A";0;ET;ES
```

which allocates drive 0, track ET and sector ES: the next free block.

Transferring the Buffer to the Disk

Another disk command, Block Write, must be used, specifying the command channel, the drive number, the track and the sector, which we already know to be ET and ES. So, we must : =

```
PRINT£15,"B-W";2;0;ET;ES
```

assuming we're using command channel 2, of course.

Keeping Track of the Index

The index associates the keyword with the track and sector number of the information you put on the disk, and the ideal way to handle all this would be in an array, holding the surname, and the associated track and sector.

Thus our entries might look like this : =

DOYLE	1	2
PARKINSON	1	1
TIMM	1	0

in alphabetical order. The track and sector can then be read off using the VAL and STR$ commands to go from string data to numeric data and back again.

As the index grows, searching through it becomes a tedious process, so it would be a wise idea to keep it all in alphabetical order, and then use a binary search to get to the specified item quickly.

This involves guessing at the middle location in the array, and then (if there were 100 elements in it) guessing at 75 if your guess is too low, or 25 if it's too high: do you see why it should be alphabetical now? So that the computer can alphabetically compare the name it finds there with the name that it's after, and branch accordingly.

Reading Data Back Again

Thus, when we do track the name down we can get the track and sector numbers and read the data back again, using the U1 command, which requires the channel number, drive number, the track and the sector.

This command transfers the contents of a specified track and sector into the associated channel buffer, and a simple INPUT # down that channel will transfer the information into Basic.

Like this (assuming channel 2) : =

```
PRINT£15,"U1";2;0;T;S
```

which will transfer the information from track T sector S into the buffer associated with channel 2.

```
INPUT£2,A$
```

which will put the contents of the buffer into the variable A$

To Conclude

A couple of things to watch for. Try to keep your records less than 80 characters long, as Basic Input # can normally only handle 80 character chunks at a time. Any more than that and the machine usually hangs up!

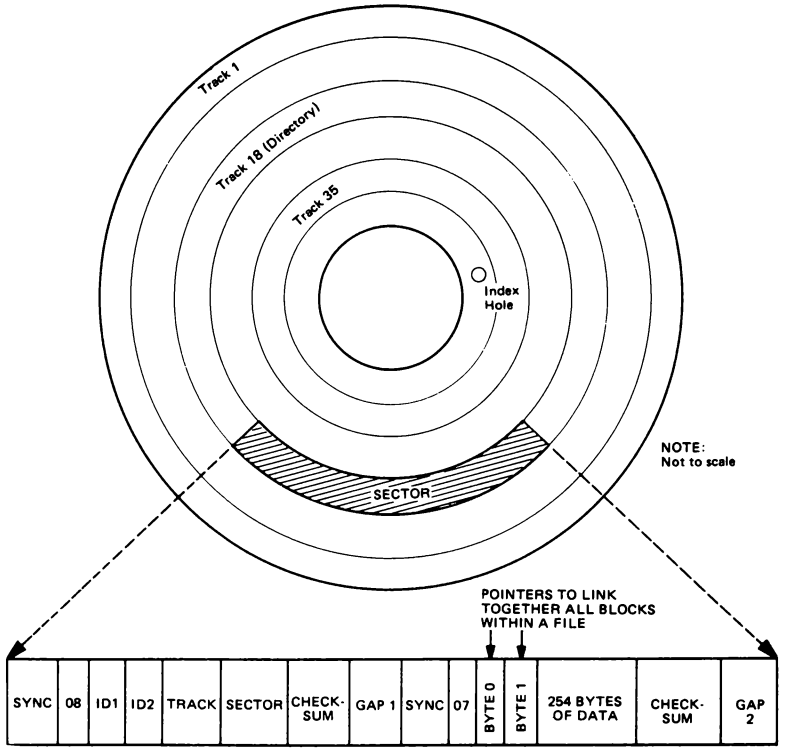
Don't put blank strings as part of the disk information, convert them into shifted spaces or something.

Validating disks will cause all kinds of chaos, and if you have done that you'll have to re-create everything from your key index array.

And that's that!

How Data is Stored on Disk

We've described this verbally, in terms of tracks and sectors, but a typical disk set-up might look something like this : =



The manual you got with your disk drive includes most of the information you'll need about what's stored where, and how the commands work.

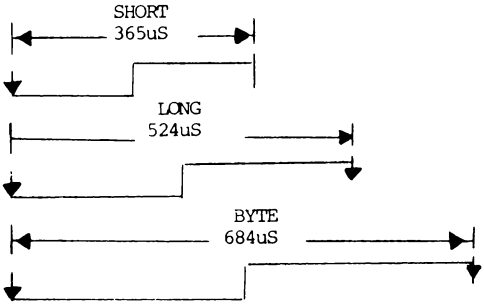
If you're using a 1541, make sure you get a copy of the new manual, not the old 1540 one!

Thanks to Mike-Gross Niklaus for the random access information.

How Data is Stored on Tape

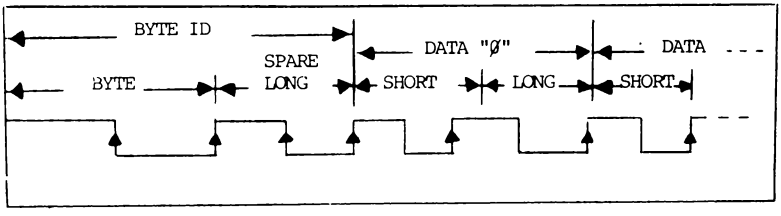
Marin Maynard takes up the story : =

"The Commodore cassette decks use an unequalised recording method of placing data on tape, by switching the direction of current through the record head, saturating the tape either negatively or positively. The encoding scheme uses three distinct full cycle pulses, thus : =



"A data zero or one is represented by a pairing of a long and short pulse. If the short pulse is first, the pair is considered a one.

"The byte marker provides reference for byte identification.



"In the playback circuit the recorded signal passes through equalisation and squaring circuits, and logic level signals are presented to the 64. The 64 measures between negative going edges of signals and decodes the data from these measurements.

"Slew rate of these negative going pulses is critical, because if the slew rate is slow there is a larger area of indecision presented to the squaring circuits, whose threshold is set about the zero crossing point of magnetic flux on tape.

Listening to a 64 tape played through your stereo is quite revealing, and can give an interesting check on how well the cassette deck (the 64 one, not yours!) is working.

"Lack of high frequency content will indicate dirty or magnetised tape heads, and speed variations can be detected as overall pitch drifting."

So now you know!

Using Different Cassette Decks

Although the Commodore cassette decks are designed specifically for Commodore machines, it is annoying if you already own a cassette deck to be told, 'Sorry guv, but you need one of these', and to have to hand over the extra money.

The only reason why the Commodore cassette decks are unique is that the computer supplies the power to them. Given a cassette deck with its own supply, all we have to alter is the edge connector that links us up to the computer. Looking at that connector, the terminals on it are numbers A,B,C,D,E and F on the bottom, and 1,2,3,4,5 and 6 on the top. The pinouts are in pairs, thus :

A-1 : ground (connect this to computer ground)

B-2 : + 5 volts (this one's not used)

C-3 : cassette motor (use a remote source i.e. a plug!)

D-4 : cassette read (earphone)

E-5 : cassette write (microphone!)

F-6 : cassette switch (this one's not used either)

The only problem might be getting a suitably sized edge connector: Commodore use an odd size for their machine. Since Commodore presumably won't be too happy about supplying you with one themselves, the easiest solution is to get a larger one of the same spacing and cut it to fit.

Not too bad really.

Some Disk Utilities

Four pages of disk utility programs for the 64, a couple of which are hangovers from the early Commodore days, but new 64 owners might as well have a share of the fun as well.

In order, they simply read the name of a disk and the name of every file on it, and store that in an array in memory; they allow you to check the disk processor and the IEEE processor, a very short program for displaying how much room is left on a disk, and finally a most useful Butterfield utility for doing all kinds of things to disk. Instructions are contained in the program.

So, have fun!

```
50 DIML$(100),FT$(100)
100 OPEN 1,8,0,"$":FORI=1TO33:GET£1,A$:L$=L$+A$:NE
XT:CLOSE1
110 PRINT"[CLR]DISK NAME = ";L$
120 PRINT"[2CD]FILE NAME       = ";
130 OPEN 1,8,0,"$":GET£1,A$:GET£1,A$:GOSUB60000
140 GOSUB60000:IFFL=1THEN170
150 M=M+1:L$(M)=B$:FT$(M)=FT$
160 PRINTB$;LEFT$("                    ",16-LEN(B$));
"[16CL]";:GOTO140
170 CLOSE1
59999 END
60000 B$="":FORI=1TO4:GET£1,A$:NEXT
60010 GET£1,A$:IFA$=""THENFL=1:RETURN
60020 IFA$<>CHR$(34)THEN60010
60030 GET£1,A$
60040 IFA$=CHR$(34)THEN60060
60050 B$=B$+A$:GOTO60030
60060 GET£1,A$:IFA$=CHR$(32)THEN60060
60070 FT$=A$
60080 GET£1,A$:FT$=FT$+A$:GET£1,A$:FT$=FT$+A$
60090 GET£1,A$:IFA$<>""THEN60090
60095 RETURN
```

```
100 PRINT"[CLR]DISK MEMORY DISPLAY
105 PRINT"BY JIM BUTTERFIELD
110 DATA77,45,87,0,18,16,162,0,189
120 DATA157,64,6,232,224,16,208,245
125 DATA76,193,254
130 FORJ=1TO9:READX:C$=C$+CHR$(X):NEXT
140 FORJ=1TO11:READX:D$=D$+CHR$(X):NEXT
150 PRINT"[CD]THERE ARE TWO PROCESSORS:
160 PRINT"  1) THE IEEE PROCESSOR
170 PRINT"  2) THE DISK PROCESSOR
180 INPUT"WHICH DO YOU WANT TO PEEK (1/2)    [3CL]";D
190 OPEN1,8,15:P$=CHR$(4)+CHR$(16):R$=CHR$(224)
200 PRINT"INPUT MEMORY ADDRESS":PRINT"IN HEXADECIMAL:
210 PRINT"  [4CL]                                [CU]"
220 INPUTZ$
240 PRINT"[CU]";:IFLEN(Z$)<>4THEN220
250 FORJ=1TO4:Y=ASC(MID$(Z$,J))
260 IFY<58THENY=Y-48
270 IFY>64THENY=Y-55
280 IFY<0ORY>16THEN220
290 Y(J)=Y:NEXT:K=0:PRINT"[6CR]";
300 U=Y(3)*16+Y(4):V=Y(1)*16+Y(2)
310 IFD<>2THEN360
320 PRINT#1,C$;CHR$(U)+CHR$(V);D$
330 PRINT#1,"M-W";P$;CHR$(1);R$
340 PRINT#1,"M-R";P$:GET#1,X$
345 IFX$=R$THEN340
350 U=64:V=18
360 PRINT#1,"M-R";CHR$(U);CHR$(V)
370 GET#1,X$:IFX$=""THENX$=CHR$(0)
380 PRINT" ";:X=ASC(X$)/16
390 FORJ=1TO2:X%=X:X=(X-X%)*16
395 IFX%>9THENX%=X%+7
400 PRINTCHR$(X%+48);:NEXT
410 U=U+1:IFU=256THENU=0:V=V+1
420 K=K+1:IFK<8THEN360
430 Y(0)=0:Y(4)=Y(4)+8:J=4
440 IFY(J)<=15THEN450
441 Y(J)=Y(J)-16:J=J-1:Y(J)=Y(J)+1
442 GOTO440
450 PRINT:PRINT"  ";
455 FORJ=1TO4:Y=Y(J):IFY>9THENY=Y+7
460 PRINTCHR$(Y+48);:NEXT
465 PRINT"[CU]":GOTO220
480 REM IT MAY BE NECESSARY TO CHANGE LINE 120 TO RESET SEQUENCE (108,252,255
490 REM ON SOME DISKS. WAS ORIGINALLY (76,193,254)
```

```
10 OPEN 1,8,0,"$0:!£*%&"
12 FORI=1TO8:GET£1,X$:NEXT
14 GET£1,X$:IFX$<>""THEN14
20 GET£1,X$:GET£1,X$
30 GET£1,X$:X=LEN(X$):IFXTHENX=ASC(X$)
40 GET£1,Y$:Y=LEN(X$):IFYTHENY=ASC(Y$)
50 L=X+Y*256:PRINT"THERE ARE ";L;"BLOCKS FREE"
60 CLOSE1:END
```

```
100 PRINT"DISK CHECKER  - JIM BUTTERFIELD"
110 DIM A(255),C%(77,28),D%(1),N$(224),T%(224,1),S%(224,1),L%(224),R%(77)
120 D%(0)=58:D%(1)=42:Z$=CHR$(0)
130 DATA 1,17,20,24,19,30,17,35,16,0
140 DATA 65,17,20,24,18,30,17,35,16,0
150 DATA 67,39,28,53,26,64,24,77,22,0
160 B$=CHR$(17):INPUT"DRIVE#";D$:IFD$="$"THEND$="0":B$=CHR$(3)
170 IFD$<>"0"ANDD$<>"1"GOTO160
180 OPEN15,8,15,"I"+D$:GOSUB3020
190 OPEN3,8,3,"$"+D$:GOSUB3020
200 A0=1:GET#3,A$:A=ASC(A$+Z$)
210 READA1:IFA=A1GOTO250
220 F%=F%+1:IFF%=3GOTO290
230 READA1:IFA1=0GOTO210
240 GOTO230
250 READA1:IFA1=0GOTO270
260 READB1:FORJ=A0TOA1:R%(J)=B1:NEXTJ:A0=J:GOTO250
270 IFA=1ORA=65THEND1=1:T9=35:S9=3:D9=18
280 IFA=67THEND1=257:T9=77:S9=4:D9=39
290 IFT9=0THENCLOSE3:PRINT"?? DISK NOT RECOGNIZED ??":STOP
300 REM    GET AND PRINT BAM
310 PRINT"       FREE BLOCK MAP"
320 FORJ=1TOD1:GET#3,A$:NEXTJ
330 FORJ=1TOT9:T1=0
340 IFJ=51THENGET#3,A$,A$,A$,A$
350 GET#3,A$:C=ASC(A$+Z$)
360 PRINTRIGHT$(" "+STR$(J),2);" ";
370 K1=0:FORK=0TOS9-1:GET#3,A$:A=ASC(A$+Z$)
380 FORL=0TO7:A%=A/2:D1=A-A%*2:IFK1<=R%(J)THENC%(J,K1)=D1:PRINTCHR$(D%(D1));
390 A=A%:T1=T1+D1:K1=K1+1:NEXTL,K:PRINT
400 IFT1<>CTHENPRINT"?";
410 NEXTJ
420 REM    DO SPECIFIC JOB
430 PRINT:CLOSE3:PRINT "    CHOOSE --"
440 PRINT "1. CHECK ALL FILES"
450 PRINT "2. CHECK FOR BAD SPOTS"
460 PRINT "3. RECOVER SCRATCHED FILE"
470 PRINT "  YOUR CHOICE? ";
480 GETX$:IFX$=""GOTO480
490 X=ASC(X$)-48:IFX<1ORX>3GOTO480
500 PRINTX$:OPEN2,8,2,"#0":GOSUB3020
510 ONXGOTO520,790,890
520 REM     CHECK FILES
530 T=D9:S=1
540 GOSUB2000
550 FORD=2TO255STEP32:IFA(D)<128GOTO590
560 D3=D3+1:T%(D3,0)=A(D+1):S%(D3,0)=A(D+2):L%(D3)=A(D+28)+A(D+29)*256
570 IFA(D)=132THENT%(D3,1)=A(D+19):S%(D3,1)=A(D+20)
580 N$="":FORK=D+3TOD+18:N$=N$+CHR$(A(K)):NEXTK:N$(D3)=N$
590 NEXTD
600 T=A(0):S=A(1):IFT=D9GOTO540
610 FORD=1TOD3:L%=0
620 PRINTN$(D);
630 T=T%(D,0):S=S%(D,0)
640 IFT>T9ORS<0THENT=0
650 IFT<1ORS>R%(T)THENPRINT" BAD CHAIN":GOTO770
660 IFC%(T,S)=1THENPRINT" UNALLOCATED BLOCKS":GOTO770
670 IFC%(T,S)>1THENPRINT" CONFLICT ";N$(C%(T,S)-1):GOTO770
680 C%(T,S)=1+D
690 GOSUB3000
700 L%=L%+1
710 FORJ=0TO1:PRINT#15,"M-R";CHR$(J);B$:GET#15,A$
720 A(J)=ASC(A$+Z$):NEXTJ
730 T4=T:S4=S:T=A(0):S=A(1):IFT<>0ANDE=0GOTO640
740 T=T%(D,1):S=S%(D,1):T%(D,1)=0:IFT<>0GOTO640
750 IFL%<>L%(D)THENPRINT" INCORRECT BLOCK COUNT":GOTO770
760 PRINT:PRINT"[CU,23SPC,CU]"
770 NEXTD
780 PRINT:PRINTD3;"FILES":GOTO1270
790 REM    SCAN SECTORS
800 IFF%=0THENPRINT"SORRY .. CAN'T DO IT":GOTO1270
810 FORT=1TOT9:PRINT"TRACK";T
820 FORS=0TOR%(T)
830 PRINT"[CU,10CR] SECT";S
840 GOSUB3000
850 NEXTS
860 PRINT"[CU,20SPC,CU]"
870 NEXTT
880 PRINT"DISK OK":GOTO1270
890 REM    UNSCRATCH
900 K=0:PRINT"I WILL LOOK FOR DISCARDED FILES..."
910 T=D9:S=1
920 GOSUB2000
930 FORD=2TO255STEP32:IFA(D)<>0ORA(D+1)=0GOTO980
940 IFK=0THENPRINT"DO YOU WANT TO RECOVER:"
950 GETX$:FORK=D+3TOD+18:PRINTCHR$(A(K));:NEXTK:PRINT"? ";
960 GETX$:IFX$<>"Y"ANDX$<>"N"GOTO960
970 PRINTX$:IFX$="Y"GOTO1010
980 NEXTD
```

```
990 T=A(0):S=A(1):IFT=D9GOTO920
1000 PRINT"THAT'S ALL ":GOTO1270
1010 T6=T:S6=S:D6=D:T=A(D+1):S=A(D+2):L%(0)=A(D+28)+A(D+29)#256:L%=0
1020 GETX$:PRINT"IS THIS FILE:"
1030 PRINT" 1. SEQUENTIAL"
1040 PRINT" 2. PROGRAM"
1050 PRINT" 3. USR"
1060 IFA(D+19)=0GOTO1080
1070 PRINT" 4. RELATIVE"
1080 PRINT"  WHICH NUMBER? ";
1090 GETX$:IFX$=""GOTO1090
1100 X=ASC(X$)-48:IFX<1ORX>4GOTO1090
1110 PRINTX$:X=X+128
1120 IFX=132THENT%(0,1)=A(D+19):S%(0,1)=A(D+20):IFT%(0,1)=0GOTO1020
1130 IFT>T9ORS<0THENT=0
1140 IFT<1ORS>R%(T)THENPRINT" BAD CHAIN!":GOTO1260
1150 IFC%(T,S)=0THENPRINT" ALLOCATED BLOCKS!":GOTO1260
1160 GOSUB3000:L%=L%+1
1170 FORJ=0TO1:PRINT#15,"M-R";CHR$(J);B$:GET#15,A$
1180 A(J)=ASC(A$+Z$):NEXTJ
1190 T4=T:S4=S:T=A(0):S=A(1):IFT<>0GOTO1130
1200 T=T%(0,1):S=S%(0,1):T%(0,1)=0:IFT<>0GOTO1130
1210 IFL%<>L%(0)THENPRINT" INCORRECT BLOCK COUNT!":GOTO1260
1220 T=T6:S=S6:D=D6
1230 GOSUB3000
1240 PRINT#15,"M-W";CHR$(D);B$;CHR$(1);CHR$(X)
1250 PRINT#15,"U2:2,";D$;T;S:GOSUB3020:GOTO1300
1260 PRINT "SORRY - IT WON'T WORK"
1270 CLOSE2
1280 INPUT"** GOT TIME TO VERIFY/COLLECT DISK";X$
1290 IFASC(X$)=78THENEND
1300 CLOSE2:PRINT#15,"V":D$:END
2000 REM    GRAB FULL DISK BLOCK
2010 GOSUB3000
2020 FORJ=0TO255:PRINT#15,"M-R";CHR$(J);B$:GET#15,A$
2030 A(J)=ASC(A$+Z$):NEXTJ:RETURN
3000 REM    READ BLOCK
3010 PRINT#15,"B-R:2,";D$;T;S
3020 REM   GET ERROR STATUS
3030 INPUT#15,E,E$,E1,E2
3040 IFE<>0THENPRINT"[RVS]DISK ERROR:[OFF]"E;E$;E1;E2
3050 RETURN
```

Choosing a Printer

The main things that people want to do with printers are to a) list programs, b) list data, and c) use them as a link in the word processing chain.

If your only interest is the first two groups, then any dot matrix printer will do the trick, although it's doubtful whether many people will be satisfied with the one chosen by Commodore to be the official Commodore 64 printer.

It is slow, and extremely noisy, and the quality of the output is pretty bad as well.

Speed is an important factor, especially if you're going to be using the printer a lot, and one should really be looking for a print speed in excess of 120 characters a second.

Noise is almost equally important, because the work environment influences very highly the amount of work done in it, and a noisy printer blaring away all day will do little to encourage hard work.

Print quality needs at least to be legible, and preferably something more than that as well, although colour isn't particularly vital for most fields of work.

If you're going to be doing a lot of business work with it you'll obviously need a high quality daisy wheel printer, but most of these tend to be extremely slow, and can't double up as anything else.

A good bet would be a compromise printer, and the best one I've seen to date would be the Epson FX80 at a basic price of around 400 pounds. It prints at 160 characters a second in dot matrix mode, but a special double striking of the print head produces an almost passable daisy wheel quality output, at a cost of halving the print speed. Still much faster than most daisy wheel printers though.

It is also reasonably quiet!

The only problem with it is that it comes equipped with a Centronics interface, and of course the 64 has RS232 output, so somehow you've got to get from one to the other!

Whatever you get though, the following few programs should all work quite happily on your printer.

They are concerned with data manipulation between the computer and the printer, and are designed to be used as subroutines in a main program.

Screen Dumps to Printers

We've already seen one program for doing this in our chapter on machine code programming, but a simple Basic program to achieve the same purpose could go something like this :=

```
10 OPEN6,4,6:PRINT£6,CHR$(18):REM CLOSE LINE SPA
CING
11 REM THIS MIGHT NOT WORK ON YOUR PRINTER!
15 OPEN 4,4:CMD4
20 FORI=0TO24:FORJ=0TO39
30 A=PEEK(1024+I*40+J)
40 GOSUB60000
50 PRINTA$;B$;C$;
60 NEXT J:PRINT:NEXTI
70 PRINT£4:CLOSE4:END
```

```
60000 A$="":B$="":C$=""
60010 IFA>127THENA$=CHR$(18):C$=CHR$(146):A=A-128
:RETURN
60020 IFA<32THENA2$=CHR$(A+64):RETURN
60030 IFA>31ANDA<64THENB$=CHR$(A):RETURN
60040 IFA>63ANDA<96THENB$=CHR$(A+128):RETURN
60050 B$=CHR$(A+64):RETURN
```

Neither this nor the earlier machine code program will copy programmable graphics, sprites or high resolution graphics onto the printer, as each printer would require a totally different program.

Checking for Data Output

There comes a time in every program when what you've tabulated nicely for the screen wants to go to the printer, so you simply change all the PRINT statements to PRINT # statements to re-direct the output, OPEN a file to the printer, a quick CMD and voila!

A perfect mess is the usual result.

This is because the printer looks on TABs in a rather different way from the computer.

When using Tab to print data out onto the screen, the 64 looks at the current cursor position with the command POS(0).

If the Tab argument is less than the cursor's current position on the line, then the data is just printed in the spaces immediately following the last character printed.

However, if the Tab argument is equal to or greater than the current cursor position, the computer subtracts the result of the Pos from the Tab argument, prints the relevant number of cursor rights, and then prints the data.

All well and good, and you spend a long time developing nice looking screen layouts.

But, to the printer the cursor is usually in column zero, or in other words the left hand side of the paper, and so Tab acts like the SPC command does on the 64: it just prints that number of cursor rights and then the data.

There are a number of ways around this problem.

One is to define a string A$ = "[39CR " for a 40 column printout, or A$ = "[79CR " for the more usual 80 column printout.

When it comes to printing data, use the LEFT$ statement to select the relevant number of cursor rights, and then when you're switching from screen to printer you're not using Tab at all, and so everything works correctly.

Another way is to have what is often termed 'dynamic PRINT# statements'.

We've used the term device number on quite a few occasions now, but it may not have occurred to you that the screen, as far as the 64 is concerned, is just another device, and it in fact is device number 3.

So we could have something like this :=

```
10 OPEN 3,3,1
15 OPEN 4,4,0
20 PRINT£3+X,"ABCDEFGHIJKLMNOPQRST";
50 X=1-X:IFXTHEN 20
```

Line 50 just toggles X between 0 and 1, and in line 20 we print either to device number 3 (the screen) if X equals zero, and to device number 4 (the printer), if X equals 1.

Where we might require a carriage return on the printer, a simple line :=

```
30 IF X THEN PRINT£4,CHR$(13);
```

This method is only efficient if the data to be printed is contained within quotes. Otherwise you might just as well have two totally different PRINT# statements.

Of course, you could always use a screen dump program.

Formatting Numbers

Another common problem, when manipulating data, is to get in an acceptable format.

If we want to print to, say, three decimal places, and the answer to a particular problem is 501, then we have a problem and our layout gets affected.

On the other hand, if a number is very small, it gets printed in the 1E-03 format, another problem.

So the following program will round a number to three significant digits to preserve clarity when outputting numerical data.

```
1000 Q$=".000":LQ=LEN(Q$):LT=LOG(10)
1010 P=LQ-1:IFQ$=""THENP=0
1020 INPUT"TYPE NUMBER NOW ",A
1030 A1:INT(A*10 to the power of P+.5)/10 to the
power of P
1040 IFA1=0THEN 1120
1050 B1$=STR$(A1)
1060 B2$=STR$(A1*10 to the power of P)
1070 LG=INT(LOG(ABS(A1))/LT)
1080 IFABS(A1)=1THEN B$="1" + Q$
1090 IF ABS(A1)<1THEN B$=LEFT$(Q$,ABS(LG))+RIGHT$
(B2$,LEN(B2$)-1)
1090 IF ABS(A1)>1THEN B$=MID$(B1$,2,LG+1)+"."+RIG
HT$(B2$,P)
1100 IFABS(A)=1THEN B$="1"+Q$
1110 IFA1=0THEN B$=Q$
1120 IFA1<0THEN B$="-"+B$
1130 PRINT"[2CD]"A1,B$
1140 GOTO 1020
```

Thanks to L.D. Gardner for that one.

Looking at Listing

There was a slight debate over this program, and whether it ought to go with the disk drive section, or here with the programs for printers.

As it stands, it works purely for disk drive and computer, but every PRINT statement contained in the program could just as easily be changed to a PRINT# statement, so we put it here.

One other thing to watch for when using this with a printer. The program checks, using PEEK(211), for the end of a line, so this will have to be changed to a simple incrementing counter.

Or, conversely, print it out on the screen and to the printer at the same time, when PEEK(211) will still work.

It is intended to be used as a subroutine, within a main program, to

be called up when you want to check on the working of another program.

A neat little input routine that you know you've used before but can't get it right this time, or anything of that nature, where you want to view a program previously used but don't want to lose the one currently in memory.

Just run this subroutine, type in the name of the program you want to look at, and it is displayed up on the screen (or printer!), for you to check on the appropriate bit of syntax.

One caution: when the program reaches an over-long line, i.e. one that wraps round onto the third line of the screen, it just repeats the line number and prints out the rest of the line.

```
10 DIMA$(90):FORI=0TO90:READA$(I):NEXT
15 INPUT "PROGRAM FILENAME ";FI$
18 OPEN 15,8,15
20 OPEN 2,8,2,FI$+",P":GOSUB500:GET£2,A$,A$
25 SL=0:GET£2,A$,A$:IFA$=""THEN999
30 GET£2,A$,B$
35 N=ASC(A$+CHR$(0))+ASC(B$+CHR$(0))*256:PRINTN;
37 GET£2,A$:P=ASC(A$+CHR$(0)):IFP=0THENPRINT:GOTO2
5
40 IF(PEEK(212)<>0)OR(P<128)THENPRINTCHR$(P);:GOTO
50
45 PRINTA$(P-128);
50 IF(A$=":"ORA$=",")AND(PEEK(211)>65)THEN65
55 IFPEEK(211)>75THEN65
60 GOTO37
65 PRINT:PRINTN;:SL=SL+1:GOTO37
500 INPUT£15,EN$,EM$,ES$,ET$:IFEN$="00"THEN RETURN
502 PRINT"*** DISK ERROR *** ";EM$
999 CLOSE2:CLOSE15:END
1000 DATA END,FOR,NEXT,DATA,INPUT£,INPUT,DIM,READ,
LET,GOTO,RUN,IF,RESTORE,GOSUB
1010 DATA RETURN,REM,STOP,ON,WAIT,LOAD,SAVE,VERIFY
,DEF,POKE,PRINT£,PRINT,CONT
1020 DATA LIST,CLR,CMD,SYS,OPEN,CLOSE,GET,NEW,TAB(
,TO,FN,SPC(,THEN,NOT,STEP,+,-
1030 DATA*,/,^,AND,OR,>,=,<,SGN,INT,ABS,USR,FRE,PO
S,SQR,RND,LOG,EXP,COS,SIN
1040 DATA TAN,ATN,PEEK,LEN,STR$,VAL,ASC,CHR$,LEFT$
,RIGHT$,MID$,GO,CONCAT
1050 DATA DOPEN,DCLOSE,RECORD,HEADER,COLLECT,BACKU
P,COPY,APPEND,DSAVE,DLOAD
1060 DATA CATALOG,RENAME,SCRATCH,DIRECTORY
```

8

6500 Chips in General and the 6566 in Particular

The family of chips inside the Commodore 64 is a new generation of processors, based on the venerable old 6502 as used by Commodore Business Machines in the early PETs, and many other manufacturers as well: Apple, for instance.

The 6502 first appeared in 1976, with an internal architecture similar in many ways to Motorola's 6800 family of chips. However, the 6502 was not without various features of its own.

Principally, it is a fairly fast processor, although events since 1976 have certainly produced speedier models. At the time, it was a landmark in microprocessor history.

It gets its speed from a technique known as pipelining, which can be compared to a conveyor belt in the way that it handles information. Before the last piece of information is finished with, the next one is already rolling in, so that the effect is of a continuous stream of data, with no pauses to stop and think whilst waiting for the next piece.

One of the initial drawbacks of the 6502 seemed to be the very small number of internal registers: we encountered these in the chapter on machine code programming.

However, these registers can be loaded and changed so quickly that the small number becomes much less of a problem. Because of the simplicity of the instructions used to alter the state of these registers, load, store, and so on, the 6502 was a relatively easy processor to program with.

And so it is with the 6510, a chip that is all bar two bytes identical to the 6502.

This chip has an input/output port built into it, with 8 pins of this available to the programmer. Address 0000 is used as the direction register for the I/O port, and 0001 is the actual port itself.

Thus these two bottom bytes are no longer part of RAM, and they can be used to test and control some of the 64's external activities. Cassette motors, as we've seen, can be controlled from byte 0000.

Byte 0001 can also be used to bank switch in or out the various amounts of ROM which can be designated as user RAM if required, although this should never be done from Basic: you'll only crash the machine if you try it.

So, a similar chip to the old 6502, but with a couple of subtle differences.

We'll come to the 6510 in much greater detail in chapter 10.

The 6566 Video Interface Chip

We've already studied this in some detail in the chapter on sprites and high resolution graphics, but the time has come to get a lot more technical!

The diagrams on the facing page show how the 6566's internal architecture works.

You'll see that the first register mentioned is memory location 53248. For the rest of this discussion on the 6566 this will be referred to as register zero, and any reference to registers after this will refer to, for example, register 22.

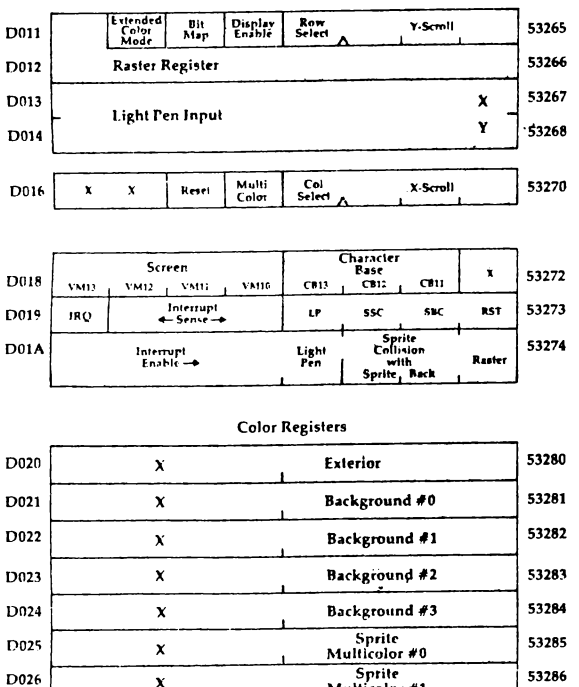
This will imply the 22nd register above this zero level, or in other words memory location 53248 + 22, or 53270.

When referring to various 'bits' in this register, e.g. the second bit, or the DEN bit, the reference can always be confirmed by looking at the above maps to find precisely where we mean.

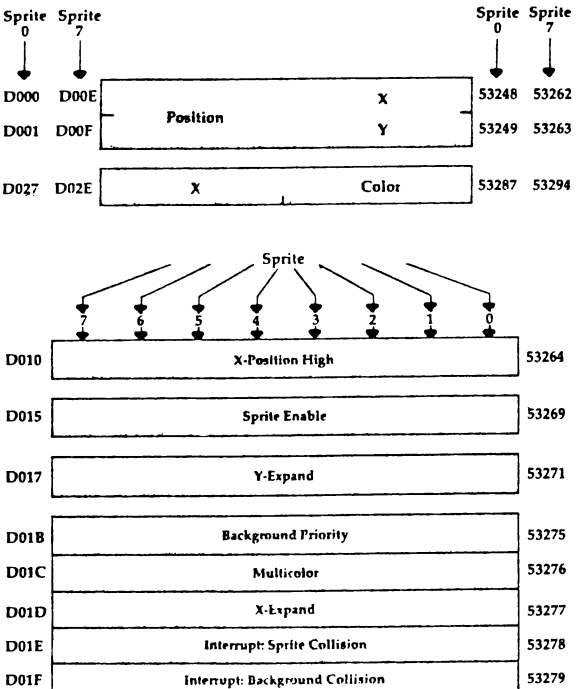
These memory locations, in common with anything else in the 64, can only address, or look at, 16K of memory at a time, and so we'll quite often want to switch memory configurations in and out as we go along.

So, let's take a look at 64 memory map control, before going on to individual chips in great detail.

6566 Video Chip
C64 Control and Miscellaneous Registers



6566 Video Chip
C64 Sprite Registers



Commodore 64 Memory Map Control

As we've seen, the 6510 can only look at 16K of memory at a time, although it is capable of addressing the full 64K, but not all at the same time. Since our 64K also consists of some 20K of ROM, and 4K of Input/Output, it is obviously necessary to bank switch some of this in and out for various different needs, thus giving us a combination of totally different memory map configurations.

Five control lines are provided to enable us to do this, three in the I/O port of the 6510 itself, and two in the plug-in cartridge slot, which also lets us see what type of cartridge has been inserted.

The three 6510 lines are as follows :=

LORAM (bit 0000). This is usually programmed to bank 8K of Basic ROM out of the processor's address space. If bit 0000 is programmed high, we're in normal Basic operation mode, and if programmed low Basic ROM disappears from the memory map, and is replaced with 8K of RAM from $A000 - $BFFF.

HIRAM (bit 0001). This is usually programmed to bank out the 8K of Kernal ROM out of the processor's address space. If bit 0001 is programmed high, we're in normal Basic operation mode, and if programmed low then the 8K of Kernal ROM disappears from the memory map, and is replaced with 8K of RAM from $E000 - $FFFF.

CHAREN (bit 0002). We've used this when designing our own characters, and this is its only purpose: to switch the 4K of character ROM into and out of the processor's address space. If bit 0002 is programmed high, we're in normal Basic operation mode, and the character ROM is not accessible. If programmed low, then the charcter ROM appears in the processor's address space, and we can't handle any I/O devices.

The two cartridge control lines are :=

GAME (pin 8). This was utilised with the fabled Ultimax game machine in mind, and was designed to be set low when a game cartridge was inserted, in which case the 64 memory map would be re-configured to resemble that of the Ultimax. However, it is now always held high, which means that either no cartridge is installed, or a non-Ultimax one is in use.

EXROM (pin 9). This is used to bank out the 8K of RAM from $8000 - $9FFF out of the processor's address space, and replace it with the 8K of ROM in the plug-in cartridge. This line is low if a Basic extension cartridge is installed (e.g. the always soon-to-be-released Simon's Basic), and high when either no cartridge or a non-expansion one is in use.

The following diagrams illustrate how these various memory configurations work : =

COMMODORE 64 FUNDAMENTAL MEMORY MAP

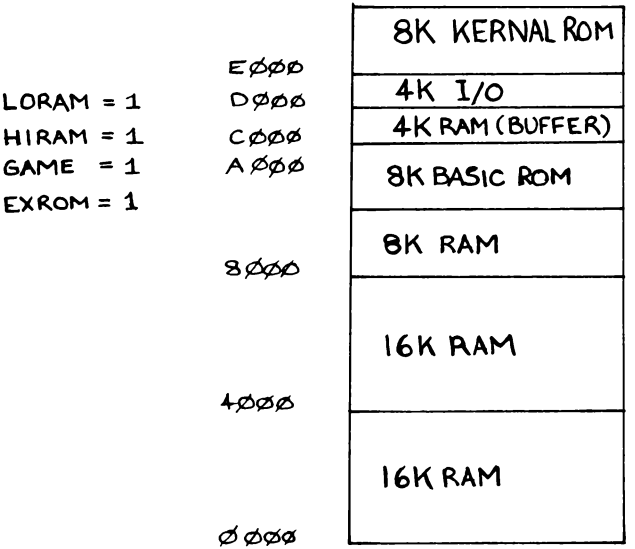
E ØØØ-FFFF	8K <	KERNAL ROM or RAM
DØØØ-DFFF	4K <	I/O or RAM or CHARACTER ROM
CØØØ-CFFF	4K <	RAM
AØØØ-BFFF	8K <	BASIC ROM or RAM or ROM PLUG-IN
8ØØØ- 9FFF	8K <	RAM or ROM PLUG-IN
4ØØØ- 7FFF	16K <	RAM
ØØØØ-3FFF	16K <	RAM

I/O BREAKDOWN

DØØØ-D3FF	VIC (Video Controller)	1K Bytes
D4ØØ-D7FF	SID (Sound Sythesizer)	1K Bytes
D8ØØ-DBFF	COLOUR RAM	1K Nybbles
DCØØ-DCFF	CIA1 (Keyboard)	256 Bytes
DDØØ-DDFF	CIA2 (Serial Bus, User Port/RS-232)	256 Bytes
DEØØ-DEFF	Open I/O slot #1 (CP/M Enable)	256 Bytes
DFØØ-DFFF	Open I/O slot #2 (Cheap Disk)	256 Bytes

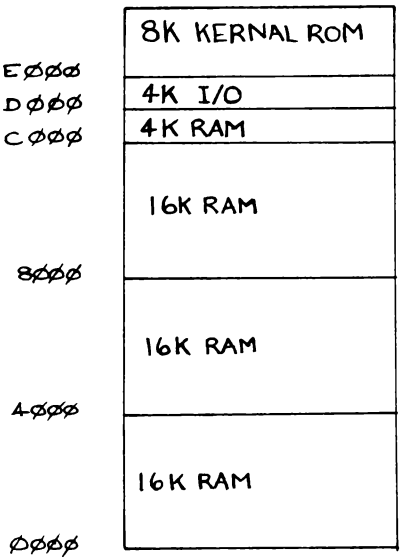
The following tables list the various memory configurations available on the Commodore 64, the states of the control lines which select each memory map, and the intended use of each map.

X = don't care, Ø = low, 1 = high.



This is the default BASIC memory map, which provides BASIC 2.0 and 38K continuous Bytes of user RAM.

LORAM = Ø
HIRAM = 1
GAME = 1
EXROM = X

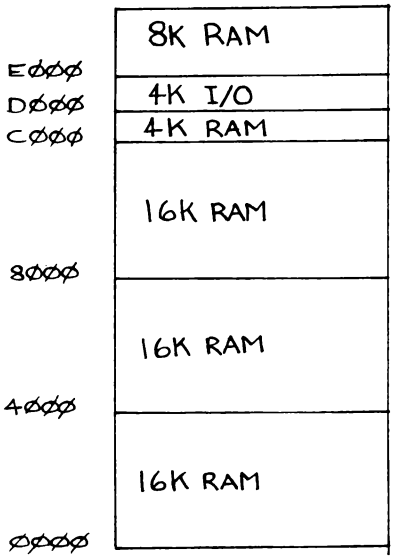


This map is intended for use with soft-load Languages (including CP/M), providing 52K continuous Bytes of user RAM, I/O devices and I/O driver routines.

LORAM = 1
HIRAM = Ø
GAME = 1
EXROM = X

or

LORAM = 1
HIRAM = Ø
GAME = Ø *
EXROM = Ø



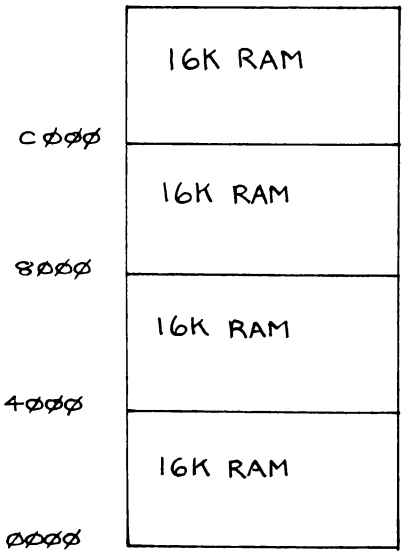
This map provides 60K Bytes of RAM and I/O devices. The user must write his own I/O driver routines.

* The character ROM is not accessible by the CPU in this map.

LORAM = Ø
HIRAM = Ø
GAME = 1
EXROM = X

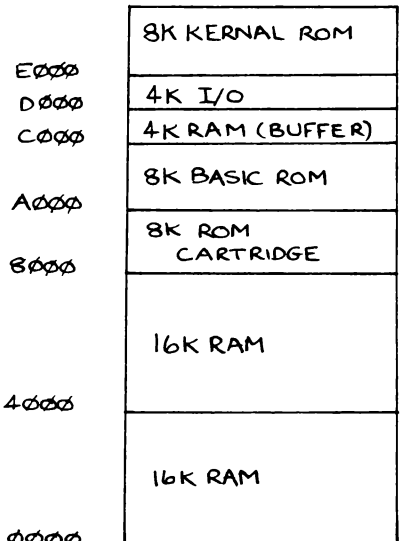
or

LORAM = Ø
HIRAM = Ø
GAME = X
EXROM = Ø

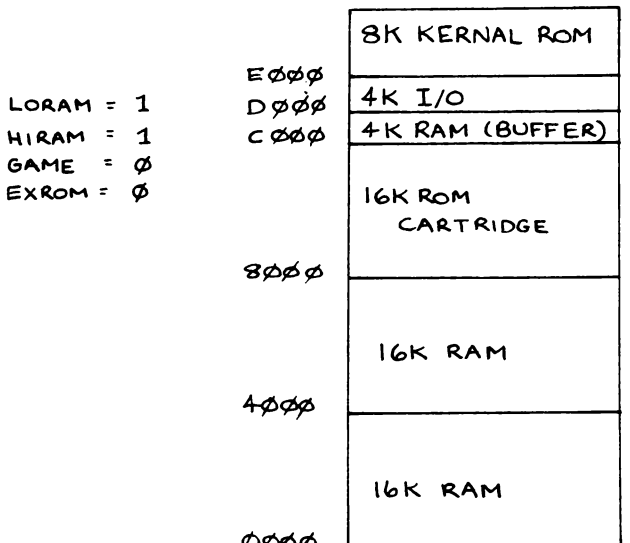


This map gives access to all 64K Bytes of RAM. The I/O devices must be banked back into the processor's address space for any I/O operation.

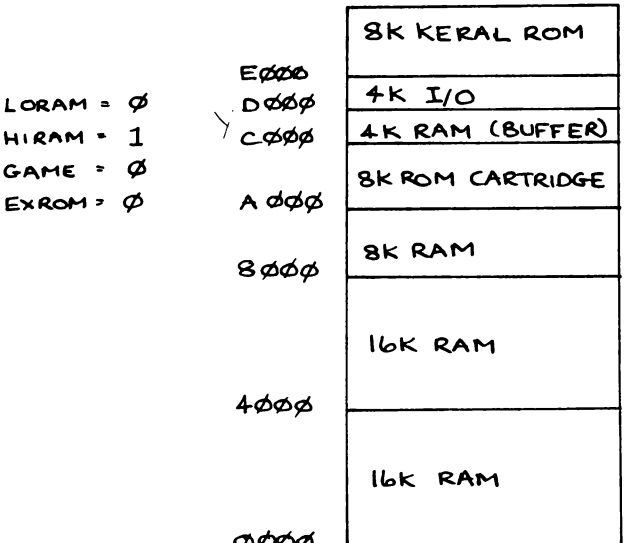
LORAM = 1
HIRAM = 1
GAME = 1
EXROM = Ø



This is the standard configuration for a BASIC system with a BASIC expansion ROM. This map provides 32K continuous Bytes of user RAM and upto 8K Bytes of BASIC "Enhancement".



This map provides 32K continuous Bytes of user RAM and up to 16K Bytes of plug-in ROM for special ROM-based applications which don't require BASIC (word processors, other languages, etc.).



This map provides 40K continuous Bytes of user RAM and up to 8K Bytes of plug-in ROM for special ROM-based applications which don't require BASIC.

6566 Video Interface Chip

The 6566 is a multi-purpose colour video controller device, capable of being used in quality arcade game terminals, which we have control over in the Commodore 64. It has 47 control registers, which are accessed by any 6502 compatible 8 bit microprocessor, in this case the 6510. It can address up to 16K of memory, like all 8 bit devices, and in this chapter we'll be taking a look at the various operating modes and options.

Character Display Mode

In this particular mode, the 6566 fetches character pointers from the video matrix area of memory, and translates that into character dot addresses in the 2K character base of memory. The video matrix consists of 1,000 locations in memory, each containing an 8 bit character pointer.

The location of this video matrix in memory is defined by VM13-VM10 in register 24 ($19), which are used as the four most significant bytes of the video matrix address. The lower order 10 bits are provided by an internal counter, in VC3-VC0, which steps through each of the 1000 character locations.

The 6566 has, in total, some 14 address outputs, as follows :=

Character Pointer Address

A13 A12 A11 A10 A09 A08 ..... A00

VM13 VM12 VM11 VM10 VC9 VC8 ..... VC0

The 8 bit character pointer permits up to 256 character definitions to be available simultaneously. In other words, under normal operating conditions, we are capable of displaying up to 256 different characters on the screen at once.

Each character consists of an 8 by 8 bit dot matrix, and is stored in the character base as eight consecutive bytes.

The location of this character base is defined by CB13-CB10, which

are also installed in register 24. These are used for the three most significant bits of the character base address.

The 11 lower order addresses are formed by the 8 bit character pointer from the video matrix (D7-D0), which selects a particular character, and a 3 bit raster counter (RC2-RC0), which selects one of the eight character bytes.

The resulting characters are formatted onto the screen in 40 columns of 25 rows: a total of 1000 screen locations, as we saw above.

In addition to this 8 bit character pointer, there is a 4 bit colour nibble (a nibble is half a byte, and on the 64 this is equal to 4 bits : someone must have had fun thinking up names like byte, nibble, and so on. I'm suprised a bit isn't called a munch or something!) associated with each video matrix location. The video matrix must be 12 bits wide.

The colour nibble defines one of the 16 character colours for each character.

The character data address table looks like this : =

Character Data Address

A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
CB13	CB12	CB11	D7	D6	D5	D4	D3	D2	D1	D0	RC2	RC1	RC0

Standard Character Mode

In standard character mode, the 8 sequential bytes from the character base are displayed directly on the 8 lines in each character space.

A '0' bit causes the background colour #0 (from register 33, or $21) to be displayed, while the colour selected by the colour nibble, known as the foreground colour, is displayed for a bit that is set to a '1' (see table of colour codes in the section on colour).

In other words, each character has a unique colour determined by the 4 bit colour nibble, and all characters must share the same background colour.

One of our earlier programs illustrated this, when displaying a collection of randomly coloured spaces on the screen.

To illustrate the point further, it would be possible to change the reverse space printed to be a reverse letter of the alphabet, or indeed any one of the characters available.

Function	Character Bit	Colour Displayed
Background	0	Background Colour #0
Foreground	1	Colour chosen by colour nibble

Multi-Colour Character Mode

Multi-colour mode gives us a much greater flexibility in choosing our displays, as it allows up to four different colours to be printed within each character space, but our character resolution is now halved in the horizontal direction.

The multi-colour mode is selected the MCM bit in register 23, or $16, to a '1', which causes all the dot data stored in the character base to be interpreted totally differently.

If the most significant bit of the colour nibble is a '0', then the character will be displayed as described above, in standard character mode. When it is set to a '0', it's displayed in multi-colour mode as displayed at the bottom of this page.

Thus the two different modes can be displayed on the same screen, but it is not possible to use any of the colours other than the first 8.

Function	Character Bit	Colour Displayed
Background	00	Background Colour #0-register 33
Background	01	Background Colour #1-register 34
Foreground	10	Background Colour #2-register 35
Foreground	11	Colour from 3 LSB of colour nibble

Since we now require two bits to specify the colour of a dot, each character space is reduced to a 4 pixel by 8 pixel, with each horizontal pixel twice as wide as in standard mode. Still, we can now have two background and two foreground colours per character space.

Extended Colour Mode

Extended colour mode allows the selection of background colours for each character space within the normal 8 pixel by 8 pixel resolution. Thus greater flexibility of colour choice is given than in standard character mode, without the loss of resolution given by multi-colour mode.

However, extended colour mode and multi-colour mode cannot be used at the same time.

This mode is selected by setting the ECM bit of register 17, or $11, to a '1'. The character dot data is displayed exactly as in standard character mode, in that the foreground colour as determined by the colour nibble is displayed for every character bit set to a '1', but the two most significant bits of the character pointer are used to select the background colour for each character space as follows : =

Character Pointer MSB Pair	Background Colour Displayed for a '0' bit
00	Background Colour #0-register 33
01	Background Colour #1-register 34
10	Background Colour #2-register 35
11	Background Colour #3-register 36

Since the two most significant bits of the character pointers are used for colour definition, this means that only 64 characters can be displayed on the screen in extended colour mode. As the 6566 forces CB10 and CB9 to be a '0' regardless of anything else going on, we are limited still further to the first 64 characters. However, these can be either the first 64 characters of ROM, or your own character set.

This mode allows us to choose any one of 16 foreground colours, and one of four background colours, for each character space.

Bit Map Mode

The most powerful of graphical modes on the 64, it is also the slowest to operate, and is usually handled from machine code rather than the much slower Basic.

Needless to say, in this mode everything is handled totally differently from all the other modes, and it works in the following way : =

In bit map mode the 6566 fetches data from an 8000 byte block of memory, and displays it on the screen in a one-to-one ratio. In other words, each bit of those 8000 bytes relates exactly to a bit as it appears on the screen.

This gives us a maximum resolution of 320 pixels by 200 pixels, or 64000 pixels, or bits. Each bit being one-eighth of a byte, this gives us our (64000/8) 8000 bytes of memory required.

Bit map mode is selected by setting the BMM bit in register 17, or $11, to a '1'.

The video matrix is still accessed as before, but the data contained there is no longer interpreted as character pointers, but instead it is read as colour data.

The video matrix counter is then also used as an address to fetch the dot data for display from the 8000 byte display base.

The display base address is made up like this : =

```
A13  A12 A11 A10 A09 A08 A07 A06 A05 A04 A03 A02 A01 A00
-------------------------------------------------------
CB13 VC9 VC8 VC7 VC6 VC5 VC4 VC3 VC2 VC1 VC0 RC2 RC1 RC0
```

Where VC denotes the video matrix counter output, RC the 3 bit raster line counter, and CB comes from register 24. The video counter goes through the same 40 locations for eight raster lines, going onto the next 40 locations every 8 lines, while the raster counter is incremented for each horizontal line on the screen (otherwise known as a raster line).

Because of this, each block of 8 sequential memory locations in the 8000 byte base are formatted as an 8 pixel by 8 pixel block on the screen. The first byte is the top line, the second byte the second line, and so on.

Standard Bit Map Mode

When standard bit map mode is initialised, the colour information comes only from the data stored in the video matrix, and anything that the colour nibble tries to do is ignored.

The 8 bits are divided into two 4 bit nibbles, which allows two colours to be independently selected for each 8 pixel by 8 pixel block. When a bit in the display memory is set to a '0' the colour of the output dot is set by the least significant, or lower, nibble, and when it's set to a '1' the colour is selected by the most significant bit, or highest nibble. Thus : =

Bit	Colour Displayed
0	Lower nibble of video matrix pointer
1	Higher nibble of video matrix pointer

Multi-Colour Bit Map Mode

Multi-colour bit map mode is selected by setting the MCM bit in register 22, or $16, to a '1', in addition to setting the BMM bit in register 17.

This mode uses the same 8000 byte block of memory to display its characters, but this time the data is handled in a totally different way, like this : =

Bit Pair	Display Colour
00	Background Colour #0-register 33
01	Higher nibble of video matrix pointer
10	Lower nibble of video matric pointer
11	Video matrix colour nibble.

This time we are using the colour nibble, along with the video matrix pointer and the background colour #0 from register 33. Thus we can have three separately chosen colours in each 8 pixel by 8 pixel block, along with one standard background colour.

However, due to the bit pairing to get the colour information, each horizontal pixel is twice as wide as it was, and so our maximum resolution is halved to become 160 dots by 200.

Sprites

A sprite, sometimes known as a MOB (for Moveable Object Block), is a special type of graphical character that can be displayed anywhere on the screen without all the usual restraints imposed upon us when manipulating simple bit patterns.

Since it somehow seems better to talk about manipulating sprites, rather than manipulating MOBs, which sounds rather like police crowd control at a football ground on a Saturday afternoon, we'll stick to the term sprites.

Up to 8 totally different sprites can be displayed on the screen at the same time, with each sprite taking a total of 63 bytes of data to define it. These bytes are arranged as a 24 pixel by 21 pixel block, as follows: =

Byte	Byte	Byte
00	01	02
03	04	05
06	07	08
.	.	.
.	.	.
.	.	.
.	.	.
60	61	62

Why 63 bytes ? Well, 24 pixels across is equal to 3 bytes, and 21 lines of these things gives us 21*3 or 63 bytes.

These blocks of 63 bytes are stored all over the place in the 64's memory, and a look at the memory map in Appendix A will tell you where the first 13 locations are. You can store them anywhere you like (within limits!), but it makes sense when getting started to use the areas that the 64 has already set aside.

The 8 bytes from location 2040 onward contain information which tells the 6566 where the data for each sprite is stored. Thus the value in location 2040 relates to sprite 1, in 2041 to sprite 2, and so on.

If PEEK(2040) returns a 13, this tells us that the data for sprite 1 is to be found in the 63 bytes starting at the 13th data block. A look at the memory map tells us that this starts in memory at location 832.

We've already seen how sprites can be displayed and formed, now let's get a little more technical and go into further detail on all the wonderful things we can do with a sprite!

Sprites On and Off

Enabling a Sprite

Each sprite can be turned on or off independently of any other sprites that happen to be around, and they are turned on by selecting the corresponding enable bit in register 21 to be a '1'. If this bit is set to be '0', nothing will happen to that particular sprite.

The SID chip memory map was explained in some detail in the chapter on sprites and high resolution graphics.

It sounds wonderful to talk about disabliing a sprite, shades of Norman Hunter, but as we've seen this is just done by setting the appropriate bit in register 21 to a zero.

Positioning a Sprite

Each sprite is positioned according to X and Y co-ordinates, on a 512 by 256 scale respectively. However, not all of these locations can be seen on the screen, which does allow you smooth scrolling off the screen in both horizontal and vertical directions.

For practical purposes, the display should be confined to a vertical scale of 50 to 250, and a horizontal scale of 25 to 350.

The number put into the X and Y co-ordinate registers determines where the sprite will appear on the screen. We gave the memory map for these registers in the section on sprites and high resolution graphics.

One other register must be considered, register 16, or $10. This must be set to a '1' if the X position exceeds 255, and back to a '0' again when the operation is finished. This allows a sprite to travel across the full width of the screen, without coming to an abrupt halt near the right hand edge.

Colouring a Sprite

Each sprite has a separate 4 bit register to determine its colour, and as usual there are two different colour modes : =

Standard Colour Mode

In this particular mode, a '0' bit of sprite data allows the background colour to show through, and this is referred to as being 'transparent'.

If the bit is set to a '1', then the sprite colour is shown as dictated by the corresponding sprite colour register: see earlier memory map.

Multi-Colour Mode

Each sprite, regardless of its neighbours, can be selected as a multi-colour sprite by setting the MCS bits in the sprite multi-colour register 29, or $1C.

When this bit is set to a '1', then the sprite will be displayed as a multi-colour sprite, with the colour coming in as follows : =

Bit Pair	Colour Displayed
00	Transparent
01	Sprite Multi-colour #0-register 37
10	Sprite Colour-registers 39 through 46
11	Sprite Multi-colour #1-register 38

As we now require two bits to define the colour of each sprite, that sprite now becomes a 12 pixel by 21 pixel character, since each horizontal pixel is now twice as wide as it was in standard mode.

It does give us control over three colours per sprite, plus one background colour, but the multi-colours must be the same for all sprites in multi-colour mode.

Magnifying Sprites

Sprites can be doubled in size either horizontally or vertically, or both, and reduced back to normal size again.

Two registers control sprite expansion, and if the relevant sprite bit is set to a '1', the sprite is expanded, and if set to a '0' it goes back to normal again.

Register	Function
29 ($1D)	Expand horizontally
23 ($17)	Expand vertically.

Despite expanding the sprite, we don't get any increase in resolution, as the same 24 pixel by 21 pixel grid is displayed, but expanded in the appropriate direction.

Thus if we're using multi-coloured expanded sprites, one pixel can be four times its normal horizontal resolution.

Priority amongst Sprites

Nothing to do with which one is 'boss', but instead this determines which sprites has priority over what i.e., if sprites pass over each other, or over anything else that happens to be on the screen, register 27 ($1B) determines what will be displayed.

This can be individually selected for each sprite, by setting the appropriate bit in register 21 to be a '0' or a '1', and functions like this : =

Register Bit	Function
0	Non-transparent part of sprite will be displayed
1	Non-transparent part displayed only instead of background colour #0 or multi-colour bit pair 01

Sprites also have a fixed priority amongst themselves, in that sprite number 0 will always be displayed over sprite number 1, 1 over sprite number 2, and so on.

Sprite to sprite priority is always sorted out before sprite to data-on-screen priority.

Sprites Colliding

It is possible to detect two types of sprite collision, sprite to sprite and sprite to anything else on the screen.

A collision between two sprites is said to occur when two non-transparent parts of each sprite want to occupy the same screen area.

When this happens, the appropriate bits in the sprite collision register, register 30 or $1E, are set to '1' for both sprites.

As more sprites collide, the appropriate bits in register 31 continue to be set, until a read of this collision register, when all the bits are set back to '0' again.

Sprites can even collide off-screen, so watch out!

The second type of collision is between a sprite and anything else on the screen.

When this occurs the appropriate bit for the sprite concerned is set in register 31, or $1F, to '1', although transparent data does not generate a setting of this register.

The display data from a 01 multi-colour bit pair also does not generate a setting to '1' of this register.

Again, collisions can take place off-screen, and the register is cleared back to zeros again the next time it is read.

Accessing Sprites in Memory

We've touched on this one already, but for the sake of complete clarity here we go again.

The data for each sprite is stored in 63 consecutive memory locations, and the position of these can be found from the memory maps in Appendix A.

Naturally we have to tell the 6566 where the data for each sprite is stored, and this is done using memory locations 2040 to 2047, the 8 bytes immediately after the screen RAM, and each byte refers to one sprite.

Thus location 2040 refers to sprite 0, 2041 to sprite 1, and so on, up to 2047 which refers to sprite 7.

If a value of 13 is stored in location 2042, it means that the data for sprite 2 is to be found at the 63 memory locations starting at the 13th

block of sprite data, which happens to be memory location 832.

The eight-bit sprite pointer, together with the six bits from the sprite byte counter (to address 63 bytes), define the entire 14-bit address field, like this :=

```
A13 A12 A11 A10 A09 A08 A07 A06 A05 A04 A03 A02 A01 A00
---------------------------------------------------------
SP7 SP6 SP5 SP4 SP3 SP2 SP1 SP0 SC5 SC4 SC3 SC2 SC1 SC0
```

Where SP are the sprite pointer bits, and the SCs are the internally generated sprite counter bits. The sprite pointers are read from the video matrix at the end of every raster line.

When the Y position register of a sprite matches the current raster line count, then the sprite data is fetched, with internal counters stepping through the 63 bytes of data, displaying 3 bytes on each raster line.

Other Screen Features

As well as all these graphical and sprite features, the 6566 is capable of much more, as we shall see.

Screen Blanking

The display can be blanked off by setting the DEN bit of register 17, or $11, to a zero. POKE 53265,11 achieves this.

When we blank the display area, the entire screen is filled with the exterior colour as set in register 32, or $20. This allows us to perform full processor utilization of the system bus, or in other words access things like cassette decks, Vic disk drives, and so on.

Sprites, however, unless specifically disabled, continue to shine through.

To get the screen back, the DEN bit must be set to a '1' again, and POKE 53265,27 achieves this.

Selecting Rows and Columns

As we've seen, we normally get a 40 column by 25 row screen, but for some purposes it would be desirable to change this. For instance, to enable smooth scrolling of the screen.

This is achieved by altering the RSEL bit in register 17, or $11, and the CSEL bit in register 22, or $16, and works like this :=

RSEL	Number of Rows	CSEL	Number of Columns
0	24	0	38
1	25	1	40

This effectively moves the border over onto the screen area, but leaves the characters previously displayed there still intact, but no longer visible.

As an example, POKE 53265,19, loses the top half of the top line of the screen, and the bottom half of the bottom line of the screen. POKE 53265 gets us back to normal again.

Scrolling the Screen

The entire screen display can be scrolled either horizontally or vertically, one pixel at a time, up to a maximum of one character space. Using this in conjunction with the screen window (screen disply minus border) facilities mentioned on the last page, enables us to produce smooth scrolling of the display area, whilst updating the system memory only when a new character row or column is required.

This method is also used to centre a fixed display within the screen window.

Bits	Register	Function
X2,X1,X0	22 ($16)	Horizontal Position
Y2,Y1,Y0	17 ($11)	Vertical Position

Light Pens

The light pen input stores the current screen position in two registers, labelled LPX and LPY.

The X position is stored in register 19, or $13, and will contain the 8 most significant bytes of the X position at the time of incident.

As the X position is defined by a 512 (9 bit) state counter, resolution to two pixels is provided.

The Y position, stored in register 20, or $14, allows us a single raster resolution on the screen display, or down to one pixel.

This light pen input may be triggered only once per frame, or screen scan, and subsequent triggers with that frame will have no effect.

Raster Register

Nothing to do with people wearing red, green and yellow hats, this is a dual function register.

Reading the raster register 18, or $12, returns the lower 8 bits of the current raster position. The higher 8 bits are stored in register 17, or $11.

The visible display window is from raster 51 to raster 251, or $033 to $0FB.

A write to the raster bits, including RC3, is stored for use in an internal raster compare, and when the current raster matches this written value, the raster interrupt latch is set.

Interrupt Registers

The interrupt register shows the status of the four sources of interrupt, which are : =

Latch Bit	Enable Bit	When Set
IRST	ERST	Raster Count = Stored Raster Count
ISDC	ESDC	Sprite collision with data on screen
ISSC	ESSC	Sprite collision with another sprite
ILP	ELP	Negative transition of LP input (once per frame)
IRQ		

In order for the interrupt request to set the IRQ output to '0', the

corresponding interrupt enable bit in register 26, or $1A, must be set to a '1'.

Again, once an interrupt latch has been set, it may only be cleared by writing a '1' to the appropriate latch in the interrupt register.

Dynamic Screen Refresh

Five 8 bit row addresses are refreshed every raster line, and this guarantees a maximum delay of 2.02 ms between the refresh of any single row address in a 129 address system, or 3.66 ms in a 256 address system.

Reset

The reset bit RES in register 22, or $16, is not used in the normal mode of operation.

Thus it is normally set low, and setting it high suspends the entire operation of the 6566!

To be used with caution.

Theory of Operation

The 6566 interacts with everything else on board the 64 in a special way.

The 6510 requires access to and from the system buses only during that portion of its cycle known as phase 2, when the clock is set high.

The 6566 takes advantage of this system, and therefore only accesses memory during phase 1, or when the clock is set low.

Therefore, such operations as getting character data, or refreshing the screen, or anything else that the 6566 handles are totally transparent to the 6510, and thus don't reduce the speed of processor operation. The 6566 itself provides the various interface control signals necessary to perform and maintain this kind of bus sharing.

The 6566 also provides the signal to enable address control, used to disable the 6510 address bus drivers, thus allowing the 6566 to access the address bus for itself. Address Enable Control is active (set low)

during phase 1 of the clock cycle, so that again the 6510 is not affected in its speed of operation.

However, because of all this all memory accesses must be completed in at most half a cycle, or 500ns, as the 6566 provides a 1MHz clock.

This could become a problem, since some of the operations of the 6566 require much longer (relatively) than a mere half a cycle. In particular, sprite generation requires that the 6566 also grabs a slice of the action during phase 2, which means that the 6510 must itself be disabled somehow.

This is achieved with the BA, or Bus Active, signal, which is connected to the ROY input of the 6510.

This is normally set high, but can be set low to indicate that the 6566 wants to do some processing during phase 2. In all, the 6566 has three phase 2 times after BA has been set low in order to complete all its data access.

On the fourth phase 2 after BA being set low, the Address Enable Control remains low until the 6566 has finished.

More manipulation must take place during the fetching of the character pointers, which takes some 40 consecutive phase 2 accesses to fetch the video matrix pointers.

Sprites, as we've said, also require more than one phase 2 access, and in fact require four accesses in total, as follows :=

Phase	Data	Condition
1	Sprite Pointer	Every Raster Scan
2	Sprite Byte 1	Each raster whilst sprite is displayed.
1	Sprite Byte 2	As above
2	Sprite Byte 3	As above

Thus sprite pointers are fetched every other phase 1 at the end of every raster line.

All this bus control is handled internally by the 6566 itself.

Memory Interfacing

The 6566 has thirteen fully decoded addresses for direct connection to the system address bus, and can be accessed in the same way as any other peripheral device.

The following 6510 interface signals are provided : =

Data Bus DB7 - DB0

These 8 data bus pins combine to form the bi-directional data port, which can only be accessed while the Address Enable Control and Phase 0 are high, and chip select is low.

Chip Select CS

This is brought low to enable access to the device registers in conjunction with the address and Read Write pins. It is only recognised as being low when Address Enable Control and Phase 0 are high.

Read Write R/W

This is used to determine the direction of data transfer on the data bus, in conjunction with CS. When R/W is high, data is transferred from the selected register to the data bus output, and when it is low data presented on the data pins is loaded into the chosen register.

Address Bus A05-A00

These lower six pins are bi-directional, and are used as inputs during a processor read or write to the video device. The data on the address inputs selects the register for read or write as defined in the register map.

Clock Out PH0

The clock output, or phase 0, is the 1 MHz clock used as the 6510 processor phase 0 in. All system bus activity is referenced to this clock, the frequency of which is generated by dividing the 8MHz video input clock by 8.

Interrupts IRQ

The interrupt output is brought low when an enabled source of interrupt occurs within the device. It requires an external pull-up register.

Video Interface

The output signal from the 6566 consists of two signals, which require mixing together.

SYNC/LUM contains all the video data, and requires an external pull-up of 500 ohms.

The Colour output, containing all the colour information for screen display, is terminated with 1,000 ohms to ground.

These two signals are then mixed before being fed through to your television set.

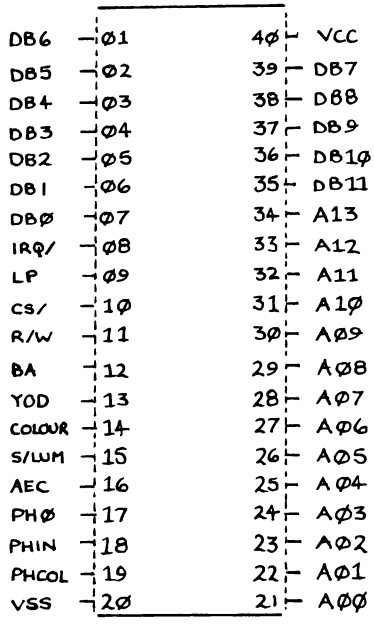
The following set of diagrams round off our look at the 6566 : =

AEC	PHØ	CS/	R/W	ACTION
Ø	Ø	X	X	Phase 1 Fetch, Refresh
Ø	1	X	X	Phase 2 Fetch (Processor Off)
1	Ø	X	X	No Action
1	1	Ø	Ø	Write to Selected Register
1	1	Ø	1	Read from Selected Register
1	1	1	X	No Action

6566/6567 Bus Activity

Colour Codes

D4	D3	D1	DØ	HEX	COLOUR
Ø	Ø	Ø	Ø	Ø	BLACK
Ø	Ø	Ø	1	1	WHITE
Ø	Ø	1	Ø	2	RED
Ø	Ø	1	1	3	CYAN
Ø	1	Ø	Ø	4	PURPLE
Ø	1	Ø	1	5	GREEN
Ø	1	1	Ø	6	BLUE
Ø	1	1	1	7	YELLOW
Ø	Ø	Ø	Ø	8	ORANGE
1	Ø	Ø	1	9	BROWN
1	Ø	1	Ø	A	LT. RED
1	Ø	1	1	B	DARK GREY
1	1	Ø	Ø	C	MED. GREY
1	1	Ø	1	D	LT. GREEN
1	1	1	Ø	E	LT. BLUE
1	1	1	1	F	LT. GREY



6566 PINOUT DIAGRAM

9

The 6581 Sound Interface Device

The 6581 is one of the most versatile and powerful of the musical chips currently available in any home computer, and can even compete with full blown musical synthesisers costing many times the price, as the following specifications show : =

3 tone oscillators, or voices, each with a range of 0 to 4 KHz

4 waveforms per voice, covering triangle, sawtooth, variable pulse and noise.

3 amplitude modulators, with a range of 48 decibels.

3 envelope generators, featuring exponential response, an attack rate in the range 2ms to 8 seconds, decay rate in the range 6ms to 24s, sustain level from 0 up to peak volume, and a release rate from 6 ms to 24s.

Oscillator synchronization.

Ring modulation.

Programmable filtering, featuring a cut off range of 30 Hz to 12 KHz, a 12 decibel octave roll off, low pass, high pass, band pass and notch outputs, and a variable resonance.

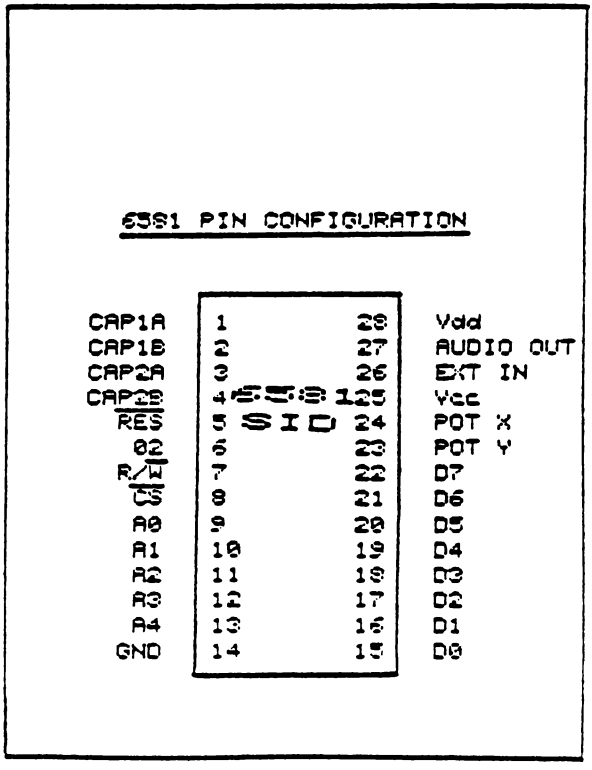
A master volume control (the one failing of this chip! It should have had separate volume controls)

2 A to D POT interfaces
A random number modulation generator, and

External audio output, allowing you to link the 6581 up to an external speaker perhaps, or play it through a guitar and then into a speaker, or even link up a couple of 6581s together.

6581 Pin Configuration

The 6581 pin configuration looks like this : =



More General Description

Here we'll outline in more human terms some of the capabilities of the 6581, and introduce a few terms that will be useful throughout the rest of this chapter.

The 6581 consists of three synthesiser voices, which can be used either independently or in conjunction with one another, in order to create some amazing sounds.

Each voice consists of a tone oscillator and waveform generator, an envelope generator and an amplitude modulator.

The tone oscillator controls the pitch of each voice, whilst each voice can also choose from one of four different waveforms at the tone, or pitch, selected, with complicated harmonic structure possible with each waveform.

The amplitude modulator, in conjunction with the envelope generator, produces the overall quality of the noise heard, and can be used to re-create the sounds of many musical instruments.

A programmable filter is also provided to enable complex tone colours to be produced: true synthesis on a home computer!

The 6581 allows the 6510 to read the changing output of the third oscillator and third envelope generator.

This can be used to create a variety of effects, including vibrato and frequency/filter sweeps. We've covered a few simple examples of this earlier.

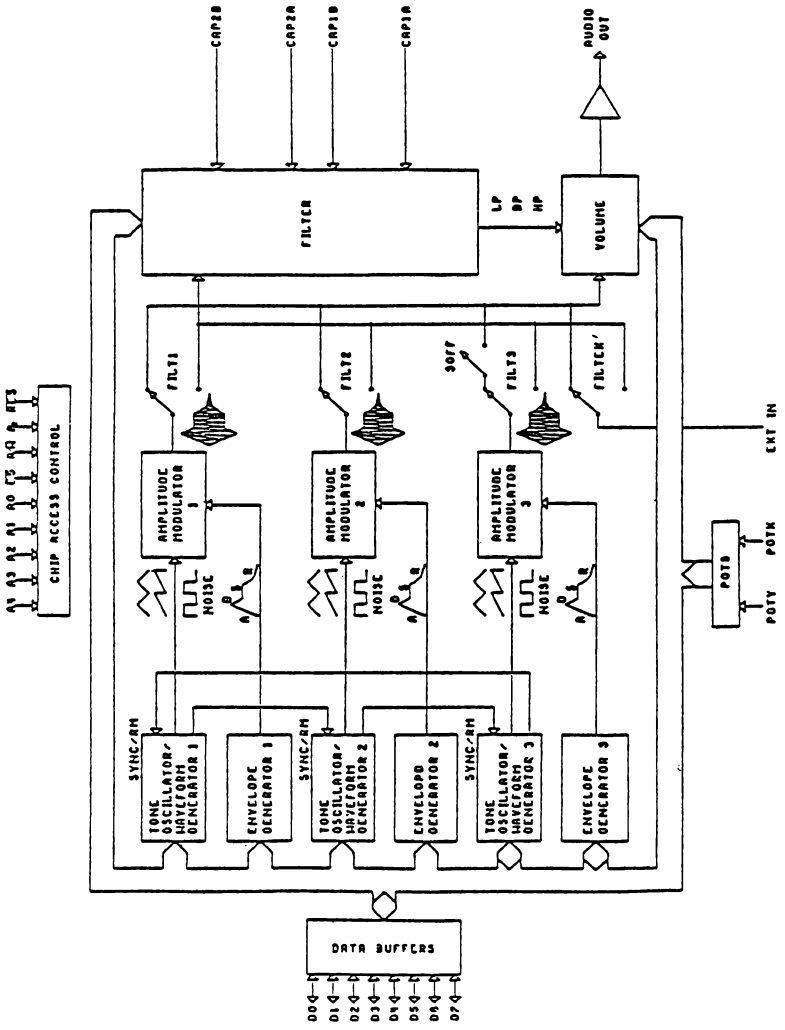
This third oscillator can also act as a random number generator.

The two A to D convertors are provided for linking the 6581 up to a number of potentiometers, for a variety of uses: perhaps as controls for some kind of external musical synthesiser.

In addition, we can link several 6581s up together, or mix them up into other external audio equipment.

The rest of this chapter goes into more detail on all of these features.

6581 Block Diagram



6581 Control Register

	ADDRESS					REG #	DATA								REG	REG
	A4	A3	A2	A1	A0	(HEX)	D7	D6	D5	D4	D3	D2	D1	D0	NAME	TYPE
															VOICE 1	
0	0	0	0	0	0	00									FREQ LO	WRITE-only
1	0	0	0	0	1	01									FREQ HI	WRITE-only
2	0	0	0	1	0	02									PW LO	WRITE-only
3	0	0	0	1	1	03									PW HI	WRITE-only
4	0	0	1	0	0	04									CONTROL REG	WRITE-only
5	0	0	1	0	1	05									ATTACK/DECAY	WRITE-only
6	0	0	1	1	0	06									SUSTAIN/RELEASE	WRITE-only
															VOICE 2	
7	0	0	1	1	1	07									FREQ LO	WRITE-only
8	0	1	0	0	0	08									FREQ HI	WRITE-only
9	0	1	0	0	1	09									PW LO	WRITE-only
10	0	1	0	1	0	0A									PW HI	WRITE-only
11	0	1	0	1	1	0B									CONTROL REG	WRITE-only
12	0	1	1	0	0	0C									ATTACK/DECAY	WRITE-only
13	0	1	1	0	1	0D									SUSTAIN/RELEASE	WRITE-only
															VOICE 3	
14	0	1	1	1	0	0E									FREQ LO	WRITE-only
15	0	1	1	1	1	0F									FREQ HI	WRITE-only
16	1	0	0	0	0	10									PW LO	WRITE-only
17	1	0	0	0	1	11									PW HI	WRITE-only
18	1	0	0	1	0	12									CONTROL REG	WRITE-only
19	1	0	0	1	1	13									ATTACK/DECAY	WRITE-only
20	1	0	1	0	0	14									SUSTAIN/RELEASE	WRITE-only
															FILTER	
21	1	0	1	0	1	15									FC LO	WRITE-only
22	1	0	1	1	0	16									FC HI	WRITE-only
23	1	0	1	1	1	17									RES/FILT	WRITE-only
24	1	1	0	0	0	18									MODE/VOL	WRITE-only
															MISC	
25	1	1	0	0	1	19									POTX	READ-only
26	1	1	0	1	0	1A									POTY	READ-only
27	1	1	0	1	1	1B									OSC3/RANDOM	READ-only
28	1	1	1	0	0	1C									ENV3	READ-only

6581 Register Descriptions

We'll now spend a few pages going through each register in detail, using location 54272 as our base register, or register 0, starting with : =

Voice 1 : Frequency Low/Frequency High
These two registers combine together to form a 16 bit number which linearly controls the frequency of voice 1.

This frequency is determined by the following equation : =

Fout = (Fn *Fclk/16777216) Hz

where Fn is the 16 bit number in the frequency registers, and Fclk is the system clock applied to the 02 input, pin 6.

For our standard 1 MHz clock, this formula comes down to : =

Fout = (Fn * 0.05961) Hz

Later on in this chapter we'll give you a much more complete table of musical notes, frequencies and values of Fn than were given earlier in the chapter on sound on the 64.

These values are all meant to be taken with a pinch of salt, and can be varied slightly as you, the listener, see fit.

It should also be noted here that the frequency resolution of the 6581 is such that sweeping from note to note on an even tempered scale is possible, without any noticeable frequency steps.

Pulse Width Low and High

These two registers combine together to form a 12 bit number, which linearly controls the pulse width of the pulse waveform of voice one. Bits 4 to 7 of Pulse high are not used.

This pulse width is determined by the following equation :=

PWout = (PWn/40.95) %

where PWn is the 12 bit number in the PW registers.

Again, the pulse width resolution is such that the width can be smoothly swept along without any noticeable stepping effects.

For constant pulse widths, a value of 0 or 4095 will produce a constant DC output, whilst a value of 2048 will produce a perfect square wave.

Obviously, these features cannot be used without having previously selected the pulse waveform for voice 1.

Control Register

The most important register of them all, containing eight control bits with the following functions :=

Gate - Bit 0

This controls the envelope generator for voice 1, and when this bit

is set to a '1' the envelope generator is triggered and the Attack/Decay/Sustain (or ADS) cycle is begun.

When this bit is reset to a zero, then the release part of the cycle begins.

This envelope generator controls the amplitude of voice 1 as it appears at the audio output, and must therefore be triggered in order for the selected output of voice 1 to be audible.

Sync - Bit 1

When set to a '1' this synchronizes the fundamental frequency of voice 1 with the fundamental frequency of voice 3, producing what are known as 'hard sync' effects.

Varying the frequency of voice 1 with respect to voice 3 produces a wide range of complex harmonic structures from voice 1 at the frequency of voice 3.

In order for this to take place, obviously voice 3 must be set to some frequency or other, preferably lower than that of voice 1, but naturally higher than zero.

Nothing else connected with voice 3 has any effect on Sync.

Ring Mod - Bit 2

When set to a '1' this bit replaces the triangle waveform of voice 1 with a ring modulation combination of voices 1 and 3 : obviously, one must previously have selected the waveform of voice 1 to be a triangle one!

Varying the frequency of voice 1 with respect to voice 3 produces a wide range of non-harmonic overtone structures.

Again, nothing else connected with voice 3 has any effect on Ring Mod.

Test - Bit 3

This bit, when set to a '1', resets and holds voice 1 at zero, until the bit is cleared.

The noise waveform of voice 1 is also reset, and if a pulse wave has been selected this is held at a DC level.

Normally only used for testing purposes, and hence the name, it can be used to synchronize voice 1 to external events.

Triangle - Bit 4

When this is set to a '1', the triangle waveform is selected for voice 1. This is low in harmonics, and thus produces a mellow, reed-like note.

Sawtooth - Bit 5

When this is set to a '1', the sawtooth waveform is selected for voice 1. This is rich in harmonics, and thus produces a brassy, trumpet-like note.

Pulse - Bit 6

When this is set to a '1' the pulse waveform is selected for voice 1. The harmonic content of this waveform can be varied by altering the pulse width registers, producing a wide variaty of different musical (and not so musical) sounds.

Sweeping through the pulse widths can produce some dynamic effects, and can add a sense of motion to the sound.

Rapidly altering from one pulse width to another can also be used to produce some interesting harmonic effects.

Noise - Bit 7

When set to a '1', the noise waveform is selected for voice 1. This is a totally random signal which changes at the frequency of voice 1, and thus is of most use in generating purely sound 'effects', like missiles taking off, engines revving, or vast explosions.

The sound of waves lapping on the beach, or of a cymbal being rapidly hit, can be achieved by sweeping through the different frequencies.

One of the above waveforms must be selected in order for voice 1 to

produce any audible sound, although that sound can be turned off without un-selecting a waveform, as the voice at the end is a funcion of the envelope generator only.

Also, you cannot add more than one waveform together to produce something totally different from the above four.

You are welcome to try, but the most likely result is that voice 1 will be switched off, and can only be reset by the Test bit, or by setting pin 5 to low, or '0'.

Attack/Decay

Bits 4 to 7 of this register, known as ATK0 to ATK3, select an attack rate from 0 to 15 for the voice 1 envelope generator. The Attack rate determines how fast the output of voice 1 rises from zero to peak amplitude, when the envelope generator is triggered.

Bits 0 to 3 of this register, known as DCY0 to DCY3, allow you to select a decay rate from 1 to 15 for the envelope generator. The decay cycle comes after the attack cycle, and determines how quickly the output falls from the peak amplitude to the selected sustain level.

Sustain/Release

Bits 4 to 7 of this register, known as STN0 to STN3, allow you to select a sustain level from 0 to 15 for the envelope generator for voice 1. The sustain cycle follows the decay cycle, and determines at what amplitude voice 1 will remain as long as the trigger bit remains set. This is all done on a linear basis, so, for example, a sustain level of 8 would cause voice 1 to sustain at exactly half the peak amplitude reached by the attack cycle.

Bits 0 to 3 of this register, known as RLS0 to RLS3, allow you to select a release rate from 0 to 15 for the envelope generator of voice 1. The release cycle follows the sustain cycle, and determines how rapidly the amplitude of voice 1 will fall from the sustain level to zero amplitude.

The 16 release rates are identical to the decay rates, shown below.

Envelope Rates

The cycling of this envelope generator can be altered at any point in the cycle by the gate bit, as the generator can be gated and released at any time, without restriction.

So, if the gate bit is set whilst half way through an attack cycle, the release cycle will begin immediately, and if the gate is reset again whilst the release cycle is still continuing, another attack cycle will start from whatever amplitude had been reached during release.

As you might imagine, this gets a bit hairy after a while, but does allow quite complex effects to be achieved.

Envelope Rates

Value Dec	Hex	Attack Rate (Time/Cycle) ms	Decay/Release Rate (Time/Cycle) ms
0	0	2	6
1	1	8	24
2	2	16	48
3	3	24	72
4	4	38	114
5	5	56	168
6	6	68	204
7	7	80	240
8	8	100	300
9	9	250	750
10	A	500	1.5 sec
11	B	800	2.4 sec
12	C	1 sec	3 sec
13	D	3 sec	9 sec
14	E	5 sec	15 sec
15	F	8 sec	24 sec

Voice 2 and 3

Voice 2

The registers $07 to $0D control voice 2, and function in the same way as registers $00 to $06 for voice 1, with the following two exceptions : =

1) When SYNC is selected, it synchronizes voice 2 with voice 1.
2) When RING MOD is selected, it replaces the triangle output of voice 2 with the ring modulated combination of voices 2 and 1.

Voice 3

The registers $0E to $14 control voice 3, and function in the same way as registers $00 to $06 for voice 1, with the following two exceptions : =

1) When SYNC is selected, it synchronizes voice 3 with voice 2.
2) When RING MOD is selected, it replaces the triangle output of voice 3 with the ring modulated combination of voices 3 and 2.

Filtering

Freq Lo/Freq Hi - registers $15 and $16.

As bits 3 to 7 of register $15 are not used, these two combine together to form an 11 bit number which linearly controls the cutoff, or centre frequency of the programmable filter. The approximate cutoff frequency is obtained from : =

FCout = ((6.6E-8 + FCn * 1.28E-8)/C) Hz

where FCn is the 11 bit number in the above two registers and C is the value of the two filter capacitors connected to pins 1 to 4, or in our case 2200 picoFarads.

This gives an approximate filter range of 30 Hz to 12 KHz, according to : =

FCout = (30 + FCn * 5.8) Hz

This frequency response can be altered for specific applications, but I don't recommend you doing it!

Res/Filt - Register $17

Bits 4 to 7 of this register control the resonance of the filter, where resonance emphasises components of the frequency at the cutoff frequency of the filter, thus causing a sharper sound.

There are 16 resonance settings ranging linearly from no resonance, when this is set to zero, or maximum resonance, when it is set to 15.

Bits 0 to 3 determine which signals will be routed through the filter : =

Bit 0: When this is set to zero, voice 1 appears directly at the audio output, and there is no filtering effect. When set to 1, voice 1 is processed through the filter, and the harmonic content of voice 1 is altered according to the selected filter parameters.

Bit 1: Ditto for voice 2

Bit 2: Ditto for voice 3

Bit 3: Ditto for external audio input on pin 26.

Mode/Vol - Register $18

We've already seen this one as the master volume control, but it actually does a whole lot more.

Bits 0 to 3 are the actual volume settings, and allow you to select an overall volume ranging from 0 (silence) to 15 (maximum).

Bits 4 through 7 select various filter modes and output options : =

Bit 4 - When set to a '1', the low pass output of the filter is selected and sent to the audio output. For a given filter input signal, all components of the frequency below the filter cut off are passed

through unaltered, whilst all those above the cutoff are attenuated at a rate of 12 decibels per octave.

Bit 5 - As above for band pass output, but attenuation above and below the cutoff is at a rate of 6 decibels per octave.

Bit 6 - As above for high pass output, and attenuation below the cutoff is back to 12 decibels per octave.

Bit 7 - When this is set to '1' the output of voice 3 is disconnected from the direct audio path, so setting voice 3 to bypass the filter and setting 3 OFF to a '1' stops voice 3 from reaching the audio output. Thus voice 3 can be used for modulation purposes without any extraneous noises coming out.

These filter modes are additive, in that one can combine a number of different modes at the same time. Playing with, and understanding, these frequency alterations is the key to getting the most out of the 6581.

Miscellaneous Information

POTX - Register $19

This allows the processor to read the position of the potentiometer connected to POTX, or pin 24, with a range of 0 at minimum resistance to 255 at maximum resistance. This value is constantly being updated.

POTY - Register $1A

As above, for POTY, at pin 23.

OSC 3/Random - Register $1B

This allows the processor to read the upper 8 output bits of oscillator three, and it is worth experimenting to read the numbers generated when producing the various waveforms. Noise produces a series of random numbers, and thus this can also be used as a random number generator in preference to RND.

Its chief purpose is to act as a modulation generator, by combining the numbers produced with something like the filter frequency registers, or the pulse width, to produce a virtually unlimited range of effects.

Voice 3 should be set to zero when using this mode.

ENV 3 - Register $1C

As above, but for the voice 3 envelope generator, and this is usually used in conjunction with the frequency filter.

The voice 3 envelope generator must be gated in order to produce any effects from this process.

6581 Pin Description

CAP1A,CAP1B (Pins 1&2)/CAP2A,CAP2B (Pins 3&4)

These four pins are used to connect the two integrating capacitors required by the programmable filter. C1 connects pins 1 and 2, and C2 connects pins 3 and 4.

The maximum cutoff frequency FCmax is given by : =

FCmax = 2.6E-5 / C

where C is the capacitor value, in our case equal to 2200 pico farads.

RES - Pin 5

This is the reset control for the 6581, and is connected to the reset line of the 6510.

02 - Pin 6.

This is the master clock for the 6581, and all oscillation frequencies and envelope rates are referenced from this clock. It also controls data transfer from the 6581 to the 6510,

R/W - Pin 7

This controls the direction of data transfer between the 6581 to the 6510. When this line is set to a '1' the 6510 can read from the selected 6581 register, and when set to a '0' the 6510 can write to the selected 6581 register. Thus this pin is connected to the system read/write line.

Note the read/write capabilities of the various registers as described earlier.

CS - Pin 8

This is a low-active chip select which controls data transfer between the 6581 and the 6510. Data transfer can only take place when this is set to low, and indeed when R/W is high, we get the 6510 reading from the 6581, and when R/W is low we get the 6510 transferring data to the 6581.

A0 - A4 - Pins 9 to 13

These inputs are used to select one of the 29 registers in the 6581. Sharp-eyed readers will observe that we have in fact the capacity to select any one of 32 registers. However, three are not used (perhaps they were once intended to be seperate volume controls ?). PEEKing these registers will only return a 0.

GND - Pin 14

This is simply the ground line.

D0 - D7 - Pins 15 to 22.

These 8 bi-directional lines are used in the transfer of data between the 6581 and the 6510. When writing data, the data buffers remain in the off state, and in read mode they are on, when the 6581 supplies data to the 6510.

POTY, POTX - Pins 23 and 24

These are the inputs to the A/D convertors used to digitize the position of any potentiometers connected up to the system.

Vcc - Pin 25

This is just used to minimise noise, and is linked up to the power supply with a 5 volt line.

EXT IN - Pin 26

This analog input allows external audio signals to be mixed with the audio output of the 6581, or processed through the filter. This can be fed to mix outputs of multiple 6581s by daisy chaining them. The maximum number of chips that can be linked together in this way is limited only by the amount of noise and distortion allowable at the final output.

AUDIO OUT - Pin 27

This is the line that carries the final audio output of the 6581, including the voices, filtering, and any external output. The output level of all the output is selected by the master volume control, register 24.

Vdd - Pin 28

Again this is there to minimise noise, and requires a 12V line.

6581 Characteristics

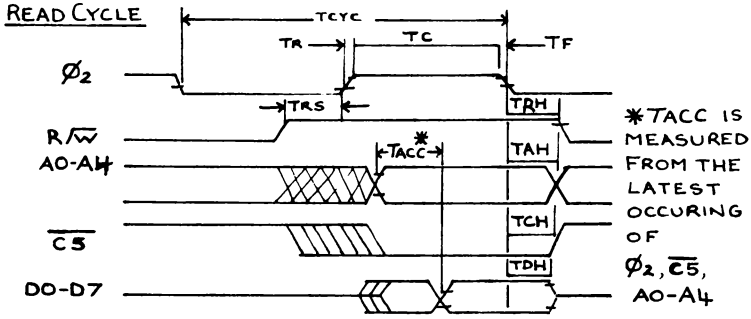
ABSOLUTE MAXIMUM RATINGS

RATING	SYMBOL	VALUE	UNITS
Supply Voltage	Vdd	-0.3 to +17	VDC
Supply Voltage	Vcc	-0.3 to +7	VDC
Input Voltage (analog)	Vina	-0.3 to +17	VDC
Input Voltage (digital)	Vind	-0.3 to +7	VDC
Operating Temperature	Ta	0 to +70	°C
Storage Temperature	Tstg	-55 to +150	°C

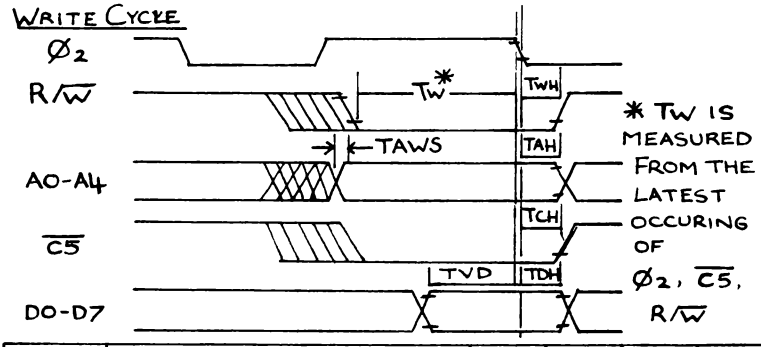
ELECTRICAL CHARACTERISTICS (Vdd=12±5% VDC, Vcc=5±5% VDC, Ta=0 to 70°C)

CHARACTERISTIC		SYMBOL	MIN	TYP	MAX	UNITS
Input High Voltage	(RES, Ø2, R/W, CS, A0-A4, D0-D7)	Vih	2	-	Vcc	VDC
Input Low Voltage		Vil	-0.3	-	0.8	VDC
Input Leakage Current	(RES, Ø2, R/W, CS, A0-A4; Vin=0-5 VDC)	Iin	-	-	2.5	uA
Three-State (Off) Input Leakage Current	(D0-D7; Vcc=max, Vin=0.4-2.4 VDC)	Itsi	-	-	10	uA
Output High Voltage	(D0-D7; Vcc=min, Iload=200 uA)	Voh	2.4	-	Vcc-0.7	VDC
Output Low Voltage	(D0-D7; Vcc=max, Iload=3.2 mA)	Vol	GND	-	0.4	VDC
Output High Current	(D0-D7; Sourcing, Voh=2.4 VDC)	Ioh	200	-	-	uA
Output Low Current	(D0-D7; Sinking, Vol=0.4 VDC)	Iol	3.2	-	-	mA
Input Capacitance	(RES, Ø2, R/W, CS, A0-A4, D0-D7)	Cin	-	-	10	pF
Pot Trigger Voltage	(POTX, POTY)	Vpot	-	Vcc/2	-	VDC
Pot Sink Current	(POTX, POTY)	Ipot	500	-	-	uA
Input Impedance	(EXT IN)	Rin	100	150	-	KOhms
Audio Input Voltage	(EXT IN)	Vin	5.7 -	6 0.5	6.3 3	VDC VAC
Audio Output Voltage	(AUDIO OUT, 1 KOhm load, volume=max) One Voice on: All Voices on:	Vout	5.7 0.4 1.0	6 0.5 1.5	6.3 0.6 2.0	VDC VAC VAC
Power Supply Current	(Vdd)	Idd	-	20	25	mA
Power Supply Current	(Vcc)	Icc	-	70	100	mA
Power Dissipation	(Total)	Pd	-	600	1000	mW

6581 Timing



SYMBOL	NAME	MIN	TYP	MAX	UNITS
TCYC	Clock Cycle Time	1	–	20	µS
TC	Clock High Pulse Width	450	500	10,000	nS
TR, TF	Clock Rise/Fall Time	–	–	25	nS
TRS	Read Set-up Time	0	–	–	nS
TRH	Read Hold Time	0	–	–	nS
TACC	Access Time	–	–	300	nS
TAH	Address Hold Time	10	–	–	nS
TCH	Chip Select Hold Time	0	–	–	nS
TDH	Data Hold Time	20	–	–	nS



SYMBOL	NAME	MIN	TYP	MAX	UNITS
TW	Write Pulse Width	350	–	–	nS
TWH	Write Hold Time	0	–	–	nS
TAWS	Address Set-up Time	0	–	–	nS
TAH	Address Hold Time	10	–	–	nS
TCH	Chip Select Hold Time	0	–	–	nS
TVD	Valid Data	80	–	–	nS
TDH	Data Hold Time	10	–	–	nS

Equal Tempered Musical Scale Values

MUSICAL NOTE		FREQ (HZ)	OSC Fn (DECIMAL)	OSC Fn (HEX)	MUSICAL NOTE		FREQ (HZ)	OSC Fn (DECIMAL)	OSC Fn (HEX)
0	C0	16.35	274	0112	48	C4	261.63	4389	1125
1	C0#	17.32	291	0123	49	C4#	277.18	4650	122A
2	D0	18.35	308	0134	50	D4	293.66	4927	133F
3	D0#	19.44	326	0146	51	D4#	311.13	5220	1464
4	E0	20.60	346	015A	52	E4	329.63	5530	159A
5	F0	21.83	366	016E	53	F4	349.23	5859	16E3
6	F0#	23.12	388	0184	54	F4#	370.00	6207	183F
7	G0	24.50	411	019B	55	G4	392.00	6577	19B1
8	G0#	25.96	435	01B3	56	G4#	415.30	6968	1B38
9	A0	27.50	461	01CD	57	A4	440.00	7382	1CD6
10	A0#	29.14	489	01E9	58	A4#	466.16	7821	1E8D
11	B0	30.87	518	0206	59	B4	493.88	8286	205E
12	C1	32.70	549	0225	60	C5	523.25	8779	224B
13	C1#	34.65	581	0245	61	C5#	554.37	9301	2455
14	D1	36.71	616	0268	62	D5	587.33	9854	267E
15	D1#	38.89	652	028C	63	D5#	622.25	10440	28C8
16	E1	41.20	691	02B3	64	E5	659.26	11060	2B34
17	F1	43.65	732	02DC	65	F5	698.46	11718	2DC6
18	F1#	46.25	776	0308	66	F5#	740.00	12415	307F
19	G1	49.00	822	0336	67	G5	783.99	13153	3361
20	G1#	51.91	871	0367	68	G5#	830.61	13935	366F
21	A1	55.00	923	039B	69	A5	880.00	14764	39AC
22	A1#	58.27	978	03D2	70	A5#	932.33	15642	3D1A
23	B1	61.74	1036	040C	71	B5	987.77	16572	40BC
24	C2	65.41	1097	0449	72	C6	1046.50	17557	4495
25	C2#	69.30	1163	048B	73	C6#	1108.73	18601	4899
26	D2	73.42	1232	04D0	74	D6	1174.66	19708	4CFC
27	D2#	77.78	1305	0519	75	D6#	1244.51	20887	518F
28	E2	82.41	1383	0567	76	E6	1318.51	22121	5669
29	F2	87.31	1465	05B9	77	F6	1396.91	23436	5B8C
30	F2#	92.50	1552	0610	78	F6#	1479.98	24830	60FE
31	G2	98.00	1644	066C	79	G6	1567.98	26306	66C2
32	G2#	103.83	1742	06CE	80	G6#	1661.22	27871	6CDF
33	A2	110.00	1845	0735	81	A6	1760.00	29528	7338
34	A2#	116.54	1955	07A3	82	A6#	1864.65	31284	7A34
35	B2	123.47	2071	0817	83	B6	1975.53	33144	8178
36	C3	130.81	2195	0893	84	C7	2093.00	35115	892B
37	C3#	138.59	2325	0915	85	C7#	2217.46	37203	9153
38	D3	146.83	2463	099F	86	D7	2349.32	39415	99F7
39	D3#	155.56	2610	0A32	87	D7#	2489.01	41759	A31F
40	E3	164.81	2765	0ACD	88	E7	2637.02	44242	ACD2
41	F3	174.61	2930	0B72	89	F7	2793.83	46873	B719
42	F3#	185.00	3104	0C20	90	F7#	2959.95	49660	C1FC
43	G3	196.00	3288	0CD8	91	G7	3135.96	52613	CD85
44	G3#	207.65	3484	0D9C	92	G7#	3322.44	55741	D9ED
45	A3	220.00	3691	0E6B	93	A7	3520.00	59056	E6B0
46	A3#	233.08	3910	0F46	94	A7#	3729.31	62567	F467
47	B3	246.94	4143	102F	95	B7	3951.06	*66286	*1F2E

The table above provides a simple and quick method for generating the equal tempered scale. However, it is not very efficient on memory, since it requires 192 bytes just for the table. It would be better to determine each note algorithmically, using the fact that each note in an octave is half the frequency of that note in the next octave, which would bring us down to a table entry of just 12 entries rather than 92 as featured here.

Having done this, we could then specify each note by a single byte. Four bits could specify which of the 12 notes we wish to play in the octave (tones and semi-tones), and three bits to specify which octave

we wish to play in. In other words, use two separate nibbles for each note.

6581 Envelope Generators

The four-part envelope generator used in the 6581, the Attack Decay Sustain Release (ASDR) one, is one of the easiest of the generators to program, as well as giving some of the best results of any electronically generated musical sound.

By appropriate selection of the various envelope parameters the simulation of many popular musical instruments is possible, particularly of percussion instruments, or those where the sound is sustained for an appreciable period.

For instance, the piano. This immediately reaches full volume as soon as the key is struck, and then begins to die away again equally quickly. As long as the key is held down the decay rate is rather slow, but if the key is released, then decay is immediate.

The ADSR diagram might look something like this :=

Attack : 02ms
Decay: 9 750ms
Sustain : 0
Release : 06ms

A percussion instrument, such as a cymbal, is characterised again by an immediate attack, but no sustain period as it gradually decays away to silence, and the parameters for that might be something like :=

Attack : 02ms
Decay: 9 750ms
Sustain : 0
Release : 9 750ms

Wind instruments, such as flutes and oboes, have a characteristically slow rise to maximum volume, then an intermediate sustain for as long as someone keeps supplying the air, and then a slow decay, perhaps like this :=

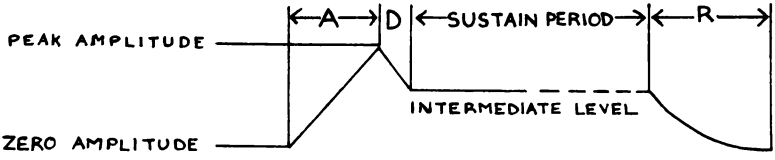
Attack : 10 500ms
Decay: 8 300ms
Sustain : 10
Release : 9 750ms

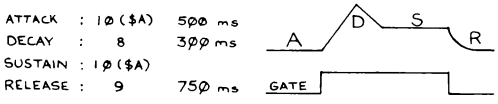
The simplest one of all would be the organ, which reaches and remains at full volume whilst the key is pressed and until it is released again, whence it drops immediately to zero volume, rather like this :=

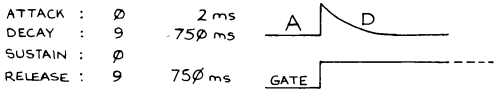
Attack : 02ms
Decay: 06ms
Sustain : 15
Release : 06ms

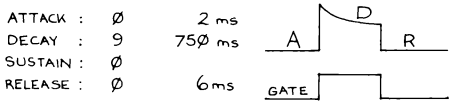
Of course, the 6581 is capable of much more than just mere impersonation, and the many unique sounds that can be generate by it can only really be discovered by a lot of experimentation.

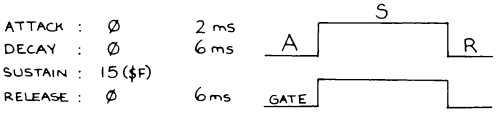
However, a good knowledge of envelope generation will certainly smooth the path, and playing about with ring modulation and filtering can again only make life a lot easier.

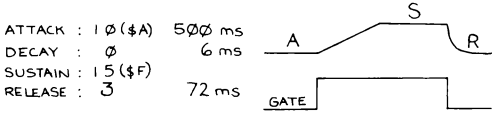




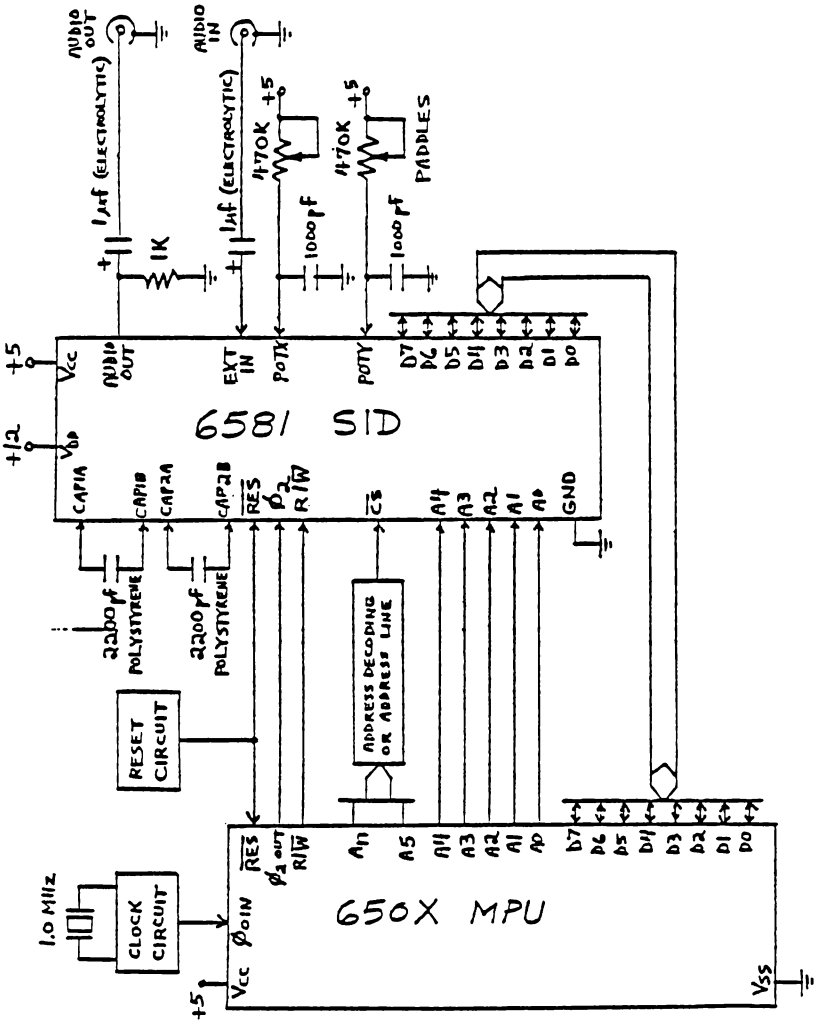








Typical 6581 Application



10

The 6510 and 6526 Input/Output Chips

Introduction to the 6510

The 6510 has an 8 bit bi-directional output port located on-chip, with the Output Register at address 0000 and the Data Direction Register at address 0001. These are the only two differences between this chip and its predecessor the 6502, used for so long by Commodore in their earlier computers.

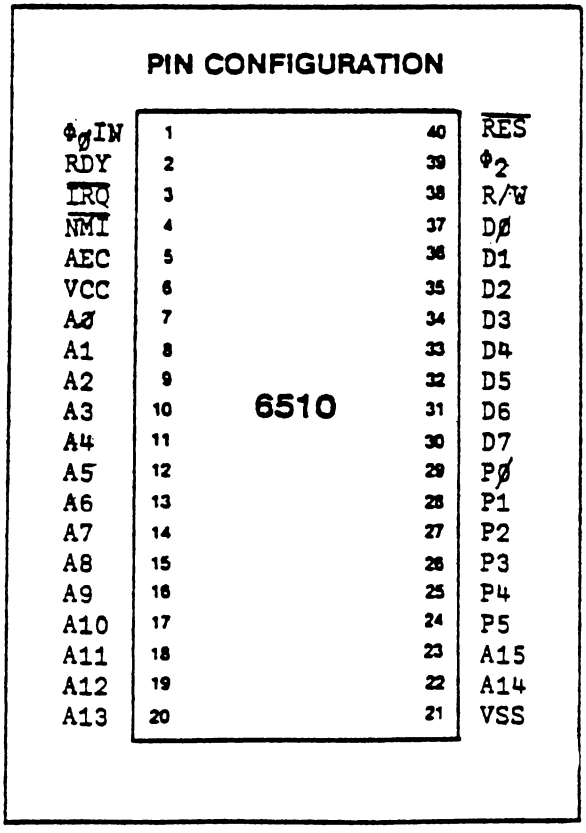
The three state sixteen bit address bus allows direct memory accessing (usually abbreviated to DMA), and multiprocessor systems sharing a common memory.

The internal procesor architecture is, as we shall see, identical to the 6502, which provides the user with a great degree of software compatability.

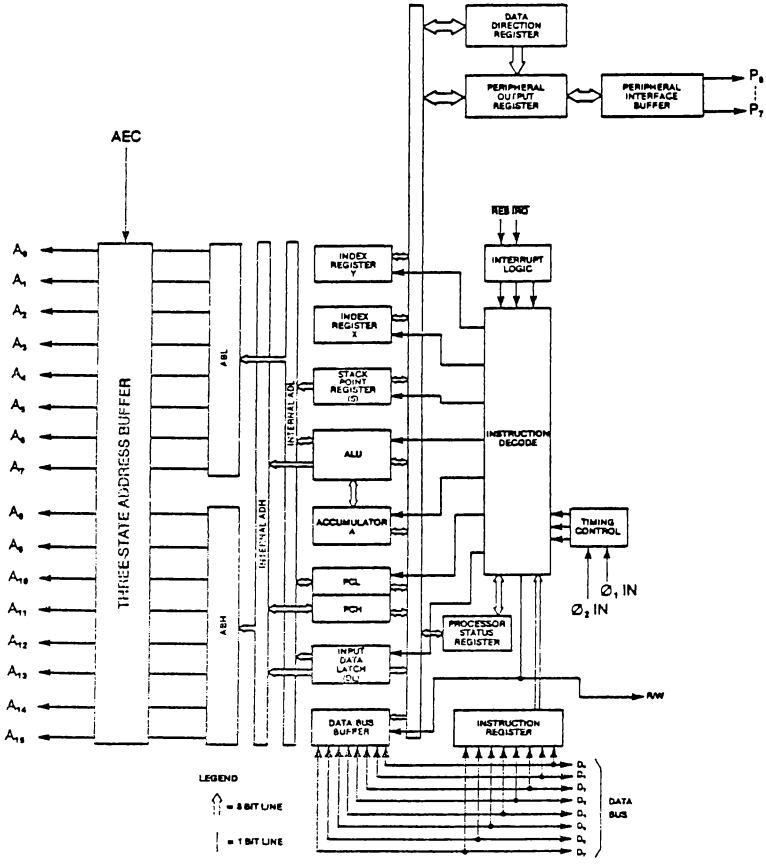
Some of the features of the 6510 include : =

8 bit bi-directional Input/Output port.
Single +5 volt power supply.
Eight bit parallel processing.
56 instructions in the instruction set.
Decimal and binary arithmetic.
13 different addressing modes.
True indexing capability.
Programmable stack pointer.
Variable length of stack.
Interrupt capability.
8 bit bi-directional data bus.
Capability to address up to 65K of memory.
Plus direct memory access capability.

Pin Configuration



6510 Block Diagram



6510 Characteristics

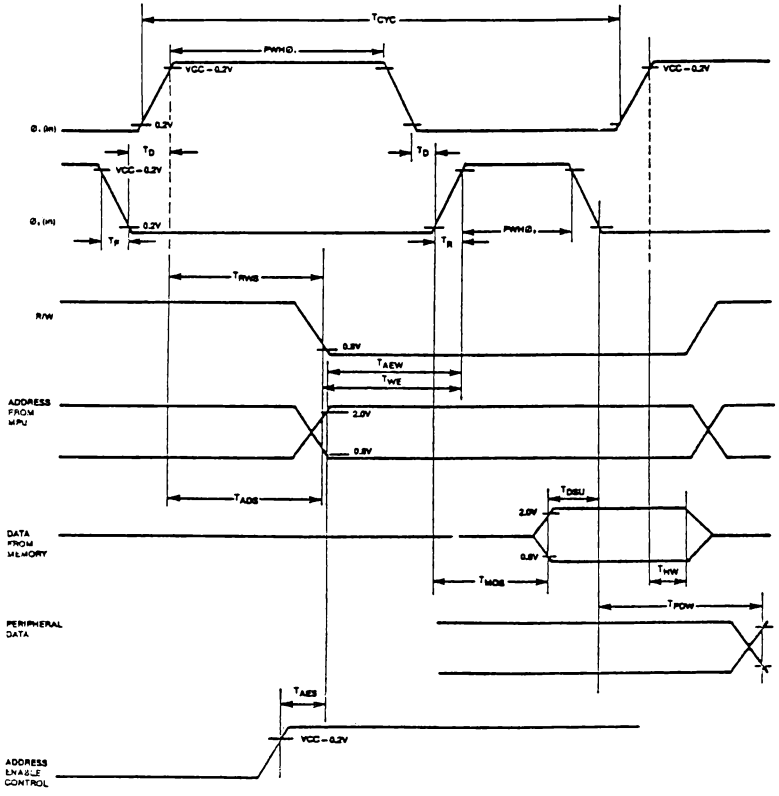
MAXIMUM RATINGS

RATING	SYMBOL	VALUE	UNIT
SUPPLY VOLTAGE	Vcc	−0.3 to +7.0	Vdc
INPUT VOLTAGE	Vin	−0.3 to +7.0	Vdc
OPERATING TEMPERATURE	T_A	0 to +70	°C
STORAGE TEMPERATURE	T_{STG}	−55 to +150	°C

ELECTRICAL CHARACTERISTICS (Vcc = 5.0V ± 5%, Vss = 0, T_A = 0° to + 70°C)

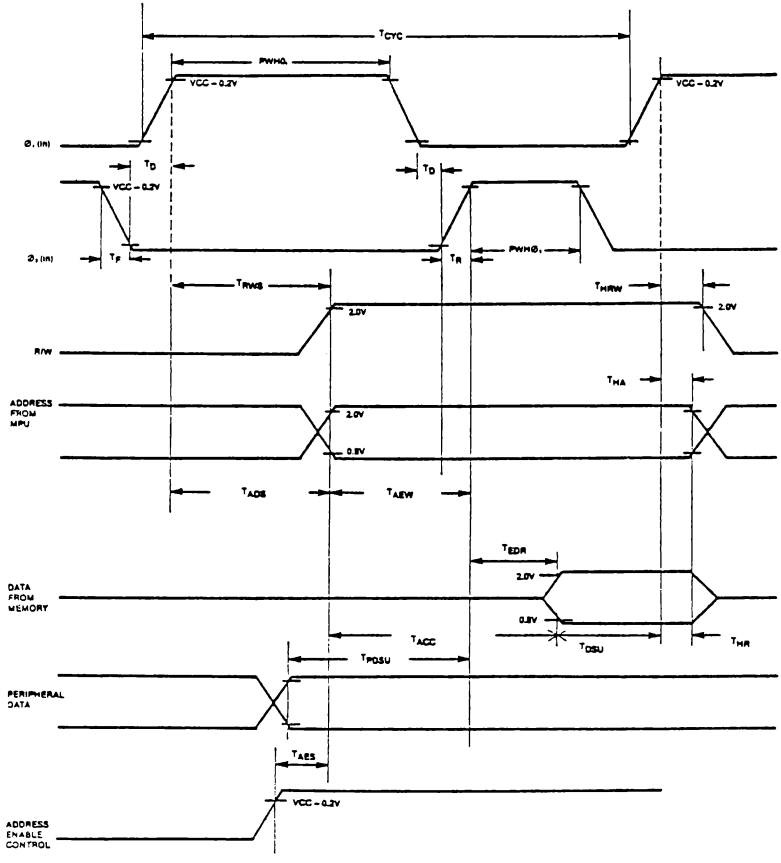
CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	UNIT
Input High Voltage Ø0, Ø1(in)	VIH	Vcc − 0.2	—	Vcc + 1.0V	Vdc
Input High Voltage RES, P0-P7, IRQ, Data		Vss + 2.0	—	—	Vdc
Input Low Voltage Ø0, Ø1(in)	VIL	Vss − 0.3	—	Vss + 0.2	Vdc
RES, P0-P7, IRQ, Data		—	—	Vss + 0.8	Vdc
Input Leakage Current (V_{in} = 0 to 5.25V, Vcc = 5.25V) Logic Ø0, Ø1(in)	Iin	— —	— —	2.5 100	μA μA
Three State (Off State) Input Current (V_{in} = 0.4 to 2.4V, Vcc = 5.2 V) Data Lines	ITSI	—	—	10	μA
Output High Voltage (I_{OH} = −100μAdc, Vcc = 4.75V) Data, A0-A15, R/W, P0-P7	VOH	Vss + 2.4	—	—	Vdc
Out Low Voltage (I_{OL} = 1.6mAdc, Vcc = 4.75V) Data, A0-A15, R/W, P0-P7	VOL	—	—	Vss + 0.4	Vdc
Power Supply Current	ICC	—	125		mA
Capacitance V_{in} = 0, T_A = 25°C, f = 1MHz Logic, P0-P7 Data A0-A15, R/W Ø0 Ø1	C C_{in} C_{out} $C_{Ø0}$ $C_{Ø1}$	— — — — —	— — — 30 50	10 15 12 50 80	pF

Clock Timing - Writing



TIMING FOR WRITING DATA TO MEMORY OR PERIPHERALS

Clock Timing - Reading



TIMING FOR READING DATA FROM MEMORY OR PERIPHERALS

AC Characteristics

1 MHz TIMING

ELECTRICAL CHARACTERISTICS (V_{CC} = 5V ± 5%, V_{SS} = 0V, T_A = 0° – 70°C)

CLOCK TIMING

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.
Cycle Time	T_{CYC}	1000	—	—
Clock Pulse Width Ø1 (Measured at VCC – 0.2V) Ø2	PWHØ1 PWHØ2	430 470	— —	— —
Fall Time, Rise Time (Measured from 0.2V to VCC – 0.2V)	T_F, T_R	—	—	25
Delay Time between Clocks (Measured at 0.2V)	T_D	0	—	—

READ/WRITE TIMING (LOAD = 1TTL)

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.
Read/Write Setup Time from 6508	T_{RWS}	—	100	300
Address Setup Time from 6508	T_{ADS}	—	100	300
Memory Read Access Time	T_{ACC}	—	—	575
Data Stability Time Period	T_{DSU}	100	—	—
Data Hold Time-Read	T_{HR}		—	—
Data Hold Time-Write	T_{HW}	10	30	—
Data Setup Time from 6510	T_{MDS}	—	150	200
Address Hold Time	T_{HA}	10	30	—
R/W Hold Time	T_{HRW}	10	30	—
Delay Time, Address valid to Ø2 positive transition	T_{AEW}	180	—	—
Delay Time, Ø2 positive transition to Data valid on bus	T_{EDR}	—	—	395
Delay Time, Data valid to Ø2 negative transition	T_{DSU}	300	—	—
Delay Time, R/W negative transition to Ø2 positive transition	T_{WE}	130	—	—
Delay Time, Ø2 negative transition to Peripheral Data valid	T_{PDW}	—	—	1
Peripheral Data Setup Time	T_{PDSU}	300	—	—
Address Enable Setup Time	T_{AES}			60

Signal Description

There are 8 basic signals to and from the 6510, and these are as follows :=

1) Clocks (01,02)

The 6510 requires a two phase non-overlapping clock that runs at the Vcc voltage level.

2) Address Bus (A0 - A15)

These outputs are TTL compatible, and are capable of driving one standard TTL load and 130 pf.

3) Data Bus (D0 - D7)

These eight pins form a bi-directional data bus, transferring data to and from the device and peripherals. These outputs are tri-state buffers.

4) Reset

This input is used to reset or start the Commodore 64 from a power down condition. During the time that this line is held low, it is not possible to write to or from the 6510. When a positive edge is detected in the input, the 6510 will immediately begin the reset sequence.

After a system initialisation time of six clock cycles, the mask interrupt flag is set and the 6510 loads the program counter as stored in memory locations $FFFC and $FFFD. This is the start location for program control.

After Vcc reaches 4.75 volts in the power up routine, reset must be held low for at least two cycles. At this time the Read/Write signal becomes valid (see later).

When the reset signal goes high following these two cycles, the 6510 proceeds with the normal reset procedure described above.

5) Interrupt Request (IRQ)

The input requests that an interrupt sequence begin in the 6510. The 6510 completes the current instruction being executed before

recognising that request. When it does, the interrupt mask bit in the status code register is looked at. If the mask flag is not set, the 6510 begins the interrupt sequence.

This stores the program counter and processor status register on the stack. The 6510 then sets the interrupt mask flag high so that no other interrupts can occur, and at the end of this cycle the low program counter will be loaded from memory location $FFFE and the high program counter from $FFFF, therefore transferring program control to the address indicated in those two locations.

6) Address Enable Control (AEC)

The address bus is valid only when the AEC line is high. When it's low, the address bus is in a high impedance state, facilitating direct memory access.

7) Input/Output Port (P0 - P7)

Eight pins are used for this peripheral port, which can transfer data to or from peripheral devices. The output register is held in RAM at location $0000, and the data direction register at location $0001. Do not attempt to program these from Basic!

8) Read/Write (R/W)

This signal is generated by the 6510 to control the direction of data transfer on the data bus. This line is held high, except when the 6510 is writing to memory or some peripheral device, in which case it is set low.

Addressing Modes

There are a number of different ways of addressing the 6510 in machine code on the Commodore 64, and for the sake of reference these are : =

Accumulator Addressing

This is represented by a one byte instruction, implying an operation on the accumulator.

Immediate Addressing

The operand is contained in the second byte of the instruction, and no further memory addressing is required.

Absolute Addressing

The second byte of the instruction specifies the eight low order bits of the effective address, while the third byte specifies the eight high order bits. Thus absolute addressing allows access to the entire 65K of addressable memory.

Zero Page Addressing

This allows shorter code and execution times by only fetching the second byte of the instruction and assuming a high address byte of zero. Using zero page can considerably speed up program efficiency, but alas the 64 wants to use this space as much as you do, so careful programming is called for!

Indexed Zero Page Addressing

This is used in conjunction with the index register. The effective address is calculated by adding the second byte to the contents of the index register. As this is a form of zero page addressing, the content of the second byte references a location in zero page. Again, as we're using zero page, no carry is added to the high order 8 bits of memory, and crossing of boundaries does not occur.

Indexed Absolute Addressing

This is used in conjunction with the X and Y index registers. The effective address is calculated by by adding the contents of X and Y to the address contained in the second and third bytes of the instruction. This allows the index register to contain the index, or count value, and the instruction to contain the base address. This also allows any location referencing and the index to modify multiple fields, thus speeding up program execution.

Implied Addressing

The address containing the operand is implicitly stated in the operation code of the instruction.

Relative Addressing

This is used only with branch instructions, and gives a destination for the conditional branch. The second byte of the instruction becomes the operand, which is an offset added to the the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is +127 to -128 bytes from the next instruction.

Indexed Indirect Addressing

The second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero, whose content is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the effective address must be in page zero.

Indirect Indexed Addressing

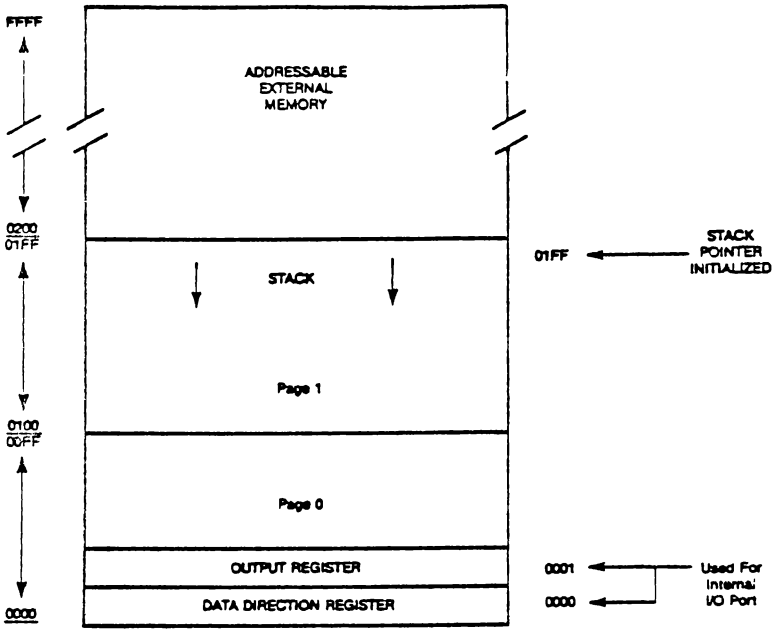
The second byte of the instruction points to a memory location in page zero. The contents of that memory location are added to the contents of the Y register. The result of this is the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

Absolute Indirect

The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that location are contained in the third byte of the instruction. The content of this fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte, which is then loaded into the sixteen bits of the program counter.

A full list of the 6510 instruction set can be found in the appendices at the back of this book.

6510 Memory Map



The 6526 Complex Interface Adaptor

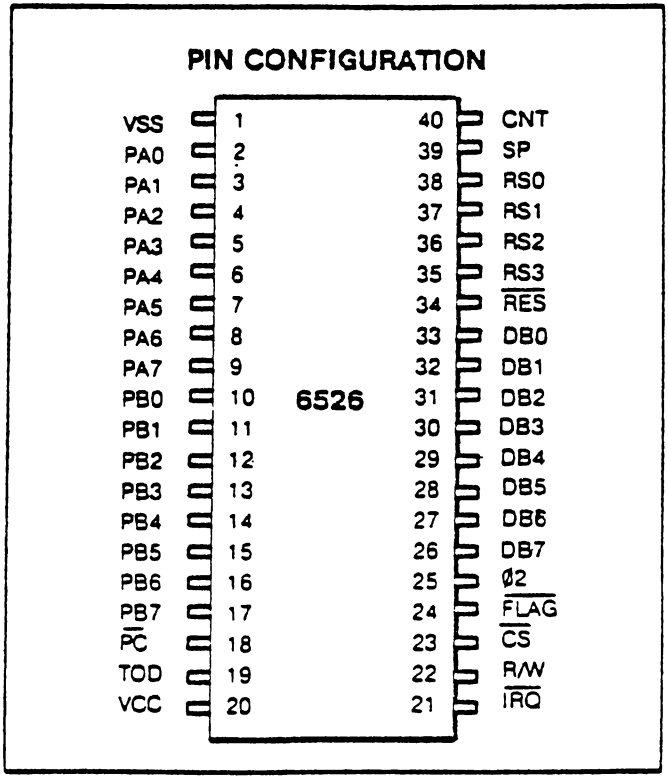
Introduction

The 6526 Complex Interface Adaptor have some powerful features, including : =

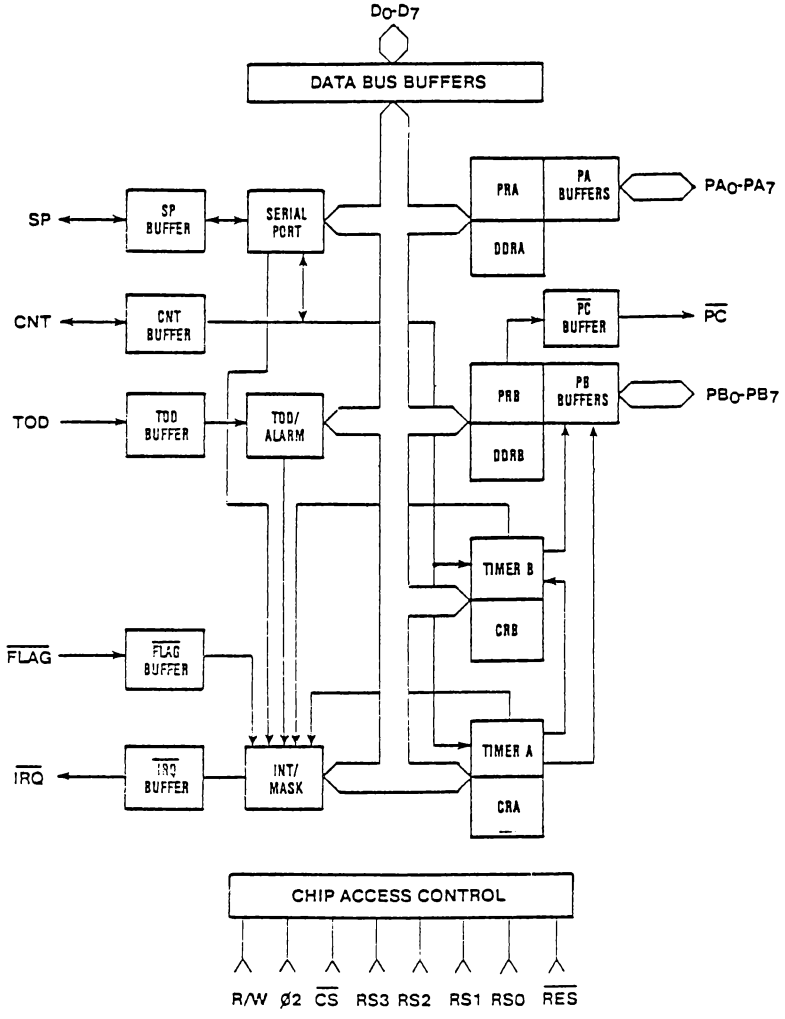
16 individually programmable Input/Output lines.
8 or 16 bit handshaking on read or write.
2 independent, linkable, 16 bit interval timers.
24 hour time of day clock, with programmable alarm.
8 bit shift register for serial input/output.

It can be purchased in either a 1Mz or 2Mz version, but as the 64 uses the 1Mz one, that's the one we'll concentrate on here.

Pin Configuration



6526 Block Diagram



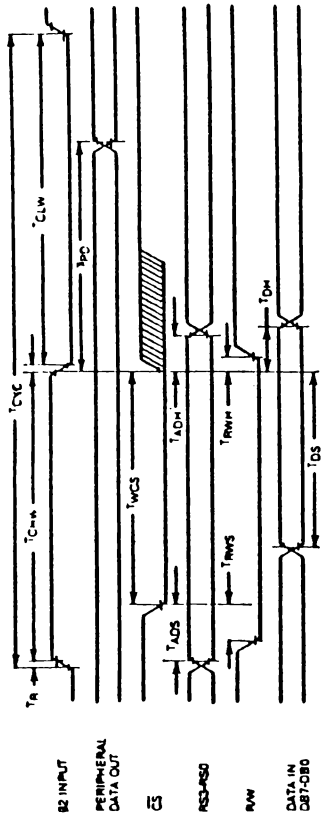
6526 Electrical Characteristics

ELECTRICAL CHARACTERISTICS (V_{CC} ± 5%, VSS = 0v, T_A = 0-70°C)

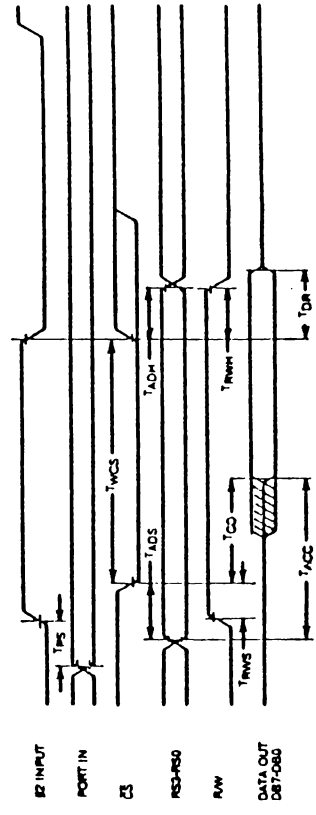
Characteristic	Symbol	Min.	Typ.	Max.	Unit
Input High Voltage	V_{IH}	+ 2.4	—	V_{CC}	V
Input Low Voltage	V_{IL}	− 0.3	—	—	V
Input Leakage Current: V_{IN} = VSS + 5v (TOD, R/W, $\overline{FLAG}$, Ø2, $\overline{RES}$, RS0-RS3, $\overline{CS}$)	I_{IN}	—	1.0	2.5	μA
Port Input Pull-up Resistance	R_{PI}	3.1	5.0	—	KΩ
Output Leakage Current for High Impedance State (Three State); V_{IN} = 4v to 2.4v; (DB0-DB7, SP, CNT, $\overline{IRQ}$)	I_{TSI}	—	± 1.0	± 10.0	μA
Output High Voltage VCC=MIN, I_{LOAD} < −200uA (PA0-PA7, $\overline{PC}$ PB0-PB7, DB0-DB7)	V_{OH}	+ 2.4	—	V_{CC}	V
Output Low Voltage VCC = MIN, I_{LOAD} < 3.2mA	V_{OL}	—	—	+ 0.40	V
Output High Current (Sourcing); VOH > 2.4v (PA0-PA7, PB0-PB7, PC, DB0-DB7)	I_{OH}	−200	−1000	—	μA
Output Low Current (Sinking); VOL < .4 v (PA0-PA7, $\overline{PC}$, PB0-PB7, DB0-DB7)	I_{OL}	3.2	—	—	mA
Input Capacitance	C_{IN}	—	7	10	pf
Output Capacitance	C_{OUT}	—	7	10	pf
Power Supply Current	I_{CC}	—	70	100	mA

6526 Read and Write diagrams

6526 WRITE TIMING DIAGRAM



6526 READ TIMING DIAGRAM



6526 Interface Signals

There are seven different read/write signals on the 6526, and these are as follows : =

1) 02 - Clock Input

The 02 clock is a TTL compatible input used for internal device operation, and as a timing reference for communicating with the system data bus.

2) CS - Chip Select Input

The CS input controls the 6526 activity. A low level on CS, while 02 is high, causes the device to respond to signals on the R/W and RS lines. A high on CS prevents these lines from controlling the 6526. The CS line is normally activated low at 02 by the appropriate address combination.

3) R/W - Read/Write Input

This signal is supplied by the 6510, and controls the direction of data transfer to and from the 6526. A high on this signal indicates a read operation, and a low indicates a write operation.

4) RS3 - RS0 - Address Inputs

These select the internal registers, as described in the register map (later).

5) DB7 - DB0 - Data Bus Input/Output

These eight data bus pins transfer information between the 6526 and the system data bus. These are high impedance inputs, unless CS is low and R/W and 02 are high, to read the device. During a read, the data bus output buffers are enabled, driving the data from the selected register onto the system data bus.

6) IRQ - Interrupt Request Output

This is an open drain output connected to the 6510 interrupt input. An external pullup resistor holds the signal high, allowing multiple IRQ outputs to be linked together. The IRQ output is normally off (high).

7) RES - Reset Input

A low on this pin resets all internal registers. The port pins are set as inputs and port registers to zero during this phase, although a read of the ports will return them all as high because of passive pullups. The timer control registers are set to zero and the timer latches are all set to one, whilst all other registers are set to zero, to complete the sequence.

6526 Timing Characteristics

SYMBOL	CHARACTERISTIC	1MHz		UNIT
		MIN	MAX	
	$Ø_2$ Clock			
TCYC	Cycle Time	1,ØØØ	2Ø,ØØØ	nS
TR,TF	Rise and Fall Time	–	25	nS
TCHW	Clock Pulse Width (High)	42Ø	1Ø,ØØØ	nS
TCLW	Clock Pulse Width (Low)	42Ø	1Ø,ØØØ	nS
	Write Cycle			
TPD	Output Delay from Ø2	–	1,ØØØ	nS
TWCS	$\overline{CS}$ low while $Ø_2$ high	42Ø	–	nS
TADS	Address Setup Time	Ø	–	nS
TADH	Address Hold Time	1Ø	–	nS
TRWS	R/W Setup Time	Ø	–	nS
TRWH	R/W Hold Time	Ø	–	nS
TDS	Data Bus Setup Time	15Ø	–	nS
TDH	Data Bus Hold Time	Ø	–	nS
	Read Cycle			
TPS	Port Setup Time	3ØØ	–	nS
TWCS(2)	$\overline{CS}$ Low while O_2 high	42Ø	–	nS
TADS	Address Setup Time	Ø	–	nS
TADH	Address Hold Time	1Ø	–	nS
TRWS	R/W Setup Time	Ø	–	nS
TRWH	R/W Hold Time	Ø	–	nS
TACC	Data Access from RS3-RS0	–	55Ø	nS
TCO(3)	Data Access from $\overline{CS}$	–	32Ø	nS
TDR	Data Release Time	5Ø	–	nS

Register Map and Functional Description

RS3	RS2	RS1	RSØ	REG		
Ø	Ø	Ø	Ø	Ø	PRA	Penpheral Data Reg A
Ø	Ø	Ø	1	1	RRB	Penpheral Data Reg B
Ø	Ø	1	Ø	2	DDRA	Data Direction Reg A
Ø	Ø	1	1	3	DDRB	Data Direction Reg B
Ø	1	Ø	Ø	4	TA LO	Timer A Low Register
Ø	1	Ø	1	5	TA HI	Timer A High Register
Ø	1	1	Ø	6	TB LO	Timer B Low Register
Ø	1	1	1	7	TBHI	Timer B High Register
1	Ø	Ø	Ø	8	TOD IØTHS	IØths of Seconds Register
1	Ø	Ø	1	9	TOD SEC	Seconds Register
1	Ø	1	Ø	A	TOD MIN	Minutes Register
1	Ø	1	1	B	TOD HR	Hours - AM PM Register
1	1	Ø	Ø	C	SDR	Serial Data Register
1	1	Ø	1	D	ICR	Interrupt Control Register
1	1	1	Ø	E	CRA	Control Reg A
1	1	1	1	F	CRB	Control Reg B

I/O Ports (PRA,PRB,DDRA,DDRB)

Ports A and B each consist of an eight bit peripheral data register (PR), and an eight bit data direction register (DDR). If the DDR bit is set to a zero, then the corresponding PR bit is set to an input, and setting a DDR bit to one causes the corresponding PR bit to act as an output. On a read, PR reflects all the information present on the port pins (PA0-PA7, PB0-PB7) for both input and output bits.

Miscellaneous Information

Handshaking

Handshaking on data transfers can be accomplished using the PC output pin and the FLAG input pin. PC will go low for one cycle following a read or write of port B. This signal can be used to indicate data ready at port B, or data accepted from port B. Handshaking on 16 bit data transfers (using both ports A and B) is possible by always reading or writing to port A first. FLAG is a negative edge sensitive

input which can be used for receiving the PC output from another 6526, or as a general purpose interrupt input. Any negative transition on FLAG will set the FLAG interrupt bit.

REG	NAME	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	PRA	PA_7	PA_6	PA_5	PA_4	PA_3	PA_2	PA_1	PA_0
1	PRB	PB_7	PB_6	PB_5	PB_4	PB_3	PB_2	PB_1	PB_0
2	DDRA	DPA_7	DPA_6	DPA_5	DPA_4	DPA_3	DPA_2	DPA_1	DPA_0
3	DDRB	DPB_7	DPB_6	DPB_5	DPB_4	DPB_3	DPB_2	DPB_1	DPB_0

Interval Timers A and B

Each interval timer consists of a 16 bit read only timer counter, and a 16 bit write only timer latch. Data written to the timer is latched in the timer latch, whilst data read from the timer are the present contents of the timer counter. These timers can be used independently or linked for extended operations. The various different timer modes allow generation of such diverse features as long time delays, variable width pulses, pulse trains, and many others. Using the CNT input, the timers can count external pulses or measure frequencies of external signals. Each has an associated control register, providing independent control of the following features : =

1) Start/Stop

A control bit allows the timer to be started or stopped by the 6510 at any time.

2) PB On/Off

A control bit allows the timer output to appear on a port B output line (PB6 for timer A, PB7 for timer B). This overrides the DDRB control bit and forces the appropriate PB line to an output.

3) Toggle/Pulse

A control bit selects the output applied to port B. On every timer underflow the output can either toggle or generate a single positive pulse of one cycle duration. The toggle output is set high whenever the timer is stared, and is set low by RES.

4) One Short/Continuous

A control bit selects either timer mode. In short mode the timer will

count down from the latched value to zero, generate an interrupt, reload the latched value, and stop. In continuous mode it doesn't stop, and just repeats the procedure continuously.

5) Force Load

A strobe bit allows the timer latch to be loaded into the timer counter at any time, whether that timer happens to be running or not.

6) Input Mode

Control bits allow selection of the clock used to decrement the counter. Timer A can count 02 clock pulses or external pulses applied to the CNT pin. Timer B can count 02 pulses, external CNT pulses, timer A underflow pulses, or timer A underflow pulses whilst CNT is high.

The timer latch is loaded into the timer on any timer underflow, on a force load, or following a write to the high byte of the pre-scaler while the timer is stopped. If the timer is running, a write to the high byte will load the timer latch, but not reload the counter.

READ (TIMER)

REG	NAME								
4	TA LO	TAL_7	TAL_6	TAL_5	TAL_4	TAL_3	TAL_2	TAL_1	TAL_0
5	TA HI	TAH_7	TAH_6	TAH_5	TAH_4	TAH_3	TAH_2	TAH_1	TAH_0
6	TB LO	TBL_7	TBL_6	TBL_5	TBL_4	TBL_3	TBL_2	TBL_1	TBL_0
7	TB HI	TBH_7	TBH_6	TBH_5	TBH_4	TBH_3	TBH_2	TBH_1	TBH_0

WRITE (PRESCALER)

REG	NAME								
4	TA LO	PAL_7	PAL_6	PAL_5	PAL_4	PAL_3	PAL_2	PAL_1	PAL_0
5	TA HI	PAH_7	PAH_6	PAH_5	PAH_4	PAH_3	PAH_2	PAH_1	PAH_0
6	TB LO	PBL_7	PBL_6	PBL_5	PBL_4	PBL_3	PBL_2	PBL_1	PBL_0
7	TB HI	PBH_7	PBH_6	PBH_5	PBH_4	PBH_3	PBH_2	PBH_1	PBH_0

Time of Day Clock (TOD)

This is a special purpose timer for real time applications. TOD consists of a 24 hour clock with resolution of one tenth of a second, organised into 4 registers: 10ths of seconds, seconds, minutes and hours. Often wondered how you worked, Mike!

The AM/PM flag in the SMB of the hours register allows for easy bit

testing. Each register reads out in BCD format, which makes it easy to convert for display purposes. A programmable alarm is also provided, for generating an interrupt at a required time. The alarm registers are contained in the same addresses as the corresponding TOD registers, with access to the alarm governed by a Control Register bit. The alarm is write only, and any read of a TOD address will read the time, regardless of the state of the alarm sequence bit.

A preset sequence must be followed to correctly set and read the TOD clock. TOD is automatically stopped whenever a write to the hours register occurs, and will not start again until after a write to the 10ths of second register. This assures that TOD will always start at the desired time. As a carry can occur from one stage to the next at any time with respect to a read operation, a latching function is included to keep all TOD information constant during a read function.

All four TOD registers latch on a read of hours, and remain latched until the 10ths of seconds is read. The TOD clock continues to count when the output registers are latched. If only one register is to be read, then there's no carry problem, provided that any read of hours is followed by a read of 10ths of seconds to disable the latching.

READ

REG	NAME								
8	TOD 10THS	0	0	0	0	T_8	T_4	T_2	T_1
9	TOD SEC	0	SH_4	SH_2	SH_1	SL_8	SL_4	SL_2	SL_1
A	TOD MIN	0	MH_4	MH_2	MH_1	ML_8	ML_4	ML_2	ML_1
B	TOD HR	PM	0	0	HH	HL_8	HL_4	HL_2	HL_1

WRITE
CRB7=0 TOD
CRB7=1 ALARM
(SAME FORMAT AS READ)

Serial Port - SDR

The serial port is a buffered, 8 bit synchronous shift register system. A control bit selects either input or output mode. On input, data on the SP pin is shifted into the shift register on the rising edge of the signal applied to the CNT pin. After eight CNT pulses, the data in the shift register is dumped into the Serial Data register, and an interrupt is generated. In output mode, timer A is used as the baud rate generator. Data is shifted out on the SP pin at half the underflow rate of timer A.

The maximum baud rate possible is 02 divided by 4, but the maximum

useable rate will be determined by line loading and the speed at which the receiver can respond to the input of data. Transmission commences after a write to the serial data register (SDR), assuming timer A is running and in continuous mode. The clock signal from timer A appears on the CNT pin as an output. THe data in the SDR will be loaded into the shift register, then moved out to the SP pin when a CNT pulse occurs. Data moved out becomes valid on the falling edge of the CNT, and remains valid until the next falling edge. After 8 CNT pulses, an interrupt is generated to indicate that more data can now be sent.

READ (INT DATA)

REG	NAME								
D	ICR	IR	0	0	FLG	SP	ALRM	TB	TA

WRITE (INT MASK)

REG	NAME								
D	ICR	S/C	X	X	FLG	SP	ALRM	TB	TA

Control Registers

REG	NAME	TOD IN	SP MODE	IN MODE	LOAD	RUN MODE	OUT MODE	PB ON	START
E	CRA	0=60Hz	0=INPUT	0=ϕ_2	1=FORCE LOAD	0=CONT.	0=PULSE	0=PB6 OFF	0=STOP
		1=50Hz	1=OUTPUT	1=CNT	(STROBE)	1=O.S.	1=TOGGLE	1=PB6 ON	1=START

TA

REG	NAME	ALARM	IN MODE		LOAD	RUN MODE	OUT MODE	PB ON	START
F	CRB	0=TOD	0	0=ϕ_2	1=FORCE LOAD	0=CONT.	0=PULSE	0=PB7 OFF	0=STOP
			0	1=CNT					
			1	0=TA					
		1=ALARM	1	1=CNT•TA	(STROBE)	1=O.S.	1=TOGGLE	1=PB7 ON	1=START

TB

Appendices - General Introduction

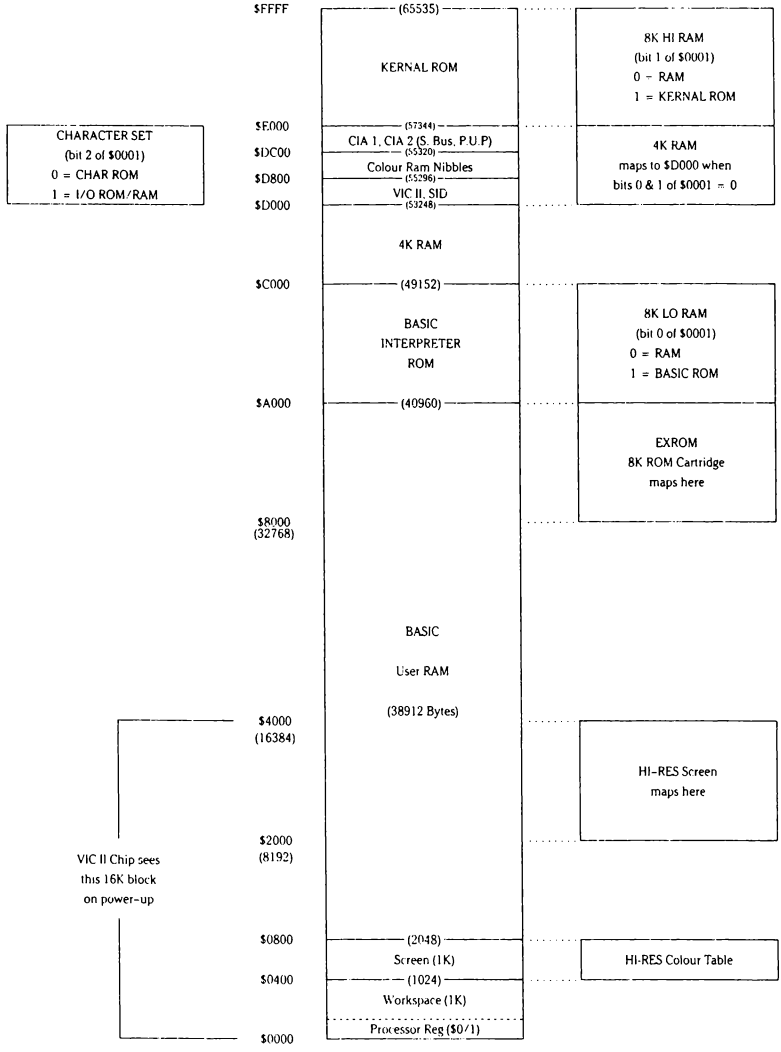
These appendices are an attempt to gather together, as far as possible, all the information the programmer will need that you will not find contained within the rest of this book.

As a result there will, of necessity, be some duplication between these pages and some of those in the User's Manual supplied with your 64.

However, if they save you diving about from book to book in an effort to find the particular bit of information that you require, then they will have served their purpose!

Appendix A : Commodore 64 Memory Maps

Commodore-64 Architecture Map



0000		0	Chip directional register
0001		1	Chip I/O; memory & tape control
0003	-0004	3-4	Float-Fixed vector
0005	-0006	5-6	Fixed-Float vector
0007		7	Search character
0008		8	Scan-quotes flag
0009		9	TAB column save
000A		10	0 = LOAD, 1 = VERIFY
000B		11	Input buffer pointer/# subscrpt
000C		12	Default DIM flag
000D		13	Type: FF = string, 00 = numeric
000E		14	Type: 80 = integer, 00 = floating point
000F		15	DATA scan/LIST quote/memry flag
0010		16	Subscript/FNx flag
0011		17	0 = INPUT;$40 = GET;$98 = READ
0012		18	ATN sign/Comparison eval flag
0013		19	Current I/O prompt flag
0014	-0015	20-21	Integer value
0016		22	Pointer: temporary string stack
0017	-0018	23-24	Last temp string vector
0019	-0021	25-33	Stack for temporary strings
0022	-0025	34-37	Utility pointer area
0026	-002A	38-42	Product area for multiplication
002B	-002C	43-44	Pointer: Start-of-Basic
002D	-002E	45-46	Pointer: Start-of-Variables
002F	-0030	47-48	Pointer: Start-of-Arrays
0031	-0032	49-50	Pointer: End-of-Arrays
0033	-0034	51-52	Pointer: String-storage(moving down)
0035	-0036	53-54	Utility string pointer
0037	-0038	55-56	Pointer: Limit-of-memory
0039	-003A	57-58	Current Basic line number
003B	-003C	59-60	Previous Basic line number
003D	-003E	61-62	Pointer: Basic statement for CONT
003F	-0040	63-64	Current DATA line number
0041	-0042	65-66	Current DATA address
0043	-0044	67-68	Input vector
0045	-0046	69-70	Current variable name
0047	-0048	71-72	Current variable address
0049	-004A	73-74	Variable pointer for FOR/NEXT
004B	-004C	75-76	Y-save; op-save; Basic pointer save
004D		77	Comparison symbol accumulator
004E	-0053	78-83	Misc work area, pointers, etc
0054	-0056	84-86	Jump vector for functions
0057	-0060	87-96	Misc numeric work area
0061		97	Accum#1: Exponent
0062	-0065	98-101	Accum#1: Mantissa
0066		102	Accum#1: Sign
0067		103	Series evaluation constant pointer
0068		104	Accum#1 hi-order (overflow)
0069	-006E	105-110	Accum#2: Exponent, etc.
006F		111	Sign comparison, Acc#1 vs #2

0070		112	Accum#1 lo-order (rounding)
0071	-0072	113-114	Cassette buff len/Series pointer
0073	-008A	115-138	CHRGET subroutine; get Basic char
007A	-007B	122-123	Basic pointer (within subrtn)
008B	-008F	139-143	RND seed value
0090		144	Status word ST
0091		145	Keyswitch PIA: STOP and RVS flags
0092		146	Timing constant for tape
0093		147	Load = 0, Verify = 1
0094		148	Serial output: deferred char flag
0095		149	Serial deferred character
0096		150	Tape EOT received
0097		151	Register save
0098		152	How many open files
0099		153	Input device, normally 0
009A		154	Output CMD device, normally 3
009B		155	Tape character parity
009C		156	Byte-received flag
009D		157	Direct = $80/RUN = 0 output control
009E		158	Tp Pass 1 error log/char buffer
009F		159	Tp Pass 2 err log corrected
00A0	-00A2	160-162	Jiffy Clock HML
00A3		163	Serial bit count/EOI flag
00A4		164	Cycle count
00A5		165	Countdown,tape write/bit count
00A6		166	Tape buffer pointer
00A7		167	Tp Wrt ldr count/Rd pass/inbit
00A8		168	Tp Wrt new byte/Rd error/inbit cnt
00A9		169	Wrt start bit/Rd bit err/stbit
00AA		170	Tp Scan;Cnt;Ld;End/byte assy
00AB		171	Wr lead length/Rd checksum/parity
00AC	-00AD	172-173	Pointer: tape bufr, scrolling
00AE	-00AF	174-175	Tape end adds/End of program
00B0	-00B1	176-177	Tape timing constants
00B2	-00B3	178-179	Pntr: start of tape buffer
00B4		180	1 = Tp timer enabled; bit count
00B5		181	Tp EOT/RS232 next bit to send
00B6		182	Read character error/outbyte buf
00B7		183	# characters in file name
00B8		184	Current logical file
00B9		185	Current secndy address
00BA		186	Current device
00BB	-00BC	187-188	Pointer to file name
00BD		189	Wr shift word/Rd input char
00BE		190	# blocks remaining to Wr/Rd
00BF		191	Serial word buffer
00C0		192	Tape motor interlock
00C1	-00C2	193-194	I/O start address
00C3	-00C4	195-196	Kernel setup pointer
00C5		197	Last key pressed
00C6		198	# chars in keybd buffer
00C7		199	Screen reverse flag
00C8		200	End-of-line for input pointer
00C9	-00CA	201-202	Input cursor log (row, column)
00CB		203	Which key: 64 if no key

00CC	204	0 = flash cursor
00CD	205	Cursor timing countdown
00CE	206	Character under cursor
00CF	207	Cursor in blink phase
00D0	208	Input from screen/from keyboard
00D1 -00D2	209-210	Pointer to screen line
00D3	211	Position of cursor on above line
00D4	212	0 = direct cursor, else programmed
00D5	213	Current screen line length
00D6	214	Row where curosr lives
00D7	215	Last inkey/checksum/buffer
00D8	216	# of INSERTs outstanding
00D9 -00F2	217-242	Screen line link table
00F3 -00F4	243-244	Screen color pointer
00F5 -00F6	245-246	Keyboard pointer
00F7 -00F8	247-248	RS-232 Rcv pntr
00F9 -00FA	249-250	RS-232 Tx pntr
00FF -010A	256-266	Floating to ASCII work area
0100 -013E	256-318	Tape error log
0100 -01FF	256-511	Processor stack area
0200 -0258	512-600	Basic input buffer
0259 -0262	601-610	Logical file table
0263 -026C	611-620	Device # table
026D -0276	621-630	Sec Adds table
0277 -0280	631-640	Keybd buffer
0281 -0282	641-642	Start of Basic Memory
0283 -0284	643-644	Top of Basic Memory
0285	645	Serial bus timeout flag
0286	646	Current color code
0287	647	Color under cursor
0288	648	Screen memory page
0289	649	Max size of keybd buffer
028A	650	Repeat all keys
028B	651	Repeat speed counter
028C	652	Repeat delay counter
028D	653	Keyboard Shift/Control flag
028E	654	Last shift pattern
028F -0290	655-656	Keyboard table setup pointer
0291	657	Keyboard shift mode
0292	658	0 = scroll enable
0293	659	RS-232 control reg
0294	660	RS-232 command reg
0295 -0296	661-662	Bit timing
0297	663	RS-232 status
0298	664	# bits to send
0299 -029A	665	RS-232 speed/code
029B	667	RS232 receive pointer
029C	668	RS232 input pointer
029D	669	RS232 transmit pointer
029E	670	RS232 output pointer
029F -02A0	671-672	IRQ save during tape I/O
02A1	673	CIA 2 (NMI) Interrupt Control
02A2	674	CIA 1 Timer A control log
02A3	675	CIA 1 Interrupt Log
02A4	676	CIA 1 Timer A enabled flag

02A5	677	Screen row marker	
02C0 -02FE	704-766	(Sprite 11)	
0300 -0301	768-769	Error message link	
0302 -0303	770-771	Basic warm start link	
0304 -0305	772-773	Crunch Basic tokens link	
0306 -0307	774-775	Print tokens link	
0308 -0309	776-777	Start new Basic code link	
030A -030B	778-779	Get arithmetic element link	
030C	780	SYS A-reg save	
030D	781	SYS X-reg save	
030E	782	SYS Y-reg save	
030F	783	SYS status reg save	
0310 -0312	784-785	USR function jump	(B248)
0314 -0315	788-789	Hardware interrupt vector	(EA31)
0316 -0317	790-791	Break interrupt vector	(FE66)
0318 -0319	792-793	NMI interrupt vector	(FE47)
031A -031B	794-795	OPEN vector	(F34A)
031C -031D	796-797	CLOSE vector	(F291)
031E -031F	798-799	Set-input vector	(F20E)
0320 -0321	800-801	Set-output vector	(F250)
0322 -0323	802-803	Restore I/O vector	(F333)
0324 -0325	804-805	INPUT vector	(F157)
0326 -0327	806-807	Output vector	(F1CA)
0328 -0329	808-809	Test-STOP vector	(F6ED)
032A -032B	810-811	GET vector	(F13E)
032C -032D	812-813	Abort I/O vector	(F32F)
032E -032F	814-815	Warm start vector	(FE66)
0330 -0331	816-817	LOAD link	(F4A5)
0332 -0333	818-819	SAVE link	(F5ED)
033C -03FB	828-1019	Cassette buffer	
0340 -037E	832-894	(Sprite 13)	
0380 -03BE	896-958	(Sprite 14)	
03C0 -03FE	960-1022	(Sprite 15)	
0400 -07FF	1024-2047	Screen memory	
0800 -9FFF	2048-40959	Basic RAM memory	
8000 -9FFF	32768-40959	Alternate: ROM plug-in area	
A000 -BFFF	40960-49151	ROM: Basic	
A000 -BFFF	49060-59151	Alternate: RAM	
C000 -CFFF	49152-53247	RAM memory, including alternate	
D000 -D02E	53248-53294	Video Chip (6566)	
D400 -D41C	54272-54300	Sound Chip (6581 SID)	
D800 -DBFF	55296-56319	Color nybble memory	
DC00 -DC0F	56320-56335	Interface chip 1, IRQ (6526 CIA)	
DD00 -DD0F	56576-56591	Interface chip 2, NMI (6526 CIA)	
D000 -DFFF	53248-53294	Alternate: Character set	
E000 -FFFF	57344-65535	ROM: Operating System	
E000 -FFFF	57344-65535	Alternate: RAM	
FF81 -FFF5	65409-65525	Jump Table, Including:	
FFC6		- Set Input channel	
FFC9		- Set Output channel	
FFCC		- Restore default I/O channels	
FFCF		- INPUT	
FFD2		- PRINT	
FFE1		- Test Stop key	
FFE4		- GET	

A000; ROM control vectors
A00C; Keyword action vectors
A052; Function vectors
A080; Operator vectors
A09E; Keywords
A19E; Error messages
A328; Error message vectors
A365; Misc messages
A38A; Scan stack for FOR/GOSUB
A3B8; Move memory
A3FB; Check stack depth
A408; Check memory space
A435; 'out of memory'
A437; Error routine
A469; BREAK entry
A474; 'ready.'
A480; Ready for Basic
A49C; Handle new line
A533; Re-chain lines
A560; Receive input line
A579; Crunch tokens
A613; Find Basic line
A642; Perform [NEW]
A65E; Perform [CLR]
A68E; Back up text pointer
A69C; Perform [LIST]
A742; Perform [FOR]
A7ED; Execute statement
A81D; Perform [RESTORE]
A82C; Break
A82F; Perform [STOP]
A831; Perform [END]
A857; Perform [CONT]
A871; Perform [RUN]
A883; Perform [GOSUB]
A8A0; Perform [GOTO]
A8D2; Perform [RETURN]
A8F8; Perform [DATA]
A906; Scan for next statement
A928; Perform [IF]
A93B; Perform [REM]
A94B; Perform [ON]
A96B; Get fixed point number
A9A5; Perform [LET]
AA80; Perform [PRINT#]
AA86; Perform [CMD]
AAA0; Perform [PRINT]
AB1E; Print string from (y.a)
AB3B; Print format character
AB4D; Bad input routine
AB7B; Perform [GET]
ABA5; Perform [INPUT#]
ABBF; Perform [INPUT]
ABF9; Prompt & input
AC06; Perform [READ]
ACFC; Input error messages
AD1E; Perform [NEXT]
AD78; Type match check
AD9E; Evaluate expression
AEA8; Constant - pi
AEF1; Evaluate within brackets
AEF7; ')'
AEFF; comma..
AF08; Syntax error
AF14; Check range
AF28; Search for variable
AFA7; Setup FN reference
AFE6; Perform [OR]
AFE9; Perform [AND]
B016; Compare
B081; Perform [DIM]
B08B; Locate variable
B113; Check alphabetic
B11D; Create variable
B194; Array pointer subrtine
B1A5; Value 32768
B1B2; Float-fixed
B1D1; Set up array
B245; 'bad subscript'
B248; 'illegal quantity'
B34C; Compute array size
B37D; Perform [FRE]
B391; Fix-float
B39E; Perform [POS]
B3A6; Check direct
B3B3; Perform [DEF]
B3E1; Check fn syntax
B3F4; Perform [FN]
B465; Perform [STR$]
B475; Calculate string vector
B487; Set up string
B4F4; Make room for string
B526; Garbage collection
B5BD; Check salvageability
B606; Collect string
B63D; Concatenate
B67A; Build string to memory
B6A3; Discard unwanted string
B6DB; Clean descriptor stack
B6EC; Perform [CHR$]
B700; Perform [LEFT$]
B72C; Perform [RIGHT$]
B737; Perform [MID$]
B761; Pull string parameters
B77C; Perform [LEN]
B782; Exit string-mode
B78B; Perform [ASC]
B79B; Input byte paramter
B7AD; Perform [VAL]
B7EB; Parameters for POKE/WAIT
B7F7; Float-fixed
B80D; Perform [PEEK]
B824; Perform [POKE]
B82D; Perform [WAIT]

B849; Add 0.5
B850; Subtract-from
B853; Perform [subtract]
B86A; Perform [add]
B947; Complement FAC#1
B97E; 'overflow'
B983; Multiply by zero byte
B9EA; Perform [LOG]
BA2B; Perform [multiply]
BA59; Multiply-a-bit
BA8C; Memory to FAC#2
BAB7; Adjust FAC#1/#2
BAD4; Underflow/overflow
BAE2; Multiply by 10
BAF9; +10 in floating pt
BAFE; Divide by 10
BB12; Perform [divide]
BBA2; Memory to FAC#1
BBC7; FAC#1 to memory
BBFC; FAC#2 to FAC#1
BC0C; FAC#1 to FAC#2
BC1B; Round FAC#1
BC2B; Get sign
BC39; Perform [SGN]
BC58; Perform [ABS]
BC5B; Compare FAC#1 to mem
BC9B; Float-fixed
BCCC; Perform [int]
BCF3; String to FAC
BD7E; Get ascii digit
BDC2; Print 'IN..'
BDCD; Print line number
BDDD; Float to ascii
BF16; Decimal constants
BF3A; TI constants
BF71; Perform [SQR]
BF7B; Perform [power]
BFB4; Perform [negative]
BFED; Perform [EXP]
E043; Series eval 1
E059; Series eval 2
E097; Perform [RND]
E0f9; ?? breakpoints ??
E12A; Perform [SYS]
E156; Perform [SAVE]
E165; Perform [VERIFY]
E168; Perform [LOAD]
E1BE; Perform [OPEN]
E1C7; Perform [CLOSE]
E1D4; Parameters for LOAD/SAVE
E206; Check default parameters
E20E; Check for comma
E219; Parameters for open/close
E264; Perform [COS]
E26B; Perform [SIN]
E2B4; Perform [TAN]
E30E; Perform [ATN]
E37B; Warm restart
E394; Initialize
E3A2; CHRGET for zero page
E3BF; Initialize Basic
E447; Vectors for $300
E453; Initialize vectors
E45F; Power-up message
E500; Get I/O address
E505; Get screen size
E50A; Put/get row/column
E518; Initialize I/O
E544; Clear screen
E566; Home cursor
E56C; Set screen pointers
E5A0; Set I/O defaults
E5B4; Input from keyboard
E632; Input from screen
E684; Quote test
E691; Setup screen print
E6B6; Advance cursor
E6ED; Retreat cursor
E701; Back into previous line
E716; Output to screen
E87C; Go to next line
E891; Perform <return>
E8A1; Check line decrement
E8B3; Check line increment
E8CB; Set color code
E8DA; Color code table
E8EA; Scroll screen
E965; Open space on screen
E9C8; Move a screen line
E9E0; Synchronize color transfer
E9F0; Set start-of-line
E9FF; Clear screen line
EA13; Print to screen
EA24; Synchronize color pointer
EA31; Interrupt - clock etc
EA87; Read keyboard
EB79; Keyboard select vectors
EB81; Keyboard 1 - unshifted
EBC2; Keyboard 2 - shifted
EC03; Keyboard 3 - 'comm'
EC44; Graphics/text contrl
EC4F; Set graphics/text mode
EC78; Keyboard 4
ECB9; Video chip setup
ECE7; Shift/run equivalent
ECF0; Screen ln address low
ED09; Send 'talk'
ED0C; Send 'listen'
ED40; Send to serial bus
EDB2; Serial timeout
EDB9; Send listen SA
EDBE; Clear ATN
EDC7; Send talk SA
EDCC; Wait for clock
EDDD; Send serial deferred
EDEF; Send 'untalk'

EDFE; Send 'unlisten'
EE13; Receive from serial bus
EE85; Serial clock on
EE8E; Serial clock off
EE97; Serial output '1'
EEA0; Serial output '0'
EEA9; Get serial in & clock
EEB3; Delay 1 ms
EEBB; RS-232 send
EF06; Send new RS-232 byte
EF2E; No-DSR error
EF31; No-CTS error
EF3B; Disable timer
EF4A; Compute bit count
EF59; RS232 receive
EF7E; Setup to receive
EFC5; Receive parity error
EFCA; Receive overflow
EFCD; Receive break
EFD0; Framing error
EFE1; Submit to RS232
F00D; No-DSR error
F017; Send to RS232 buffer
F04D; Input from RS232
F086; Get from RS232
F0A4; Check serial bus idle
F0BD; Messages
F12B; Print if direct
F13E; Get..
F14E; ..from RS232
F157; Input
F199; Get.. tape/serial/rs232
F1CA; Output..
F1DD; ..to tape
F20E; Set input device
F250; Set output device
F291; Close file
F30F; Find file
F31F; Set file values
F32F; Abort all files
F333; Restore default I/O
F34A; Do file open
F3D5; Send SA
F409; Open RS232
F49E; Load program
F5AF; 'searching'
F5C1; Print filename
F5D2; 'loading/verifying'
F5DD; Save program
F68F; Print 'saving'
F69B; Bump clock
F6BC; Log PIA key reading
F6DD; Get time
F6E4; Set time
F6ED; Check stop key
F6FB; Output error messages
F72D; Find any tape headr
F76A; Write tape header
F7D0; Get buffer address
F7D7; Set buffer start/end pointers
F7EA; Find specific header
F80D; Bump tape pointer
F817; 'press play..'
F82E; Check tape status
F838; 'press record..'
F841; Initiate tape read
F864; Initiate tape write
F875; Common tape code
F8D0; Check tape stop
F8E2; Set read timing
F92C; Read tape bits
FA60; Store tape chars
FB8E; Reset pointer
FB97; New character setup
FBA6; Send transition to tape
FBC8; Write data to tape
FBCD; IRQ entry point
FC57; Write tape leader
FC93; Restore normal IRQ
FCB8; Set IRQ vector
FCCA; Kill tape motor
FCD1; Check r/w pointer
FCDB; Bump r/w pointer
FCE2; Power reset entry
FD02; Check 8-rom
FD10; 8-rom mask
FD15; Kernal reset
FD1A; Kernal move
FD30; Vectors
FD50; Initialize system constnts
FD9B; IRQ vectors
FDA3; Initialize I/O
FDDD; Enable timer
FDF9; Save filename data
FE00; Save file details
FE07; Get status
FE18; Flag status
FE1C; Set status
FE21; Set timeout
FE25; Read/set top of memory
FE27; Read top of memory
FE2D; Set top of memory
FE34; Read/set bottom of memory
FE43; NMI entry
FE66; Warm start
FEB6; Reset IRQ & exit
FEBC; Interrupt exit
FEC2; RS-232 timing table
FED6; NMI RS-232 in
FF07; NMI RS-232 out
FF43; Fake IRQ
FF48; IRQ entry
FF81; Jumbo jump table
FFFA; Hardware vectors

Processor I/O Port (6510)

$0000	IN	IN	OUT	IN	OUT	OUT	OUT	OUT	DDR 0
$0001			Tape Motor	Tape Sense	Tape Write	D-ROM Switch	EF RAM Switch	AB RAM Switch	PR 1

SID (6581)

Voice 1	Voice 2	Voice 3		Voice 1	Voice 2	Voice 3
$D400	$D407	$D40E	Frequency L	54272	54279	54286
$D401	$D408	$D40F	Frequency H	54273	54280	54287
$D402	$D409	$D410	Pulse Width L	54274	54281	54288
$D403	$D40A	$D411	0 0 0 0 H	54275	54282	54289
$D404	$D40B	$D412	Voice Type: NSE PUL SAW TRI Key	54276	54283	54290
$D405	$D40C	$D413	Attack Time 2ms – 8ms / Decay Time 6ms – 24 sec	54277	54284	54291
$D406	$D40D	$D414	Sustain Level / Release Time 6ms – 24 sec	54278	54285	54292

Voices (write only)

$D415	0 0 0 0 0 L	54293
$D416	Filter Frequency H	54294
$D417	Resonance / Filter Voices: Ext V3 V2 V1	54295
$D418	Passband: V3 off HI BP LO / Master Volume	54296

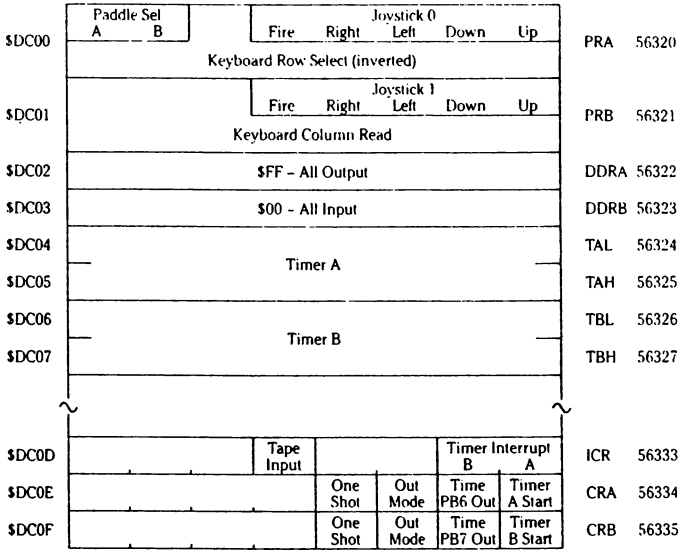
Filter & Volume (write only)

$D419	Paddle X (A/D #1)	54297
$D41A	Paddle Y (A/D #2)	54298
$D41B	Noise 3 (random)	54299
$D41C	Envelope 3	54300

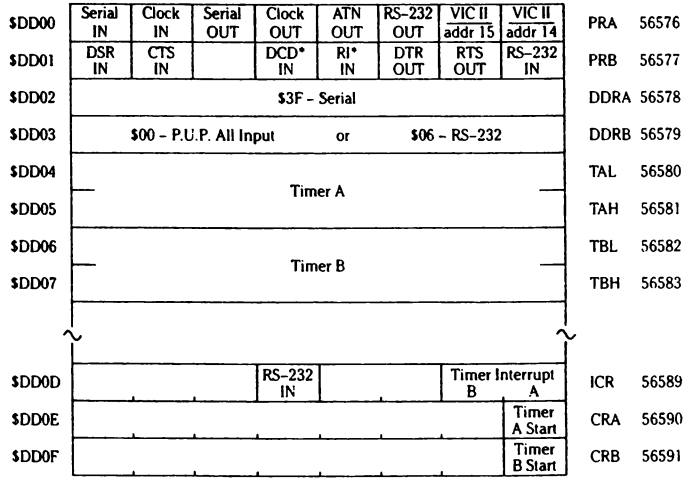
Sense (read only)

Note: Special Voice Features
(TEST, RING MOD, SYNC)
are omitted from the above diagram.

CIA 1 (IRQ) (6526)



CIA 2 (NMI) (6526)



* Connected but not used by O.S.

Appendix B : The Basic Language

Basic Keywords

SIMPLE VARIABLES

Type	Name	Range
Real	XY	±1.70141183E+38
		±2.93873588E−39
Integer	XY%	±32767
String	XY$	0 to 255 characters

X is a letter (A-Z), Y is a letter or number (0-9). Variable names can be more than 2 characters, but only the first two are recognized.

ARRAY VARIABLES

Type	Name
Single Dimension	XY(5)
Two-Dimension	XY(5,5)
Three-Dimension	XY(5,5,5)

Arrays of up to eleven elements (subscripts 0-10) can be used where needed. Arrays with more than eleven elements need to be DIMensioned.

ALGEBRAIC OPERATORS

= Assigns value to variable
− Negation
↑ Exponentiation
* Multiplication
/ Division
+ Addition
− Subtraction

RELATIONAL AND LOGICAL OPERATORS

= Equal
<> Not Equal To
< Less Than
> Greater Than
<= Less Than or Equal To
>= Greater Than or Equal To
NOT Logical "Not"
AND Logical "And"
OR Logical "Or"
Expression equals 1 if true, 0 if false.

SYSTEM COMMANDS

LOAD "NAME"	Loads a program from tape
SAVE "NAME"	Saves a program on tape
LOAD "NAME",8	Loads a program from disk
SAVE "NAME",8	Saves a program to disk
VERIFY "NAME"	Verifies that program was SAVEd without errors
RUN	Executes a program
RUN xxx	Executes program starting at line xxx
STOP	Halts execution
END	Ends execution
CONT	Continues program execution from line where program was halted
PEEK(X)	Returns contents of memory location X
POKE X,Y	Changes contents of location X to value Y
SYS xxxxx	Jumps to execute a machine language program, starting at xxxxx
WAIT X,Y,Z	Program waits until contents of location X, when FORed with Z and ANDed with Y, is nonzero.
USR(X)	Passes value of X to a machine language subroutine

EDITING AND FORMATTING COMMANDS

LIST	Lists entire program
LIST A-B	Lists from line A to line B
REM Message	Comment message can be listed but is ignored during program execution
TAB(X)	Used in PRINT statements. Spaces X positions on screen
SPC(X)	PRINTs X blanks on line
POS(X)	Returns current cursor position
CLR/HOME	Positions cursor to left corner of screen
SHIFT CLR/HOME	Clears screen and places cursor in "Home" position
SHIFT INST/DEL	Inserts space at current cursor position
INST/DEL	Deletes character at current cursor position
CTRL	When used with numeric color key, selects text color. May be used in PRINT statement.
CRSR Keys	Moves cursor up, down, left, right on screen
Commodore Key	When used with SHIFT selects between upper/lower case and graphic display mode. When used with numeric color key, selects optional text color

ARRAYS AND STRINGS

DIM A(X,Y,Z)	Sets maximum subscripts for A; reserves space for (X+1)*(Y+1)*(Z+1) elements starting at A(0,0,0)
LEN (X$)	Returns number of characters in X$
STR$(X)	Returns numeric value of X, converted to a string
VAL(X$)	Returns numeric value of A$, up to first nonnumeric character
CHR$(X)	Returns ASCII character whose code is X
ASC(X$)	Returns ASCII code for first character of X$
LEFT$(A$,X)	Returns leftmost X characters of A$
RIGHT$(A$,X)	Returns rightmost X characters of A$
MID$(A$,X,Y)	Returns Y characters of A$ starting at character X

INPUT/OUTPUT COMMANDS

INPUT A$ OR A	PRINTs '?' on screen and waits for user to enter a string or value
INPUT "ABC";A	PRINTs message and waits for user to enter value. Can also INPUT A$
GET A$ or A	Waits for user to type one-character value; no RETURN needed
DATA A,"B",C	Initializes a set of values that can be used by READ statement
READ A$ or A	Assigns next DATA value to A$ or A
RESTORE	Resets data pointer to start READing the DATA list again
PRINT "A= ";A	PRINTs string 'A= ' and value of A ';' suppresses spaces - ',' tabs data to next field.

PROGRAM FLOW

GOTO X	Branches to line X
IF A=3 THEN 10	IF assertion is true THEN execute following part of statement. IF false, execute next line number
FOR A=1 TO 10 STEP 2 : NEXT	Executes all statements between FOR and corresponding NEXT, with A going from 1 to 10 by 2. Step size is 1 unless specified
NEXT A	Defines end of loop. A is optional
GOSUB 2000	Branches to subroutine starting at line 2000
RETURN	Marks end of subroutine. Returns to statement following most recent GOSUB
ON X GOTO A,B	Branches to Xth line number on list. If X = 1 branches to A, etc.
ON X GOSUB A,B	Branches to subroutine at Xth line number in list

Abbreviations for Keywords

Command	Abbreviation	Looks like this on screen
ABS	A SHIFT B	A
AND	A SHIFT	A
ASC	A SHIFT S	A
ATN	A SHIFT T	A
CHR$	C SHIFT H	C
CLOSE	CL SHIFT O	CL
CLR	C SHIFT L	C
CMD	C SHIFT M	C
CONT	C SHIFT O	C
DATA	D SHIFT A	D
DEF	D SHIFT E	D
DIM	D SHIFT I	D
END	E SHIFT N	E
EXP	E SHIFT X	E
FOR	F SHIFT O	F
FRE	F SHIFT R	F
GET	G SHIFT E	G
GOSUB	GO SHIFT S	GO
GOTO	SHIFT O	G
INPUT#	I SHIFT N	I
LET	L SHIFT E	L
LEFT$	LE SHIFT F	LE
LIST	L SHIFT I	L
LOAD	L SHIFT O	L
MID$	M SHIFT I	M
NEXT	N SHIFT E	N
NOT	N SHIFT O	N

Command	Abbreviation	Looks like this on screen
OPEN	O SHIFT P	O
PEEK	P SHIFT E	P
POKE	P SHIFT O	P
PRINT	?	?
PRINT#	P SHIFT R	P
READ	R SHIFT E	R
RESTORE	RE SHIFT S	RE
RETURN	RE SHIFT T	RE
RIGHT$	R SHIFT I	R
RND	R SHIFT N	R
RUN	R SHIFT	R
SAVE	S SHIFT A	S
SGN	S SHIFT G	S
SIN	S SHIFT I	S
SPC(	S SHIFT P	S
SQR	S SHIFT Q	S
STEP	ST SHIFT E	ST
STOP	S SHIFT T	S
STR$	ST SHIFT R	ST
SYS	S SHIFT Y	S
TAB	T SHIFT A	T
THEN	T SHIFT H	T
USR	U SHIFT S	U
VAL	V SHIFT A	V
VERIFY	V SHIFT E	V
WAIT	W SHIFT A	W

Error Messages

BAD DATA String data was received from an open file, but the program was expecting numeric data.

BAD SUBSCRIPT The program was trying to reference an element of an array whose number is outside of the range specified in the DIM statement.

CAN'T CONTINUE The CONT command will not work, either because the program was never RUN, there has been an error, or a line has been edited.

DEVICE NOT PRESENT The required I/O device was not available for an OPEN, CLOSE, CMD, PRINT#, INPUT#, or GET#.

DIVISION BY ZERO Division by zero is a mathematical oddity and not allowed.

EXTRA IGNORED Too many items of data were typed in response to an INPUT statement. Only the first few items were accepted.

FILE NOT FOUND If you were looking for a file on tape, and END-OF-TAPE marker was found. If you were looking on disk, no file with that name exists.

FILE NOT OPEN The file specified in a CLOSE, CMD, PRINT#, INPUT#, or GET#, must first be OPENed.

FILE OPEN An attempt was made to open a file using the number of an already open file.

FORMULA TOO COMPLEX The string expression being evaluated should be split into at least two parts for the system to work with.

ILLEGAL DIRECT The INPUT statement can only be used within a program, and not in direct mode.

ILLEGAL QUANTITY A number used as the argument of a function or statement is out of the allowable range.

LOAD There is a problem with the program on tape.

NEXT WITHOUT FOR This is caused by either incorrectly nesting loops or having a variable name in a NEXT statement that doesn't correspond with one in a FOR statement.

NOT INPUT FILE An attempt was made to INPUT or GET data from a file which was specified to be for output only.

NOT OUTPUT FILE An attempt was made to PRINT data to a file which was specified as input only.

OUT OF DATA A READ statement was executed but there is no data left unREAD in a DATA statement.

OUT OF MEMORY There is no more RAM available for program or variables. This may also occur when too many FOR loops have been nested, or when there are too many GOSUBs in effect.

OVERFLOW The result of a computation is larger than the largest number allowed, which is 1.70141884E+38.

REDIM'D ARRAY An array may only be DIMensioned once. If an array variable is used before that array is DIM'd, an automatic DIM operation is performed on that array setting the number of elements to ten, and any subsequent DIMs will cause this error.

REDO FROM START Character data was typed in during an INPUT statement when numeric data was expected. Just re-type the entry so that it is correct, and the program will continue by itself.

RETURN WITHOUT GOSUB A RETURN statement was encountered, and no GOSUB command has been issued.

STRING TOO LONG A string can contain up to 255 characters.

?SYNTAX ERROR A statement is unrecognizable by the Commodore 64. A missing or extra parenthesis, misspelled keywords, etc.

TYPE MISMATCH This error occurs when a number is used in place of a string, or vice-versa.

UNDEF'D FUNCTION A user defined function was referenced, but it has never been defined using the DEF FN statement.

UNDEF'D STATEMENT An attempt was made to GOTO or GOSUB or RUN a line number that doesn't exist.

VERIFY The program on tape or disk does not match the program currently in memory.

Basic Timings

TIMING TABLES

BASIC STATEMENTS

CONSTRUCT	APPROX TIME (MILLISEC)
FRE	1 to 10
PEEK,POKE	1
TI$	3 to 4
TI	1
GET	1 to infinity
POS	1
PRINT X or PRINT	15 to 19
PRINT X$;	14 + LEN (X$)/2
READ X and DATA 3	9
REM	0.2 to 2
RESTORE	0.3
TAB	2
SPC(N)	1 + 0.6*N
FOR 1 = ... NEXT 1	4.0 + (1.6 each)
STEP	1.3
IF	0.4
GOTO or GOSUB	1.1
ON A GOTO or GOSUB	
L L	0.5 + (0.3*A) + (0.2*M)
RETURN	0.9

Using colon,:, saves 0.6 over new line.
SAVE or LOAD
15 sec + (2 sec per 100 char)
i.e. 500 baud.

STRING FUNCTIONS

FUNCTION	APPROX TIME (MILLISEC)
+	0.5 + (0.2 per char)
ASC	1
CHR$	1.2
LEFT$, RIGHT$	3 + (0.025 per char)
LEN	0 to 8
MID$	4 + (0.025 per char)
STR$	7 to 10
VAL	1.3
=,<>,<=,>,>=	3 to 4

ARITHMETIC FUNCTIONS

FUNCTION	APPROX TIME (MILLISEC)
ABS	0.6
ATN	42
COS	27
EXP	27
INT	1.2
LOG	23
RND RND (-1)	1.0
RND (0)	0.9
RND (1)	4.1
SGN	1.1
SIN	25
TAN	50
user FN	2.4

ARITHMETIC OPERATORS

SYMBOL	APPROX TIME (MILLISEC)
0 B,1 B	0.3
2 B	32
else	50 to 100
/ 0/B,A/1	0.5
else	2 to 5
* 0*B, A*0	0.4
else	1.5 to 3
+	0.3 to 1
-	0.3 to 1
=,<>,<,<=,>,>=	0.7
AND, OR	1.7
NOT	1.4

VARIABLES AND CONSTANTS

ITEM	APPROX TIME (MILLISEC)
A,A$,A=,A$=	0.7 to (0.7 + nv*0.1) nv = no. of variables in program
AA,AA$,AA=,AA$=	0.2 more than above
A%	0.3 more than above
A%=	0.6 more than above
999	1 per digit
.999	0.7 + (4.2 per digit)
E16	0.2 + (0.4* exponent)
E-16	0.2 + (3.0* exponent)
"ABCDE"	(0.6 to 0.7) + 0.02 per char)
M (I,J....)	(1 to 1.5)* (no. of subscripts)

```
10 REM ****** TIMING PROGRAM *****
100 N=50
110 T1=TI
120 FOR I=1 TO N
130 REM
140 NEXT I
150 T2=TI
160 FOR I=1 TO N
170 NEXT I
180 T3=TI:REM TIME TAKEN TO DO LOOP
190 PRINT"[CLR,6CD,6CR]TIME  = "1000*((T2-T1)-(T3-
T2))/60/N
200 END
```

BASIC 1Ø28 (1/Ø buffers, tables etc)
each statement
    4 for line number and following space, regardless for the line number
    1 for each BASIC keyword
    1 for each character, including RETURN
each variable with a value assigned, regardless of spelling or value takes 7 bytes; for string variables, add the length of the string
each array (N.B., size includes Øth element) take f* (size + 1) + (2 per dimension) where f=5 for floating point arrays, f=2 for integer arrays, and f=3 for string arrays.
The system slows down noticeably when memory is nearly full.

Appendix C : Machine Code Instruction Set

The following notation applies to this summary:

A	Accumulator
X, Y	Index registers
M	Memory
P	Processor status register
S	Stack Pointer
√	Change
–	No change
+	Add
∧	Logical AND
−	Subtract
V	Logical Exclusive-OR
→, ←	Transfer to
∀	Logical (inclusive) OR
PC	Program counter
PCH	Program counter high
PCL	Program counter low
#dd	8-bit immediate data value (2 hexadecimal digits)
aa	8-bit zero page address (2 hexadecimal digits)
aaaa	16-bit absolute address (4 hexadecimal digits)
↑	Transfer from stack (Pull)
↓	Transfer onto stack (Push)

ADC

Add to Accumulator with Carry

Operation: A + M + C → A, C

N Z C I D V
✓ ✓ ✓ – – ✓

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	ADC	#dd	69	2	2
Zero Page	ADC	aa	65	2	3
Zero Page, X	ADC	aa,X	75	2	4
Absolute	ADC	aaaa	6D	3	4
Absolute, X	ADC	aaaa,X	7D	3	4*
Absolute, Y	ADC	aaaa,Y	79	3	4*
(Indirect, X)	ADC	(aa,X)	61	2	6
(Indirect), Y	ADC	(aa),Y	71	2	5*

*Add 1 if page boundary is crossed.

AND

AND Memory with Accumulator

Logical AND to the accumulator
Operation: A ∧ M → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	AND	#dd	29	2	2
Zero Page	AND	aa	25	2	3
Zero Page, X	AND	aa,X	35	2	4
Absolute	AND	aaaa	2D	3	4
Absolute, X	AND	aaaa,X	3D	3	4*
Absolute, Y	AND	aaaa,Y	39	3	4*
(Indirect, X)	AND	(aa,X)	21	2	6
(Indirect), Y	AND	(aa),Y	31	2	5*

*Add 1 if page boundary is crossed.

ASL

Accumulator Shift Left

Operation: C ← [7][6][5][4][3][2][1][0] ← 0

N Z C I D V
✓ ✓ ✓ – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ASL A	0A	1	2
Zero Page	ASL aa	06	2	5
Zero Page, X	ASL aa,X	16	2	6
Absolute	ASL aaaa	0E	3	6
Absolute, X	ASL aaaa,X	1E	3	7

BCC

Branch on Carry Clear

Operation: Branch on C = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCC aa	90	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BCC aaaa), and convert it to a relative address.

BCS

Branch on Carry Set

Operation: Branch on C = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCS aa	B0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to next page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BCS aaaa), and convert it to a relative address.

Branch on Result Equal to Zero

Operation: Branch on Z = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BEQ aa	F0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to next page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BEQ aaaa), and convert it to a relative address.

BIT

Test Bits in Memory with Accumulator

Operation: A M, $M_7 \rightarrow N$, $M_6 \rightarrow V$
Bit 6 and 7 are transferred to the Status Register. If the result of A M is zero then Z = 1, otherwise Z = 0

N Z C I D V
M_7 ✓ – – – M_6

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	BIT aa	24	2	3
Absolute	BIT aaaa	2C	3	4

BMI

Branch on Result Minus

Operation: Branch on N = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BMI aa	30	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BMI aaaa), and convert it to a relative address.

BNE

Branch on Result Not Equal to Zero

Operation: Branch on Z = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BNE aa	D0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BNE aaaa), and convert it to a relative address.

BPL

Branch on Result Plus

Operation: Branch on N = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BPL aa	10	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BPL aaaa), and convert it to a relative address.

BRK

Force Break

Operation: Forced Interrupt PC + 2 ↓ P ↓

B N Z C I D V
1 – – – 1 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	BRK	00	1	7

BVC

Branch on Overflow Clear

Operation: Branch on V = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVC aa	50	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BVC aaaa), and convert it to a relative address.

BVS

Branch on Overflow Set

Operation: Branch on V = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVS aa	70	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BVS aaaa), and convert it to a relative address.

CLC

Clear Carry Flag

Operation: 0 → C

N Z C I D V
– – 0 – – –

Addressing Mode	Assembly Language Form	OP CODE	No Bytes	No. Cycles
Implied	CLC	18	1	2

CLD

Clear Decimal Mode

Operation: 0 → D

N Z C I D V
– – – – 0 –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLI

Clear Interrupt Disable Bit

Operation: 0 → I

N Z C I D V
– – – 0 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLV

Clear Overflow Flag

Operation: 0 → V

N Z C I D V
– – – – – 0

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CMP

Compare Memory and Accumulator

Operation: A − M

N Z C I D V
√ √ √ − − −

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	CMP	#dd	C9	2	2
Zero Page	CMP	aa	C5	2	3
Zero Page, X	CMP	aa,X	D5	2	4
Absolute	CMP	aaaa	CD	3	4
Absolute, X	CMP	aaaa,X	DD	3	4*
Absolute, Y	CMP	aaaa,Y	D9	3	4*
(Indirect, X)	CMP	(aa,X)	C1	2	6
(Indirect), Y	CMP	(aa),Y	D1	2	5*

*Add 1 if page boundary is crossed.

CPX

Compare Memory and Index X

Operation: X − M

N Z C I D V
√ √ √ − − −

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	CPX	#dd	E0	2	2
Zero Page	CPX	aa	E4	2	3
Absolute	CPX	aaaa	EC	3	4

CPY

Compare Memory and Index Y

Operation: Y − M

N Z C I D V
√ √ √ − − −

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	CPY	#dd	C0	2	2
Zero Page	CPY	aa	C4	2	3
Absolute	CPY	aaaa	CC	3	4

DEC

Decrement Memory by One

Operation: M − 1 → M

N Z C I D V
√ √ − − − −

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	DEC aa	C6	2	5
Zero Page, X	DEC aa,X	D6	2	6
Absolute	DEC aaaa	CE	3	6
Absolute, X	DEC aaaa,X	DE	3	7

DEX

Decrement Index X by One

Operation: X − 1 → X

N Z C I D V
√ √ − − − −

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEX	CA	1	2

DEY

Decrement Index Y by One

Operation: Y − 1 → Y

N Z C I D V
√ √ − − − −

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR

Exclusive-OR Memory with Accumulator

Operation: A V M → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	EOR	#dd	49	2	2
Zero Page	EOR	aa	45	2	3
Zero Page, X	EOR	aa,X	55	2	4
Absolute	EOR	aaaa	4D	3	4
Absolute, X	EOR	aaaa,X	5D	3	4*
Absolute, Y	EOR	aaaa,Y	59	3	4*
(Indirect, X)	EOR	(aa,X)	41	2	6
(Indirect), Y	EOR	(aa),Y	51	2	5*

*Add 1 if page boundary is crossed.

INC

Increment Memory by One

Operation: M + 1 → M

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Zero Page	INC	aa	E6	2	5
Zero Page, X	INC	aa,X	F6	2	6
Absolute	INC	aaaa	EE	3	6
Absolute, X	INC	aaaa,X	FE	3	7

INX

Increment Index X by One

Operation: X + 1 → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY

Increment Index Y by One

Operation: Y + 1 → Y

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP Code	No. Bytes	No. Cycles
Implied	INY	C8	1	2

JMP

Jump

Operation: (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JMP aaaa	4C	3	3
Indirect	JMP (aaaa)	6C	3	5

JSR

Jump to Subroutine

Operation: PC + 2↓, (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JSR aaaa	20	3	6

LDA

Load Accumulator with Memory

Operation: M → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDA	#dd	A9	2	2
Zero Page	LDA	aa	A5	2	3
Zero Page, X	LDA	aa,X	B5	2	4
Absolute	LDA	aaaa	AD	3	4
Absolute, X	LDA	aaaa,X	BD	3	4*
Absolute, Y	LDA	aaaa,Y	B9	3	4*
(Indirect, X)	LDA	(aa,X)	A1	2	6
(Indirect), Y	LDA	(aa),Y	B1	2	5*

*Add 1 if page boundary is crossed.

LDX

Load Index X with Memory

Operation: M → X

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDX	#dd	A2	2	2
Zero Page	LDX	aa	A6	2	3
Zero Page, Y	LDX	aa,Y	B6	2	4
Absolute	LDX	aaaa	AE	3	4
Absolute, Y	LDX	aaaa,Y	BE	3	4*

*Add 1 when page boundary is crossed.

LDY

Load Index Y with Memory

Operation: M → Y

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDY	#dd	A0	2	2
Zero Page	LDY	aa	A4	2	3
Zero Page, X	LDY	aea,X	B4	2	4
Absolute	LDY	aaaa	AC	3	4
Absolute, X	LDY	aaaa,X	BC	3	4*

*Add 1 when page boundary is crossed.

LSR

Local Shift Right

Operation: 0 → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | → C

N Z C I D V
0 √ √ – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Accumulator	LSR	A	4A	1	2
Zero Page	LSR	aa	46	2	5
Zero Page, X	LSR	aa,X	56	2	6
Absolute	LSR	aaaa	4E	3	6
Absolute, X	LSR	aaaa,X	5E	3	7

NOP

No Operation

Operation: No Operation (2 cycles)

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	NOP	EA	1	2

ORA

OR Memory with Accumulator

Operation: A V M → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	ORA	#dd	09	2	2
Zero Page	ORA	aa	05	2	3
Zero Page, X	ORA	aa,X	15	2	4
Absolute	ORA	aaaa	0D	3	4
Absolute, X	ORA	aaaa,X	1D	3	4*
Absolute, Y	ORA	aaaa,Y	19	3	4*
(Indirect, X)	ORA	(aa,X)	01	2	6
(Indirect), Y	ORA	(aa),Y	11	2	5*

*Add 1 on page crossing.

PHA

Push Accumulator on Stack

Operation: A ↓

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHA	48	1	3

PHP

Push Processor Status on Stack

Operation: P ↓

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHP	08	1	3

PLA

Pull Accumulator from Stack

Operation: A ↑

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP

Pull Processor Status from Stack

Operation: P ↑

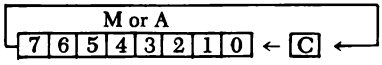
N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL

Rotate Left

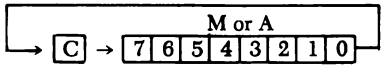
Operation:



N Z C I D V
✓ ✓ ✓ – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Accumulator	ROL	A	2A	1	2
Zero Page	ROL	aa	26	2	5
Zero Page, X	ROL	aa,X	36	2	6
Absolute	ROL	aaaa	2E	3	6
Absolute, X	ROL	aaaa,X	3E	3	7

Rotate Right

Operation: 

N Z C I D V
✓ ✓ ✓ – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Accumulator	ROR	A	6A	1	2
Zero Page	ROR	aa	66	2	5
Zero Page, X	ROR	aa,X	76	2	6
Absolute	ROR	aaaa	6E	3	6
Absolute, X	ROR	aaaa,X	7E	3	7

RTI

Return from Interrupt

Operation: P↑ PC↑

N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTI	40	1	6

RTS

Return from Subroutine

Operation: PC↑, PC + 1 → PC

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTS	60	1	6

SBC

Subtract from Accumulator with Carry

Operation: A − M − $\overline{C}$ → A

Note: $\overline{C}$ = Borrow

N Z C I D V
✓ ✓ ✓ – – ✓

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	SBC #dd	E9	2	2
Zero Page	SBC aa	E5	2	3
Zero Page, X	SBC aa,X	F5	2	4
Absolute	SBC aaaa	ED	3	4
Absolute, X	SBC aaaa,X	FD	3	4*
Absolute, Y	SBC aaaa,Y	F9	3	4*
(Indirect, X)	SBC (aa,X)	E1	2	6
(Indirect), Y	SBC (aa),Y	F1	2	5*

*Add 1 when page boundary is crossed.

SEC

Set Carry Flag

Operation: 1 → C

N Z C I D V
– – 1 – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEC	38	1	2

SED

Set Decimal Mode

Operation: 1 → D

N Z C I D V
– – – – 1 –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI

Set Interrupt Disable Status

Operation: 1 → I

N Z C I D V
– – – 1 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA

Store Accumulator in Memory

Operation: A → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STA aa	85	2	3
Zero Page, X	STA aa,X	95	2	4
Absolute	STA aaaa	8D	3	4
Absolute, X	STA aaaa,X	9D	3	5
Absolute, Y	STA aaaa,Y	99	3	5
(Indirect, X)	STA (aa,X)	81	2	6
(Indirect), Y	STA (aa),Y	91	2	6

STX

Store Index X in Memory

Operation: X → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STX aa	86	2	3
Zero Page, Y	STX aa,Y	96	2	4
Absolute	STX aaaa	8E	3	4

STY

Store Index Y in Memory

Operation: Y → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STY aa	84	2	3
Zero Page, X	STY aa,X	94	2	4
Absolute	STY aaaa	8C	3	4

TAX

Transfer Accumulator to Index X

Operation: A → X

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAX	AA	1	2

TAY

Transfer Accumulator to Index Y

Operation: A → Y

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAY	A8	1	2

TSX

Transfer Stack Pointer to Index X

Operation: S → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TSX	BA	1	2

TXA

Transfer Index X to Accumulator

Operation: X → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXA	8A	1	2

TXS

Transfer Index X to Stack Pointer

Operation: X → S

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXS	9A	1	2

TYA

Transfer Index Y to Accumulator

Operation: Y → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TYA	98	1	2

Appendix D : Miscellaneous

Screen Display Codes

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	[		27	6		54
A	a	1	£		28	7		55
B	b	2	]		29	8		56
C	c	3	↑		30	9		57
D	d	4	←		31			58
E	e	5	SPACE		32	;		59
F	f	6	!		33	<		60
G	g	7	"		34	=		61
H	h	8	#		35	>		62
I	i	9	$		36	?		63
J	j	10	%		37			64
K	k	11	&		38	♠	A	65
L	l	12	'		39		B	66
M	m	13	(		40		C	67
N	n	14	)		41		D	68
O	o	15	*		42		E	69
P	p	16	+		43		F	70
Q	q	17	,		44		G	71
R	r	18	–		45		H	72
S	s	19			46		I	73
T	t	20	/		47		J	74
U	u	21	0		48		K	75
V	v	22	1		49		L	76
W	w	23	2		50		M	77
X	x	24	3		51		N	78
Y	y	25	4		52		O	79
Z	z	26	5		53		P	80

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
	Q	81			97			113
	R	82			98			114
	S	83			99			115
	T	84			100			116
	U	85			101			117
	V	86			102			118
	W	87			103			119
	X	88			104			120
	Y	89			105			121
	Z	90			106			122
		91			107			123
		92			108			124
		93			109			125
		94			110			126
		95			111			127
SPACE		96			112			

Codes from 128-255 are reversed images of codes 0-127.

Ascii Values

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0	CRSR ↓	17	"	34	3	51
	1	RVS ON	18	#	35	4	52
	2	CLR HOME	19	$	36	5	53
	3	INST DEL	20	%	37	6	54
	4		21	&	38	7	55
WHT	5		22		39	8	56
	6		23	(	40	9	57
	7		24	)	41		58
DISABLES SHIFT C=	8		25	*	42	;	59
ENABLES SHIFT C=	9		26	+	43	<	60
	10		27	,	44	=	61
	11	RED	28	-	45	>	62
	12	CRSR →	29		46	?	63
RETURN	13	GRN	30	/	47	@	64
SWITCH TO LOWER CASE	14	BLU	31	0	48	A	65
	15	SPACE	32	1	49	B	66
	16	!	33	2	50	C	67

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
D	68		97		126		155
E	69		98		127		156
F	70		99		128		157
G	71		100		129		158
H	72		101		130		159
I	73		102		131		160
J	74		103		132		161
K	75		104	f1	133		162
L	76		105	f3	134		163
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		149		178
£	92		121		150		179
]	93		122		151		180
↑	94		123		152		181
←	95		124		153		182
	96		125		154		183

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	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

Dec/Hex/Binary Conversions

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

These neat little programs will allow you to convert from hex to decimal and back again, and from decimal to binary, although the latter is restricted in size to numbers lying between 0 and 225.

1) Decimal to Hex, where D contains the decimal number, and H$ = "" before calling this routine. On exit, H$ contains the hex value :=

```
10 IFDTHENA=INT(D/16):H$=MID$("0123456789ABCDEF",1+
D-A*16,1)+H$:D=A:GOTO10
```

2) Hex to Decimal, where H$ contains the hexadecimal value, and D holds the decimal value on exit :=

```
10
D=0:IFH$>""THENFORI=1TOLEN(H$):A=ASC(MID$(H$,I,1))-48
:D=D*16+A+(A>9)*7:NEXT
```

3) Decimal to Binary, where B is the decimal number lying between 0 and 225, and A$ contains the binary value on exit :=

```
10 A$="":FORI=0TO7:T=B-INT(B/2)*2:IFT=0THENA$="0"+A$
15 A$="1"+A$:B=INT(B/2):NEXTN
```

4) Other Hyperbolic Functions

Deriving Mathematical Functions

FUNCTION	BASIC EQUIVALENT
SECANT	SEC(X)= 1/COS(X)
COSECANT	CSC(X)= 1/SIN(X)
COTANGENT	COT(X)= 1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(−X*X+1))
INVERSE COSINE	ARCCOS(X)= −ATN(X/SQR (−X*X +1)) +π/2
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X−1))
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X−1)) +(SGN(X)−1*π/2
INVERSE COTANGENT	ARCOT(X)=ATN(X)+π/2
HYPERBOLIC SINE	SINH(X)=(EXP(X)−EXP(−X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(−X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(−X)/(EXP(x)+EXP (−X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(−X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)−EXP(−X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(−X)/(EXP(X) −EXP(−X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X−1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1−X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR (−X*X+1)+1/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR (X*X+1/x)
INVERSE HYPERBOLIC COTAN-GENT	ARCCOTH(X)=LOG((X+1)/(x−1))/2

Extramon: A Machine Code Assembler

```
100 PRINT"TINY PEEKER/POKER"
110 X$="*":INPUTX$:IFX$="*"THENEND
120 GOSUB500
130 IF E GOTO280
140 A=V
150 IFJ>LEN(X$)GOTO300
160 FORI=0TO7
170 P=J:GOSUB550
180 C(I)=V
190 IF E GOTO 280
200 NEXTI
210 T=0
220 FORI=0TO7
230 POKE A+I,C(I)
240 T=T+C(I)
250 NEXT I
260 PRINT"CHECKSUM=";T
270 GOTO110
280 PRINTMID$(X$,1,J);"??":GOTO110
300 T=0
310 FORI=0TO7
320 V=PEEK(A+I)
330 T=T+V
340 V=V/16
350 PRINT " ";
360 FORJ=1TO2
370 V%=V
380 V=(V-V%)*16
390 IFV%>9THENV%=V%+7
400 PRINTCHR$(V%+48);
410 NEXT J
420 NEXT I
430 PRINT "/";T
440 GOTO110
500 P=1
510 L=4
520 GOTO600
550 P=J
560 L=2
600 E=0
610 V=0
620 FORJ=P TO LEN(X$)
630 X=ASC(MID$(X$,J))
640 IFX=32 THEN NEXT J
650 IFJ>LEN(X$)THEN790
660 P=J
670 FORJ=PTOLEN(X$)
680 X=ASC(MID$(X$,J))
```

```
690 IF X<>32 THEN NEXT J
700 IF J-P<>L THEN 790
710 FORK=PTOJ-1
720 X=ASC(MID$(X$,K))
730 IF X<58 THENX=X-48
740 IF X>64 THEN X=X-55
750 IF X<0 OR X>15 THEN790
760 V=V*16+X
770 NEXT K
780 RETURN
790 E=-1
800 RETURN
```

```
0800 00 1A 04 64 00 99 22 93  0A00 02 20 48 FA 00 AD 3A 02  0C00 60 A2 02 2C A2 00 00 B4
0808 12 1D 1D 1D 1D 53 55 50  0A08 20 48 FA 00 20 B7 F8 00  0C08 C1 D0 08 B4 C2 D0 02 E6
0810 45 52 20 36 34 2D 4D 4F  0A10 20 8D F8 00 F0 5C 20 3E  0C10 26 D6 C2 D6 C1 60 20 3E
0818 4E 00 31 04 6E 00 99 22  0A18 F8 00 20 79 FA 00 90 33  0C18 F8 00 C9 20 F0 F9 60 A9
0820 11 20 20 20 20 20 20 20  0A20 20 69 FA 00 20 3E F8 00  0C20 00 00 8D 00 00 00 01 20 CC
0828 20 20 20 20 20 20 20 20  0A28 20 79 FA 00 90 28 20 69  0C28 FA 00 20 8F FA 00 20 7C
0830 00 4B 04 78 00 99 22 11  0A30 FA 00 A9 90 20 D2 FF 20  0C30 FA 00 90 09 60 20 3E F8
0838 20 2E 2E 4A 49 4D 20 42  0A38 E1 FF F0 3C A6 26 D0 38  0C38 00 20 79 FA 00 B0 DE AE
0840 55 54 54 45 52 46 49 45  0A40 A5 C3 C5 C1 A5 C4 E5 C2  0C40 3F 02 9A A9 90 20 D2 FF
0848 4C 44 00 66 04 82 00 9E  0A48 90 2E A0 3A 20 C2 F8 00  0C48 A9 3F 20 D2 FF 4C 47 F8
0850 28 C2 28 34 33 29 AA 32  0A50 20 41 FA 00 20 8B F8 00  0C50 00 20 54 FD 00 CA D0 FA
0858 35 36 AC C2 28 34 34 29  0A58 F0 E0 4C ED FA 00 20 79  0C58 60 E6 C3 D0 02 E6 C4 60
0860 AA 31 32 37 29 00 00 00  0A60 FA 00 90 03 20 80 F8 00  0C60 A2 02 B5 C0 48 B5 27 95
0868 AA AA AA AA AA AA AA AA  0A68 20 B7 F8 00 D0 07 20 79  0C68 C0 68 95 27 CA D0 F3 60
0870 AA AA AA AA AA AA AA AA  0A70 FA 00 90 EB A9 08 85 1D  0C70 A5 C3 A4 C4 38 E9 02 B0
0878 AA AA AA AA AA AA AA AA  0A78 20 3E F8 00 20 A1 F8 00  0C78 0E 88 90 0B A5 28 A4 29

0880 A5 2D 85 22 A5 2E 85 23  0A80 D0 F8 4C 47 F8 00 20 CF  0C80 4C 33 FB 00 A5 C3 A4 C4
0888 A5 37 85 24 A5 38 85 25  0A88 FF C9 0D F0 0C C9 20 D0  0C88 38 E5 C1 85 1E 98 E5 C2
0890 A0 00 A5 22 D0 02 C6 23  0A90 D1 20 79 FA 00 90 03 20  0C90 A8 05 1E 60 20 D4 FA 00
0898 C6 22 B1 22 D0 3C A5 22  0A98 80 F8 00 A9 90 20 D2 FF  0C98 20 69 FA 00 20 E5 FA 00
08A0 D0 02 C6 23 C6 22 B1 22  0AA0 AE 3F 02 9A 78 AD 39 02  0CA0 20 0C FB 00 20 E5 FA 00
08A8 F0 21 85 26 A5 22 D0 02  0AA8 48 AD 3A 02 48 AD 3B 02  0CA8 20 2F FB 00 20 69 FA 00
08B0 C6 23 C6 22 B1 22 18 65  0AB0 48 AD 3C 02 AE 3D 02 AC  0CB0 90 15 A6 26 D0 04 20 28
08B8 24 AA A5 26 65 25 48 A5  0AB8 3E 02 40 A9 90 20 D2 FF  0CB8 FB 00 90 5F A1 C1 81 C3
08C0 37 D0 02 C6 38 C6 37 68  0AC0 AE 3F 02 9A 6C 02 A0 A0  0CC0 20 05 FB 00 20 33 FB 00
08C8 91 37 8A 48 A5 37 D0 02  0AC8 01 84 BA 84 B9 88 84 B7  0CC8 D0 EB 20 28 FB 00 18 A5
08D0 C6 38 C6 37 68 91 37 18  0AD0 84 90 84 93 A9 40 85 BB  0CD0 1E 65 C3 85 C3 98 65 C4
08D8 90 B6 C9 4F D0 ED A5 37  0AD8 A9 02 85 BC 20 CF FF C9  0CD8 85 C4 20 0C FB 00 A6 26
08E0 85 33 A5 38 85 34 6C 37  0AE0 20 F0 F9 C9 0D F0 38 C9  0CE0 D0 3D A1 C1 81 C3 20 28
08E8 00 4F 4F 4F 4F AD E6 FF  0AE8 22 D0 14 20 CF FF C9 22  0CE8 FB 00 80 34 20 B8 FA 00
08F0 00 8D 16 03 AD E7 FF 00  0AF0 F0 10 C9 0D F0 29 91 BB  0CF0 20 BB FA 00 4C 7D FB 00
08F8 8D 17 03 A9 80 20 90 FF  0AF8 E6 B7 C8 C0 10 D0 EC 4C  0CF8 20 D4 FA 00 20 69 FA 00

0900 00 00 D8 68 8D 3E 02 68  0B00 ED FA 00 20 CF FF C9 0D  0D00 20 E5 FA 00 20 69 FA 00
0908 8D 3D 02 68 8D 3C 02 68  0B08 F0 16 C9 2C D0 DC 20 88  0D08 20 3E F8 00 20 88 FA 00
0910 8D 3B 02 68 AA 68 A8 38  0B10 FA 00 29 0F F0 E9 C9 03  0D10 90 14 85 1D A6 26 D0 11
0918 8A E9 02 8D 3A 02 98 E9  0B18 F0 E5 85 BA 20 CF FF C9  0D18 20 2F FB 00 90 0C A5 1D
0920 00 00 8D 39 02 BA BE 3F  0B20 0D 60 6C 30 03 6C 32 03  0D20 81 C1 20 33 F8 00 D0 EE
0928 02 20 57 FD 00 A2 42 A9  0B28 20 96 F9 00 D0 D4 A9 90  0D28 4C ED FA 00 4C 47 F8 00
0930 2A 20 57 FA 00 A9 52 D0  0B30 20 D2 FF A9 00 00 20 EF  0D30 20 D4 FA 00 20 69 FA 00
0938 34 E6 C1 D0 06 E6 C2 D0  0B38 F9 00 A5 90 29 10 D0 C4  0D38 20 E5 FA 00 20 69 FA 00
0940 02 E6 26 60 20 CF FF C9  0B40 4C 47 F8 00 20 96 F9 00  0D40 20 3E F8 00 A2 00 00 20
0948 0D D0 F8 68 68 A9 90 20  0B48 C9 2C D0 BA 20 79 FA 00  0D48 3E F8 00 C9 27 D0 14 20
0950 D2 FF A9 00 00 85 26 A2  0B50 20 69 FA 00 20 CF FF C9  0D50 3E F8 00 9D 10 02 E8 20
0958 0D A9 2E 20 57 FA 00 A9  0B58 2C D0 AD 20 79 FA 00 A5  0D58 CF FF C9 0D F0 22 E0 20
0960 05 20 D2 FF 20 3E F8 00  0B60 C1 85 AE A5 C2 85 AF 20  0D60 D0 F1 F0 1C 8E 00 00 01
0968 C9 2E F0 F9 C9 20 F0 F5  0B68 69 FA 00 20 CF FF C9 0D  0D68 20 8F FA 00 90 C6 9D 10
0970 A2 0E DD B7 FF 00 D0 0C  0B70 D0 98 A9 90 20 D2 FF 20  0D70 02 E8 20 CF FF C9 0D F0
0978 8A 0A AA BD C7 FF 00 48  0B78 F2 F9 00 4C 47 F8 00 A5  0D78 09 20 88 FA 00 90 B6 E0

0980 BD C6 FF 00 48 60 CA 10  0B80 C2 20 48 FA 00 A5 C1 48  0D80 20 D0 EC 86 1C A9 90 20
0988 EC 4C ED FA 00 A5 C1 8D  0B88 4A 4A 4A 4A 20 60 FA 00  0D88 D2 FF 20 57 FD 00 A2 00
0990 3A 02 A5 C2 8D 39 02 60  0B90 AA 68 29 0F 20 60 FA 00  0D90 00 A0 00 00 B1 C1 DD 10
0998 A9 08 85 1D A0 00 00 20  0B98 48 8A 20 D2 FF 68 4C D2  0D98 02 D0 0C C8 E8 E4 1C D0
09A0 54 FD 00 B1 C1 20 48 FA  0BA0 FF 09 30 C9 3A 90 02 69  0DA0 F3 20 41 FA 00 20 54 FD
09A8 00 20 33 F8 00 C6 1D D0  0BA8 06 60 A2 02 B5 C0 48 B5  0DA8 00 20 33 F8 00 A6 26 D0
09B0 F1 60 20 88 FA 00 90 0B  0BB0 C2 95 C0 68 95 C2 CA D0  0DB0 8D 20 2F FB 00 B0 DD 4C
09B8 A2 00 00 81 C1 C1 C1 F0  0BB8 F3 60 20 88 FA 00 90 02  0DB8 47 F8 00 20 D4 FA 00 85
09C0 03 4C ED FA 00 20 33 F8  0BC0 85 C2 20 88 FA 00 90 02  0DC0 20 A5 C2 85 21 A2 00 00
09C8 00 C6 1D 60 A9 3B 85 C1  0BC8 85 C1 60 A9 00 00 85 2A  0DC8 86 28 A9 93 20 D2 FF A9
09D0 A9 02 85 C2 A9 05 60 98  0BD0 20 3E F8 00 C9 20 D0 09  0DD0 90 20 D2 FF A9 16 85 1D
09D8 48 20 57 FD 00 68 A2 2E  0BD8 20 3E F8 00 C9 20 D0 0E  0DD8 20 6A FC 00 20 CA FC 00
09E0 4C 57 FA 00 A9 90 20 D2  0BE0 18 60 20 AF FA 00 0A 0A  0DE0 85 C1 84 C2 C6 1D D0 F2
09E8 FF A2 00 00 BD EA FF 00  0BE8 0A 0A 85 2A 20 3E F8 00  0DE8 A9 91 20 D2 FF 4C 47 F8
09F0 20 D2 FF E8 E0 16 D0 F5  0BF0 20 AF FA 00 05 2A 38 60  0DF0 00 A0 2C 20 C2 F8 00 20
09F8 A0 3B 20 C2 F8 00 AD 39  0BF8 C9 3A 90 02 69 08 29 0F  0DF8 54 FD 00 20 41 FA 00 20
```

```
0E00 54 FD 00 A2 00 00 A1 C1   1000 00 BD 2A FF 00 20 B9 FE
0E08 20 D9 FC 00 48 20 1F FD   1008 00 D0 65 CA D0 D1 F6 0A
0E10 00 68 20 35 FD 00 A2 06   1010 20 B8 FE 00 D0 AB 20 B8
0E18 E0 03 D0 12 A4 1F F0 0E   1018 FE 00 D0 A6 A5 28 C5 1D
0E20 A5 2A C9 E8 B1 C1 B0 1C   1020 D0 A0 20 69 FA 00 A4 1F
0E28 20 C2 FC 00 88 D0 F2 06   1028 F0 28 A5 29 C9 9D D0 1A
0E30 2A 90 0E BD 2A FF 00 20   1030 20 1C FB 00 90 0A 98 D0
0E38 A5 FD 00 BD 30 FF 00 F0   1038 04 A5 1E 10 0A 4C ED FA
0E40 03 20 A5 FD 00 CA D0 D5   1040 00 C8 D0 FA A5 1E 10 F6
0E48 60 20 CD FC 00 AA E8 D0   1048 A4 1F D0 03 B9 C2 00 00
0E50 01 C8 98 20 C2 FC 00 8A   1050 91 C1 88 D0 F8 A5 26 91
0E58 86 1C 20 48 FA 00 A6 1C   1058 C1 20 CA FC 00 85 C1 84
0E60 60 A5 1F 38 A4 C2 AA 10   1060 C2 A9 90 20 D2 FF A0 41
0E68 01 88 65 C1 90 01 C8 60   1068 20 C2 FB 00 20 54 FD 00
0E70 A8 4A 90 0B 4A B0 17 C9   1070 20 41 FA 00 20 54 FD 00
0E78 22 F0 13 29 07 09 80 4A   1078 A9 05 20 D2 FF 4C B0 FD

0E80 AA BD D9 FE 00 B0 04 4A   1080 00 A8 20 BF FE 00 D0 11
0E88 4A 4A 4A 29 0F D0 04 A0   1088 98 F0 0E 86 1C A6 1D DD
0E90 80 A9 00 00 AA BD 1D FF   1090 10 02 08 E8 86 1D A6 1C
0E98 00 85 2A 29 03 85 1F 98   1098 28 60 C9 30 90 03 C9 47
0EA0 29 8F AA 98 A0 03 E0 8A   10A0 60 38 60 40 02 45 03 D0
0EA8 F0 0B 4A 90 08 4A 4A 09   10A8 08 40 09 30 22 45 33 D0
0EB0 20 88 D0 FA C8 88 D0 F2   10B0 08 40 09 40 02 45 33 D0
0EB8 60 B1 C1 20 C2 FC 00 A2   10B8 08 40 09 40 02 45 B3 D0
0EC0 01 20 FE FA 00 C4 1F C8   10C0 08 40 09 00 00 22 44 33
0EC8 90 F1 A2 03 C0 04 90 F2   10C8 D0 8C 44 00 00 11 22 44
0ED0 60 A8 B9 37 FF 00 85 28   10D0 33 D0 8C 44 9A 10 22 44
0ED8 B9 77 FF 00 85 29 A9 00   10D8 33 D0 08 40 09 10 22 44
0EE0 00 A0 05 06 29 26 28 2A   10E0 33 D0 08 40 09 62 13 78
0EE8 88 D0 F8 69 3F 20 D2 FF   10E8 A9 00 00 21 81 82 00 00
0EF0 CA D0 EC A9 20 2C A9 0D   10F0 00 00 59 4D 91 92 86 4A
0EF8 4C D2 FF 20 D4 FA 00 20   10F8 85 9D 2C 29 2C 23 28 24

0F00 69 FA 00 20 E5 FA 00 20   1100 59 00 00 58 24 24 00 00
0F08 69 FA 00 A2 00 00 86 28   1108 1C 8A 1C 23 5D 8B 1B A1
0F10 A9 90 20 D2 FF 20 57 FD   1110 9D 8A 1D 23 9D 8B 1D A1
0F18 00 20 72 FC 00 20 CA FC   1118 00 00 29 19 AE 69 A8 19
0F20 00 85 C1 84 C2 20 E1 FF   1120 23 24 53 1B 23 24 53 19
0F28 F0 05 20 2F FB 00 B0 E9   1128 A1 00 00 1A 5B 5B A5 69
0F30 4C 47 F8 00 20 D4 FA 00   1130 24 24 AE AE A8 AD 29 00
0F38 A9 03 85 1D 20 3E F8 00   1138 00 7C 00 00 15 9C 6D 9C
0F40 20 A1 F8 00 D0 F8 A5 20   1140 A5 69 29 53 84 13 34 11
0F48 85 C1 A5 21 85 C2 4C 46   1148 A5 69 23 A0 D8 62 5A 48
0F50 FC 00 C5 28 F0 03 20 D2   1150 26 62 94 88 54 44 C8 54
0F58 FF 60 20 D4 FA 00 20 69   1158 68 44 E8 94 00 00 B4 08
0F60 FA 00 8E 11 02 A2 03 20   1160 84 74 B4 28 6E 74 F4 CC
0F68 CC FA 00 48 CA D0 F9 A2   1168 4A 72 F2 A4 8A 00 00 AA
0F70 03 68 38 E9 3F A0 05 4A   1170 A2 A2 74 74 74 72 44 68
0F78 6E 11 02 6E 10 02 88 D0   1178 B2 32 B2 00 00 22 00 00

0F80 F6 CA D0 ED A2 02 20 CF   1180 1A 1A 26 26 72 72 88 C8
0F88 FF C9 0D F0 1E C9 20 F0   1188 C4 CA 26 48 44 44 A2 C8
0F90 F5 20 D0 FE 00 B0 0F 20   1190 3A 3B 52 4D 47 58 4C 53
0F98 9C FA 00 A4 C1 84 C2 85   1198 54 46 48 44 50 2C 41 42
0FA0 C1 A9 30 9D 10 02 E8 9D   11A0 F9 00 35 F9 00 CC F8 00
0FA8 10 02 E8 D0 DB 86 28 A2   11A8 F7 F8 00 56 F9 00 89 F9
0FB0 00 00 86 26 F0 04 E6 26   11B0 00 F4 F9 00 0C FA 00 3E
0FB8 F0 75 A2 00 00 86 1D A5   11B8 FB 00 92 FB 00 C0 FB 00
0FC0 26 20 D9 FC 00 A6 2A 86   11C0 38 FC 00 5B FD 00 8A FD
0FC8 29 AA BC 37 FF 00 BD 77   11C8 00 AC FD 00 46 F8 00 FF
0FD0 FF 00 20 B9 FE 00 D0 E3   11D0 F7 00 ED F7 00 0D 20 20
0FD8 A2 06 E0 03 D0 19 A4 1F   11D8 20 50 43 20 20 53 52 20
0FE0 F0 15 A5 2A C9 E8 A9 30   11E0 41 43 20 58 52 20 59 52
0FE8 B0 21 20 BF FE 00 D0 CC   11E8 20 53 50 AA AA AA AA AA
0FF0 20 C1 FE 00 D0 C7 88 D0
0FF8 EB 06 2A 90 0B BC 30 FF
```

Entering Extramon : Commodore 64 version

Use the program Tiny Peeker/Poker as follows.

Type POKE 8192,0:POKE44,32 (return)

Enter and run the peeker/poker program.

In response to the program prompts, type in the data as given in the following tables. When you've finished, type POKE44,8:POKE45,232:POKE46,17:CLR.

Save Extramon with a normal SAVE before attempting to run it.

Then type NEW. and use the following checksum program to enable you to identify and locate any errors:

```
100 REM EXTRAMON64 CHECKSUM PROGRAM
110 DATA10170,13676,15404,14997,15136,16221,16696
115 DATA12816,16228,14554
120 DATA14677,15039,14551,15104,15522.16414,15914
125 DATA8958,11945:S=2048
130 FORB=1TO19:READX:FORI=1TOS:N=PEEK(I):Y=Y+N
140 NEXTI:IFY<>XTHEN?"ERROR IN BLOCK "B:GOTO160
150 PRINT"BLOCK "B" CORRECT"
160 S=I:Y=0:NEXTB:REM CHECK LAST BLOCK BY HAND
```

This program must be run after the first set of POKEs, and before the second set.

If errors are found, type NEW (you won't lose Extramon!), re-load Tiny Peeker/Poker, enter block of memory again, type NEW, re-load checksum program, run it, if errors found type NEW and re-load Tiny Peeker/Poker, and so on until no errors remain. Then, issue last set of POKEs and SAVE Extramon!

Extramon Instruction Set

This will be given in the form COMMAND, followed by the syntax.

1) Simple Assembler
```
.A 2000 LDA£12
```
start assembly at 2000 hex.

2) Disassembler
```
.D 2000
```
disassemble hex from 2000 onwards.

3) Printing Disassembler
```
.P 2000,2040
```
engage printer beforehand with OPEN4,4:CMD4.

4) Fill memory
```
.F 1000 1100 FF
```
fill memory from 1000 to 1100 hex with the byte FF.

5) Go run
```
.G 1000
```
go to hex 1000 and executre program there.

6) Hunt memory
```
.H C000 D000 'READ
```
look from C000 to D000 for the ASCII string READ.

7) Load
```
.L "FRED",08
```

8) Memory display
```
.M 0800 0820
```
display memory from hex 0800 to 0820.

9) Register display
```
.R
```
displays register values when Extramon was entered.

10) Save
```
.S "0:FRED",08,0800,0820
```
save memory from hex 0800 to 0820 onto device 08 drive 1, and call that portion of memory FRED.

11) Transfer memory
```
.T 1000 1100 5000
```
transfer memory in the range hex 1000 to 1100 and start storing it at hex 5000 onwards.

12) Exit to Basic
```
.Q
```
return to Basic ready mode. Perform a CLR before doing anything.

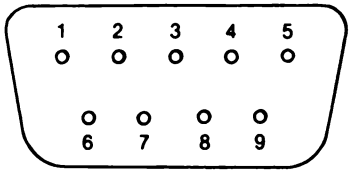
Appendix E : Pinouts for Input/Output

This appendix is designed to show you what connections may be made to the Commodore 64.

1) Game I/O
2) Cartridge Slot
3) Audio/Video
4) Serial I/O (Disk/Printer)
5) Modulator Output
6) Cassette
7) User Port

Control Port 1

Pin	Type	Note
1	JOYA0	
2	JOYA1	
3	JOYA2	
4	JOYA3	
5	POT AY	
6	BUTTON A/LP	
7	+5V	MAX. 100mA
8	GND	
9	POT AX	



Control Port 2

Pin	Type	Note
1	JOYB0	
2	JOYB1	
3	JOYB2	
4	JOYB3	
5	POT BY	
6	BUTTON B	
7	+5V	MAX. 100mA
8	GND	
9	POT BX	

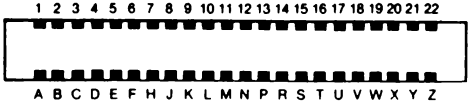
Cartridge Expansion Slot

Pin	Type
22	GND
21	CD0
20	CD1
19	CD2
18	CD3
17	CD4
16	CD5
15	CD6
14	CD7
13	$\overline{\text{DMA}}$
12	BA

Pin	Type
11	$\overline{\text{ROML}}$
10	I/O2
9	$\overline{\text{EXROM}}$
8	$\overline{\text{GAME}}$
7	I/O1
6	Dot Clock
5	R/$\overline{\text{W}}$
4	$\overline{\text{IRQ}}$
3	+ 5V
2	+ 5V
1	GND

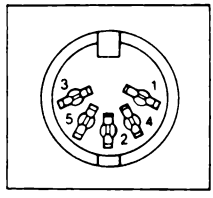
Pin	Type
Z	GND
Y	CA0
X	CA1
W	CA2
V	CA3
U	CA4
T	CA5
S	CA6
R	CA7
P	CA8
N	CA9

Pin	Type
M	CA10
L	CA11
K	CA12
J	CA13
H	CA14
F	CA15
E	$\overline{\text{S02}}$
D	$\overline{\text{NMI}}$
C	$\overline{\text{RESET}}$
B	$\overline{\text{ROMH}}$
A	GND



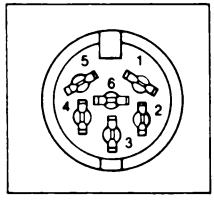
Audio/Video

Pin	Type	Note
1	LUMINANCE	
2	GND	
3	AUDIO OUT	
4	VIDEO OUT	
5	AUDIO IN	



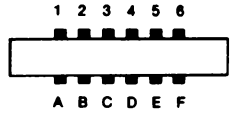
Serial I/O

Pin	Type
1	SERIAL $\overline{\text{SRQIN}}$
2	GND
3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	$\overline{\text{RESET}}$



Cassette

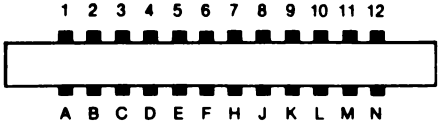
Pin	Type
A-1	GND
B-2	+5V
C-3	CASSETTE MOTOR
D-4	CASSETTE READ
E-5	CASSETTE WRITE
F-6	CASSETTE SENSE



User I/O

Pin	Type	Note
1	GND	
2	+5V	MAX. 100 mA
3	$\overline{\text{RESET}}$	
4	CNT1	
5	SP1	
6	CNT2	
7	$\overline{\text{SP2}}$	
8	$\overline{\text{PC2}}$	
9	SER. ATN IN	
10	9 VAC	MAX. 100 mA
11	9 VAC	MAX. 100 mA
12	GND	

Pin	Type	Note
A	$\overline{\text{GND}}$	
B	$\overline{\text{FLAG2}}$	
C	PB0	
D	PB1	
E	PB2	
F	PB3	
H	PB4	
J	PB5	
K	PB6	
L	PB7	
M	PA2	
N	GND	



DUCKWORTH HOME COMPUTING

a new series

All books written by Pete Gerrard, former editor of *Commodore Computing International*, author of two top-selling adventure games for the Commodore 64 and a regular contributor to *Personal Computer News, Which Micro? and Software Review.*

EXPLORING ADVENTURE GAMES ON THE COMMODORE 64

The complete guide to computer adventure games: playing, writing and solving them. Starting with an introduction to adventures, and their early history, it takes you gently through the basic programming necessary on the 64 before you can start writing your own games. Inputting of information, room mapping, movement, vocabulary, and everything required to write an adventure game are explored in full detail. Then follow a number of adventure scenarios, and finally three complete listings, written specially for the 64.

September £6.95

THE BEGINNER'S GUIDE TO COMPUTERS AND COMPUTING by Pete Gerrard

Writing for the person who knows absolutely nothing about computers, and assuming no prior knowledge, Pete Gerrard introduces you gently to this exciting and fast-moving world.

This book guides you through the history of computers into the 1980s and introduces you to many of the personalities who dictate how computers will develop in the future. It comes complete with a glossary of computing terms, including all the often used 'buzz words', and even an 'alternative' computer glossary.

You may know nothing about computers or computing when you pick up this book, but by the last page you'll be thoroughly conversant with every aspect of them – you may even go out and buy one!

October £6.95

Other titles in the series include *Using the Commodore 64, Sprites & Sound on the 64, 12 Simple Electronic Projects for the VIC, Will You Still Love Me When I'm 64, Advanced Basic & Machine Code Programming on the VIC, Advanced Basic & Machine Code Programming on the 64, Exploring Adventures on the VIC,* as well as *Pocket Handbooks for the VIC, 64, Dragon, Spectrum and BBC Model B.*

Write in for a descriptive leaflet.



DUCKWORTH

The Old Piano Factory, 43 Gloucester Crescent, London NW1 7DY

Tel: 01-485 3484

Index

This index usually only shows the first appearance of a subject in the book, but if a second (and subsequent) entry is important, it is also noted down.

6510 chip : 201,250
6510 pin description : 251
6510 characteristics : 256
6510 memory map : 261
6526 complex interface adaptor : 261
6526 pin description : 262
6526 characteristics : 264
6526 interface signals : 266
6526 timing : 267
6566 video interface chip : 202,210
6581 sound interface device : 229
6581 registers : 233
6581 pin description : 242
6581 timing : 245
Accumulator : 75
Addressing modes : 83
Animation : 87
ASC : 28
Bit mapping : 122,124
Bit map modes : 214
Calculations : 8
Carry flag : 79
Character definition : 114
Character modes : 212
Charget : 91
Colour : 103,113
Cursor control : 12,35
CHR$: 28
CONtinue : 15
CONTROL key : 6
Defining functions : 53
Defining function keys : 99
Dimension statements : 37
Disk Usage : 47
Disk storage : 189
Disk utilities : 192,193,194,195
DATA : 16
Expanding memory : 92
Exponentialism : 59
END : 15
Formatting numbers : 198
FOR ... NEXT loops : 29
FRE command : 58
Games listings : 66,109,145
Graphic modes : 112,117,120,121
GET : 18
GOSUB : 32
GOTO : 31
Hexadecimal : 72
Hyperbolic functions : 55,57
Input : 11,14,65
Interrupts : 91
IF : 18
Joystick control : 43
Keyboard control : 2
Line numbers : 7
Linking to Basic : 96
Listing program : 199
Logical Operators : 19,49,52,86
LEFT$: 24
LEN : 22
LOGO key : 5
Machine language monitor : 69
Mathematical functions : 57
Mathematics in machine code : 81,94
Memory map control : 204
Memory interfacing : 226
Menu options : 34
Merging programs : 179
Multiple voices : 150,155
Music programs : 153,171
MID$: 23
Operands : 73
ON ... GOTO : 33
Physics of sound : 157
Printer utilities : 196,197
Program editing : 9,64
Program counter : 76
PEEK : 39
POKE : 39
PRINT command : 8

Random numbers : 36
Relative files : 181
Repeat keys : 63
REM statements : 20
RESTORE key : 4
RESTORE command : 17
RETURN key : 3
RETURN command : 32
RIGHT$: 24
RUN/STOP key : 4
Screen blanking : 48
Screen manipulation : 125
Screen features : 221
Sequential files : 178,180
Sprites : 127,128,129,132,134,
137,139
Stack pointer : 77
Status register : 78
Subroutines : 21
SHIFT/LOCK Key : 6
SID memory map : 149
SID manipulation : 165,166
STOP : 15
STR$: 26
SYS : 60
Tape Control : 45
Tape storage : 190
Tape decks, using non-
Commodore : 191
THEN : 18
TI$: 54
USR : 60
Variables : 13
VAL : 26
Waveforms : 160
WAIT statement : 42

Duckworth Home Computing

USING THE COMMODORE 64 by Peter Gerrard

This is an essential book for any user of the latest home computer from Commodore Business Machines.

Starting with a refresher course in Basic programming, it moves on through Machine Code, before considering in great detail sprites, graphics and sound. The heart of the book is an in-depth look at the chips that make it work, including the 6581 Sound Interface Device and the 6566 Video Controller Chip, and the 6510. Appendices cover the full Basic and Machine Code Instruction sets and a complete Machine Code assembler/disassembler listing.

The book is illustrated throughout by many example listings, all specially designed for the Commodore 64.

Peter Gerrard, former editor of *Commodore Computing International,* is author of two top-selling adventure games for the Commodore 64 and a regular contributor to *Personal Computer News* and *Which Micro? and Software Review.*

Duckworth
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 1777 X

IN UK ONLY £9.95 NET



This was brought to you

from the archives of

http://retro-commodore.eu