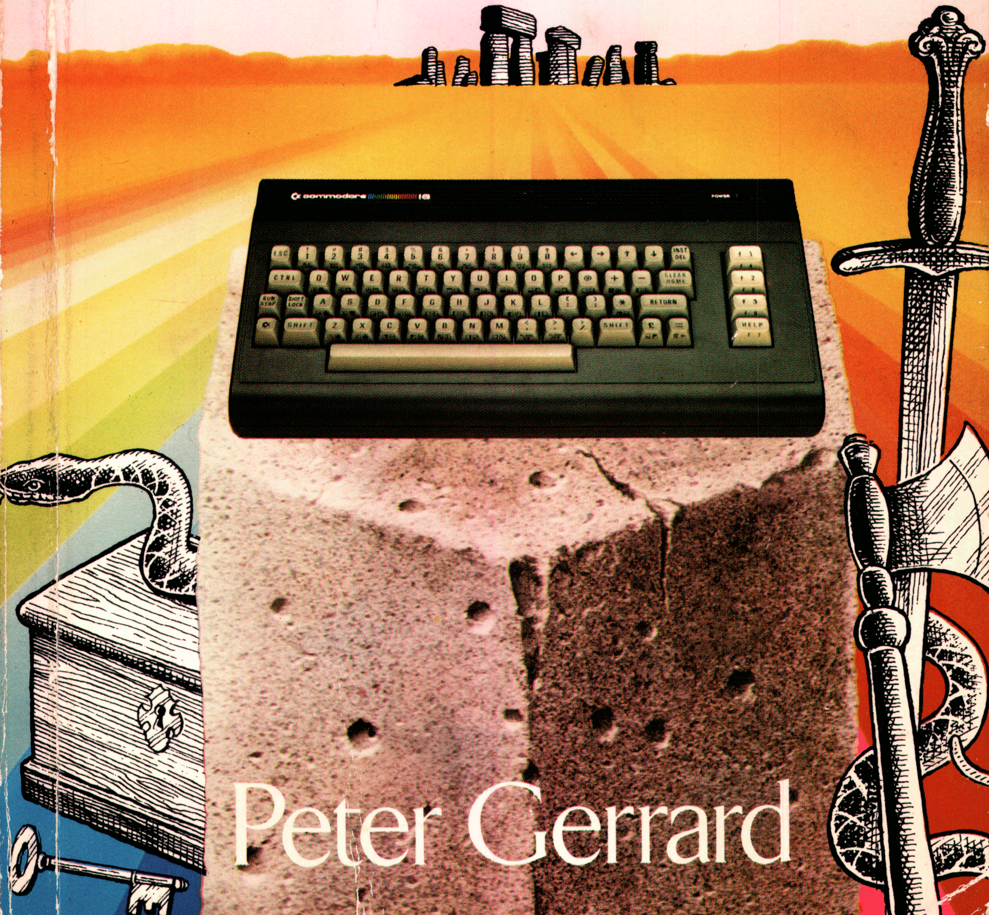
USING THE COMMODORE 16



Peter Gerrard

Using the Commodore 16

USING THE COMMODORE 16

Peter Gerrard

Duckworth

First published in 1986 by
Gerald Duckworth & Co. Ltd.
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 2003 7

British Library Cataloguing in Publication Data

Gerrard, Peter
Using the Commodore 16
1. Commodore 16 (Computer)
I. Title
001.64'04 QA76.8.C64

ISBN 0-7156-2003-7

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

Contents

Preface

This book is for all C16 owners, whether beginner or expert. It starts with a refresher course in Basic programming and an introduction to new Basic commands on the C16. There are details of how to use the handy monitor built into the machine, and the book also explores colour, graphics and sound. Three important chapters show you how to write an arcade game, build up a database and write an adventure. By the end of the careful explanation of machine code you should be able to write your own machine-code marvels! Half-a-dozen games provide some relaxation, and finally there is a chapter on peripherals.

P.G.

1

Getting Started

Okay, you've got yourself a Commodore 16. Perhaps you were persuaded by friends, by television advertisements, or by reviews in the popular computing press. Why you bought the machine isn't particularly important. What is important is the fact that you've got it home and are probably beginning to wonder what it is really capable of doing.

The exterior of the machine strongly resembles the Commodore 64. However, the interior is very different, as we shall see. Try typing in PRINT PEEK(198) on the 64, and you'll probably get an answer of zero. The same command on the C16 will probably give you an answer of 64. We'll see why later.

The cassette deck used with the C16 is electronically very similar to the one used with the Commodore 64, or Vic 20, or indeed any of the earlier Commodore computers. But for reasons probably known only to Commodore Business Machines themselves, earlier cassette decks will not work on the C16. It requires its own little interface, and consequently the C16 starter pack arrives complete with its own cassette deck.

It also arrives with the grandly-titled *Introduction to Basic,* and a User's Manual as well. If you only want to learn how to count to ten, or discover what all the Basic keywords are, then those two publications are sufficient and you might as well put this book down now. If you want to make your new computer do something useful, then carry on reading.

One thing the C16 does have in common with the 64 is its use of disk drives and printers. Most of the peripherals which can be connected up to the 64 can also be connected to the C16, and indeed the listings in this book were produced using an interface for the 64, my old 64 disk drive, and an Epson FX80 printer which the 64 has happily been driving for a couple of years now. However, all three devices ran quite readily from the C16, so there is a high degree of compatability between the two on that score.

On the software side, simple programs for the 64 that don't take up too much memory will run on the C16. Try running any programs of more than 12K, however, and you'll be in trouble. Also, any programs using machine code, or any which PEEK and POKE to the screen will certainly not work. Again, we'll see why later.

Despite all that, the C16 can be a useful computer, and is certainly an excellent introduction to the fast moving world of microcomputers. The version of Basic used to program it is a highly advanced one, and one which is certainly capable of competing with any other home computer currently available. It has a sensible keyboard, one which any touch-typist should soon feel at home on, and if one or two keys are not in the 'right' place for users of other computers, well, that's only to be expected. It seems to be the rule that new computers always move a few keys around, just to maintain individuality.

Wiring up

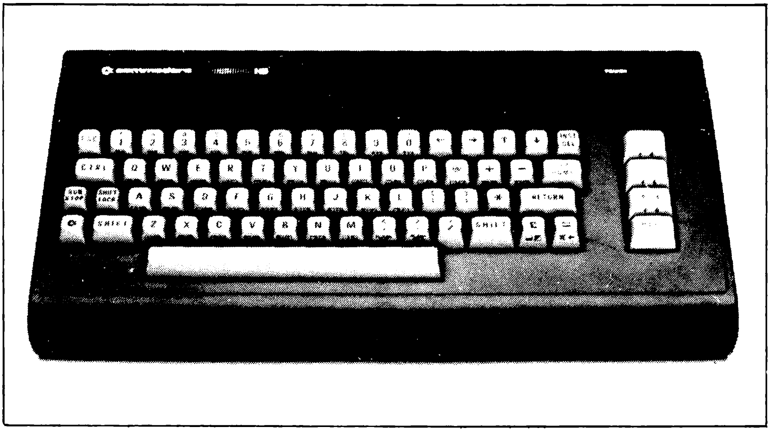
Before you start using your new computer it is worthwhile taking a few minutes to set it up properly. Presumably it is meant to be something more than a passing fad, and will hopefully become a device that you will use quite often. You wouldn't (I hope) get a new stereo system and then proceed to put both speakers in front of the main deck, obscuring your view of the television in the process. You will also want to avoid the almost inevitable collection of spaghetti wires trailing around all over the place.

Circumstances will obviously dictate how the computer is set up, but a few rules are worth observing. You'll want to be at a reasonable distance from the television screen, and to look at it from a sensible angle. You'll want to be able to use the keyboard in a manner which makes it easy to touch-type (or at least to type fast with one finger), and you'll also want to be able to connect up any peripherals without causing miles of wire and cables to go everywhere. Bear in mind that you'll be using the machine a lot, so a little bit of time and patience at the beginning will save an awful lot of time later on.

Assuming you've got everything connected up properly, what happens next?

In order to make the C16 work to its full potential, we first of all need to know what the various keys on the keyboard do. Without them, we won't be capable of doing anything, and so ...

The keyboard



As you can see from our picture and from the computer that I hope is in front of you at the moment, the C16 has a standard QWERTY layout. Pressing a key in normal mode will give you an upper-case letter on the screen, and pressing a key in conjunction with the shift key (of which there are two) will give you the graphic character that appears on the right of the key. To get the graphic character on the left of the key, you must press the CBM logo key (bottom left) in conjunction with any of the alphabetic keys (and one or two of the other keys as well: the multiplication symbol, for instance).

To obtain some more figures on the screen, press the CONTROL key (marked CTRL) and the '9' key simultaneously. This will put us into what is known as REVERSE FIELD mode, and any key now pressed will come out as a reverse of itself. To get out of this mode, press CTRL and the '0' key together. To change the colour of any character to be displayed on the screen, we need to use the CTRL key again. Pressing this together with one of the numeric keys 1 to 8 will produce a change in colour. As you are probably aware, though, the C16 is capable of displaying many different colours. In order to get the second lot of eight major colours we must go back to the CBM logo key again, and this time press that in conjunction with one of the first eight numeric keys. Any of the major 16 colours can be achieved in this way.

As well as producing the display that we've seen so far, the C16 has another character set that we haven't yet looked at. All the keys pressed so far have been either upper-case letters or graphics characters (or numbers, of course). If you now hold down the shift key and press the CBM logo key, you will see that the display flips into upper-case characters and lower-case ones. Pressing the two keys together again will take us back to the more familiar upper-case and graphics character set. Holding them both down continuously will result in a slowly repeating display.

There are a number of ways of achieving this from within a program. One could type:

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

to get the upper-case and lower-case character set. Normality is achieved by typing:

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

However, this doesn't stop anyone who is running your programs from hitting the CBM logo key in conjunction with the shift key, and upsetting the display as a consequence. To protect your programs from this happening, type:

```
10 PRINT CHR$(8)
```

which will inhibit the function of the shift and CBM logo keys. To make them work again, we need:

```
10 PRINT CHR$(9)
```

All these little one-liners can be typed in in what is referred to as 'immediate' mode. That is, they can be typed in without a line number in front of them, followed by pressing the RETURN key. Pressing the (RETURN) key indicates to the computer that we want something to happen, and the computer will then trot off to a part of its own memory and interpret precisely what is going to take place.

Function keys

Over to the right of the keyboard are the four (or eight, using the shift key) so-called function keys. At power on they all have little functions assigned to them, although these can be changed quite easily using the KEY command. To make them behave like Commodore 64 function keys, you need something like:

```
1 FORI=0TO7
2 KEY I,CHR$(I+133)
3 NEXT
```

Then they could be detected in a program like this:

```
5 A=1
10 GETA$:IFA$="[F1]"THEN A=1
20 IFA$="[F3]"THEN A=2
30 IFA$="[F5]"THEN A=3
40 IFA$="[F7]"THEN A=4
50 COLOR 0,A,7
60 GOTO 10
```

where the symbol [F1] represents the control character that is produced by pressing the first function key when inside quotes, and so on.

You may have noticed this quote mode before. Pressing the quotes key (shift and 2) will produce a set of quote marks on the screen. If you then attempt to press any of the cursor keys, the numeric keys in conjunction with the CTRL key or the CBM logo key, the function keys, and a few others, nothing will happen other than a strange graphic character appearing on the screen. As with seemingly everything else on the C16, there are various ways of getting out of this mode, but probably the easiest is to press the shift key and RETURN key together. This has the effect of moving the cursor (the little flashing blob that greets you all the time the C16 is waiting for something to happen) down on to the next line of the screen, without disturbing anything else.

Other keys

There are plenty of other keys on the C16 keyboard that we haven't looked at yet, and among these we find the INST/DEL key. Pressing this on its own has the effect of deleting everything to the left of the cursor, and dragging anything that happens to be to the right of it along as well. If we use this key in conjunction with the shift key, almost the reverse happens, and everything to the right of the cursor is moved even further to the right (up to a certain maximum point), while everything to the left stays where it was.

The CLEAR/HOME key also has two functions. To move the cursor to the top left-hand corner (the home position) of the screen, simply press the CLEAR/HOME key by itself. To do this and clear the screen at the same time, press the CLEAR/HOME key in conjunction with the shift key. Since this wipes out anything that happens to be on the screen at the time, you should use it with caution.

The ESC key (top left) has little function in life, and seems to do nothing other than provide the program with a means of detecting when it has been pressed. PEEK(198) returns a 52 when the ESC key is pressed, so by including a check in a program to see whether PEEK(198) = 52, you can make the program branch off to an appropriate point.

The reset switch on the right of the machine can be used if the computer has come across something that it dislikes and has hung up (i.e. is in a state of temporary distress, where nothing seems to be happening). It won't actually clear everything out of the computer's memory, so you won't lose any machine code program that might be there.

There is an even more drastic way to achieve a similar effect, and that is to type in:

```
10 SYS 62116
```

which, when run, will simulate the effect of switching the C16 off and on again. As with pressing the restore switch, it doesn't remove very much from the computer's memory, and most of what is in there can be retrieved relatively easily.

The RUN/STOP key itself is another dual-purpose key. Pressed by itself it will do nothing unless a program that hasn't disabled the key is running (to disable it, type POKE 806,103 ; to enable, it type POKE 806,101), in which case it will stop execution of the program with a BREAK IN XXX error, where XXX is the line number that was being acted on.

If RUN/STOP is pressed in conjunction with the shift key, the computer will attempt to load the first file from disk, and the only way to get out of that is simply to sit back and wait. It saves you typing DLOAD"prog name",8 and then RUN, I suppose.

Of the other keys, the shift lock acts like a conventional typewriter shift lock key, and everything else simply displays something on the screen: usually a letter, a number, or a graphics character. However, there are exceptions.

Calculations

If we want to make the computer act like an ordinary calculator, there are four keys that perform mathematical operations. These are the multiplication key ('*'), the division key ('/'), the addition key ('+') and the subtraction key ('–'), so if we wanted to know what 27 times 15 plus 4 minus 256 was, we would type something like:

```
PRINT 27*15+4–256
```

followed by RETURN, which would give us our answer of 253. There are other symbols we could use, the up-arrow key for exponentiation for instance, as well as mathematical keywords such as SIN, COS and so on. However, we'll come to those in later chapters.

What we must beware of when using all these keys is the order in which we perform these mathematical calculations. The above sum, for instance, could be expressed in many different ways:

```
PRINT (27*15)+4–256
```

```
PRINT 27*(15+4)–256
```

```
PRINT 27*(15+4–256)
```

all of which would give different answers. The important thing here is the use of brackets, which determine in what order the calculations will be carried out. As with everything else on the C16, experimentation is the key, so try doing a few sums and seeing what the answers are.

Before we introduce you to some of the features of the machine, type in the following short program and see what happens.

```
1  A$="[CU,CD,CL,CR]":PRINTMID$(A$,INT(RND(.5)*4+1),1)"
+[CL]";:FORI=1TO50:NEXT:PRINT"[RVS,SP,CL]";:GOTO1
```

By using ? as shorthand for PRINT, N shifted E as shorthand for NEXT, and F shifted O as shorthand for FOR, you will be able to get all this on one line.

Please note that CU, CL, CD and CR are just notations used to express the result of typing the cursor up, cursor left, cursor down and cursor right keys respectively when in quotes mode (don't type in the commas!), RVS means type CTRL and '9' simultaneously, and SP means leave a space. Again, don't enter the commas.

Some golden rules

Try modifying as many programs as you can, learning from the original along the way, because this will teach you more about programming than anything else ever can. Buy some computer magazines and type in the listings from them, get some of the tape-based magazines that are on the market, get hold of as much public domain software as possible, and see how it all works.

And don't just restrict yourself to programs written specifically for the C16. Many other interesting programs are printed for other computers which, with a little time, effort and patience on your part, can be made into working programs for your computer. Most of the better magazines give program conversion hints anyway, although it will not be possible to convert every program that you come across.

Before long you'll be building up a useful set of program subroutines. Of program what? We'll get to that one in a moment.

Running programs

When you finally set about writing your own programs for your own use, you must first of all decide what area of program writing you are going to concentrate on during any one particular writing session. You can't start out with the idea of writing a game of Space Invaders and then half way through suddenly decide that your time could be better spent by writing an integrated accounts package, for instance! Well, perhaps some of you could, but if so I very much doubt that you'll be spending time and money on this book.

So, settle on the area and decide that for the next hour, three hours, or however long you can spare at the keyboard, that is what you are going to write. Having done that, there are a number of other points that must be settled before you actually start typing in the code.

Most important, you must have a good idea in your head, and preferably a written idea on paper, of precisely what you are going to do, and how you are going to structure the program. Some people may sit down at the keyboard and just type, modifying the program as they go along, with no pre-conceived idea of how the program is going to develop. People like that probably do produce reasonable programs that work, but inevitably they could be far better structured, could work faster and could achieve more. But, for the majority of us, it is impossible to work in this way, and we must resort to primitive aids like paper and pens to help us out.

In order to do this, you must map out precisely what you want the program to do, and not be deterred when a particular section of it stubbornly refuses to work in quite the way you want it, as will almost always happen. No doubt you're doing something wrong, tackling the problem in the wrong way, and should perhaps be approaching it in a different manner. With luck, the computer itself may be able to give you a hint about where you're going wrong, but then again ...

By mapping out beforehand everything that you want to do, the risk of this sort of problem occurring is lessened.

Sorting out programs

And you must write down rather more than just 'I think I'll write an Asteroids program, there'll be forty aliens and one spaceship'. This tells you nothing about the program, nor how it will ultimately be put together. Each section of the program should be thoroughly written out before sitting down at the computer. For instance, in our Asteroids example, you would need a routine to move the ship around, which takes into account the fact that you don't want the ship to move off either the left, right, top or bottom sides of the screen. How would you write that routine, how would you make the ship move, how will you check whether you're at the edge of the screen?

Think about it, rather than just ploughing in and hoping that, by changing a few numbers in the program every now and again, it will eventually work. Not only will this probably not work, but it will also teach you absolutely nothing about programming. By some fluke you might get it right, but you'll have no idea how or why it works.

When you get used to writing everything down beforehand, you'll soon realise that there is a lot more involved in writing a program than you might at first think. All sorts of problems will occur that you hadn't thought about, and five minutes' pencil work sorting it out will save much more time than that at the keyboard. Look at some commercial programs such as databases and word processors, and then you'll realise that they can't possibly have been written by just sitting down at the keyboard and thinking 'I'll write a word processor today'.

Again, when you're firmly in the habit of writing everything out beforehand, so that the program organises itself into various well-defined blocks - move ship left, move aliens towards the centre of the screen, etc. - you'll soon realise that various parts of the program are familiar from other programs that you may have written earlier.

Rather than re-inventing the wheel, why not use the routine that you wrote in that other program? Answer: because you can't find it.

Some programming lessons

This should teach you another lesson about program writing. Always make sure that you write your programs so that, when you come back to them in six months, you know precisely which part of the program is doing what.

To make this easier, the C16 has a statement called REMARK, usually shortened to REM, that allows you to insert comments into your program listings. In other words, at the start of our 'move ship left' routine, you could have a couple of program lines something like:

```
1000 REM THIS PART OF THE PROGRAM
1010 REM MOVES THE SHIP TO THE LEFT
```

Thus you'll never be in any doubt as to what that section of the program does. Inserting comments like this throughout your listings, you'll soon be able to find your way through them again and spot useful routines that could be used under a similar set of circumstances in a different program.

This habit of putting comments in your programs will not only help you, but also anyone else who might be looking at them. You might submit a program to a magazine for possible publication: it will certainly help your chances of acceptance if the people at the magazine can read your program and see how it works without having to spend hours ploughing through an unintelligible maze of line numbers.

As we said earlier, you'll soon realise that the same sort of routines crop up again and again, and an extremely important program-writing 'trick' is to save all these common routines on a separate tape or disk from your more usual program-writing one. In other words, build yourself up a collection of useful routines, or, as they're more commonly called, subroutines. And don't forget to label the tape or disk. This is something that I personally don't do as often as I should, and I know from bitter experience that the time spent wading through 30 disks, loading up the directory of each one while trying to find a particular program, is wasted time indeed.

So what, precisely, is a subroutine ?

Subroutines

A subroutine is essentially a small program in its own right, which performs a specific function. For instance, it might be a routine to format numbers to 2 decimal points. You have a numeric variable, say A, equal to 12.6785434, which you want to look a little neater than that when you print it out on the screen, and so your routine truncates A so that it becomes equal to 12.68 (rounding the last number up or down as appropriate).

In a lengthy program you'll probably want to use this sort of routine time and again, and it would be rather tedious to type it out again and again. This is the purpose of a subroutine: a short program that performs a useful function, and which performs that function many times within a major program. The Basic statement used to call up one of these subroutines is the word GOSUB (hence SUBroutine), which transfers control of the program to the short sub-set in order to perform whatever function is required. To return to the main flow of the program, we use the word RETURN. We'll be coming to these in more detail later.

By building up a set of routines such as this your program writing time can be cut down quite effectively, and by writing everything out beforehand you can also make considerable savings on development time.

Where do all these routines come from? Obviously most of them will be written by you, but there are many publications available that detail programs for all manner of uses.

Magazines, books (*Five Billion Subroutines for the C16,* that type of thing), and what is referred to as public domain software, are all available to help you in your own program writing. With magazines and books, you're going to have to enter the code yourself, and perhaps modify the program along the way, but if you want the lazy way of building up a good set of programming subroutines, perhaps public domain software is your answer.

This kind of software is usually made available through some kind of user group, a collection of individuals dedicated to the use (usually) of one type of computer in particular, but occasionally encompassing a much wider range of microcomputers. By their own efforts they write, collate and gather as much programming material together as possible, and, for a small fee, this is usually presented to anyone who cares to join their group.

Examination of this software will almost certainly repay the effort (and small amount of finance) involved. Programming ideas, tips and hints will all be there, and they can all be accommodated into your own programs.

As I said earlier, why re-invent the wheel? If someone else has written a routine for performing XYZ, why should you bother to write one? It may be lazy, but it will save you time and money, and time and money are important in a fast-growing industry like this.

As time goes by and your programming efforts produce better and better results, you will ultimately develop routines for use in many areas of software development. A whole collection of financial routines, screen movement routines, and so on, will be there to be included in programs as and when you require them.

This does not mean that you will be able to write a program in five seconds flat. Routines may have to be altered slightly, maybe changed around to fit the requirements of a particular area, and you will still have to come up with the ideas in the first place. No one else is going to do that for you. None the less, you will find programming becoming significantly easier.

Many of the routines you come across will not be written for your own particular computer. They can (usually) be easily adapted for your own machine, and are always worth examining.

Having all these routines at your disposal, you might think that you know all about your machine. But the story, as usual, goes much further than that.

By building up a collection of useful program subroutines, we lessen the length of time taken to produce a working program, and also cut down extensively on program development time.

All these subroutines are, ideally, stored away on tape or disk for later recall whenever appropriate. Something that you will learn almost as soon as you start developing, or simply coming across, these programs is the need for an organised filing system of some sort, which makes it easy to retrieve the right program at the right time.

In an ideal world, we would all have access to exactly the right kind of computer equipment for every job, but in the real world we must recognise that most people will not be able to afford all the equipment that they would like, and so sacrifices have to be made.

Program tidiness

You might have a set of programs for use in financial programming, all stored on the same disk or tape. If disk, wouldn't it be nice to have a printed list of all the programs on it (disks have something called a Directory, which is a fancy name for - yes, a list of all the programs on the disk) stored with the disk?

Or, if you're using tape, you might want a list of everything on that tape, together with, if your cassette deck is up to it, a record of the place on the tape where each program starts. It is always useful to note what number the tape counter has reached whenever you get to the end of a program.

If you can't afford a printer, all this work will have to be done longhand, which can, of course, soon become a tedious process. However, as with many things in the computer world, the time that you ultimately save will be ample compensation for the initial drudgery involved.

Inevitably, there will come a time when the routine you need to perform a particular function just will not fit into another program without some drastic operations being performed on it. This will probably occur a good many times in the course of program development, and it will probably be a darn sight quicker to write a new routine from scratch rather than trying to bend another routine to fit.

But enough of theory. Let's get down to practice and start writing some simple programs, introducing new commands gradually as we do so.

2

Basic Programming

When first starting to program, there are some important lessons to be learnt, and chief among these at first are the concepts of line numbers and program editing. Fortunately the C16 has got one of the finest program editors of all home computers, so the time taken to learn how to alter a program is minimal.

Type in the following, exactly as shown:

```
10 PRINT "I AM A COMPUTER"
20 PRINT "AND I'M ONE TOO!"
30 PINT "BUT I'M NOT"
40 PRINT:PRINT:GOTO10
```

When you run this program, the result will be that the first two sentences ('I AM A COMPUTER' and 'AND I'M ONE TOO!') will be printed, and then the program will halt with the words SYNTAX ERROR IN 30.

Using the cursor keys, move back so that the cursor is over the letter I in PINT in line 30. Press the shift key and the INST/DEL key once, to move everything one character to the right, insert the missing letter R, and hit RETURN. Voilà, a new line is entered, and if we run the program now everything will be printed out properly.

To repeat a line, for example have a line 35 the same as line 10, move the cursor until it's flashing over the 1 of 10, type 35 and hit RETURN. Again, if we now LIST the program we'll see that our new line is now in place.

To remove a line, simply type the line number by itself and hit RETURN. Try it now, by deleting line 35, and then type LIST just to prove it.

And line numbers? They're just there to tell the computer in what order you want things to be executed. They should also serve to remind you as well!

Variables

Variables are a way of retaining information in the computer, so that it can be used again and again. As with all other aspects of programming, there are some important rules to be observed when dealing with variables. The first of these is that although a variable name can be as long as you like, always remember that the C16 is only going to recognise the first two letters of that name. Secondly, a variable name cannot contain one of the so-called reserved words (words that the computer itself uses), so that names like AND1, OR3, NOTALOT, are not acceptable, since they contain the reserved words AND, OR and NOT respectively.

There are three types of variables allowed on the C16: numeric, string, and integer. The following program uses all three, and the point to note here is the way we differentiate between them.

```
10 A=27.12345
20 A$="HELLO MA BOY"
30 A%=27
40 PRINTA:PRINTA$:PRINTA%
```

If we now RUN this program, the values 27.12345, HELLO MA BOY, and 27 will be printed out on the screen. These are the values contained in the numeric variable A, the string variable A$, and the integer variable A%. It should, from this example, be obvious that string variables are suffixed by a dollar sign, integer variables by a percent sign, and numeric variables have no suffix whatsoever.

One could equally have had something like:

```
10 A1=27.12345
20 A2$="HELLO MA BOY"
30 A3%=27
40 PRINTA1:PRINTA2$:PRINTA3%
```

and the result would have been exactly the same. Just remember the rules about recognising two letters only, and not using reserved words.

Getting information in

There are a number of ways of getting information into the C16, and one of these involves using the INPUT statement, which works in the following way:

```
10 INPUTA$
20 PRINTA$
```

Here the computer would print a question mark up on the screen, and wait until you type something and hit the RETURN key. Whatever you type in will be stored in the variable A$, and printed out in line 20. If you typed nothing, but just hit the return key straight away, A$ would hold nothing, and would be called a 'null' string. If you altered the program to read:

```
10 INPUTA
20 PRINTA
```

the computer would be expecting you to type in a number, since the variable A is a numeric one. If you typed in a string instead (such as 'HELLO') the message 'BAD DATA ERROR IN 10' would appear, and you'd have to try again. If it's a numeric variable that you're declaring, then it's a numeric variable that you must type in, and the computer won't let you type in anything else.

There are other methods of getting information into the computer, but we'll come to them later.

STOP, END and CONTinue

Although most commercial programs are designed so that you can't break into them, when writing and developing your own programs you will often find it necessary to break into them to examine 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 (which means, of course, that 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 exist on the C16 to enable you to stop and start programs in this way: STOP, END and CONT.

STOP can be used as in the following example:

```
10 PRINT "[CLR]"
20 X=5:Y=15:Z$="LAUREL"
30 STOP
40 X=X+1:Y=Y+10:Z$=Z$+" AND HARDY"
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 X,Y,Z$, you will see that they have the values as listed in the program, namely 5, 15 and 'LAUREL'. 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, the computer will just return to READY mode and the flashing cursor will be sitting there waiting for you to type something in.

If we now print out the three variables again, we'll see that X is now equal to 6, B is equal to 25, and Z$ is now equal to "LAUREL AND HARDY". Strings can be added just like anything else, but instead of producing greater numbers they just get longer. This process of adding strings together is termed string concatenation. If you keep adding strings together, you will eventually get 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, so don't create a string with any more characters than that.

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, because 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. Some computers, like the Spectrum, do allow you to change things in this way and continue with a program afterwards, but alas the C16 isn't one of them.

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. Another technique, useful when trying to trap an error in a long program, is to have a dummy line that periodically prints out the values of various variables. In this way you don't have to keep stopping the program to see what's happening, and the dummy line can always be removed afterwards.

DATA and READ

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, or entering a lot of information. 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 in that same program. Usually, data cannot be passed from one program to another, although there are a couple of ways of doing this which we'll look at later.

Not only do data statements save you typing in vast amounts of data each time you run the program, they also allow 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 X
25 IF X=0THEN40
30 Y=Y+X
35 GOTO20
40 Z=Y/10
45 PRINT "[CD]THE TOTAL IS ";Y
50 PRINT "[CD]AVERAGE NUMBER READ = ";Z
60 END:REM NOT NECESSARY, BUT USUALLY A GOOD IDEA.
100 DATA9,8,7,6,5,4,3,2,1,9,0
```

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. A word of caution, though. If you're using string data, it isn't necessary to put that data within quotes. However, if you miss the quotes out, you won't be able to use a mixture of upper- and lower-case characters, just lower case only.

What you must remember is that data is read as it is encountered, so wherever it happens 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 a command available on the C16, called the RESTORE command.

RESTORE command

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

```
62 RESTORE
63 GOTO 20
```

Alternatively, you can RESTORE to a specific line of data by using the command:

RESTORE 20 (or whatever line number you wish).

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 62 and 63. 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 computer 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 W$,X,Y,Z
25 IFW$="END"THENEND
30 PRINT "[CD]DATA READ = ";W$;X;Y;Z
40 GOTO20
50 DATA STRING,1,2,3
60 DATA ROPE,4,3,6
70 DATA END,0,0,0
```

When run, this will just print out the data as it is encountered. This continues until W$ contains the word 'END', at which point the program will finish.

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 on the C16, as we've seen, 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 Z=2:Y=10:X=40:W$="HELLO THERE"
40 REM ADD NUMBERS TOGETHER
50 V=Z+Y+X
60 REM AND PRINT OUT RESULT
70 PRINT "THE TOTAL IS ";V
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 *C16 REM PROGRAM          *
30 REM * WRITTEN ON MAY 9 1985*
40 REM *BY PETE GERRARD          *
50 REM *FOR DUCKWORTH            *
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 without affecting what comes before them:

```
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 hide experimental pieces of code 'temporarily' within a program, for bringing out later:

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

GET, IF and THEN

The GET command can be used in many ways. One such is by 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. A$ still contains a null string because nothing's been pressed) 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. Part of our program might look something like:

```
100 GET A$:IFA$< >" "THEN100
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 that someone to press the Y or N keys. There are a number of ways of doing this, but here is just one way you might code that particular part of a program:

```
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 loop back to line 100 and wait until one of them is.

Another way of doing this might be as follows:

```
100 GET A$:IFA$< >"Y" AND A$< >"N"THEN100
110 IFA$="Y"THEN go to 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 than the previous example, 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:

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

Thus our program might contain a line something like this:

```
100 IF X <= Y THEN200
```

Thus, if X is less than or equal to Y then we go to line 200. If X is greater than Y we simply slip through to the next line of the program. Strings on the C16 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. For instance:

```
100 A$="FRED":B$="BERT"
110 IFA$<B$THEN200
120 PRINT"BERT IS GREATER THAN FRED!":END
200 REM PROGRAM NEVER GETS HERE
```

Subroutines and strings

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 Z=X+Y
20 U=V+W
30 R=U+Z
40 PRINT R
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. Even a simple 'wait until the space bar is pressed' routine may crop up time and time again, and the memory that a hundred such routines soaks up is quite large.

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.

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

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) 12.345. The command:

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

will print out the string 12.345, although the number A has remained the same.

This command is more useful when assigning variables, so the following program shows this in action along with lots of other commands to make life more interesting:

```
10 X=12.345678
20 X$=STR$(X)
30 PRINT X$
40 PRINT LEN(X$)
50 PRINT MID$(X$,1,2)
60 PRINT MID$(X$,4)
```

When run, this program will print out the following:

```
12.345678
9
12
345678
```

So you can see that by finding the position of the decimal point, we can split a number up into its two components. A simple way to do this from within a program would be as follows:

```
10 X=12.34568
20 X$=STR$(X)
30 FORI=1TOLEN(X$)
40 Y$=MID$(X$,I,1)
50 IFY$<>"."THENA$=A$+Y$:GOTO70
60 IFY$="."THENB$=RIGHT$(X$,I+1):I=LEN(X$)
```

```
70 NEXT
80 PRINTA$,B$
```

Here we've used a FOR ... NEXT loop, which is something we'll be looking at more closely later on.

The VAL command is essentially the reverse of the STR$ command, as this takes a string and converts it into a number. It takes the following format:

```
PRINT VAL("12.12345")
```

This would print out the number 12.12345, not exactly a useful exercise. However, if we have a number of strings assigned, VAL becomes a lot more useful. For instance:

```
10 A$="12.43532"
20 B$="77AA77"
30 PRINTVAL(A$):PRINTVAL(B$)
```

This would print out the numbers 12 and 77, since VAL stops as soon as it comes across something that is not a number. Thus our earlier program for finding the position of a decimal point within a program could be amended, and considerably shortened, if one were to use VAL to find out where the decimal point occurs. Try it, as a programming exercise.

As with most other things in the programming world, VAL can also be used to assign values to other variables, like this:

```
10 A$="54323.323"
20 B=VAL(A$)
30 PRINTB
```

When run, this would print out the value now stored in the variable B, which would be 54323.

LEN

LEN, as one might imagine, is associated with the LENgth of a string. For instance, if we assign a string X$ to be equal to "I AM A STRING", the command:

```
PRINT LEN(X$)
```

would return a value of 13, this being the number of characters (including spaces), contained within the string X$. We can also assign another variable to be equal to the length of a string:

```
10 X$="ANOTHER STRING"
20 Z=LEN(X$)
30 PRINTZ
```

Running this would give us the result 14, this being the value of the variable Z, in other words the number of characters in the string X$. LEN comes into its own when taken in conjunction with the next set of string commands.

MID$

This is the most powerful and flexible of all the string handling commands, and is 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 the place where this book was written, Manorfield. So, if we say A$="MANORFIELD", A$ 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$="MANORFIELD"
20 PRINT MID$(A$,1,5)
```

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

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

```
10 A$="MANORFIELD"
20 B$=MID$(A$,6,5)
30 PRINT B$
```

would result in the string FIELD 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$="MANORFIELD", the command:

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

would print out the word ORFIELD.

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 towns, we'll assign the string A$ to equal "AXMOUTH".

When we issue the following command:

```
PRINT LEFT$(A$,2)
```

the result is printed to the screen as AX. 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$="AXMOUTH"
20 B$=LEFT$(A$,4)
30 PRINT B$
```

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

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

RIGHT$

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$="LONDON", the command:

```
PRINT RIGHT$(A$,3)
```

would print out the word DON.

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

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

```
10 A$="LONDON"
20 B$=RIGHT$(A$,5)
30 PRINT B$
```

and print it out as ONDON.

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

Take the following short program:

```
10 A$="DUCKWORTH"
20 B$=LEFT$(A$,4)
30 C$=MID$(A$,5,5)
40 D$=RIGHT$(A$,3)
50 PRINT B$;C$;D$
```

When run, this would define the variables to be, respectively, DUCK, WORTH and RTH.

To illustrate further, one could go about reversing the direction of a word, like this:

```
10 A$="DOLLARS"
20 B$=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 SRALLOD is printed out. Of course, you could get smart and start using palindromes!

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

CHR$ and ASC

These are similar to the STR$ and VAL commands that we have already encountered, in that both give values for numbers and strings respectively and, like those two commands, they are in effect the reverse of each other.

CHR$ takes a number as its argument, and has the following format:

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

This will print out the character string 147 (the equivalent of clearing the screen). We encountered CHR$ earlier, when we saw that CHR$(8) and CHR$(9) would inhibit and enable the action of the shift and CBM logo keys. Some CHR$ characters aren't nearly as interesting as this though. For instance:

```
PRINT CHR$(65)
```

would just print the letter A on the screen. As with everything else, we can define strings to be equal to various CHR$ characters. For example:

```
10 C$=CHR$(13)
20 PRINT C$
```

When run, it doesn't look as if this program is doing very much, but it has assigned the CHR$(13) to be stored in C$. CHR$(13) is in fact the number for a carriage return.

ASC is short for ASCII, which itself stands for American Standard Code for Information Interchange. However, like most home computers, the Commodore version of ASCII is not amazingly standard, and you'll have to look in the user manual to find out what all the ASC characters are.

To take just a couple of examples:

```
10 A=ASC("A")
20 PRINTA
```

or

```
10 A$="B"
20 PRINT ASC(A$)
```

This would result in the numbers 65 and 66 being printed out respectively, these being the ASCII codes for the letters A and B.

FOR ... NEXT loops

A FOR ... NEXT loop enables us to perform a particular action FOR a certain number of times. The general syntax to be observed is as follows:

```
10 FOR I = 1 TO 100
20 PRINT I
30 NEXT
```

Here the action to be performed is a simple one: print out the value of the variable I. All that will result in this program is that the numbers 1 to 100 will be printed out in rapid succession. Line 10 starts the loop up, by instructing the computer that we're going to be doing something 100 times. Line 20 is obvious: just print out the value of the variable I. Line 30 then says NEXT. In other words, if the value of I is not yet equal to 100, go back and perform the NEXT step around the loop.

There is one peculiarity about this in Commodore, and indeed other, Basics. When it has finished performing this loop, I will actually hold the value 101. Why? Because, when it gets to the NEXT statement the value of I is incremented by one, and then control goes back to line 10 to check whether or not I has reached the final value, in this case 100. If it has exceeded that then program control passes to whatever statement happens to come after line 30. If it hasn't, then carry on through the loop again. So, when I holds a value of 100, the loop will continue to be performed, and then I will be incremented by the NEXT in line 30. Back to line 10 again to check on the value of I, and since it now holds the value 101 program control goes to the next part of the program. You can check this by inserting another line in the program:

```
40 PRINT I
```

which will indeed print out the value 101.

Actually the syntax in line 30 is not quite correct. It should read:

```
30 NEXTI
```

This is because we can have more than one loop active at a time, like this:

```
10 FORI=1TO3
20 FORJ=1TO3
30 PRINTJ;I;
40 NEXTJ
45 PRINT
50 NEXTI
```

If we run through this program, on the first run through we'll get the values:

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

On the second time around the I loop we'll get:

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

And finally, on the third time around:

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

If it wasn't for that PRINT statement in line 45 we could have abbreviated line 40 to read:

```
40 NEXTJ,I
```

Up to 26 of these loops can be active at any one time, but don't exceed that value as you will get an OUT OF MEMORY error message.

We're not just limited to incrementing in steps of one either. For example:

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

Here we're going up in steps of two, so the display will be something like:

```
1
3
5
7
:
:
:
99
```

at which point the program will come to an end. You can also, if you wish, count backwards, like this:

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

in which case, we'll see something like:

```
100
98
96
94
:
:
:
2
```

Obviously, you'll have to be careful to match the FOR conditions with the right STEP instruction. You couldn't, for instance, have something like:

```
10 FORI=1TO100STEP-1
20 PRINTI
30 NEXTI
```

although it's worth running a program like that just once to see what happens.

GOTO

We've already encountered this statement in another setting. Basically it sends command of a program to somewhere else within the program, or back to the same line (as in earlier examples when we were discussing the GET command).

The syntax used is GOTO xxxx, where xxxx is an existing line number. If it isn't, you'll get the UNIDENTIFIED STATEMENT error for attempting to go to a line that doesn't exist. Unlike, say, Spectrum Basic, which would just drop through to the next line of the program if it couldn't find the specific one that you asked it to go to, C16 Basic is nothing like as tolerant, so you have to be a bit more careful.

As a short example of the GOTO statement, type in the following:

```
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 as the program attempts to clear the screen every few microseconds. To use one of the other features of the machine, how about this example:

```
10 PRINT "[CLR]"
20 PRINT "[HOME]"TI$
30 GOTO 20
```

GOSUB and RETURN

We have already encountered subroutines as small, or maybe even large, segments of programs that have to be repeated many times.

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 xxxx, where xxxx 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. Just 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. For example:

```
10 FORI=1TO10
20 GOSUB1000
30 GOSUB2000
40 NEXTI
50 A=A+1
60 PRINTA
70 END
1000 A=A+1
1100 RETURN
2000 A=A+2
2100 RETURN
```

When run, this program will leap about all over the place. Can you work out what value will be stored in A when it's finished? It'll probably be easier to run it than to try to work it out in your head. Of course, one can get a lot more complicated than that.

```
10 PRINT"[CLR]"
20 X=2:Y=4:Z=8
30 GOSUB1000
40 GOSUB2000
50 GOSUB3000
60 PRINTX,Y,Z
70 END
1000 X=X+Y+Z
1100 GOSUB2000
1200 RETURN
2000 GOSUB3000
```

```
2100 X=X+Y+Z
2200 GOSUB3000
2300 RETURN
3000 X=X+1
3100 RETURN
```

What value will X 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 X = 1 THEN 100
20 IF X = 2 THEN 200
30 IF X = 3 THEN 300
40 IF X = 4 THEN 400
50 IF X = 5 THEN 500
60 etc.
```

Although this works, it can 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 X GOTO 100,200,300,400,500
```

Here, if X has the value 1, the program continues execution at line 100 onwards, a value of 2 and it goes to line 200, and so on up to a value of 5, when it goes to line 500.

The value used for X can be varied, like this:

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

Unfortunately, we can't use programmable GOTOs or GOSUBs.

Just one example of this command in operation could be something like this - an interesting use of string handling:

```
10 B$="ABDCDEF"
20 GETA$:IFA$=""THEN20
```

```
30 FORI=1TOLEN(B$)
40 IFA$=MID$(B$,I,1)THEN80
50 NEXT I
60 PRINT "NO KEY SELECTED."
70 GOTO20
80 ONASC(A$)-64GOTO100,200,300,400,500,600
90 END
100 REM WHATEVER COMES HERE
```

and so on, for all of the other options.

RaNDom INTegers

Like most of the home computers currently available, the C16 comes supplied with a random number generator. In common with them, it isn't particularly random, and so a few 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 random number in the range 0 to 1. To start things off differently every time, we'll need to use the C16's internal clock. 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 from the base number seed. 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 basically chops off the numbers after the decimal point, so INT(2.12) becomes 2, as does INT (2.99). INT of a negative number returns the next lower number. Thus, INT (−2.12) 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(RND(.5)*(X-Y+1)+Y)
```

DIMension statements

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 letters 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 like this:

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

We now have the eleven different numbers stored in A(0), A(1) etc. up to A(10) (since 0 to 10 does, of course, cover eleven numbers).

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
```

Or even, much more simply:

```
25 PRINT A(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 want to use a lot more than eleven elements in an array, and this is where the DIM statement comes in. An array, in C16 Basic, defaults to have eleven elements in it, numbered from 0 to 11. If you want to use any more than that, you must use the DIM statement.

The syntax for this is DIM A(200), 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 and effectively.

A useful trick, if running low on memory space, is to use something like DIM A(2), if we're only going to need to store a maximum of three numbers in the array A. Any array you refer to in your program automatically has 11 elements of memory space reserved for it, 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, as fans of Einstein will be pleased to know. 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 eighth column of the sixth 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. The following short program will serve to illustrate this.

```
10 DIMA(10,10)
20 FORI=0TO10
30 FORJ=0TO10
40 A(I,J)=J
50 PRINTA(I,J);
60 NEXTJ,I
```

3

More Basic Programming

WAIT command

One of the most under-used commands in the C16's vocabulary is the WAIT command, although this is probably because the manual accompanying the C16 hardly mentions it at all, and even when it does is extremely difficult to understand.

The WAIT command performs exactly the sort of function that you'd think it would: it waits for something to happen. That something can be a wide variety of different things, such as a key being pressed, some external device being operated, or a joystick being moved. It doesn't matter what it is, provided that we've used the command properly. The syntax to follow is:

```
WAIT X,Y,Z
```

where X lies in the range 0 to 65535 (you may recognise this number as the largest line number you can have in a program), and Y and Z lie between 1 and 255 (another number that will become all too familiar, as we will see fairly shortly).

You don't have to use the final (,Z) argument, in which case it defaults to zero. So what are these three values? 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 it works, then read on. If not, turn to page XXX before coming back here again.

Technically, WAIT looks at the content of memory location X, exclusively ORs it with Y, and then ANDs the result with Z, 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 4 KEYS TO CONTINUE"
20 WAIT 239,4
40 PRINT "[CD]WELL DONE."
50 REM CARRY ON WITH PROGRAM
```

After printing a message on the screen, line 20 looks at memory location 239, and the result of ANDing this with 4 is to wait until 4 keys have been pressed. We'll see later, in a different way, 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 anything that requires something to be pressed, 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 while pressing the keys, you may think you've only pressed 'Y' once, but as far as the C16 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, someone must already have pressed it', and moves on through the program: usually the last thing you want to happen. This can be cured by using a line like:

```
10 FORI = 1TO10:GETA$:NEXT
```

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 163: increments every 65536 jiffies (18.2 minutes)
Location 164: increments every 256 jiffies (4.2 seconds)
Location 165: increments every 1/60 second (1 jiffy)
Location 198: returns a unique value for the key being pressed
Location 239: 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 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 perhaps is that 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= 32768 TO X + 8191:PRINTX,PEEK(X):WAIT 198,64: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.

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 198,64:WAIT 198,64,64 for any key control.

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 X>20 AND Y<10 THEN 200
```

This means that if the variable X has a value greater than 20, AND the variable Y has a value less than 10, then program execution branches to program line 200. Otherwise, just carry on to the next line of the program.

And again:

```
10 IF X>20 OR Y<10 THEN 200
```

Almost the same line of code, but not quite. In this case, the program will branch to line 200 if X is greater than 20, OR if Y is less than 10.

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 (X<20 OR Y>10) AND A$="Y" THEN 200
```

Here, the program will branch off to line 200 if X is less than 20 or Y is greater than 10, but only if A$ is equal to Y as well. If all 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 X=20
20 PRINT (X>15)
```

When run, this program will print out the value -1, because the statement X>15 is a true one: we've just defined X 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 example:

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

will result in 0 being printed, as the statement is patently not true. We've defined X in line 10 to be equal to 20, and this is certainly not greater than 25.

All of this is based on what are called Truth Tables, and the table for AND is as follows:

X	Y	Z
0	0	0
0	−1	0
−1	0	0
−1	−1	−1

The table works in the following way: if the first statement X is false (zero) and the second one Y is false (zero), then the result Z is also false.

If the first statement X is true (-1) and the second one Y is false, the result Z is also false. Only if both statements are true is the final result also true. Any other combination produces a false result.

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

X	Y	Z
0	0	0
0	–1	–1
–1	0	–1
–1	–1	–1

Here, if either X OR Y 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 X=12:Y=6:A$="Y"
10 IF (X<10 OR Y>5) AND A$="Y" THEN 200
```

To explain. X is not less than ten, so we have a false (0). Y is certainly greater than 5, so we have a true (–1). A false ORed with a true give 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, in program listings you may have seen statements like:

```
X=Y AND Z
X=Y OR Z
```

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 X equal to 53, and Y equal to 153. 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, 53 becomes 00110101
153 " 10011011

Remembering the rules for AND and OR, 53 AND 153 now becomes 00010001 (only twice do we have two 1s together), and 53 OR 153 becomes 10111111 (if a 1 occurs in either number, the result is a 1). In plain English then, 53 AND 153 equals 17, and 53 OR 153 equals 191.

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 later on, and explain in more detail how this method of counting works when we encounter PEEKs and POKEs.

Other logical operators

These have been mentioned earlier in statements like:

IF X >5

and so on.

Knowing how truth tables work, it now becomes a simple matter to understand all these logical operations, 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

A large number of trigonometrical functions are available on the C16, and there are also ways of defining your own funcitons. We'll cover both these topics here.

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 20*(40-1.2*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 20*(40-1.2*17)
PRINT 20*(40-1.2*5.9)
PRINT 20*(40-1.2*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 C16 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 defining a function is:

```
DEF FNA(Z)=20*(40-1.2*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 560 (40 – 1.2*10, all multiplied by 20)

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

```
DEF FNA(A)
DEF FNA(1)
DEF FNA(B)
```

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

Playing for time

The reserved variable TI, and its string counterpart TI$, have appeared before, but with little explanation of what they do. TI and TI$ both relate to the real-time clock which is built into the C16. TI is updated every 1/60 of a second, starting from when you turn the computer on, or whenever you reset TI$, as this can be treated like an ordinary variable. 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 (such as disabling the stop key), 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, as in the following:

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

which sets the time at 12 hours, 30 minutes, and 45 seconds. This will also alter TI, but alas you can't say LET TI=1. TI$ can be split up into its component parts in the following manner:

```
10 PRINT"[CLR]THE TIME IS NOW :"
20 TI$="000000"
30 A$=LEFT$(TI$,2)
40 B$=MID$(TI$,3,2)
50 C$=RIGHT$(TI$,2)
60 PRINTA$;"HOURS ";B$;" MINUTES AND ";C$;"
SECONDS":PRINT"PRECISELY[2CU]"
70 GOTO 30
```

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

SIN, COS and TAN

We mentioned earlier on that your C16 could be used as an extremely expensive calculator, and showed you how to perform various mathematical calculations on it.

Built into the C16'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 they are the only trigonometric functions that are actually readily available, apart from ATN (see below). The rest of them you'll have to define yourself.

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.

In case your knowledge of mathematical functions is a little bit rusty, or has perhaps seized up altogether, the following diagram illustrates what we're talking about.

```
X
*
* *
*  *
*   *
*    *
*     *
*      *
******** Z
Y
```

The sine of the angle at X is equal to the length of the OPPosite side YZ divided by the HYPotenuse (the side opposite the right angle) XZ.

The cosine of that angle is equal to the ADJacent side (XY) divided by the HYPotenuse (XZ), and the tangent is equal to the OPPosite (YZ) divided by the ADJacent (XY).

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

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:

X degrees = (X * PI)/180 radians
Y radians = (Y * 180)/PI degrees

PI is the well-known mathematical symbol (accessible from the keyboard under the equals key), 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 C16.

SQR

This returns the square root of a number, and the syntax for using it 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

Remember the days of naperian and natural logarithms? 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 later on.

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 A is positive, 0 if it's zero, and −1 if it's 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

This tells you how much memory is still left in the computer, and takes the syntax:

```
PRINT FRE(0)
```

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

PEEK and POKE: a technical interlude

We will now take a close look at PEEK and POKE, counting systems, and how binary really works. We'll need some of the knowledge gleaned from the last few sections, since we're also going to be mentioning AND and OR again, in order to explain how some of the more complicated aspects of these commands work. In addition, we'll round off with a brief look at SYS and USR commands.

In their simplest form, PEEK and POKE are quite straightforward commands. POKE allows us to alter the contents of a memory location, and one or two extensions to the command allow us to alter those contents selectively. In order to know what we're altering though, we need to know a little bit more about how other counting systems work.

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.

Bits and bytes

This is the last bit of definition, before we get down to the nitty-gritty. 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 C16 is called an 8-bit computer precisely because each byte contains 8 bits.

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

For instance, supposing we POKE 3072 with 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 3072 with 128, we're simply altering bit 7, and so on.

We've glibly been talking about PEEK and POKE here, but what do these commands actually do? To give very brief definitions, POKE allows you to alter the contents of a memory location, and PEEK allows you to see what numbers are stored in various memory locations. They take the following syntax. For POKE:

```
POKE XXXX,Y
```

where XXXX is any one of the 65536 memory locations that the C16 can look at, and Y is any number between 0 and 255. No larger than 255 as we've seen, since an 8-bit memory location cannot hold any more information.

And for PEEK:

```
PRINT PEEK(XXXX)
```

where XXXX is again any one of the 65536 memory locations on the C16, but bear in mind that PEEK won't work correctly beyond the bottom 16K of memory. We can also define variables in this way as well. For example:

```
A=PEEK(XXXX)
```

would place the content of memory location XXXX into the variable A.

Some computers are fortunate enough to have the commands DEEK and DOKE. This is the 16-bit equivalent of PEEK and POKE, which, as we'll be seeing in a moment, is a convenient command to have at times. However, the C16 doesn't allow us to use these commands, so we must stick to just PEEKs and POKEs.

If we want to alter the content of a memory location selectively, we must use the commands AND and OR again. Say we want to alter the content of memory location 3072, the top left-hand corner of the screen. If, for instance, we want to put a 65 into that location, the straightforward command:

```
POKE 3072,65
```

will do the job for us. However, there is a problem in doing this. Not only have we now switched on bits 0 and 6, but we have also switched off bits 1,2,3,4,5 and 7. If we want to set bits 0 and 6 (which combine to give us the value of 65) and leave the rest the same, we'd have to:

```
POKE 3072,PEEK(3072) OR 65
```

In other words, look at what is already in location 3072 and OR it with the number 65. Remembering our earlier discussion on AND and OR, you'll see that this will leave all the other bits as they were, and just set bits 0 and 6.

By being able to alter the contents of various locations, we can produce some interesting results. On the C16 we can create a 2K block of spare memory, starting at location 14336, by typing in the commands POKE52,55:POKE56,55. Since this memory is now spare, it means that we can POKE values into it, and leave them there for other programs to operate on. This is just one way in which we can transfer variables from one program to another, normally an impossible task.

If we have three variables, X, Y, and Z, equal to 1,2 and 4 respectively, and we want to keep those variables for use in another program, the following would achieve that:

```
A=14336:POKEA,X:POKEA+1,Y:POKEA+2,Z
```

We could then load our new program, or indeed alter the existing one (as you know, this resets all existing variables), and then have a line either in our new program or in the old, altered one, that reads:

```
A=14336:X=PEEK(A):Y=PEEK(A+1):Z=PEEK(A+2)
```

We mentioned earlier the commands DEEK and DOKE, and lamented their unavailability on the C16. Why should they be so useful to us?

Double precision

Quite often we want to put very large numbers into various memory locations, and as we've seen we can't POKE a number greater than 255 into a location. So what's the solution? The answer is to split the number up into two parts, and proceed from there.

Let's take a practical example. Say we want to put the number 1024 into a memory location. Obviously we can't do it, so the first step is to convert that number into hexadecimal format. In this case that is pretty straightforward, as 1024 in decimal is equal to \$0800 in hexadecimal. Splitting this up into two parts, we now get the numbers \$08 and \$00. This can then be POKEd into memory locations in the following way:

```
POKEXXXX,00:POKEXXXX + 1,08
```

Why the wrong way round? Memory locations always require that the smaller part of the number goes into the first location, and the larger part goes in the second location. For one thing, it's a darn sight easier for the computer to handle. For us, it's a minor inconvenience, as you just have to get used to reading everything back to front. To convert a two-byte number held in this way is as straightforward as the earlier procedure.

```
A = PEEK(XXXX):B = PEEK(XXXX + 1)
```

Say A was equal to 0, and B was equal to 8. Converting this into hexadecimal again gives us the number $0800, or decimal 1024, the same number as in our earlier example.

Screen animation

Since we can alter memory locations in this way, this allows us to display things on the screen. The screen, after all, is only another block of memory, and by altering the locations that go to make up the screen we can quite easily create moving images. Not as easily as with some of the graphics commands, of course, but then we haven't got to them yet!

The screen occupies memory locations 3072 to 4071, and so to put something in the top left-hand corner of the screen, we must POKE 3072 with something: say 65. This is the screen code for a heart, as a glance at the manual that came with your C16 will show you. In order to see that heart on the screen you'll need to give it some colour, so POKE 2048,1, for example, for a white heart. (More on colour later on).

To fill the screen with lots of hearts, try this:

```
FORI = 0TO999:POKE3072 + I,65:POKE2048 + I,1:NEXT
```

What if you want an animated heart? The secret, as any cartoonist will tell you, is to display the heart, keep it in place for a second, and then wipe it out before moving on to another location. As in the following short program:

```
10 S=3072:C=2048
20 FORI=0TO24
30 POKES+I*40,65:POKEC+I*40,1
40 FORK=1TO50:NEXTK
50 POKES+I*40,32
60 NEXT
70 FORI=24TO0STEP-1
80 POKES+I*40,65:POKEC+I*40,1
90 FORK=0TO50:NEXTK
100 POKES+I*40,0
110 NEXT
120 GOTO20
```

Not very exciting, but at least it works and displays the principle in action. Lines 30 and 80 actually plot the white heart, lines 40 and 90 are just a short display to enable you to see it, and lines 50 and 100 then blank out the heart by display character 32: a blank space.

The whole thing continues ad infinitum, so before you get too bored with it, pressing the Stop key will break into the program. Later on, we'll see how we can define individual characters to display, for example, little aliens bouncing about the screen.

SYS and USR

You may have seen program listings with SYS in them, and wondered what it meant. USR you probably won't have seen, since few people seem to use it, but for the sake of completeness we'll include it here.

SYS is short for SYStem call, and transfers program execution to a machine code routine that starts at the address specified in the SYS call. For instance,

```
SYS 62116
```

transfers program execution to the routine starting at location 62116. As this routine wipes everything out and virtually re-sets the machine, it is not advisable to use it just for the sake of something to do. The C16 has a whole host of such routines built into it, but most of these are inaccessible unless you're a budding machine-code programmer. Randomly calling up routines will not do much for your C16 (although you can't damage it in any way) and it certainly won't do much for your temper.

USR is an easier one to use, and is one way of passing variables from a Basic program to a machine-code one. The syntax to be observed is:

X = USR(Y)

which transfers the variable Y into a machine-code program whose starting address is indicated by memory locations 1280 and 1281. You can do a PEEK on these locations to see what they hold at power-on, and they can easily be altered with a POKE command. When the machine-code program returns back to Basic mode, the new value of the variable Y, presuming it has been altered by the program, or even if it hasn't, will be stored into the variable X. This can then be printed out with a straightforward PRINT command.

Well, that was all fairly technical, but knowledge of these things is important if we're to understand some of the more useful aspects of the C16 - graphics, for example. We'll be coming to that later, but before getting there we're going to take a look at some of the new commands available using Commodore Basic version 3.5.

4

New Basic Commands

The commands encountered in the last two chapters will be familiar enough to anyone who has used Commodore machines in the past. However, the C16 has a brand new version of Basic called Basic V3.5, and there are plenty of new commands now installed in the machine. Let's take a look at them.

Commands for convenience

Some of these new commands are there merely for convenience, and save having to type in strings of PEEKs and POKEs. For instance:

```
PRINT JOY(n)
```

where:

n = 1: position of joystick number 1.
n = 2: position of joystick number 2.

JOY returns the following values:

0: nothing's happening!
1: moving up.
2: moving up and right.
3: moving right.
4: moving down and right.
5: moving down.
6: moving down and left.
7: moving left.
8: moving left and up.

Adding 128 to any of these values means that the fire button has been pressed.

Similarly, the KEY command allows us to make active use of the function keys on the machine. Re-defining these on earlier Commodore machines required a machine-code program, but now the job is simplicity itself. The command takes the form:

KEY n,whatever you want the key to be.

Here n lies between 1 and 8, indicating which function key you want to alter. Say we wanted function key 1 to be equal to the Basic keyword GOSUB. The command to use would be:

```
KEY 1,"GOSUB"
```

If you wanted it to equal PRINT", then you'd have to enter:

```
KEY 1,"PRINT"+CHR$(34)
```

CHR$(34) being the quotes character. To add a carriage return on the end, you'll need to use CHR$(13). For instance, if we wanted key 1 to become MONITOR plus a carriage return, we'd use:

```
KEY 1,"MONITOR"+CHR$(13)
```

Now, pressing key 1 would cause us to enter the monitor.

A lot of the new commands are to do with graphics and sound, but these will be left to their appropriate chapters. For now, we'll stick to non-graphical/musical commands.

Of these, nine are to do with using a disk drive, some are utilities to aid in typing in programs and amending them, and the rest are additions and enhancements to Basic.

Using disk drives

Nine new commands in total, and taking them in alphabetical order we start with:

BACKUP

It seems a bit strange to include this command on the C16, since it will only work with a dual disk drive, and most of those cost many times more than the C16. Still, it is here, and its function is to allow you to make a direct copy of a disk in one drive onto a disk in another drive. It takes the form:

```
BACKUP D0 TO D1
```

which will copy everything on the disk in drive zero onto the disk in drive one. If you decide to change the device number of your disk unit the command can still cope with it. If your disk drive is now device number ten, for example, you'd need to use:

```
BACKUP D0 TO D1, ON U10
```

COLLECT

COLLECT basically tidies up the information stored on a disk. If, for instance, you've OPENed a file to disk and then not CLOSEd it, COLLECT will take care of this. It takes the form:

```
COLLECT D1, ON U10
```

where D1 refers to the disk in drive one, and U10 refers to disk drive unit number 10.

COPY

This allows you to copy files from one disk to another, or rename files on the same disk without deleting the original file. It looks like this:

```
COPY D0,"FRED" TO D1,"BILL",ON U10
```

which will copy the program called FRED on drive zero to drive one, renaming it BILL in the process, from the disk drive whose device number is number 10. The filename can be kept the same if required. Missing out file names copies everything from drive zero onto drive one, and missing out the drive numbers creates a new copy of the file with a different name.

DIRECTORY

This displays a list of everything that is contained on the disk in the selected drive on the selected disk unit. To be even more selective, you can ask it to look for specific files e.g. all those beginning with the letters BAS. It looks like this:

```
DIRECTORY D0,U10,"BAS*"
```

Or simply press function key 3 for the command in its basic form as the machine is turned on.

DLOAD and DSAVE

Two particularly pointless commands, since SAVE and LOAD work just as well. However, if you find it easier to type:

```
DLOAD "BILL"
```

than

```
LOAD "BILL",8
```

you may find them useful!

Their only real advantage is that you can specify strings rather than names:

```
DLOAD (A$)
```

which is a bit easier than trying to use LOAD to do the same thing.

HEADER

This is used when you want to use a new disk for the first time, and takes the form:

```
HEADER "name of disk",D0,lid,ON U10
```

or whatever. 'Name of disk' is whatever you want to call the disk, 'id' is a two-letter/number identifier, and D and U work in their usual ways.

These commands only serve to make life easier, since they could all be replaced with OPEN ... commands. For example:

```
HEADER "new disk",D0,I01,ON U10
```

could be replaced with:

```
OPEN1,10,15,"N0:new disk,01":CLOSE1
```

Each to his own, I suppose.

RENAME

This allows you to rename a file on disk, and takes the form:

```
RENAME D0,"FRED" TO "BILL",U10
```

SCRATCH

This allows you to delete a file from a disk, and takes the form:

```
SCRATCH D0,"FRED",U10
```

The equivalent of:

```
OPEN1,10,15,"S0:FRED":CLOSE1
```

Entering and checking programs

Our next set of new commands are basically just there to make life easier when entering and checking programs.

AUTO

This automatically generates line numbers. It takes the form:

```
AUTO n
```

where n indicates the increment between line numbers. I can't help feeling that some of these commands would be better omitted and the extra memory thus saved used for more RAM, but there you go.

RENUMBER

A genuinely useful one when it comes to saving on memory, RENUMBER renumbers every line in a program in a specified increment. It also takes care of all GOTOs, GOSUBs, etc.

It takes the form:

```
RENUMBER A,B,C
```

where:

A = new starting line.
B = increment between lines.
C = line of program to start with.

DELETE

This works in the same way as LIST, but instead of listing lines it deletes them from the program in memory. Thus:

```
DELETE 10-20
```

would delete lines 10 to 20 inclusive, and

```
DELETE 100-
```

would delete all lines in the program from line 100 onwards.

TRON and TROFF

These two excellently named commands turn 'trace' mode on and off respectively.

'Trace' mode will allow you to check the running of a program, and if there's an error in the program is a great help in finding where that error is. If you type:

```
TRON <RETURN>
```

and then RUN your program, you'll see each line number displayed on the screen as it is executed. By seeing which lines are executed in what order, you'll get a very good idea of where your program is or isn't going wrong.

MONITOR

This we'll come to in more detail in Chapter 5, but basically it allows us access to the C16's internal machine-code monitor.

Error handling

A number of new commands now make it a lot easier to track down errors in a program.

HELP is about as helpful as a toothpick when you're trying to fight your way through the jungle, although it does display the line in which a program error occurred. Since most error messages tell you SYNTAX ERROR IN 110, or whatever, there doesn't seem much point in including this command.

However, TRAP, ERR$, EL and ER are definitely a lot more helpful.

Take the following short program:

```
10 TRAP100
20 PINT "FRED"
30 STOP
100 PRINT ERR$(ER)" ERROR IN "EL
110 PRINT "MUST TRY HARDER!"
120 END
```

would result in:

```
SYNTAX ERROR IN 20
MUST TRY HARDER!
```

appearing on the screen. ERR$(N) prints up error message N, ER is the error message number generated by the error, and EL determines what line the error was in. By having line 10, if ever an error (apart from an UNIDENTIFIED STATEMENT ERROR, which don't get TRAPped) occurs in the program, execution will continue at line 100.

Further commands

HEX$ and DEC

Another couple of useful time savers, HEX$ returns the hexadecimal equivalent of a decimal number, and DEC returns the decimal equivalent of a hexadecimal number. They take the form:

```
PRINT HEX$(1234)
04D2
```

```
PRINT DEC("04D2")
1234
```

The two complement each other.

INSTR and GETKEY

Taking these in reverse order, GETKEY waits for a key to be pressed, and takes the form:

```
10 GETKEY A$
20 REM carry on with program.
```

The program just sits and waits until a key is pressed, and this value is then stored in A$.

INSTR is another string-associated command, and takes the form:

```
INSTR ("MY NAME IS FRED","FRED",SP)
```

which tells the C16 to look through the string "MY NAME IS FRED" and find where the string "FRED" occurs (the twelfth one in this case), starting at the SPth letter.

DOing LOOPs

Anything a FOR ... NEXT loop can do, a DO ... LOOP can do better. The DO ... LOOP can take a number of different forms, from a simple:

```
10 DO
20 A=A+1:?"FRED"
30 LOOP
```

which will just keep incrementing the variable A and printing "FRED" lots of times, to:

```
10 DO UNTIL A=50
20 A=A+1:?"FRED"
30 LOOP
```

which will continue the loop until A is equal to 50, or even:

```
10 DO WHILE A<>100
20 A=A+1:?"FRED"
30 LOOP
```

which will carry on with the loop as long as A is not equal to 100.

RESUME

This is used in conjunction with the TRAP statement discussed earlier. If you had something like;

```
10 TRAP 100
20 PINT "FRED"
30 REM CARRY ON WITH PROGRAM
40 REM ANOTHER LINE
:
:
:
100 PRINT "ERROR IN LINE ";EL
110 RESUME 40
```

program execution would go to line 40 after the error. If you had RESUME NEXT instead, execution would go to the line immediately after the error: line 30 in this case. Finally, if you just had RESUME on its own, the program would try again to execute the line with the error in it, which wouldn't do much good at all really.

What ELSE

An extension to the IF ... THEN statement encountered earlier, this is a valuable memory saver. For instance, we could have something like:

```
10 IFA=1ANDB=2THEN100
20 PRINT"FRED":STOP
100 PRINT"HELLO AGAIN."
```

Using the ELSE option, this could be changed to:

```
10 IFA=1ANDB=2THENPRINT"FRED":ELSE PRINT"HELLO AGAIN."
```

A saving of 13 bytes in only a three-line program! It makes programs a lot more legible as well.

And to finish ...

PRINT USING and PUDEF are the last two new commands on the C16, and PRINT USING is probably one of the most complicated of them all.

Basically, PRINT USING allows you to determine how text, whether alphabetic or numeric, will be printed out, and PUDEF allows you to alter the PRINT USING statement. The command takes the form:

```
PRINT USING "instructions";A,A$,12.435 etc.
```

"instructions" are taken from the following list:

Hash sign, plus sign, minus sign, decimal point, comma, dollar sign, up-arrow sign, equals sign, greater than sign.

To see how all these work, it's best to look at a few examples.

```
1 A=24.245:B=12345:C=345.1:D$="HELLO":E$="I'M A C16"
2 PRINT USING "$###.##";A,B,C
3 PRINT USING "+$#####.###";A,B,C
4 PRINT USING "###.##[4 UP-ARROW]";A,B,C
5 PRINT USING "#$,###.###";A,B,C
6 PRINT USING "###>#";D$,E$
7 PRINT USING "###>=#####";D$,E$

RUN <RETURN>
```

```
$ 24.25$******$345.10
+$   24.245+$12345.000+$  345.100
242.45E-01123.45E+02345.10E+00
   $24.245**********  $,345.100
HELLOI'M A
   HELLO   I'M A C16
```

The equals sign, the hash sign and the greater than sign are the only ones that can be used with strings, and they respectively select centring, number of characters, and right justification of a string within a field.

The others, plus the hash sign again, can all be used with numeric data. The hash sign again determines how many characters are to be printed. Plus and minus signs indicate that a plus or a minus sign is to be displayed, the dollar sign likewise indicates that a dollar sign will be displayed, the decimal point that a decimal point will be used (or the comma that a comma will be used), and finally the four-up arrows indicate that exponential notation will be used.

As I said earlier, probably the best way to get to grips with this is just to play about with it all. Like anything else in computing, it gets easier with practice.

Looking through my window

Although it is not a command as such, there is an interesting new feature on the C16. This is the ability to define a screen window, after which all action takes place within that window until you re-define it or press CLEAR/HOME twice and escape from it.

There is a simple way of defining a screen window. Using the cursor keys, position the cursor where you want the top left of the window to be, and press ESC followed by T. Then move to the position where you want the bottom right to be, and then press ESC followed by B.

Voilà! We have our window. This is all very well, but what if you want to define a screen window from within a program? You can't very well ask someone using your program to fiddle about with the cursor and define your window for you. You could use lots and lots of cursor control characters, but there is an easier way. A look at the memory map at the back of the book provides us with our answer.

```
10 POKE 2021,20:REM BOTTOM OF WINDOW
```

```
20 POKE 2022,10:REM TOP OF WINDOW
30 POKE 2023,10:REM LEFT OF WINDOW
40 POKE 2024,30:REM RIGHT OF WINDOW
50 PRINT "[HOME]HI THERE!"
```

Once you've created the window, pressing ESC followed by another key produces a myriad of functions.

KEY	FUNCTION
A	Automatic insert
B	Seen this one!
C	Cancels automatic insert
D	Deletes the current line, and
I	Inserts a line
J	Back to the beginning of the current line, or
K	Forward to the end of the current line
L	Turns on scrolling, or
M	Turns it off again
N	Returns to normal display size
O	Cancels just about everything (including quotes mode, reverse mode and flash mode)
P	Erases beginning of line to cursor position, or
Q	Erases from cursor to end of line
R	Reduces screen display
T	Seen this one!
V	Scrolls the screen up, and
W	Scrolls it down
X	Cancels whatever the last ESC function was

Unfortunately COLOR etc. commands still affect the whole screen and not just the window. Still, you could always write a machine code routine to take care of this. Do what? Read on ...

5

Using the Monitor

Introduction

One of the many features of the C16 that sets it aside from its predecessors the Commodore 64 and the Vic 20 is the inclusion of a monitor. On those two machines a monitor had to be loaded from tape or disk, or even cartridge, but on the C16 it's included as standard. So what, exactly, is a monitor?

First of all, it's nothing to do with a television monitor, or a dedicated computer monitor. That is what you use to observe the display generated by the computer. No, a monitor is an extremely valuable aid when it comes to programming in machine code.

In Chapter 12 we'll be taking a much closer look at machine code, but for now we'll concentrate on introducing the monitor, and talking about just some of the things that it can do.

To access the monitor on the C16 is nice and easy. Just type in the command MONITOR (or M followed by a shifted O) and RETURN, and there we are. The display may look a bit confusing to you at first, but there's really nothing to worry about. All it's telling you is the current values stored in a number of memory locations or registers. XR, for instance, refers to the X register. This in turn can be likened to a Basic variable. It's a bit like saying XR = 10, or whatever value you happen to choose.

As can be seen from the memory maps in the Appendices, most of the useful work done on the monitor lies from memory location $F445 to $F91F.

Monitor memory breakdown

Address ($)	Function
F445	Perform [MONITOR]
F44C	BRK/USR entry
F478	Perform ['R']
F4D7	Perform ['M']
F529	Perform ['>']
F54B	Perform ['G']
F570	Monitor Commands
F582	Addresses for monitor commands
F5CE	Perform ['C']
F5D1	Perform ['T']
F60E	Perform ['H']
F6CA	Print '?'
F6E0	Perform ['S']
F6F1	Perform ['L']
F6F3	Perform ['V']
F70A	Perform ['F']
F724	Perform ['D']
F91F	Perform ['A']

This in itself doesn't tell us very much about using the monitor. Rather, it tells us what memory locations look after the dozen or so commands available to us for using the monitor. However, if we don't know what the commands do, there's very little that we can do! So, taking each one in turn ...

Perform ['R']

Typing in R, followed by RETURN, displays the state of all the registers when the monitor was first entered. A typical display might be somthing like:

```
R
    PC   SR  AC  XR  YR  SP
;  0000  00  00  00  00  F9
```

Perform ['M']

This allows us to look at various memory locations, and returns not only the decimal value of what is stored there, but also the reverse field representation of any screen display characters that might be stored there. The command can be used in two ways:

```
M 0800 <RETURN>
```

on its own, displays the contents of the 96 memory locations starting at location $0800.

```
M 0800 0808 <RETURN>
```

displays the contents of the memory locations from $0800 to $0808. To alter them, just use the cursor keys to move the cursor to the desired byte of memory and type in the new value(s) followed by RETURN.

Perform ['>']

This is very similar to the M command mentioned above, in that it displays the contents of a range of memory locations. However, in order to save us typing:

```
M 0800 0800 <RETURN>
```

to look at just 8 locations, we can also type:

```
>0800 <RETURN>
```

A convenient shorthand.

Perform ['G']

This is similar to the Basic GOTO command. It transfers program execution to an address specified by the user, although of course in this case we're going to a machine language routine rather than a Basic one. It takes the form:

```
G F2A4 <RETURN>
```

where the address specified is naturally enough the one that program flow will go to. In the example given, $F2A4 is the start of the routine that resets the C16, so use it with caution. If you just choose random addresses, the 'G' command quite often generates a SYNTAX ERROR and drops you out of the monitor and back to Basic, so let that be a warning to you!

Just for fun, try G CDAB.

Perform ['C']

I shall be honest with you and admit that I can see no possible reason for including this command in the monitor, nor any possible use for it. Still, for the sake of completeness, it takes the form:

```
C 0800 0850 01 <RETURN>
```

where the first two addresses form a range for the command to operate on, and the last number is what it operates on them with. However, what it does is beyond me since it just prints out a vast number of memory addresses. Not the contents of them, just the addresses themselves.

Perform ['T']

A much more useful command, this is used to transfer a block of memory from one location to another. Useful if you want to copy ROM into RAM to alter it, or perhaps copy the character set from ROM to RAM for the same purpose. It works virtually instantaneously, and is a darn sight faster than Basic. The syntax used is:

```
T D000 D7FF 3800 <RETURN>
```

where the first two addresses form the block of memory to be moved, and the last one tells the computer where to put that block. The contents of the block of memory are unchanged, and (in this example) $D000 to $D7FF will also be left intact. So really you're copying a block of memory rather than transferring it, but since the letter 'C' is already used by the ridiculous ['C'] command, T is used here instead.

Perform ['H']

Analogous to the Basic FIND command supplied with some advanced Basics (but alas not the C16's), which allows you to specify a character, or set of characters, to be searched for in a given range of lines. Here we are searching through a block of memory for a specific hexadecimal number, or set of hexadecimal numbers, or an ASCII string. The command takes several forms:

```
H CDAB CDFF 88 <RETURN>
CDB5
```

or:

```
H CDAB CDFF 88 10 F5 <RETURN>
CDB5
```

or, finally :

```
H 8000 9000 'COMMODORE <RETURN>
80CF
```

Here the first two addresses specify the range to look through, and the last number (or three numbers, or string) specify what it is that we're looking for. If it's found, the starting location will be displayed. More than one, if it's found more than once.

Perform ['S']

The machine-code equivalent of the Basic SAVE command, which allows us to transfer a block of memory to disk or tape. This block of memory could be anything: the screen, for instance - it all gets saved. The command takes the form:

```
S "0:NAME",08,3800,3FFF <RETURN>
```

In this example, we're saving onto drive 0 a block of memory which will be called NAME. We're using device 8, and the block of memory starts at location $3800 and ends at $3FFF. To save to tape:

```
S "NAME",01,3800,3FFF <RETURN>
```

You must use 01, or 08, not just 1 or 8, otherwise the command won't work.

Perform ['L']

The machine-code equivalent of the Basic LOAD command, which allows us to restore to memory a previously saved portion of code. The command takes the form:

```
L "0:NAME",08 <RETURN>
```

or:

```
L "NAME",01 <RETURN>
```

for disk and tape respectively.

Perform ['V']

The machine-code equivalent of the Basic VERIFY command, which allows us to check that a block of saved memory has been saved correctly. The command takes the form:

```
V "0:NAME",08 <RETURN>
```

or:

```
V "NAME",01 <RETURN>
```

for disk and tape respectively.

Perform ['F']

No Basic equivalent for this one. It fills a block of memory with a specified character, and takes the form:

```
F 3800 38FF A0 <RETURN>
```

which fills memory from $3800 to $38FF with the character $A0 (a shifted space, in this example).

Try:

```
F 0C00 0FFF A0
```

and watch what happens!

Perform ['X']

This allows us to leave the monitor and return to Basic, and takes the form:

```
X <RETURN>
```

Perform ['A']

This command and the next one are the two most interesting commands available through the monitor, since they allow us to program the thing. Without these two, programming in machine code would be considerably more difficult than it already is.

To give us some space to put a couple of short machine-code routines, we must, before entering the monitor, type in the following two Basic commands:

```
POKE 52,55:POKE 56,55 <RETURN>
```

This fools the C16 into thinking that the top of its Basic memory is now location 14335, so we can use the 2K starting at location 14336 (or $3800) for machine code-programming. Now, enter the monitor by typing MONITOR, as usual. Then type:

```
A 3800 LDX #$41
```

the monitor takes this and replies by converting the machine code mnemonics into the following line:

```
A 3800 A2 41 LDX #$41
A 3802
```

and sits and waits for you to type the next piece of code in. So, type STX $0C00 <RETURN>, RTS <RETURN>, and you should have something like this:

```
A 3800  A2  41     LDX #$41
A 3802  8E  00 0C  STX $0C00
A 3805  60  RTS
A 3806
```

Now, by typing <RETURN> once more, we come out of 'Assemble' mode, and the C16 is sitting waiting for us to do something else. So, type:

```
G 3800 <RETURN>
```

and lo and behold, a heart is displayed in the top left-hand corner of the screen, followed by the almost inevitable SYNTAX ERROR generated by the 'G' command. Your first machine-code program! Next, 'Revenge of the Mutant Programmers'.

Perform ['D']

This allows us to look at any machine code that is in the machine, and takes the form:

```
D 3800 <RETURN>
```

If you'd typed in the little bit shown above, you'd get:

```
D 3800
. 3800  A2  41          LDX #$41
. 3802  8E  00   0C     STX $0C00
. 3804  60              RTS
. 3805  00              BRK
. 3806  00              BRK
```

and a lot more BRKs as well.

To display a specific range, type:

```
D 3800 3804 <RETURN>
```

If you want to see how Commodore's Basic 3.5 works, you can use this disassemble command. For instance, the Basic keyword PRINT is handled by a routine starting at $9000. So, typing D $9000 <RETURN> will show you how Commodore do it. A good way of getting to grips with machine code is to see how the experts do it, but don't worry if you've been totally lost by all this. We'll be going into more detail in Chapter 12.

Conclusion

The monitor is a very valuable asset on the C16, and why Commodore chose to leave one out of the Commodore 64 and Vic 20 is a strange decision known only to them. The early PET machines had one (well, all right, the very first one didn't), so why the others didn't is a bit of a mystery.

Still, it's there in the C16 and is very useful. Learning machine code with the aid of a reasonably good monitor, which this one is, is not too difficult, and we'll get started on that later. But for now, back to Basics, and a look at colour on the C16.

6

Colour

Despite the wealth of colours and colourful commands available to us on the C16, this is not going to be the longest of chapters. The reason for this is that most of the colour commands are dealt with in Chapter 7, so here we're just going to get a few things out of the way. In other words, prepare the ground for the real work to come.

Looking at colours

One of the first things that you'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
```

```
42 B$(18)=B$(18)+"[5CL.SP,CBMC,OFF,CBMV,CBMD,RVS,S
P,CD,5CL,SP,OFF,CBMK,SP]"
43 B$(18)=B$(18)+"[CBMD,RVS,CBMF,CD,5CL,SP,OFF,CBM
K,SP,RVS,2CBMK,CD,5CL]"
44 B$(18)=B$(18)+"[SP,CBMV,CBMK,OFF,CBMD,RVS,SP,CD
,5CL,2SP,2CBMK,SP,CD,5CL,SP]"
45 B$(18)=B$(18)+"[CBMC,CBMV,OFF,CBMI,RVS,SP,5CL,7
CU,OFF,5SP,CD,5CL,5SP]"
46 B$(18)=B$(18)+"[CD,5CL,5SP,CD,5CL,5SP,CD,5CL,5S
P,CD,5CL,5SP,CD,5CL,5SP]"
47 B$(18)=B$(18)+"[CD,5CL,5SP,CU]"
50 DIM PX(17),PY(17),R(17),CM$(7),A(2),B(2)
60 FORI=0TO17
70 : READ PX(I),PY(I)
75 : R(I)=I
80 : NEXTI
105 DIM M$(15)
110 FORI=0TO15
115 : READ M$(I)
120 : NEXTI
121 FORI=0TO7
122 : READCM$(I)
123 : NEXTI
130 GOSUB2000
146 IQ=.9
150 RR=3:B(0)=10:B(1)=15:B(2)=18
155 Q$="DO YOU NEED INSTRUCTIONS?":GOSUB800
160 IFA$="N"GOTO200
165 Q$="WE ARE THE EXECUTIONERS.\ PICK ONE OF US (
A B OR C)\ TO DESTROY AS MAN
166 Q$=Q$+"Y ANDROIDS\ FROM EACH ROW AS YOU WISH.\
 THEN IT IS OUR TURN TO PLAY.
167 Q$=Q$+"\ THE ONE WHO GETS THE LAST DROID WINS.
":GOSUB1500:FORI=1TO3000:NEXT
200 PRINT"[CLR]":GOSUB2000:FOR N=3TO17
205 : GOSUB1000
210 : R(N)=N
215 : NEXTN
220 RR=18:A(0)=7:A(1)=5:A(2)=3
225 TR=0:Q$="DO YOU WANT TO PLAY FIRST?":GOSUB800
228 M=0
230 IFA$="N"GOTO245
235 IFA$<>"Y"GOTO225
240 M=1-M
245 IFRR=3GOTO500
250 IFM=0GOTO400
255 TR=0:Q$="IT IS YOUR TURN.\ WHICH ROW?":GOSUB80
0
256 Z=1
260 P=ASC(A$)-65
265 IFP<0ORP>2THENGOSUB600:GOTO255
```

has a ball on the right-hand side of the key and a vertical line with a horizontal line half-way down on the left-hand side of the key. As you probably know, the graphics on the right of the key are accessed using the shift key, and those on the left are reached using the CBM logo key.

This is quite handy. If we go into lower/upper case mode using PRINT CHR$(14), we lose the characters on the right of the keys, but can still use the ones on the left. A listing given in a moment, for the program Android Nim, makes use of this, and the program called Response in Chapter 13 manages to build up a decent set of graphical displays using only the characters provided for us.

So, to see what those characters are capable of doing, let's tke a look at the Android Nim listing.

Android Nim

This may be only a game, but the use of the existing graphical characters is very interesting. By the way, if you think that I'm boasting about this program, I'm not. It was written by Don Denis, many years ago, for one of the existing Commodore PET computers. I've just updated it for the C16.

```
0 COLOR 0,8,7:COLOR 4,10,4:PRINTCHR$(14)
1 PRINT"[CLR]"TAB(10)"[2CD,BLK]***[RVS]ANDROID NIM
[OFF]***"
2 PRINTTAB(18)"[CD]BY":PRINTTAB(14)"[CD]DON DENIS[
3CD]"
3 PRINTTAB(11)"TORONTO, CANADA"
4 PRINTTAB(10)"(FOR ORIGINAL PET)":REM   153 UNDER
HILL DR
5 REM   DON MILLS, CANADA
6 REM   M3A 2K6
7 REM   (416)445-3927
8 PRINTTAB(11)"[3CD]ADAPTED FOR C16
9 PRINTTAB(11)"[CD]BY PETE GERRARD"
10 SF=64:VOL8
31 CL$="[HOME,39SP,HOME]"
34 LN=205:CN=202:KB=239
35 DEF FNE(X)=(A(P)ORE)AND(NOT(A(P)ANDE)):IQ=.7
36 DIM B$(18)
38 : FORI=0TO17
39 : READB$(I)
40 : NEXTI
41 B$(18)="[CL,RVS,SP,OFF,2CBMK,RVS,2SP,CD,5CL,SP,
CBMC,OFF,CBMD,SP,RVS,SP,CD]
```

7
Graphics

The C16 is an interesting machine when it comes to using graphics. With just a little over 12K to play with, certain graphics modes will be desirable in one circumstance, others in another. We can use the existing graphics set, we can define our own graphical characters, or we can use the excellent Basic commands provided with the machine. There are advantages and disadvantages with any of these modes.

Using the existing character set is suitable for many purposes. It doesn't take up any extra memory, there are over 80 graphics characters to play with, and they can be displayed in many different colours. They can also be made to flash on and off.

However, there will always come a time when a certain character is not available from the keyboard. Then we have to design our own characters: this can take a couple of K or so of memory, and, to begin with, a lengthy program.

For detailed graphical work we have to go into high resolution mode. This, alas, eats up 10K of our precious memory, leaving just 2K to play with: not a lot.

So whichever you decide to use must depend on the circumstances involved. We'll be looking at user-defined characters later on in this chapter, followed by a look at high resolution work, but for now we'll stick to the existing character set.

Using the graphics characters

As you can see from the keyboard of the C16, most of the keys have got two special graphics characters on them. The 'Q' key, for instance, To POKE something (a letter A) into the top left-hand corner of the screen, we must:

POKE 3072,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 simple formula:

POKE 3072 + X + 40*Y,something

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 where we want to put something.

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 2048 + X + 40*Y,colour

We use the same colour numbers as were given for the background and border colours, with one subtracted. This is a hang-over from the days of the Vic-20 and Commodore 64, when the colours were laid out slightly differently. Thus, to make something appear in black, we must use the number zero for colour, rather than the more familiar one.

So, to put a letter A 21 columns across and 12 rows down we must:

POKE 3072 + 21 + 40*10,65 or POKE 3493,65.

To change it to a red letter A, we must then POKE 2048 + 21 + 40*10,2 or POKE 2469,2.

You may find it easier to remember the colour locations as being (2048 - 3072), or 1024 locations behind. So, our earlier POKEs now become:

POKE 3493,65 : POKE 3493 - 1024,2.

However, there is a further complication. Since the C16 allows us to use luminance values for the border and background colours, you may well find that POKEing the colour memory map will not produce any discernible difference for some of the colours. In a way, the C16 is behaving like a Commodore 64, and when it POKEs into colour memory it doesn't take any notice of whatever luminance value may have been set. The only solution, I'm afraid, is trial and error.

7 BLUE 15 DARK BLUE
8 YELLOW 16 LIGHT GREEN

The following short program allows you to scan through all the border/background combinations, using the default luminance value.

```
10 FORI = 0 TO 15
20 FORJ = 0 TO 15
30 COLOR 0,J
40 COLOR 4,I
50 FORK=1TO1000:NEXTK
60 NEXTJ
70 NEXTI
```

The two outer loops change border (I) and background (J), while the inner (K) loop is just a display to let you see what's happening. Without that it all looks a little sickly!

If you want to alter the luminance as well, make the following changes:

```
25 FORK = 0 TO 7
30 COLOR 0,J,K
40 COLOR 4,I,K
50 FORL=1TO1000:NEXTL
55 NEXTK
```

Screen and colour memory maps

As we've already seen, the first screen memory location is number 3072, and since the screen is 40 columns across by 25 rows down, this means that we have 1000 locations on the screen (40 times 25). Thus the last screen location is number 4071, 1000 on from 3072.

The colour screen memory map lives totally independently of the ordinary screen memory map, and starts at memory location 2048. It finishes on 3047, 1000 locations further on.

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.

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 got by using the CHR$ command.

However, this isn't much good if we can't change anything else, like the background colour and the border colour.

COLOR

The COLOR command enables us to do this, as well as allowing us to change the character colour. A useful command.

COLOR MO,NO,LU

is the syntax to be used, where MO determines what mode of colour is going to be altered. MO can take the following values:

0 : Background colour
1 : Character colour
2 : Used in multi-colour mode
3 : Used in multi-colour mode
4 : Border colour

We'll come to multi-colour mode in Chapter 7, but for now we'll just concentrate on modes 0, 1 and 4. LO determines the luminance, or brightness, of the colour being used, and can range from 0 to 7. Finally, NO determines the actual colour itself, and the numbers to use for this are :

1 BLACK
2 WHITE
3 RED
4 CYAN
5 PURPLE
6 GREEN
9 ORANGE
10 BROWN
11 YELLOW/GREEN
12 PINK
13 BLUE/GREEN
14 LIGHT BLUE

```
270 IFA(P)=0THENGOSUB650:GOTO255
275 TR=P:Q$="HOW MANY ANDROIDS?":GOSUB800
280 Z=ASC(A$)-48
285 IFZ<1ORZ>9THENGOSUB600:GOTO255
288 POKELN,PY(P):POKECN,PX(P):PRINT"[2CU,CR]"Z
290 IFZ>A(P)THENGOSUB650:POKELN,PY(P):POKECN,PX(P)
:PRINT"[2CU,2CR,SP]":GOTO275
300 SL=25:GOSUB700
305 POKELN,PY(P):POKECN,PX(P):PRINT"[2CU,2CR,SP]"
310 GOTO240
400 E=0:F=0
405 FORP=0TO2
410 : E=FNE(0):IFA(P)>FTHENF=A(P):I1=P
415 : NEXTP
420 FORP=0TO2
425 : R=FNE(0):IFR<=A(P)GOTO470
430 : NEXTP:STOP
470 IFR=A(P)ORIQ>RND(1)THENP=I1:R=A(P)-INT(RND(1)*
(A(P)-1)+1)
475 TR=P:Z=A(P)-R:Q$="WE CHOOSE"+STR$(Z)+" ANDROID
 FROM ROW "+CHR$(P+65)+".\"
476 GOSUB1500
478 SL=5:GOSUB700
495 GOTO240
500 :PRINT"[CLR]BYE ...":END
505 Q$="YOU"+Q$
510 IFM=0THENQ$=Q$+" WE WILL PLAY BETTER NEXT TIME
.\":IQ=IQ*IQ*IQ
515 TR=0:GOSUB1500
520 Q$="WOULD YOU LIKE ANOTHER GAME?":GOSUB800
525 IFA$<>"N"GOTO200
530 Q$="THANK YOU FOR PLAYING.\\":GOSUB1500:PRINT"
[CLR]BYE ...":END
600 TR=0:R1=0:R2=0:R3=0:SL=17
605 M1$=M$(9):M2$=M$(10):M3$=M$(11)
610 GOSUB900
615 Q$="YOUR ANSWER DOES NOT MAKE SENSE.\"
616 IFZ=0THENQ$="CAN'T YOU MAKE UP YOUR MIND?\"
617 GOSUB1500
620 RETURN
650 R1=P:R2=P:R3=P:SL=25
655 M1$=M$(7):M2$=M$(8):M3$=M$(8)
660 GOSUB900
665 TR=P:Q$="SORRY, ONLY"+STR$(A(P))+" ANDROIDS LE
FT.\"
670 IFA(P)=0THENQ$="I CAN'T DO IT. I HAVE NONE LEF
T.\"
675 GOSUB1500
680 RETURN
700 R1=P:R2=P:R3=P
705 M1$=M$(6):M2$=M$(8):M3$=M$(8)
```

```
710 GOSUB900
712 II=B(P)-A(P)
715 FORI=IITOII+Z-1
720 : POKELN,PY(I):POKECN,PX(I):PRINT"[CU,CR]"B$(6
)
725 : NEXTI
726 REM
727 FORJJ=1000TO10STEP-20:SOUND1,JJ,2:NEXTJJ
730 FORI=1TOZ
735 : GOSUB950
740 : NEXTI
788 RETURN
800 POKEKB,0:QU$=Q$:GOSUB1500
805 T=TI+800
810 M1$=M$(RND(1)*16)
815 M2$=M$(RND(1)*16)
820 M3$=M$(RND(1)*16)
825 R1=R(RND(1)*RR)
830 R2=R(RND(1)*RR):IFR2=R1GOTO830
835 R3=R(RND(1)*RR):IFR3=R2ORR3=R1GOTO835
840 SL=INT(25*RND(1)+1)
845 GOSUB900
850 GETA$:IFA$<>""THENPRINTCL$:RETURN
855 IFTI>TTHEN Q$=CM$(RND(1)*8)+"\ "+QU$:GOSUB1500
:GOTO805
860 GOTO810
900 FORC=SL TO1STEP-1
910 : POKELN,PY(R1):POKECN,PX(R1):PRINT"[CU,CR]"B$
(ASC(RIGHT$(M1$,C))-SF)
920 : POKELN,PY(R2):POKECN,PX(R2):PRINT"[CU,CR]"B$
(ASC(RIGHT$(M2$,C))-SF)
930 : POKELN,PY(R3):POKECN,PX(R3):PRINT"[CU,CR]"B$
(ASC(RIGHT$(M3$,C))-SF)
940 : NEXTC
945 RETURN
950 POKELN,PY(R1):POKECN,PX(R1):PRINT"[CU,CD,4CR]"
;
954 FORJJ=100TO800STEP20:SOUND1,JJ,1:NEXTJJ
955 SP=PX(R1):EP=PX(B(P)-A(P))-5
959 SP=PX(R1):EP=PX(B(P)-A(P))-5
960 FORJ=SPTOEPSTEP2:PRINT"  -=*[3CL]";:NEXTJ
965 IFINT((EP-SP)/2)*2=EP-SPTHENPRINT"[CL]";
970 PRINT"[CU,CR]"B$(18)
974 RR=RR-1:A(P)=A(P)-1
976 A=3
977 ONP+1GOTO990,985,980
980 A=A+A(1)
985 A=A+A(0)
990 FORJ=ATO16
991 : R(J)=R(J+1)
992 : NEXTJ
```

```
998 RETURN
1000 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(1+
7*RND(1));
1010 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(0)
;
1020 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(9+
5*RND(1));
1030 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(14
+4*RND(1));
1040 RETURN
1500 PRINTCL$
1505 II=0:GOSUB1600
1510 FORI=1TOLEN(Q$)
1515 : CH$=MID$(Q$,I,1)
1517 N=N+1
1520 :  IFCH$=" "THENGOSUB1600
1525 :  IFCH$="\"THENII=I:FORJ=1TO600:NEXTJ:PRINTCL
$:GOTO1550
1530 :  POKELN,1:POKECN,I-II:PRINT"[CU,CL,RVS]"CH$"
[OFF]
1550 :  NEXTI
1560 RETURN
1600 POKELN,PY(TR):POKECN,PX(TR):PRINT"[CU,CR]"B$(
1);
1605 SOUND1,600-TR*100,2
1610 PRINT"[3CL,CBMK,CL]";:GOSUB1700
1615 PRINT"[RVS,CBMC,CL]";:GOSUB1700
1620 PRINT"[SP,CL]";:GOSUB1700
1625 PRINT"[CBMC]":GOSUB1700
1630 N=0
1650 RETURN
1700 FORJJ=1TO3*RND(1)
1702 SOUND 1,600-(TR*100+RND(1)*100),2
1706 NEXTJJ
1710 RETURN
2000 FOR N=0TO2
2010 :  GOSUB1000
2020 :  PRINT"[RVS,2CU,3CL]*[CD,CL]"CHR$(N+65)
2030 :  NEXTN:RETURN
5030 DATA"[3CD,CR,RVS,SP,CD,CL,SP,CU,CBMD,OFF,CD,C
L,CBMK,CD,2CL,2CBMK,CD
5031 DATA"[2CL,2CBMK,CD,3CL,CBMC,CBMV,RVS,CBMI,OFF
,3CL,7CU]"
5032 REM CARRIES ON FROM 5030 TO 5031
5035 DATA"[SP,2CBMK,CD,3CL,RVS,CBMV,2CBMF,OFF,CBMF
,CD,4CL,CBMC,RVS,CBMC
5036 DATA"[CBMD,OFF,SP]":REM CARRIES ON FROM 5035
TO 5036
5040 DATA"[SP,2CBMD,CD,3CL,RVS,CBMK,CBMB,CBMV,OFF,
SP,CD,4CL,RVS,CBMI,SP
5041 DATA"[CBMD,OFF,CBMV]":REM CARRIES ON FROM 504
```

```
0 TO 5041
5045 DATA"[SP,2CBMD,CD,3CL,RVS,CBMV,2SP,OFF,CBMF,C
D,4CL,RVS,CBMK,CBMB,CBMV
5046 DATA"[OFF,SP]":REM CARRIES ON FROM 5045 TO 50
46
5050 DATA"[SP,RVS,2CBMK,CD,3CL,CBMK,2CBMD,OFF,SP,C
D,4CL,OFF,CBMC,RVS,SP
5051 DATA"[CBMB,OFF,SP]":REM CARRIES ON FROM 5050
TO 5051
5055 DATA"[2SP,CBMK,CD,3CL,RVS,CBMK,SP,OFF,CBMV,SP
,CD,4CL,CBMC,RVS,SP,CBMD
5056 DATA"[OFF,SP]":REM CARRIES ON FROM 5055 TO 50
56
5060 DATA"[RVS,2CBMK,OFF,SP,CD,3CL,CBMC,RVS,CBMD,S
P,OFF,SP,CD,4CL,CBMC,RVS,CBMV
5061 DATA"[CBMD,OFF,SP]":REM CARRIES ON FROM 5060
TO 5061
5065 DATA"[SP,CBMK,2SP,CD,4CL,SP,RVS,CBMF,SP,OFF,S
P,CD,4CL,CBMC,RVS,SP,CBMD,OFF,SP
5070 DATA"[CD,CR,RVS,2SP,OFF,CR,CD]"
5075 DATA"[3CD,RVS,CBMB,CD,2CL,OFF,CBMC,CBMF,CD,2C
L,SP,CBMC]"
5080 DATA"[3CD,RVS,CBMB,CD,2CL,OFF,SP,CBMK,CD,2CL,
SP,CBMC]"
5085 DATA"[3CD,RVS,CBMB,CD,2CL,OFF,SP,CBMK,CD,2CL,
SP,CBMV]"
5090 DATA"[3CD,RVS,CBMB,CD,2CL,OFF,SP,CBMK,CD,2CL,
CBMC,SP]"
5100 DATA"[3CD,RVS,CBMK,CD,CL,CBMK,2CL,OFF,SP,CD,C
L,SP,CBMV]"
5105 DATA"[3CD,3CR,CBMF,CD,CL,RVS,CBMD,CD,2CL,CBMD
,OFF,SP]"
5110 DATA"[3CD,3CR,CBMF,CD,CBMK,CD,2CL,RVS,CBMD,OF
F,SP]"
5115 DATA"[3CD,3CR,CBMF,CD,CL,CBMK,CD,2CL,CBMK,CBM
V]"
5120 DATA"[3CD.3CR.CBMF,CD,CL,CBMK,CD,2CL,CBMK,CBM
C]"
5230 DATA0,2,3,10,0,18,5,2,10,2,14,2,19,2,24,2,29,
2,34,2,13,10,18,10,23,10,28
5240 DATA10,33,10,21,18,26,18,31,18
5330 DATA AHDEEDABACABACABACAADHDAB
5335 DATA AHDAFADAFADEDHDAHAFFHFFAA
5340 DATA AHANCAAABKPLQAKPINHACCAFG
5345 DATA JOKPLQKPJOKPLQKPJOINFJHFM
5350 DATA FGNKLJLJLJLJLFHFFADEQNJNID
5355 DATA AHAFADAFAHADFDFDFDHDAFGKN
5360 DATA AHBBBAHADEEEDABACABACADEI
5365 DATA ABBBAHADEEEDAFADAFADAFAHA
5370 DATA OJJJPPPPQPQPKKKKKKKKKKKK
5137 DATA AAAAAAAAHABBBAAACCAHAAAHA
```

```
5380 DATA AAAAAAAAIIIIIJKLLLIIIIIII
5385 DATA AAAAAAAANIIIIOPQQQNNNNNNN
5390 DATA AHABADACAFABADACFBDCFBDHD
5395 DATA ADEDADEDADEDHAFGFAFGFAFGF
5400 DATA BDBDBDBDBACFMNCACACACAHCA
5405 DATA AFGGNQPQPQFAHDEPQNDAFGLIG
5510 DATA"COME ON.","WE HAVEN'T GOT ALL DAY!"
5520 DATA"WE HAVE BETTER THINGS TO DO."
5530 DATA"JUST ANSWER THE QUESTION.","IT ISN'T THA
T DIFFICULT!"
5540 DATA"THERE IS A LIMIT TO OUR PATIENCE!"
5545 DATA"JUST GET ON WITH IT.","STOP PLAYING FOR
TIME."
```

You may be familiar with the game of Nim as played with matches, but if not, a quick explanation of the rules. Three rows of matches (or androids!) are laid out, one row of seven, one row of five, and one row of three. You and your opponent, in this case the computer, take it in turns to take as many matches as you like from one row. Whoever ultimately takes the last match wins the game.

To begin with the C16 doesn't play a very good game, and lulls you into a false sense of security. However, after a few games its standard of play begins to increase significantly and you have a markedly different opponent on your hands.

Typing the listing in

There are a number of problems with this listing, not least of which is that the printer used to list the program in the first place couldn't cope with Commodore control characters, and so special symbols had to be used instead.

Everything up to line 40 should be fairly obvious, but from there the problems begin. Lines 41 to 46 merely define the string B$(18), but take a long time doing so. To take the first line (line 41), we have:

41 B$(18)="

So far so good, just type that in as shown. Then, hit the cursor left key, CTRL and 9 to turn reverse mode on, then a space, then turn reverse mode off, then press the CBM logo key in conjunction with the K key twice, and so on.

Whatever you do, don't type in the commas! This warning has been repeated throughout the book, for the benefit of anyone who might just dive in and attempt to type in a listing.

Lines 50 to 160 are again fairly straightforward, but after that ...! The strange symbol in lines 165 to 167 (and indeed lines 255, 475, 510, 530, 615, 616, 665, 670, 855 and 1525) is meant to be the pound sign, next to the equals key. I must change my printer!

Lines 200 to 2030 are easy enough to type in, but again after that we run into a few problems.

The data statements from line 5030 to line 5120 are the ones in question. If we look in detail at lines 5030 to 5032 you'll see that we have a large number of Commodore control characters, followed by a REM statement to the effect that 'carries on from 5030 to 5031'. This means that when you're typing in line 5030 and reach the end of the current Commodore control characters, DON'T PRESS RETURN AND ATTEMPT TO ENTER LINE 5031 AS A SEPARATE DATA STATEMENT. Just keep going until you reach the end of line 5031, or in later cases the REM statement at the end of a line.

And the best of British luck!

When you get the program up and running, watch out for the two androids in the top row who seem to be more than a little bit friendly. I can't help it if the C16's screen wrap-around routine doesn't work properly, and two androids had to be squeezed up next to each other! Still, I'm sure they're just good friends.

Having seen what can be done with the existing character set, then, what about user-defined graphics?

Existing characters

Every character that the C16 is capable of displaying is built up on an 8 by 8 matrix, just like most other home computers. For example, the letter 'A' looks like this:

```
•••★★•••
••★★★★••
•★★••★★•
•★★★★★★•
•★★••★★•
•★★••★★•
•★★••★★•
••••••••
```

There are some 256 characters in all stored away somewhere in the C16's memory (we'll find out where later). One quick and easy way to see what they all are is to type in and run the following program:

```
10 COLOR 0,8,7:COLOR 4,10,4:PRINT"[CLR,BLK]"
20 FORI=0TO255
30 POKE3072+I,I
40 NEXT
50 FORI=1TO500:NEXTI
60 PRINTCHR$(14)
70 FORI=1TO500:NEXTI
80 PRINTCHR$(142):GOTO50
```

Line 10 gives us a brown border and a yellow background, while clearing the screen and setting the printing colour to black. Then all 256 characters are POKEd onto the screen (screen memory begins at location 3072), before a short delay in line 50 allows you to look at them. The command in line 60 switches the display to lower-case mode, followed by another delay and back to upper-case mode again. The program will loop round forever, so press the Run/Stop key to get out of it.

You can, if you want, try printing the CHR$(values of everything, but as lines 60 and 80 in the short program above tell us, some of them have rather interesting effects and do not just display characters on the screen.

Some of these characters (obviously the graphical ones will figure most prominently) can be used to build up fascinating displays, as was seen with the Android Nim program, and with the impressive graphical high resolution commands available on the C16 you might think that there's more than enough in the machine to keep everyone happy. However, when using high resolution the C16's memory is restricted to a paltry couple of kilobytes, hardly enough for any serious work. Similarly, when designing a graphical display, you will always find that a particular character you want is not available directly from the keyboard. So, what to do?

User-defined graphics

The obvious thing is to design your own characters. This has the happy results of (a) not using up too much memory, and (b) allowing you to create virtually any type of display on the screen.

In order to do this, however, we need to know a little bit more about the inner workings of the C16. We'll also come across one or two design faults as well!

The first thing we need to find out is what tells the C16 that when you press the 'A' key a letter 'A' will indeed appear on the screen. Somewhere in the C16's memory must be stored the data for each and every character that it is capable of displaying, and a little hunting around reveals that this 'somewhere' is in memory locations 53248 to 55295. 2048 bytes of memory, eight for each of the 256 characters available.

Unfortunately, this information is stored in ROM, not RAM, which means that we can't alter it directly. Okay then, let's copy it all into RAM where we can alter it. The following program would appear to do the trick.

```
FORI=0TO2048:POKE14336+I,PEEK(53248+I):NEXT
```

Being Basic, this takes quite some time to run. When we've got the character data in RAM we need to tell the C16 that it has now got to find its information on each character at somewhere other than its usual position. So we alter the two memory locations that inform the C16 that (a) its character data is stored in RAM, not ROM, and (b) precisely where it is now stored. These two locations are respectively 65298 and 65299, so we must:

```
POKE 65298,192:POKE 65299,56
```

The first command simply switches a bit of location 65298 off, and the second tells the C16 to look for its character data beginning at the memory location (56*256), or 14336, which is where we put it with the FOR ... NEXT loop described above.

Type in those two POKEs, and ... disaster! Some very strange characters appear on the screen, and, to make matters worse, whatever you type comes up as garbage. So, very carefully (since you can't see what you're typing), type out the following, and press RETURN.

```
POKE 65298,196:POKE 65299,208
```

If you've done that correctly, we've got our existing character set back again and the C16 thinks everything is back to normal. So, why didn't all that work?

The answer lies in the PEEK command used to look at locations 53248 and upwards. For some bizarre reason best known to Commodore, PEEK will not recognise any location behind 16384, so what we typed in earlier simply won't work properly. Everything may look all right, but as the strange display found earlier will testify, it isn't.

There are a number of ways to get around this. One is to enter the monitor and transfer a block of memory with the command:

```
T D000 D7FF 3800
```

which performs precisely the same action as the FOR ... NEXT loop earlier, but this time it works properly. However, that is not a very elegant solution, since you can't ask everyone who wants to use one of your programs with user-defined characters in it to enter the monitor and fiddle about with possibly unfamiliar commands.

An easier way is found in the Character Builder listing below.

Character Builder

How characters are built up

Although we'll look at the listing in a bit more detail in a moment, what it basically does is to move the top of Basic memory and put the information for the first 64 characters into the space created. This information is read in from the mass of data statements at the end of the listing (lines 30000 onwards). The problem now is, how did we get that information in the first place?

Remember our old friend the letter 'A':

```
ABCDEFGH
• • • * * • • •
• • * * * * • •
• * * • • * * •
• * * * * * * •
• * * • • * * •
• * * • • * * •
• * * • • * * •
• • • • • • • •
```

Each row of dots and asterisks represents one byte of memory, with in this case the letters A to H representing each of the eight bits in that byte. By looking at the diagram, we can see that bits A,B,C,F,G and H are off in the first row of data, and bits D and E are turned on. Incidentally, reverse field characters are created by turning on all the bits that were off, and vice versa.

As you probably know, the maximum number that can be stored in any memory location is 255, which gives us a possible 256 (0 to 255) values that can be put there. In order to make every one of those 256 values unique, a numbering system is used to denote whether any bit in that byte is on or off. If bit A of a byte were on, and all the rest were off, the value 128 would be stored in whatever memory location we're talking about. If bit B were on and all the rest were off then 64 would be stored there, and so on until we get to the last bit, bit H, which has a value of 1 if it's turned on and the rest are turned off.

Going back to the diagram, we see that our first row of data has bits D and E turned on, so our data value becomes 16 plus 8, or 24. If you look at program line 30001 you'll see that that is the first value stored there. Follow it through and see how the rest of the values are worked out.

```
5 POKE52,55:POKE56,55
10 REM UDG PROGRAM
20 COLOR 0,8,7:COLOR 4,10,4:PRINT"[CLR,BLK]CHARACT
ER BUILDER ... PLEASE WAIT
25 FORI=0TO511:READA:POKE14336+I,A:NEXT
30 POKE65298,192:POKE65299,56
40 PRINT"[CD]CHARACTER SET NOW READ IN.
45 PRINT"[CD]PRESS ANY KEY FOR MENU."
50 GETKEYA$
55 PRINT"[CLR]CHARACTER BUILDER"
59 XC=3152:CC=2128:XX=2
```

```
60 FORI=1TO8:FORJ=0TO7:POKEXC+I*40+J,J+8*(I-1):POK
ECC+I*40+I,1:NEXTJ,I
61 X1=40:Y1=0
62 X=12:PRINT"[HOME,3CD]"TAB(X)"[RVS]E[OFF]DIT CHA
RACTER
64 PRINTTAB(X)"[CD,RVS]Q[OFF]UIT PROGRAM
65 PRINTTAB(X)"[CD]USE CURSOR KEYS TO":PRINTTAB(X)
"SELECT CHARACTER.
66 GETA$:IFA$="Q"THEN999
68 IFA$="E"THEN100
70 IFA$="[CU]"THENPOKECC+X1+Y1,0:X1=X1-40:IFX1<40T
HENX1=320
72 IFA$="[CD]"THENPOKECC+X1+Y1,0:X1=X1+40:IFX1>320
THENX1=40
74 IFA$="[CR]"THENPOKECC+X1+Y1,0:Y1=Y1+1:IFY1=8THE
NY1=0
76 IFA$="[CL]"THENPOKECC+X1+Y1,0:Y1=Y1-1:IFY1=-1TH
ENY1=7
78 XX=PEEK(XC+X1+Y1):POKEXC+X1+Y1,32:POKEXC+X1+Y1,
XX:GOTO66
100 PRINT"[CLR]CHARACTER BUILDER[2CD]"
102 A=(X1/40-1)*8+Y1:A1=14336+A*8:Z=A
104 FORI=0TO7:FORJ=7TO0STEP-1
106 IFPEEK(A1+I)AND2^JTHENPRINT"*";:ELSEPRINT".";
110 NEXTJ:PRINT:NEXTI
112 PRINT"[HOME,3CD]TAB(X)"[RVS]+[OFF] TO ADD DOT
114 PRINTTAB(X)"[RVS]-[OFF] TO ERASE DOT
116 PRINTTAB(X)"[RVS]M[OFF]IRROR CHARACTER
118 PRINTTAB(X)"[RVS]I[OFF]NVERT CHARACTER
120 PRINTTAB(X)"[RVS]R[OFF]OTATE CHARACTER
122 PRINTTAB(X)"[RVS]E[OFF]NTER CHAR. INTO MEMORY
124 PRINTTAB(X)"[RVS]B[OFF]ASIC DATA STATEMENT
126 PRINTTAB(X)"[RVS]Q[OFF]UIT TO MAIN MENU
127 PRINTTAB(X)"[CD]USE CURSOR KEYS TO":PRINTTAB(X
)"MOVE AROUND."
128 K$="+-MIREBQ[CU,CD,CL,CR]":X1=40:Y1=0
130 XX=PEEK(XC+X1+Y1):POKEXC+X1+Y1,32:POKEXC+X1+Y1
,XX
131 GETA$:IFA$=""THEN130
132 FORI=1TOLEN(K$):IFMID$(K$,I,1)=A$THEN135
134 NEXT:GOTO130
135 XX=PEEK(XC+X1+Y1):POKEXC+X1+Y1,32:POKEXC+X1+Y1
,XX
136 ONIGOTO200,300,400,500,600,700,800,900,1000,11
00,1200,1300
199 GOTO199
200 POKEXC+X1+Y1,42:GOTO1302
300 POKEXC+X1+Y1,46:GOTO1302
400 GOSUB2000
402 FORI=0TO7:FORJ=7TO0STEP-1
404 IFA%(I,J)=1THENPOKEXC+40+I*40+(7-J),42:ELSE PO
```

```
KEXC+40+I*40+(7-J),46
406 NEXTJ,I
408 GOTO128
500 GOSUB2000
502 FORI=0TO7:FORJ=0TO7
504 IFA%(I,J)=0THENA%(I,J)=1:ELSE A%(I,J)=0
506 NEXTJ,I
508 GOSUB3000:GOTO128
600 GOSUB2000
602 FORI=0TO7:FORJ=0TO7
604 IFPEEK(XC+40+I*40+J)=42THENA%(7-J,7-I)=1:ELSE
A%(7-J,7-I)=0
606 NEXTJ,I
608 FORI=0TO7:FORJ=7TO0STEP-1
610 IFA%(I,J)=1THENPOKEXC+I*40+40+(7-J),42:ELSE PO
KEXC+40+I*40+(7-J),46
612 NEXTJ,I:GOTO128
700 FORI=0TO7:FORJ=7TO0STEP-1
702 A=PEEK(XC+40+I*40+(7-J)):IFA=42THENB=B+2^J
704 NEXTJ
706 POKEA1+I,B:B=0:NEXTI
708 GOTO128
800 FORI=0TO7:FORJ=7TO0STEP-1
802 A=PEEK(XC+40+I*40+(7-J)):IFA=42THENB=B+2^J
804 NEXTJ
806 POKEA1+I,B:Z(I)=B:B=0:NEXTI
808 PRINT"[CLR,2CD]"30000+Z"DATA";:FORI=0TO6:Z$(I)
=RIGHT$("00"+MID$(STR$(Z(I)),2),3)
809 PRINTZ$(I);",";:NEXT:Z$(7)=RIGHT$("00"+MID$(ST
R$(Z(I)),2),3):PRINTZ$(7)
810 PRINT"GOTO55[HOME]";
812 POKE1319,13:POKE1320,13:POKE239,2:END
900 GOTO55
999 POKE65298,196:POKE65299,208:PRINT"[CLR]":END
1000 POKECC+X1+Y1,0:X1=X1-40:IFX1<40THENX1=320
1001 GOTO1302
1100 POKECC+X1+Y1,0:X1=X1+40:IFX1>320THENX1=40
1101 GOTO1302
1200 POKECC+X1+Y1,0:Y1=Y1-1:IFY1<0THENY1=7
1201 GOTO1302
1300 POKECC+X1+Y1,0:Y1=Y1+1:IFY1>7THENY1=0
1302 POKECC+X1+Y1,14:GOTO130
2000 FORI=0TO7:FORJ=0TO7
2002 IFPEEK(XC+40+I*40+J)=42THENA%(I,J)=1:ELSE A%(
I,J)=0
2004 NEXTJ,I
2006 RETURN
3000 FORI=0TO7:FORJ=0TO7
3002 IFA%(I,J)=1THENPOKEXC+40+I*40+J,42:ELSE POKEX
C+40+I*40+J,46
3004 NEXTJ,I:RETURN
```

```
30000 DATA60,102,110,110,96,98,60,0
30001 DATA024,060,102,126,102,102,102,000
30002 DATA124,102,102,124,102,102,124,0
30003 DATA60,102,96,96,96,102,60,0
30004 DATA120,108,102,102,102,108,120,0
30005 DATA126,96,96,120,96,96,126,0
30006 DATA126,96,96,120,96,96,96,0
30007 DATA60,102,96,110,102,102,60,0
30008 DATA102,102,102,126,102,102,102,0
30009 DATA60,24,24,24,24,24,60,0
30010 DATA30,12,12,12,12,108,56,0
30011 DATA102,108,120,112,120,108,102,0
30012 DATA96,96,96,96,96,96,126,0
30013 DATA99,119,127,107,99,99,99,0
30014 DATA102,118,126,126,110,102,102,0
30015 DATA60,102,102,102,102,102,60,0
30016 DATA124,102,102,124,96,96,96,0
30017 DATA60,102,102,102,102,60,14,0
30018 DATA124,102,102,124,120,108,102,0
30019 DATA60,102,96,60,6,102,60,0
30020 DATA126,24,24,24,24,24,24,0
30021 DATA102,102,102,102,102,102,60,0
30022 DATA102,102,102,102,102,60,24,0
30023 DATA99,99,99,107,127,119,99,0
30024 DATA102,102,60,24,60,102,102,0
30025 DATA102,102,102,60,24,24,24,0
30026 DATA126,6,12,24,48,96,126,0
30027 DATA60,48,48,48,48,48,60,0
30028 DATA12,18,48,124,48,98,252,0
30029 DATA60,12,12,12,12,12,60,0
30030 DATA0,24,60,126,24,24,24,24
30031 DATA0,16,48,127,127,48,16,0
30032 DATA0,0,0,0,0,0,0,0
30033 DATA24,24,24,24,0,0,24,0
30034 DATA102,102,102,0,0,0,0,0
30035 DATA102,102,255,102,255,102,102,0
30036 DATA24,62,96,60,6,124,24,0
30037 DATA98,102,12,24,48,102,70,0
30038 DATA60,102,60,56,103,102,63,0
30039 DATA6,12,24,0,0,0,0,0
30040 DATA12,24,48,48,48,24,12,0
30041 DATA48,24,12,12,12,24,48,0
30042 DATA0,102,60,255,60,102,0,0
30043 DATA0,24,24,126,24,24,0,0
30044 DATA0,0,0,0,0,24,24,48
30045 DATA0,0,0,126,0,0,0,0
30046 DATA0,0,0,0,0,24,24,0
30047 DATA0,3,6,12,24,48,96,0
30048 DATA60,102,110,118,102,102,60,0
30049 DATA24,24,56,24,24,24,126,0
30050 DATA60,102,6,12,48,96,126,0
```

```
30051 DATA60,102,6,28,6,102,60,0
30052 DATA6,14,30,102,127,6,6,0
30053 DATA126,96,124,6,6,102,60,0
30054 DATA60,102,96,124,102,102,60,0
30055 DATA126,102,12,24,24,24,24,0
30056 DATA60,102,102,60,102,102,60,0
30057 DATA60,102,102,62,6,102,60,0
30058 DATA0,0,24,0,0,24,0,0
30059 DATA0,0,24,0,0,24,24,48
30060 DATA14,24,48,96,48,24,14,0
30061 DATA0,0,126,0,126,0,0,0
30062 DATA112,24,12,6,12,24,112,0
30063 DATA60,102,6,12,24,0,24,0
```

The program as a whole allows you to design your own characters and place the data for those characters in data statements. If you decide to alter the letter 'A', then the new data will simply replace the old lot, and line 30001 will be re-written. Thus, when you've finished altering your characters, you can use the DELETE command to remove lines 40 to 3004, and leave the data and the lines that read that data in as a nucleus for whatever program you're working on.

The area up to line 50 we've already covered, but what about after that?

Line 59 sets up a couple of variables to decide where the 'cursor' is in terms of screen and colour memory. Don't worry about the XX variable yet. Line 60 then prints up the first 64 characters on the screen in a convenient 8 by 8 square, and line 61 tells us where on the square our 'cursor' is in terms of X and Y co-ordinates.

The initial program menu is then displayed, and the program sits back and waits for you to do something. Pressing 'Q' will exit you from the program, which would be a bit unwise to do since we haven't altered any characters yet. Pressing 'E' allows you to edit whatever character the 'cursor' is lying over, while pressing any of the four cursor movement keys moves the 'cursor' around until you select whatever character you want to edit. The character flashes periodically to remind you of where you are on the square.

Lines 70 to 76 control movement of the 'cursor', while line 78 controls the flashing of whatever character you happen to be lying over.

To take line 76 as an example, don't type in the symbols '[CL]'. These are just used because the printer used to list the program can't cope with the Commodore character set. You press the cursor left key instead, when it comes to typing the program in.

When you decide to edit a character, lines 100 to 127 print up the secondary program menu, as well as displaying a scaled-up picture of the character on an 8 by 8 grid. Line 102 tells the program which character you've selected to edit, and the variable A1 is used to point to the correct location in memory that stores the data for that character. The next three lines build up the large image of the character, and the little Chinaman's hat symbol in line 106 (after AND2) is meant to be the up-arrow key above the 0. Another printer quirk.

Lines 128 to 135 then sit and wait for you to press a key, as well as checking that the key you've pressed is a valid one from the menu selection, before going off to the 12 sub-sections of the program. We'll look at each of those in turn.

Moving the 'cursor' around

Again, our 'cursor' is moved around by using the ordinary cursor control keys, and lines 1000 to 1302 handle that. As usual, a check is made to see that you don't move the 'cursor' off the grid.

Quitting from this section is quite straightforward. The program goes to line 900, which just sends you back to line 55 again. We could have gone straight to line 55 I suppose, but for some long-forgotten reason the program happens to work this way round.

Amending the character is another straightforward process. Line 200 puts an asterisk wherever the 'cursor' happens to be, and line 300 puts a dot there instead. Thus we determine whether a bit is to be turned on (asterisk) or off (dot).

The three most interesting parts of this section of the program are those that (a) create a mirror image of the character, (b) invert it and (c) rotate it.

Character manipulation

To create a mirror image we go to line 400, which first of all buzzes off to the subroutine commencing at line 2000. This builds up an image of the character in the array A%(I,J) in terms of ones and zeroes, rather than the screen image of dots and asterisks. It's easier to manipulate. Integer arrays (using A% instead of just A) are used instead of ordinary ones because they operate faster and take up less memory: a valuable virtue on the C16.

Using our array data, lines 402 to 406 then determine whether an asterisk or a dot is to be displayed on the screen. Note the fact that we're counting down in the J loop, which is what helps to give us our mirror image.

To invert the character is fairly simple, remembering that a reverse image of a character just turns on all the bits that are off, and vice versa. Lines 500 to 508 do the trick for us, with a quick trip to the subroutine starting at line 3000 to display the new character on the screen.

However, rotating the character is not so easy, hence the routine starting at line 600 is a little longer than the previous two. After visiting the subroutine at line 2000 we move our character data round line by line, and then display it up on the screen again. By rotating the character three more times, we get back to where we started from, just to prove that it works properly.

And into memory

To enter the character in memory is simply a question of POKEing the relevant data into the correct part of memory. Lines 700 to 708 do this, and this part of the program is repeated in lines 800 to 806 when we actually create the data statement for the new character. Again, beware of the Chinaman's hat in lines 702 and 802. Press the up-arrow key above the '0' instead.

Getting the data statement to appear on screen and become a part of the program is a bit complicated, and lines 808 to 812 achieve the desired results. By carefully printing everything on the screen, and adding a GOTO55 for good measure, we can get what we want with line 812. Locations 1319 and 1320 are the first two locations in the C16's keyboard buffer, and location 239 tells the C16 how many characters are stored in the buffer. It's a way of getting the machine to type two carriage returns, instead of you having to do it.

Conclusion

To get the best results out of this program, it's best just to play around with it. Take any character, and see the effect of rotating, inverting, and creating a mirror image of it. Design your own characters for use in your own programs, and let the computer worry about bits and bytes, data statements and memory.

To use the data in your own programs, all you'll need are lines 5, 25 and 30, plus (of course!) the data itself. If you then, from within the program itself, delete lines 30000 onwards you'll have about 10,000 bytes left in the machine, enough for quite a convincing program. How do you delete lines from within a program? Study the techniques used to generate the data statements, and that should point you gently towards the answer.

Now that we've seen just about how far we can go using the existing character set, and our own character set, let's turn our attention to high resolution graphics, and take a look at the new commands which Basic version 3.5 has given us.

High resolution graphics

Unlike earlier Commodore machines, which had no Basic keywords (other than PEEK and POKE) designed to cope with high-resolution graphics, the C16 has a wealth of them. We'll take them in turn, but before doing so here is a small program to whet your appetite.

Quite simply, this little program allows you to draw on a high resolution screen, all 320 pixels by 200 pixels of it. Each dot on the screen can be altered with this program, and perhaps by the end of the chapter you'd like to enhance it a little.

```
1 FORI=1TO8:KEY I,CHR$(I+132):NEXT
10 COLOR 0,8,7:COLOR 4,10,4
20 GRAPHIC 1,1
30 DRAW 1,0,0:X=0:Y=0:C=1:CC=1:BC=8:BB=10:BL=7:LL=
4
40 GETA$:IFA$=""THEN40
42 IFA$="C"THENC=1-C
44 IFA$="1"THENCC=CC+1:IFCC>16THENCC=1:REM DRAWING
 COLOUR
45 IFA$="[F1]"THENBC=BC+1:IFBC>16THENBC=1:REM BACK
GROUND COLOUR
46 IFA$="[F2]"THENBB=BB+1:IFBB>16THENBB=1:REM BORD
ER
47 IFA$="[F4]"THENBL=BL+1:IFBL>7THENBL=0:REM BACKG
ROUND LUMINANCE
48 IFA$="[F5]"THENLL=LL+1:IFLL>7THENLL=0:REM BORDE
R LUMINANCE
50 IFA$="I"THENY=Y-1:IFY<0THENY=200
52 IFA$="M"THENY=Y+1:IFY>200THENY=0
54 IFA$="A"THENX=X-1:IFX<0THENX=320
56 IFA$="D"THENX=X+1:IFX>320THENX=0
```

```
57 IFPEEK(198)=52THEN999
58 COLOR 1,CC:DRAW C,X,Y:COLOR 0,BC,BL:COLOR 4,BB,
LL:GOTO40
999 GRAPHICO,1:GRAPHIC CLR:END
```

This program uses the keys I, M, A and D to move the drawing cursor up, down, left and right respectively. As the REM statements in lines 44 to 48 explain, it also uses a number of other keys to alter drawing colours, background colours, luminance, etc.

To labour a point, when you're typing in line 45 don't enter "[F1]", but just re-define the function keys as shown in line 1 and press the key labelled F1 once you've opened up the quotes.

New commands

We might lose 10K when it comes to working in high-resolution graphics, but we certainly get a lot of new commands to fill the remaining 2K. To illustrate just some of them, the following two programs produce interesting displays on the screen!

```
5 GRAPHIC 1,1
6 COLOR 1,7,3
7 COLOR 0,8,7
8 COLOR 4,10,5
10 FORI=0TO360STEP15
20 CIRCLE,160,100,50,42,90,180,I
25 NEXT
26 COLOR 1,1
30 PAINT 1,160,100
40 COLOR 1,3,3
50 FORI=0TO360STEP15
60 CIRCLE,160,100,80,67,90,180,I
70 NEXT
80 COLOR 1,6,4
90 PAINT 1,210,240
100 COLOR 1,1
110 BOX 1,65,10,250,185
120 COLOR 1,5,3
130 PAINT 1,68,15
140 COLOR 1,1
150 PAINT 1,10,10
160 COLOR 0,2,7
170 FORI=0TO25
```

```
175 X=INT(RND(.5)*39+1):Y=INT(RND(.5)*24+1)
180 CHAR 1,X,Y,"*",1
190 NEXT
```

And the next one!

```
4 PRINT CHR$(14)
10 GRAPHIC 2,1
20 COLOR 0,8,7:COLOR 4,10,4
24 COLOR 1,1
25 DRAW 1,0,159 TO 320,159
30 PRINT"[CD,BLK]WHAT NOW [FLON]?[FLOFF]"
40 PRINT"[CD,D.BLU]SAY TO ELROND "CHR$(34)"MINE'S A
PINT"CHR$(34)
45 COLOR 1,1
50 FORI=1TO50
60 CIRCLE 1,I,I,I,I
70 NEXT
80 DRAW 1,200,159 TO 200,100 TO 230,100 TO 230,159
85 COLOR 1,1:CIRCLE 1,220,130,3
90 COLOR 1,6,3:PAINT 1,215,120,0
95 COLOR 1,1:CIRCLE 1,200,93,2:CIRCLE 1,195,85,5:CIRCLE
1,195,60,15
100 A$="OH LOOK, A GIANT SPIDER'S WEB!"
110 FORI=1TOLEN(A$)
120 CHAR 1,24,7,MID$(A$,I,1)
122 FORJ=1TO75:NEXTJ
125 NEXTI
130 BOX 1,300,20,320,0,,1
140 SSHAPE A$,300,20,320,0
145 A=INT(RND(.5)*300):B=INT(RND(.5)*140)
150 GSHAPE A$,A,B
160 FORI=1TO100:NEXT
170 GOTO145
```

An interesting exercise in graphics, and the second program uses just about every graphic command available to us on the C16: an impressive repertoire.

The one thing that neither of these programs has attempted to do is to produce a display in multi-colour high-resolution graphics. True to the old adage that you don't get anything for nothing, using multi-colour allows us to display more colours on the screen, but at the expense of the maximum resolution available. Whereas before we could alter pixels on the screen on a 320 by 200 basis, we're now down to

160 by 200.

To illustrate the difference, try typing in and running the two following programs.

(1) In ordinary graphics mode.

```
10 GRAPHIC 1,1
20 COLOR 1,1
30 BOX 1,140,80,180,120,45,1
40 COLOR 1,2,7
45 CIRCLE 1,160,100,5
```

Not very impressive, is it? Let's go into multi-colour mode instead.

(2) In multi-colour mode.

```
10 GRAPHIC 3,1
20 COLOR 3,1
30 BOX 3,70,80,90,120,45,1
40 COLOR 2,2,7
45 CIRCLE 2,80,100,5
```

For a more interesting effect, try adding the following lines:

```
50 A=A+1:IFA=16THENA=1
55 COLOR 3,A,3
60 FORI=1TO250:NEXT
65 GOTO50
```

Now that you've seen most of the commands in action, let's take a look at them individually and explain how they work. Most of them have got more than one mode of operation (in fact, several modes of operation), and these different modes can be used to startling effect.

GRAPHIC

If it wasn't for this one, there wouldn't be much point in having any of the rest of the commands. GRAPHIC has five different arguments, and these are as follows:

GRAPHIC 0: Normal text mode.
GRAPHIC 1: High-resolution graphics.

GRAPHIC 2: High-resolution graphics with a five line text window at the bottom of the screen.
GRAPHIC 3: Multi-colour high-res graphics.
GRAPHIC 4: Multi-colour high-res graphics with a five-line text window at the bottom of the screen.

There is yet another option to this command, which is to have something like:

```
GRAPHIC n,0
```

or

```
GRAPHIC n,1
```

The first one doesn't clear a high-resolution screen, the second one does.

What they all do is to set aside 10K of memory, which doesn't leave you with very much really. This is set aside even if you type GRAPHIC 0, so after doing some graphics work, if you want your 10K back, you'll have to type in:

```
GRAPHIC CLR
```

COLOR

Typical American computer designers: can't even spell colour correctly. Oh well.

COLOR, like GRAPHIC, has a number of different modes of operation. Indeed, there are 640 possible variations on this comand. Such generosity!

The command takes the form:

```
COLOR A,B,C
```

where A is a number from 0 to 4 and determines what colour area will be changed. B is a number from 1 to 16, and determines what colour will be used (as described in the chapter on colour). Finally, C is a number from 0 to 7, and determines how bright the colour will be. 7 is the brightest.

What are these values of A? As follows:

A = 0: alter the background colour.
A = 1: alter the drawing, or ink, colour.
A = 2: first multi-colour colour.
A = 3: second multi-colour colour.
A = 4: alter border colour.

The second multi-colour option is an interesting one, and was used in one of the short programs described above. If you draw something on a multi-colour high-resolution screen using multi-colour number 2, and you then change that colour to be something else, everything previously drawn with that colour will also change colour as well. Handy for producing some great graphical effects.

PAINT

Every other computer uses FILL, but Commodore have to use PAINT.

What it does, though, should be fairly obvious from the word itself. It PAINTs an area on a high-resolution screen in a particular colour (or color, I suppose). Like everything else, it has about five million options to it.

```
PAINT A,X,Y,B
```

A determines which colour will be used to fill the area, and has four different values:

A = 0: paint in background colour.
A = 1: paint in current drawing colour.
A = 2: paint in first multi-colour colour.
A = 3: paint in second multi-colour colour.

X and Y are the X and Y co-ordinates of the area to be coloured around. This determines what area will be coloured in.

B can be either zero or one, and determines how the PAINT command will operate. If it's set to zero, PAINT trundles on until it comes across something that isn't the same colour being used to paint with. If it's set to one, it carries on until it reaches anything that isn't the background colour.

DRAW

Obvious really, this command is used to DRAW things on the screen. Well, lines actually. It takes the form:

DRAW A,X1,Y1 TO X2,Y2 TO X3,Y3 TO ... ad infinitum.

A determines what colour the line will be drawn in, and has four different options, as usual:

A=0: draw in background colour.
A=1: draw in current drawing colour.
A=2: draw in first multi-colour colour.
A=3: draw in second multi-colour colour.

X1 and Y1, needless to say, are the starting points for drawing the line, and the line will be drawn TO the co-ordinates X2, Y2. If you want you can then carry on TO X3,Y3, X4,Y4 and so on for ever and ever.

There probably is a limit to how many TOs you can have, but I think you'd run out of line length before you reached it.

BOX

Great ones for originality, these designers. BOX draws, well, a box! However, BOX does a lot more than just that, since it's one of the commands with a vast number of options to it. Like this:

BOX A,X1,Y1,X2,Y2,B,C

A has the usual four options:

A=0: draw in background colour.
A=1: draw in current drawing colour.
A=2: draw in first multi-colour colour.
A=3: draw in second multi-colour colour.

This refers to the outline of the box.

X1,Y1 and X2,Y2 are the bottom-left and top-right-hand corners of the box respectively.

B is the angle of rotation of the box, anything from zero degrees to 360 degrees. Actually, you could type in 720 degrees if you wanted to, but there'd be little practical point in doing it. A 45 degree rotation would draw a diamond shape, for example. A 90 degree rotation would change a long, short rectangle into a thin, tall one. And so on.

C can be either zero or one. If it's set to zero, then nothing much happens. However, if it's set to one then the area inside the box will be PAINTed in in the colour used to draw the outline of the box, as in the last of our programs given earlier.

CIRCLE

Well you'd never guess, but this command draws circles! And lots of other things as well, it must be said.

The CIRCLE command has a maximum of nine parameters associated with it, although most of them thankfully have default values which do not need to be typed in. However, in its full form, the command looks like this:

```
CIRCLE A,X1,Y1,XR,YR,SA,EA,B,D
```

Here A as usual determines what colour the circle will be drawn in:

A = 0: draw in background colour.
A = 1: draw in current drawing colour.
A = 2: draw in first multi-colour colour.
A = 3: draw in second multi-colour colour.

X1 and Y1 are the co-ordinates for the centre of the circle. XR and YR are the radius of the circle in the X and Y directions respectively. This means that we can draw not only circles but ellipses and ovals as well by giving different radii in the X and Y directions. YR defaults to equal XR if you don't feel like typing it all in.

SA and EA determine the start and end angle of the part of the circle to be drawn, if you don't want to draw a whole circle. Thus you can draw various segments of a circle if you require, as we did in the two lengthier listings at the start of this section on new commands.

As with the BOX command you can rotate whatever you draw through a certain angle, which we have labelled above as B. Like BOX, this can be anything from 0 degrees to 360 degrees and beyond.

Finally, the D parameter labelled above determines how smooth or coarse the final circle (or part of one) will be. For example:

```
10 GRAPHIC 1,1
20 COLOR 1,1
25 FORI=1TO255
30 CIRCLE 1,160,100,20,20,,,,I
31 SCNCLR
32 FORJ=0TO250:NEXT
35 NEXTI
```

will eventually end up drawing a straight line, so coarse is the circle. Along the way you'll also get boxes, triangles, pentagons, and so on.

LOCATE

This simply moves the drawing position to a new X,Y co-ordinate on the high-resolution screen. It takes the form:

```
LOCATE X,Y
```

This is in case you've used lots of PAINT, DRAW, BOX etc. commands and haven't the faintest idea where the next line is going to be drawn from. By using the LOCATE command you can position everything exactly where you want it.

CHAR

Unlike some machines with high-resolution capability, the C16 allows you to draw characters onto the screen easily. This command takes the form:

```
CHAR A,X1,Y1,"string"
```

where A, as usual, determines what colour the character will be printed in. As we saw in the second of our introductory programs, "string" can be quite a complicated argument, but you could just as easily say:

```
CHAR 1,20,10,"FRED"
```

which would print the word FRED 20 columns across and 10 rows down.

Here X1 can range from 0 to 39, and Y1 can range from 0 to 24. If the string goes over the end of a line on the screen it simply wraps around to the next line.

Apart from SCNCLR, which clears a high-resolution graphics screen as effectively as it clears an ordinary one, there are just two commands left when it comes to manipulating high-resolution graphics.

GSHAPE and SSHAPE

Many people have bemoaned the fact that the C16 doesn't support sprite graphics, and quite rightly to. The ability to handle sprites is a major aid when writing games, or indeed other programs.

Still, although we haven't got sprites we have got GSHAPE and SSHAPE, and the following program shows them in action.

```
10 GRAPHIC 3,1
20 COLOR 1,1
30 CIRCLE 1,80,100,10
35 CIRCLE 1,75,98,3:CIRCLE 1,85,98,3
40 CIRCLE 1,80,110,1
45 COLOR 3,3,3
50 PAINT 3,79,97,1
60 SSHAPE A$,70,120,90,80
65 SSHAPE B$,1,1,21,41
68 SCNCLR
70 FORI=0TO140STEP5
71 A=INT(RND(.5)*16+1):COLOR 3,A,3
75 GSHAPE A$,I,100:GSHAPE B$,I,100
80 NEXT
```

SSHAPE allows us to save a block of memory from a high-resolution screen, and GSHAPE allows us to display it somewhere else. The commands take the form:

```
SSHAPE string,X1,Y1,X2,Y2
```

```
GSHAPE string,X1,Y1,A
```

'String' can be any legally acceptable string name. X1 and Y1 are the starting X and Y co-ordinates of the piece of high-resolution screen memory to be displayed or saved. X2 and Y2 are the end points of the portion of screen memory to be saved (the corner opposite X1,Y1), and A determines in what mode the portion of memory will be displayed.

A=0: display it as it is.
A=1: invert it.
A=2: OR it with whatever happens to be on the screen at the time.
A=3: AND it with whatever happens to be on the screen at the time.
A=4: EOR it with whatever happens to be on the screen at the time (see Chapter 12 for an explanation of EOR).

Add values of A to line 75 in the above program to see what happens.

Since the maximum length of any string is 255 characters, the maximum area that you can save is not that large: the above 'face' in the program is close to the limit. However, smaller areas can be moved around the screen faster, and Chapter 12, were we learn all about machine code, will tell you how to move things about the screen at an even greater speed. Swings and roundabouts, as usual.

SCALE

We haven't met this one yet. SCALE allows us to change the scale used when drawing on a high-resolution screen, whether it be a multi-colour one or not. It takes the form:

```
SCALE n
```

where n can equal either one or zero. One turns scaling on, and zero turns it off again.

When turned on, this gives us a plotting area of 1023 by 1023, considerably more than the usual 320 by 200 (or 160 by 200 in multi-colour mode). This doesn't mean that the C16 now has a high-resolution screen capable of displaying 1023 by 1023 pixels. If you've turned scaling on and typed in something like:

```
DRAW 1,0,0 TO 1023,1023
```

it would have exactly the same result as, with scaling turned off, typing:

```
DRAW 1,0,0 TO 320,200
```

This allows you to use more accurate calculations, and lets the C16 worry about scaling everything down. You can't DRAW 1,16.5,46.3 for example, but you could turn scaling on and DRAW 1,50,146, allowing the C16 to make the necessary adjustments.

R commands

We now come to four relatively useless commands, but since they exist in the C16's repertoire we might as well cover them.

RCLR (n) tells you what colour is assigned to source n, where:

n = 0: background colour.
n = 1: foreground colour.
n = 2: multi-colour one.
n = 3: multi-colour two.
n = 4: border colour.

So, if you typed in COLOR 4,10,3, PRINTRCLR(4) would return the value of 10.

PRINT RLUM(4) in this case would return a value of 3, because RLUM(n) tells you what luminance level has been assigned to colour source n.

RGR(0) tells you what graphics mode you're currently in, and finally RDOT(n) tells you position of the last point plotted, where:

n = 0: gives you the X co-ordinate.
n = 1: gives you the Y co-ordinate.
n = 2: gives you the colour source.

All of these can also be used in the form:

```
A=RDOT(0)
```

where we've assigned the variable A to be equal to RDOT(0).

Conclusion

Graphically the C16 is a powerful machine, and it's just a shame that you have so little memory left in which to use those graphics. No sprites either, but a great wealth of commands to make the best use of what you've got.

Enough of graphics then, let's start making some noises.

8

Sound

VOL and SOUND

Unlike earlier Commodore machines, which required a multitude of POKE instructions to make any sort of noise at all, the C16 has just two commands which enable you to create a musical repertoire. Unfortunately this tends to work against the C16, since its musical ability is not on a par with, say, the Commodore 64. However, it's certainly easier to make a noise now.

The two commands are:

```
VOL n
```

where n is a number between 0 and 8, and

```
SOUND v,no,du
```

where v is a number between 1 and 3, no is the value of the musical note to be played, and du is the duration of the note.

VOL, as you might imagine, controls the volume of the note(s). 8 is the maximum level at which you can set the volume, and obviously 0 is the minimum level. Silence, in other words. Of course, if someone reaches for the volume control on the television set there's precious little that you can do about it.

SOUND can cover two voices. The first one is a 'pure' tone, whereas the second can be either 'pure' or what is termed 'noise'. Type in:

```
VOL 8 <RETURN>
SOUND 3,700,300
```

and you'll find out what a noisy voice sounds like.

```
FORI=1TO1015:SOUND3,I,2:NEXT
```

sounds like the start of a Hawkwind record.

To calculate the musical note value, we need to look at the table in the manual supplied with your C16, which gives both the frequency of the note along with the actual parameter used by the sound command. If you want to play a note other than those given in the table, you'll need to use the formula:

SOUND PARAMETER = 1024−(111840.45/FREQUENCY)

In other words you'll still have to find out the frequency of the note before you can play it.

The duration of the note is measured in 60ths of a second, so if you typed in:

```
SOUND 1,596,60
```

the C16 would play middle C for one second. The value you use for the duration can be anything from zero to 65535.

One last point to note (sorry about the pun). When using the C16's second voice to generate 'noise', please remember that it will only produce white noise in the sound parameter range 600 to 940.

And that's all there is to it!

Music maker

The following program illustrates everything in action. It isn't very long, but it performs a number of interesting functions.

```
10 COLOR 0,8,7:COLOR 4,10,4:PRINT"[CLR,BLK,RVS]C16
 MUSIC MAKER!!
20 DIMA(24),AF(24)
30 FORI=1TO24:READA(I):NEXT
40 FORI=1TO24:READAF(I):AF(I)=1024-(111840.45/AF(I
)):NEXT
```

```
50 VOL8:D1=1:D2=1:D3=1:GR=1
60 FORI=0TO7:KEY 1+I,CHR$(132+I):NEXT
70 PRINT"[2CD]PLAY USING THE KEYS: Q W E R T Y U I
80 PRINT"[CD]                      A S D F G H J K
90 PRINT"[CD]                      Z X C V B N M ,
91 PRINT"[CD]'1' TURNS ON VOICE 1:'!' TURNS IT OFF
92 PRINT"'2' TURNS ON VOICE 2:'"CHR$(34)"' TURNS I
T OFF
93 PRINT"'3' TURNS ON VOICE 3:'#' TURNS IT OFF
94 PRINT"[CD]F1/F4 TO +/- DURATION OF VOICE 1
95 PRINT"F2/F5 TO +/- DURATION OF VOICE 2
96 PRINT"F3/F6 TO +/- DURATION OF VOICE 3
97 PRINT"[CD]'=' TOGGLES VOICE 1 GLISSANDO ON/OFF
98 PRINT"'_' ALTERS RATE OF GLISSANDO
100 GETA$:IFA$=""THEN100
102 IFA$="="THENG1=1-G1
104 IFA$="_"THENGR=GR*2:IFGR>64THENGR=1
110 IFA$="1"THENV1=1
120 IFA$="2"THENV2=1
130 IFA$="3"THENV3=1
140 IFA$="!"THENV1=0
150 IFASC(A$)=34THENV2=0
160 IFA$="#"THENV3=0
170 B=PEEK(198)
180 IFA=132THEND1=D1*2:IFD1>1024THEND1=1024
190 IFA=133THEND2=D2*2:IFD2>1024THEND2=1024
200 IFA=134THEND3=D3*2:IFD3>1024THEND3=1024
210 IFA=135THEND1=D1/2:IFD1<1THEND1=1
220 IFA=136THEND2=D2/2:IFD2<1THEND2=1
230 IFA=137THEND3=D3/2:IFD3<1THEND3=1
240 A=ASC(A$):FORI=1TO24:IFA<>A(I)THENNEXT:GOTO100
241 IFG1=1THEN500
250 IFV1>0THENSOUND1,AF(I),D1
260 OF=AF(I):IFV2>0THENSOUND2,AF(I),D2
270 IFV3>0THENSOUND3,AF(I),D3
280 GOTO100
290 DATA90,88,67,86,66,78,77,44,65,83,68,70,71,72,
74,75,81,87,69,82,84,89,85,73
300 DATA130.8,146.8,164.7,174.5,195.9,220.2,246.9,
261.4,261.4,293.6,330,349.6
310 DATA392.5,440.4,494.9,522.7,522.7,588.7,658,69
9,782.2,880.7,989.9,1045
500 IFOF>AF(I)THEN502
501 FORP=OFTOAF(I)STEP40:SOUND1,P,GR:NEXT:GOTO260
502 FORP=OFTOAF(I)STEP-40:SOUND1,P,GR:NEXT:GOTO260
```

The lines up to and including line 98 just print up the on-screen display and instructions, as well as reading in the frequencies for all 24 notes (from data stored in lines 290 to 310) before turning them into actual sound parameters.

Incidentally, the strange symbol in lines 98 and 104 should be the left-arrow key.

Lines 102 and 104 check for, and alter if appropriate, the 'glissando' rate for voice one, or indeed whether 'glissando' is on or off.

Lines 110 to 160 check for, and alter if appropriate, voices 1, 2 and 3 and determine whether they are to be turned on or off.

Lines 170 to 230 check for, and alter if appropriate, the length of time for which any of the three voices are to be played.

Line 240 then checks to see whether a valid key has been pressed: one that will play a musical note.

Line 241 trots off to the routine starting at line 500 if the 'glissando' effect for voice one is turned on.

Lines 250 to 280 play the appropriate note(s), as well as keeping track of the last frequency for voice one in line 260.

Lines 500 to 502 achieve the 'glissando' effect by either going up from the old frequency to the new one or going down from the old frequency to the new one, depending on what has been pressed.

Like most features on the C16, you'll have to play around with it to get the best out of it, so take it away, maestro!

9

Writing a Games Program

They say that imitation is the sincerest form of flattery, and it is true that some of the top selling games for home computers are copies of well-known arcade games, of the Asteroids/Kong/Pacman type. However, the market for that sort of thing is strictly limited, since people aren't going to go out and buy ten different copies of the same game. Consequently, the most important thing in writing your own game is to come up with a scenario that is new, and one which will stand up to repeated playing.

Just occasionally you'll get a computer game that looks as if it should spawn an arcade game, such as Manic Miner, the game that launched a thousand rip-offs. However, for the purposes of this book we'll confine ourselves to producing something a little more humble than that.

Getting started

When looking for ideas, one of the best places to start is down at the local pub or amusement arcade. Take a look at the game there, and note carefully everything that the game is doing. If it's a game of Space Invaders, go to another pub!

Recently, in the never-ending search to come up with something new, a totally different breed of game has started to appear. Not the usual 'Genghis' type shoot and destroy everything in sight, but something very different.

Remember Tutenkhamen? A mixture of traditional death and destruction, with a little bit of adventure thrown in for good measure. You're exploring the tombs underneath an old pyramid, and have to avoid all the snakes, scarab beetles and the like that are out to get you, find a key to open a door, and pick up various treasures en route. Only by getting the key can you get to the next level, and so on.

Another good example of this genre of game is Scrambled Egg, a truly bizarre game if ever there was one. In this one you control what looks a bit like a hamburger, and this hamburger has to race around the screen kicking eggs until they hatch out into little chickens, who must then be kicked off the screen. Chasing after you are bouncing microchips, demented Pacmen, and other assorted nasties. To an infuriatingly jolly soundtrack, all manner of weird and wonderful things happen as you progress through the levels.

This is another secret of the more popular games: a number of different levels of play, whether they're simply faster versions of the first level, or completely new and different scenarios. This is one point to bear in mind when thinking of your own initial ideas: can the game progress from the humble beginnings that you've given it?

So you have to think of the scenario, make it good and different, and incorporate the possibility of extending the game by moving on to different levels when you've zapped the relevant number of aliens, collected a suitable number of treasures, or whatever.

Secondly, you don't want to make it so difficult to play that people quickly become bored and move on to other things. Defender is a classic example here: a very good piece of programming, but the game never managed to achieve the same popularity as simpler games like Asteroids or Invaders. Only people with more than the allotted number of fingers can keep playing Defender for very long.

Keep it simple, must be the rule, and don't have people hunting all over the keyboard to find the right key in moments of panic. Perhaps make it playable by joystick, but always make sure that it will work on the Basic machine. The C16, although it does of course have a joystick port, doesn't come supplied with a joystick, so if you're going to include a joystick option in your game, make sure that it can still be played using only the keyboard. Don't limit your potential audience before you've even started!

To sum up so far: think of the idea, keep it simple but with the possibility of expansion into further and more interesting levels of play, make it as addictive as possible (easy really, as most simple-to-play games have the right kind of addictive quality), and make sure that it isn't too difficult to play. There's nothing more annoying than having to stop in mid-zap to consult the manual to find out which key does the hyper-jump into safe territory away from marauding aliens.

First steps

The following reasonably short listing is a starting point for a game that we'll come back to later in this chapter.

```
5 GOSUB10000:REM INSTRUCTIONS
8 S=3112:C=2088:A$="1234"
10 COLOR 0,8,7:COLOR 4,10,4:PRINT"[CLR,BLK,3CD]
40 FORI=0TO39:POKE3072+I,230:POKE2048+I,2:NEXT
42 FORI=0TO22:POKES+I*40,230:POKEC+I*40,2:POKE3151
+I*40,230
43 POKE2127+I*40,2:NEXT
45 FORI=0TO39:POKE4032+I,230:POKE3008+I,2:NEXT
50 A=500:A1$=TI$:TI$="000000"
52 B=VAL(MID$(A$,INT(RND(.5)*4+1),1))
55 POKES+A,81:POKEC+A,0
60 POKES+A,32:POKEC+A,7
65 ONBGOTO100,200,300,400
100 A=A-40:POKES+A,81:POKEC+A,0:GOTO500
200 A=A+40:POKES+A,81:POKEC+A,0:GOTO500
300 A=A-1:POKES+A,81:POKEC+A,0:GOTO500
400 A=A+1:POKES+A,81:POKEC+A,0:GOTO500
500 PRINT"[HOME]"TI$:Z=PEEK(198):IFZ=64THEN550
502 IFZ=39THEN800
504 IFZ=36THEN900
550 H=H+1:IFH=50THEN560
551 GOTO600
560 H=INT(RND(.5)*2+77):I=INT(RND(.5)*1000+1):IFPE
EK(S+I)<>32THEN560
562 POKES+I,H:POKEC+I,0:H=0:GOTO600
600 ONBGOTO700,720,740,760
700 X=S+A-40:IFPEEK(X)=230THEN1000
701 IFPEEK(X)=77THENB=3
702 IFPEEK(X)=78THENB=4
703 GOTO55
720 X=S+A+40:IFPEEK(X)=230THEN1000
721 IFPEEK(X)=77THENB=4
722 IFPEEK(X)=78THENB=3
723 GOTO55
740 X=S+A-1:IFPEEK(X)=230THEN1000
741 IFPEEK(X)=77THENB=1
742 IFPEEK(X)=78THENB=2
743 GOTO55
760 X=S+A+1:IFPEEK(X)=230THEN1000
761 IFPEEK(X)=77THENB=2
762 IFPEEK(X)=78THENB=1
763 GOTO55
```

```
800 IFB=1THENX=S+A-40:IFPEEK(X)=230THEN1000
802 IFB=1THENPOKEX,78:POKEC+A-40,0:GOTO600
804 IFB=2THENX=S+A+40:IFPEEK(X)=230THEN1000
806 IFB=2THENPOKEX,78:POKEC+A+40,0:GOTO600
808 IFB=3THENX=S+A-1:IFPEEK(X)=230THEN1000
810 IFB=3THENPOKEX,78:POKEC+A-1,0:GOTO600
812 IFB=4THENX=S+A+1:IFPEEK(X)=230THEN1000
814 IFB=4THENPOKEX,78:POKEC+A+1,0:GOTO600
816 GOTO600
900 IFB=1THENX=S+A-40:IFPEEK(X)=230THEN1000
902 IFB=1THENPOKEX,77:POKEC+A-40,0:GOTO600
904 IFB=2THENX=S+A+40:IFPEEK(X)=230THEN1000
906 IFB=2THENPOKEX,77:POKEC+A+40,0:GOTO600
908 IFB=3THENX=S+A-1:IFPEEK(X)=230THEN1000
910 IFB=3THENPOKEX,77:POKEC+A-1,0:GOTO600
912 IFB=4THENX=S+A+1:IFPEEK(X)=230THEN1000
914 IFB=4THENPOKEX,77:POKEC+A+1,0:GOTO600
916 GOTO600
1000 COLOR0,9,4:PRINT"[CLR,WHT]CRASH ...
1001 PRINT"[CD]YOU LASTED "MID$(TI$,1,2)" HOURS, "
MID$(TI$,3,2)" MINUTES
1002 PRINT"AND "MID$(TI$,5,2)" SECONDS!"
1003 FORI=1TO10:GETAG$:NEXT
1004 PRINT"[CD]ANOTHER GAME (Y OR N)?"
1006 GETAG$:IFAG$="Y"THENPRINT"[BLK]":H=0:GOTO10
1008 IFAG$="N"THENEND
1010 GOTO1006
10000 COLOR0,9,4:COLOR4,10,4:PRINT"[CLR,WHT]WIZZO!
!":PRINT"[6CBMY]"
10002 PRINT"[CD]USE THE 'N' AND 'M' KEYS TO CREATE
"
10004 PRINT"BARRIERS AND GUIDE THE BALL AROUND THE
10006 PRINT"SCREEN.  AVOID HITTING THE WALLS AT TH
E
10008 PRINT"SIDE OF THE SCREEN.
10010 PRINT"[CD]OCCASIONALLY A RANDOM BARRIER WILL
10012 PRINT"APPEAR.  THIS IS JUST TO STOP YOU
10014 PRINT"SITTING AROUND AND DOING NOTHING FOR
10016 PRINT"TOO LONG!
10018 PRINT"[CD]PRESS <[RVS]SPACE[OFF]> TO START
10020 FORI=1TO10:GETGA$:NEXT
10022 GETGA$:IFGA$<>" "THEN10022
10024 RETURN
```

Why Wizzo? Who knows, perhaps I was feeling very British at the time.

This is a (very!) simple game, in which you have to guide a ball around a screen by placing obstacles in its path. These cause it to bounce around, and the object of the game is to avoid the border of the screen and to keep going as long as possible. The instructions for steering the ball are given in lines 10000 onwards. Let's take a quick look at how the rest of it works.

Program structure

Line 5: obvious.

Line 8: set up a variable for screen memory, then colour memory, and then finally A$.

Line 10: set the border, background, and text colours.

Lines 40-45: draw a border around the screen.

Line 50: set the time to zero.

Line 52: take a random element from the variable A$.

Line 55: place our ball on the screen using the variable A from line 50.

Line 60: erase the ball so that we don't get a load of ... well, I think you'd better finish that sentence!

Line 65: governs program flow.

Lines 100-400: place ball on screen depending on which direction we're going in.

Line 500: display elapsed time and check for a key press.

Lines 502-504: go somewhere if particular key has been pressed.

Lines 550-551: update variable H and every 50th time go off to line 560, otherwise go to line 600.

Lines 560-562: place a random obstacle on the screen.

Line 600: governs program flow depending on direction of movement.

Lines 700-703: going up the screen, and checking for collisions with anything.

Lines 720-763: ditto for going down, left and right respectively.

Lines 800-816: placing one kind of obstacle on the screen, checking for a collision with the screen border, and going off to line 600.

Lines 900-916: ditto for other kind of obstacle.

Lines 1000-1016: oh dear, you've crashed!

A reasonably addictive little game, that at least forms the basis for our finished product later on. However, that's only the start. We need to do a lot more work before we arrive at the final product.

Defining characters

Assuming that we want this to finish up as some kind of space game, take a look at the real arcade games and try to decide what it is about them that makes some more popular than others. We've looked at some of the reasons already, but there's a lot more that goes into making a long-lasting and successful game.

As ever, it's the little touches that help. We'll be looking at (or listening to, perhaps) sound later, a vital factor in games playing (at least televisions have got volume controls if the sound becomes too irritating), but for now we'll just concentrate on the screen display.

In a space game, a nice touch is to have a perpetually changing starry background. With bit mapping of the screen it's not too difficult to achieve this, and this in turn achieves something else. It obeys the golden rule of graphics on the screen: always keep it moving, and always keep it changing. Stars that wink on and off and change colour help to make the game that little bit more exciting. We won't be doing any bit mapping, but we will find a way of keeping things on the move and constantly changing.

Obviously, though, the most important piece of design work involves the major characters in the game, namely the hero and the villain. Perhaps you could use the Character Builder program from Chapter 7 to help you out here? However you do it, you'll probably do it using the same routine again and again. Most programs use very similar routines time after time, and arcade games are no exception. Once you've written one, the rest follow on quite easily: like everything else, the first time is the most difficult, and takes the most work.

Anyway, how to define a character? You could have any old boring spaceship, but with the capability of the C16 it would be a great shame if a good idea were to fail at this stage. Come up with a good design that is not merely a copy of someone else's. You don't need to go so far as to have a mole or something spitting out venomous lasers and destroying everything in sight, but it would make sense to have something that is slightly different from the usual run-of-the-mill spaceship.

Take great care over its design, because it's going to be the one thing that remains on the screen all the time, and it thus becomes the item that the person playing the game will have fixed in their mind. If it's boring to look at, they'll rapidly forget the game after just a few minutes spent away from it, but if it looks good, and original, it will remain in the mind and you'll be on your way to fame and fortune! However, people remember bad ideas just as well as they remember good ones, and it's much harder to succeed after an initial failure, so get it right first time!

Okay, you've designed the hero and thought about the background display, so what are the aliens going to look like?

Alien design

Here again it's well worth going to have a look at some of the more successful pub games, and the shape and form that the aliens take in those. Cosmic Firebird is one very good example, where it is obvious that a lot of thought has gone into the alien design: the characters look good, and you remember them months after playing the game.

A form that was in vogue a while ago was that of the flying bird, and you can probably recall a number of games that had birds flapping about all over the place. I'm not suggesting that you copy those, but at least give yourself some idea of what everyone else is doing. He who watches the competition stays ahead of the competition (ancient Chinese proverb).

Decide what those aliens will do. Will they split up as they come down the screen? Will they have to be shot once, twice or three times before going to tha great invader graveyard in the sky? Will they home in on the avenging spaceship, or just do a fine impression of a bunch of lemmings as they fall down the screen? If they go off one side of the screen will they re-appear at the other, or come down again from the top of the screen?

Sort out all these things in your mind, and write the alien handling routines accordingly. No one is going to like a game where movement of the various objects on the screen follows different rules from time to time. If they disappear off the left-hand side of the screen and then re-appear on the right, that is what they should do throughout the game.

Will you have more than one sort of alien? In the vast majority of the popular games there are many different types of enemy, and so you too will probably have to resign yourself to the fact that you'll be designing more than one type of alien.

Character movement

Your spaceship must also adhere to the rules of movement, and this will possibly be one of the most difficult parts of the program to write. Not only do the images on the screen have to move, but you also have to keep track of where they've moved to. How else will you know whether an alien has been shot or not, or has left the screen, or has (horror of horrors) collided with our heroic spaceship?

When characters collide, will something happen to them, or will they just sail serenely past each other? Again, you'll have to decide on that one.

Whatever they do, always keep something on the move. We've said it earlier, but it really is the most important rule in the world of computer animation. Always, but always, keep something on the screen and keep something happening. You want to hold the attention of the player, not send him to sleep.

Right, we have now decided on our scenario, we've designed our characters, both friendly and unfriendly, and we've decided how they're going to move about the screen and what rules they will obey as they do so. All this means that we can now sit down and write the game, doesn't it?

Well, unfortunately, the answer is no! We've done the bulk of the program design, but there's an awful lot more yet that has to be done before the game is going to be a success.

Just think back to all the popular arcade games and think what they have in common. As well as superb graphics and animation, sure screen movement and alien handling, easy manipulation of the enemy spacecraft and so on, there is something else which we haven't yet considered.

What are we looking for? The answer is that we're not looking for something, we're listening for it.

Crash, bang, wallop

The hills are alive with the sound of music, as they say, and this applies to beady little aliens just as much as to Julie Andrews.

Sound can make or break a game, and if someone doesn't like it they can always turn down the volume on their television set. However, good use of sound will greatly enhance a game, and can turn a relatively ordinary game into a very exciting one. Remember the original Space Invaders? That exploited sound with great success. Their great trick was to reproduce the human heartbeat, and have that speeded up as the game progressed. Thus as it got faster and faster so your reactions seemed to speed up, and at the end of a long game you could be considerably exhausted.

Without sound it's arguable whether this would have been the case, and it was a major factor in making Space Invaders as popular and enduring as it was. Apply sound to your games, but do make sure that the sound is an addition to and not a distraction from what's going on.

When you destroy an alien, respond with a satisfying death scream. When your spaceship is zapped into smithereens, and again you must decide how that happens, respond with sound of cosmic proportions, as befits the death of a space hero.

Asteroids, like Space Invaders, was another game to feature a continually running sound, and any game that is to make its mark must do the same as well. It mustn't be a noise that irritates, like a high-pitched whine, but it must create some kind of response in the user other than making him reach immediately for the volume control.

High scoring

From sound, we come to the question of keeping score. You must decide how many points are gained by each destruction of an alien, and whether all the aliens merit the same number of points when they die. Most people seem to favour high scoring games rather than low scoring ones, so although zapping ten aliens can give you a score of anything from ten points to ten thousand points, there're something a lot more satisfying about scoring ten thousand!

Is there going to be some kind of time limit to your game, such as fuel running out or something of that nature? In a case like that, points could be gained by refuelling, for instance. However you ultimately decide that the points will be allocated, you'll certainly have to keep track of the high score: the player has to have some kind of target to aim at. One of the reasons why people keep coming back to games is to beat the previous high score, particularly if it's been scored by someone else. I know, because when I've finished this chapter I'm going to the pub to play Scrambled Egg and beat my mate Roger's highest score!

Inevitably, the game will end in destruction of some kind for the player. A rule of arcade games has to be that you just can't go on for ever and win, no matter how hard you try, so the only incentive to keep going is to try to better your previous score. You'll probably also want to keep track of more than one high score as well. Some games go a bit overboard on this, and hold anything up to 150 scores, but ten or so is usually sufficient. Perhaps room to enter the player's name as well? We've all got a streak of vanity in us.

Back to Wizzo

But enough of theory, back to practice. The final version of Wizzo as presented here is by no means a saleable and finished product. But, after having read the above, perhaps you could turn it into something really special. Anyway, let's look at the listing.

```
0 POKE 52,55:POKE 56,55
1 REMGOSUB40000
2 REMFORI=0TO2047:POKE14336+I,PEEK(53248+I):NEXT
3 VOL8:B0=24:B2=66:B3=153:B4=14392
5 FORI=0TO79:READA:POKE14336+I,A:NEXT
6 GOSUB10000:REM INSTRUCTIONS
7 FORI=0TO7:POKE14336+8*32+I,0:NEXT:FORI=0TO79:REA
DA:POKE14720+I,A:NEXT
8 S=3112:C=2088:A$="1234"
10 COLOR 0,8,7:COLOR 4,10,4:PRINT"[CLR,BLK,3CD]
11 POKE65298,192:POKE65299,56
12 SOUND3,800,110
40 FORI=0TO39:POKE3072+I,5:POKE2048+I,0:NEXT
42 FORI=0TO22:POKES+I*40,5:POKEC+I*40,0:POKE3151+I
*40,5
```

```
43 POKE2127+I*40,0:NEXT
45 FORI=0TO39:POKE4032+I,5:POKE3008+I,0:NEXT
46 FORI=1TO(Q-1)*5
47 B1=INT(RND(.5)*920+1):IFPEEK(S+B1)<>32THEN47
48 POKES+B1,7:POKEC+B1,6:SOUND2,600,10:FORJ=1TO100
:NEXTJ,I
49 A=500:A1$=TI$:TI$="000000"
50 B=VAL(MID$(A$,INT(RND(.5)*4+1),1))
51 ZZ=B:GOTO55:REM IFB=1THENZZ=3:GOTO55
52 IFB=2THENZZ=4:GOTO55
53 IFB=3THENZZ=5:GOTO55
54 ZZ=6
55 POKES+A,ZZ:POKEC+A,0
60 POKES+A,32:POKEC+A,7
65 SOUND1,B*100,10:ONBGOTO100,200,300,400
100 A=A-40:POKES+A,ZZ:POKEC+A,0:GOTO500
200 A=A+40:POKES+A,ZZ:POKEC+A,0:GOTO500
300 A=A-1:POKES+A,ZZ:POKEC+A,0:GOTO500
400 A=A+1:POKES+A,ZZ:POKEC+A,0:GOTO500
500 PRINT"[HOME]"TI$:Z=PEEK(198):B0=24-B0:B2=66-B2
:B3=153-B3:B5=231-B5
501 POKEB4,B0:POKEB4+1,B2:POKEB4+3,B3:POKE14384,B5
502 IFZ=62THEN800:REM Q PRESSED
504 IFZ=41THEN900:REM P PRESSED
506 IFZ=60THEN1100:REM FIRE
550 H=H+1:IFH=50THEN560
551 GOTO600
560 IFQ>2ANDE=0THENGOSUB1200
561 H=INT(RND(.5)*2+8):I=INT(RND(.5)*1000+1):IFPEE
K(S+I)<>32THEN560
562 POKES+I,H:POKEC+I,11:H=0
563 SOUND3,600,30
564 I=INT(RND(.5)*1000+1):IFPEEK(S+I)<>32THEN564
565 POKES+I,7:POKEC+I,6:H=0
600 IFE>0THENX1=X1+1:IFX1=3THENX1=0:GOSUB1201
601 ONBGOTO700,720,740,760
700 X=S+A-40:GOSUB775
701 IFPEEK(X)=9THENB=3
702 IFPEEK(X)=8THENB=4
703 GOTO51
720 X=S+A+40:GOSUB775
721 IFPEEK(X)=9THENB=4
722 IFPEEK(X)=8THENB=3
723 GOTO51
740 X=S+A-1:GOSUB775
741 IFPEEK(X)=9THENB=1
742 IFPEEK(X)=8THENB=2
743 GOTO51
760 X=S+A+1:GOSUB775
761 IFPEEK(X)=9THENB=2
762 IFPEEK(X)=8THENB=1
```

```
763 GOTO51
775 IFPEEK(X)=5ORPEEK(X)=6ORPEEK(X)=7ORPEEK(X)>47T
HEN1000
776 RETURN
800 IFB=1THENX=S+A-40:GOSUB775
802 IFB=1THENPOKEX,8:POKEC+A-40,11:GOTO600
804 IFB=2THENX=S+A+40:GOSUB775
806 IFB=2THENPOKEX,8:POKEC+A+40,11:GOTO600
808 IFB=3THENX=S+A-1:GOSUB775
810 IFB=3THENPOKEX,8:POKEC+A-1,11:GOTO600
812 IFB=4THENX=S+A+1:GOSUB775
814 IFB=4THENPOKEX,8:POKEC+A+1,11:GOTO600
816 GOTO600
900 IFB=1THENX=S+A-40:GOSUB775
902 IFB=1THENPOKEX,9:POKEC+A-40,11:GOTO600
904 IFB=2THENX=S+A+40:GOSUB775
906 IFB=2THENPOKEX,9:POKEC+A+40,11:GOTO600
908 IFB=3THENX=S+A-1:GOSUB775
910 IFB=3THENPOKEX,9:POKEC+A-1,11:GOTO600
912 IFB=4THENX=S+A+1:GOSUB775
914 IFB=4THENPOKEX,9:POKEC+A+1,11:GOTO600
916 GOTO600
1000 T=VAL(TI$):COLOR0,9,4:PRINT"[CLR,WHT]":POKE65
299,208:POKE65298,196:SOUND3,10,120
1001 PRINT"[CLR]CRASH ... YOU LASTED "MID$(TI$,1,2
)" HRS., "MID$(TI$,3,2)" MINUTES
1002 PRINTMID$(TI$,5,2)" SECONDS":IFT>HSTHENHS=T:G
OTO1050
1003 FORI=1TO10:GETAG$:NEXT
1004 PRINT"[CD]ANOTHER GAME - Y OR N?
1006 GETAG$:IFAG$="Y"THENGOSUB10015:PRINT"[BLK]":E
=0:GOTO8
1008 IFAG$="N"THENEND
1010 GOTO1006
1050 PRINT"[CD]GREAT, A NEW WORLD RECORD ...
1052 PRINT"[CD]THE HIGHEST SCORE NOW STANDS AT [CL
]";HS*50:GOTO1003
1100 ONBGOTO1120,1140,1160,1180
1120 FORI=S+ATOSSTEP-40:IFPEEK(I)<>40THENPOKEI,1:P
OKEC+I-S,1:POKEC+I-S,7:POKEI,32
1121 NEXT:GOTO51
1140 FORI=S+ATO1983STEP40
1141 IFPEEK(I)<>40THENPOKEI,32:POKEC+I-S,1:POKEC+I
-S,7:POKEI,32
1142 NEXT:GOTO51
1160 L=INT(A/40):L1=A-L*40:FORI=L*40+L1TOL*40+1STE
P-1
1161 IFPEEK(S+I)<>40THENPOKES+I,3:POKEC+I,1:POKES+
I,32:POKEC+I,7
1162 NEXT:GOTO51
1180 L=INT(A/40):L1=A-L*40:FORI=L*40+L1TOL*40+38
```

```
1181 IFPEEK(S+I)<>40THENPOKES+I,4:POKEC+I,1:POKES+
I,32:POKEC+I,7
1182 NEXT:GOTO51
1200 IFE=0THENE=1:QE=500:POKES+QE,6:POKEC+QE,2:BE=
INT(RND(.5)*4+1):RETURN
1201 REMCHECK ON THE MONSTER
1202 ONBEGOTO1400,1404,1408,1412
1203 POKES+QE,32:POKEC+QE,7
1204 REM
1205 IFPEEK(X)=5ORPEEK(X)=7ORPEEK(X)>47THEN1207
1206 GOTO1210
1207 IFBE=1ORBE=2THENBE=3-BE
1208 IFBE=3ORBE=4THENBE=7-BE
1210 ONBEGOTO1510,1520,1530,1540
1400 X=S+QE-40
1401 IFPEEK(X)=9THENBE=3
1402 IFPEEK(X)=8THENBE=4
1403 GOTO1203
1404 X=S+QE+40
1405 IFPEEK(X)=9THENBE=4
1406 IFPEEK(X)=8THENBE=3
1407 GOTO1203
1408 X=S+QE-1
1409 IFPEEK(X)=9THENBE=1
1410 IFPEEK(X)=8THENBE=2
1411 GOTO1203
1412 X=S+QE+1
1413 IFPEEK(X)=9THENBE=2
1414 IFPEEK(X)=8THENBE=1
1415 GOTO1203
1420 X=S+A+40:GOSUB775
1500 SOUND3,1000,10
1501 RETURN
1510 QE=QE-40:POKES+QE,6:POKEC+QE,2:GOTO1500
1520 QE=QE+40:POKES+QE,6:POKEC+QE,2:GOTO1500
1530 QE=QE-1:POKES+QE,6:POKEC+QE,2:GOTO1500
1540 QE=QE+1:POKES+QE,6:POKEC+QE,2:GOTO1500
10000 COLOR 0,9,4:COLOR 4,10,6:PRINT"[CLR,WHT]TRAP
PER":PRINT"[7CBMY]"
10002 PRINT"[CD]USE THE Q AND P KEYS TO CREATE"
10004 PRINT"BARRIERS AND GUIDE THE BALL AROUND THE
10006 PRINT"SCREEN.  AVOID HITTING THE WALLS AT TH
E
10008 PRINT"SIDE OF THE SCREEN.
10009 PRINT"[CD]PRESS SPACE TO FIRE A MISSILE AND
      CLEAR EVERYTHING OUT OF YOUR";
10010 PRINT" PATH.":PRINT:PRINT"OCCASIONALLY A RAN
DOM BARRIER/BOMB WILL
10012 PRINT"APPEAR.  THIS IS JUST TO STOP YOU
10014 PRINT"SITTING AROUND AND DOING NOTHING FOR T
OOLONG.
```

```
10015 PRINT"[CD]SKILL LEVEL - 1 TO 9. 1 IS THE EAS
IEST.
10016 GETA$:IFA$=""THEN10016
10017 Q=VAL(A$):IFQ>9ORQ<1THEN10016
10019 PRINT"[CD]LEVEL"Q"SELECTED."
10020 PRINT"[CD]PRESS <SPACE> TO START
10021 FORI=1TO10:GETGA$:NEXT
10022 GETGA$:IFGA$<>" "THEN10022
10024 RETURN
30000 DATA28,62,255,31,63,255,62,28
30001 DATA66,195,231,231,255,255,126,60
30002 DATA60,126,255,255,231,231,195,66
30003 DATA124,254,63,7,7,63,254,124
30004 DATA62,127,252,240,240,252,127,62
30005 DATA255,129,189,189,189,189,129,255
30006 DATA231,189,126,90,126,126,195,195
30007 DATA24,66,24,153,60,126,126,60
30008 DATA7,15,25,48,96,192,192,224
30009 DATA224,240,152,12,6,3,3,7
30010 DATA60,102,110,118,102,102,60,0
30011 DATA24,24,56,24,24,24,126,0
30012 DATA60,102,6,12,48,96,126,0
30013 DATA60,102,6,28,6,102,60,0
30014 DATA6,14,30,102,127,6,6,0
30015 DATA126,96,124,6,6,102,60,0
30016 DATA60,102,96,124,102,102,60,0
30017 DATA126,102,12,24,24,24,24,0
30018 DATA60,102,102,60,102,102,60,0
30019 DATA60,102,102,62,6,102,60,0
40000 COLOR0,2,6:COLOR4,2,6:PRINT"[BLU]
40001 PRINT"[CLR]WIZZO ARRIVES ...":PRINT"[2CD]PLE
ASE WAIT ... READING CHARACTERS ..."
40002 RETURN
```

There's no point going into quite as much detail as we did with version one, but a few words of explanation are called for. At the start of the program the top 2K of Basic memory are set aside to hold details for our user-defined characters (aliens, bombs, heroes etc.). The only major changes to the game occur in lines 1100-1182 and 1200-1540, which respectively allow the hero to fire, and control the movement of the enemy about the screen.

If you type in the listing and play the game, you'll note that some bombs have been added, and that they flash at you as the game goes on. This is quite easy to achieve, simply by altering the data for the bomb character as it's stored in memory. Lines 500 and 501 take care of this.

So, the challenge is on. Can you take Wizzo and turn it into the greatest game of all time? Back to the keyboard!

10

Building up a Database

One of the more useful programs on any computer is the database. Opinions seem to differ over what a database should be capable of doing, but essentially it should be able to store and retrieve information, rather like a card filing system, and it should also be able to search through the files and sort them into order on a number of different fields, rather like an overworked and underpaid secretary.

For the purposes of this book, we're going to present the database in the form of an address file, and build it up from scratch, explaining what each routine does and how it does it. In this program we've allowed enough room to store around 50 'cards', with each card holding 8 different 'fields' of information, although practical memory limitations impose an upper limit of much less than that. Perhaps, with a bit of careful re-writing and renumbering, you could improve on things? Anyway, the fields are as follows:

Name of person.
Four fields for the address.
Postcode.
Telephone number.
Date of birth.

The program can search through any of those eight fields for any bit of information within that field. For instance, if the person's name was GERRARD, and you decided to search on the key word ARD, it would still find GERRARD, and also pull out any other names that happened to have ARD in them.

The program can also sort into order on the first seven fields. You can't sort on date of birth, unfortunately, since the time taken to do that would be a mite prohibitive. Still, if anyone wants to take up the challenge ...

The listing will be broken down into eleven major 'chunks': the ten parts of the program that manipulate, store, retrieve, file, sort, display, load and save the data, and the eleventh 'chunk' will be any other part of the program that doesn't fit into those categories. Displaying the menu, for instance, and setting up some data in the first place.

There are some interesting routines contained within the main body of the listing, so without further ado we'll start with all those parts of the program that do not fit into one of the operations categories.

The peripheral parts of the program

```
5 PRINTCHR$(14):F=0
30 DIMF1$(50),F2$(50),F3$(50),F4$(50),F5$(50),F6$(
50),F7$(50),F8$(50)
5000 COLOR0,1:COLOR4,1:PRINT"[CLR,YEL]16BASE : MAI
N MENU SELECTION
5010 PRINT:PRINT:PRINT"0) LOAD FILE FROM DISK
5015 PRINT:PRINT"1) SAVE FILE TO DISK
5020 PRINT:PRINT"2) ADD FILE
5025 PRINT:PRINT"3) REMOVE FILE
5030 PRINT:PRINT"4) EXAMINE FILE
5035 PRINT:PRINT"5) AMEND FILE
5040 PRINT:PRINT"6) DISPLAY ALL FILES
5045 PRINT:PRINT"7) SEARCH FILE
5050 PRINT:PRINT"8) SORT FILE
5055 PRINT:PRINT"9) QUIT PROGRAM
5060 PRINT:PRINT"PRESS NUMERIC KEY FOR REQUIRED OP
TION."
5061 GETK$:IFK$=""THEN5061
5065 IFK$="0"THEN5500
5066 K=VAL(K$):IFK<1ORK>9THEN5061
5067 ONKGOTO6000,6500,7000,7500,8000,8500,9000,950
0,10000
```

Explanation

Line 5 - go into lower case mode, and set the variable flag F.

Line 30 - dimension field arrays to 50 each, one for each field.

Lines 5000-5060 - clear screen, set border and background colours to be black, and display the program menu option.

Lines 5061-5115 - wait for a key to be pressed and branch to the appropriate part of the program. If nothing suitable is pressed, loop back and wait until it is.

```
60000 REM INPUT ROUTINE
60002 CM$=""
60004 PRINT "[RVS]*[OFF,CL]";
60006 GETZ$:IFZ$=""THEN60006
60008 Z=ASC(Z$):IFZ<>13ANDZ<>20ANDZ<>32AND(Z<47ORZ
>57)AND(Z<65ORZ>90)THEN60006
60010 ZL=LEN(CM$):IFZL>27THEN60014
60012 IFZ>57THENZ=Z+128:Z$=CHR$(Z):CM$=CM$+Z$:PRIN
TZ$;:GOTO60004
60013 IFZ<>13ANDZ<>20THENCM$=CM$+Z$:PRINTZ$;:GOTO6
0004
60014 IFZ=13ANDZLTHENPRINT" ":RETURN
60016 IFZ=20ANDZLTHENCM$=LEFT$(CM$,ZL-1):PRINTZ$;
60018 GOTO60004
```

Lines 60000-60018 - this is our multi-purpose input routine, and deserves closer examination.

Line 60002 - declare input string to be a null one.

Line 60004 - print up prompt.

Line 60006 - wait for a key to be pressed.

Line 60008 - get the ASCII value of the key pressed. If that key wasn't the RETURN key, the DELETE key, a letter, a number, or the backslash key then we don't want to know, so back to line 60006.

Line 60010 - check the length of the input string. If it's greater than 27 characters, then check to see if the RETURN key's been pressed.

Line 60012 - if a letter has been pressed, turn it into an upper case one, add it to the input string, display it, and loop back for more.

Line 60013 - if it's numeric, or the backslash key, then add it to the input string, display it, and loop back for more.

Line 60014 - if RETURN has been pressed, and the input string actually has a character in it, then return from the subroutine. To enter a blank field then requires that you at least press the space bar.

Line 60016 - if the delete key has been pressed and the input string has some characters in it, then take off the rightmost character and echo that to the screen.

Line 60018 - back to the prompt again.

LOAD routine

This section of the program is used to load in any data previously saved onto disk. To convert this program to run on tape, you'll need to convert lines 5506 and 6006 to read as follows:

5506 OPEN1,1,1,"DATA"

6006 OPEN1,1,0,"DATA"

It will then take an exceedingly long time to file everything onto tape, but at least it will work.

```
5500 PRINT"[CLR,YEL]LOADING DATA FILE"
5502 PRINT:PRINT:PRINT"WHEN DATA DISK IS READY, PR
ESS SPACE    BAR."
5504 GETLO$:IFLO$<>" "THEN5504
5506 OPEN2,8,2,"@0:DATA,S,R"
5508 FORI=1TO50
5510 INPUT#2,F1$(I),F2$(I),F3$(I),F4$(I),F5$(I),F6
$(I),F7$(I),F8$(I)
5512 NEXTI
5514 CLOSE2:GOTO5000
```

Explanation

Line 5500 - print a message onto the screen.

Line 5502-5504 - print another message, and wait while the user gets the disk ready and puts it in the drive.

Line 5506 - open a sequential file for reading the data file imaginatively called DATA.

Line 5508 - set up a loop to be performed

Line 5510 - input the Ith element for each of the 8 data fields.

Line 5512 - continue until the end of the loop.

Line 5514 - close the file, and go back to the menu again.

SAVE routine

This is the routine that saves all your precious data onto disk (or tape if you choose to amend the program).

Since Commodore disk drives don't particularly like null strings being saved onto them, we have to incorporate a check for every field on every 'card', and if it's a null field pad it out with a single space character. This takes a little while, but at least the file gets properly prepared for saving, and prevents any headaches later.

```
6000 PRINT"[CLR,YEL]SAVING DATA FILE"
6002 PRINT:PRINT:PRINT"WHEN DATA DISK IS READY, PR
ESS SPACE    BAR."
6004 GETSA$:IFSA$<>" "THEN6004
6006 OPEN2,8,2,"@0:DATA,S,W"
6007 PRINT:PRINT:PRINT"PREPARING FILE.":GOSUB6020
6008 FORI=1TO50
6010 PRINT#2,F1$(I);CHR$(13);F2$(I);CHR$(13);F3$(I
);CHR$(13);F4$(I);CHR$(13);
6012 PRINT#2,F5$(I);CHR$(13);F6$(I);CHR$(13);F7$(I
);CHR$(13);F8$(I);CHR$(13);
6014 NEXTI
6016 CLOSE2:GOTO5000
6020 FORI=1TO50
6022 IFF1$(I)=""THENF1$(I)=" "
6024 IFF2$(I)=""THENF1$(I)=" "
6026 IFF3$(I)=""THENF1$(I)=" "
6028 IFF4$(I)=""THENF1$(I)=" "
6030 IFF5$(I)=""THENF1$(I)=" "
6032 IFF6$(I)=""THENF1$(I)=" "
6034 IFF7$(I)=""THENF1$(I)=" "
6036 IFF8$(I)=""THENF1$(I)=" "
6038 NEXTI
6040 PRINT:PRINT:PRINT"FILE READY FOR SAVING."
6042 RETURN
```

Explanation

Line 6000 - print a message onto the screen.

Line 6002-6004 - print another message, and wait while the user gets the disk ready and puts it in the drive.

Line 6006 - open a sequential file for writing the data file.

Line 6007 - tell the user that the file is being prepared, and go to the subroutine at line 6020.

Line 6008 - set up a loop to be performed 50 times.

Lines 6010-6012 - print all the data fields for each card onto disk, separating each one with a carriage return.

Line 6014 - carry on until the loop is finished.

Line 6016 - close the file and go off to the main menu again.

Lines 6020-6042 - go through every field on every 'card', and if that field is a null one then pad it out with an empty string. Then rush back to line 6008 again to carry on saving the file.

Adding a file to the record

This routine is called up whenever the user wishes to add a file to the main collection of 'cards'. The user can choose which number he wishes to call the file, and if you want to put file number 49 immediately after file 2 then that's fine by me. However, you might have to wait a long time if you then decide to flip through every file in turn before getting to the one you want.

```
6500 PRINT"[CLR,YEL]ADDING FILE":PRINT:PRINT
6502 INPUT "FILE NUMBER FOR NEW FILE";F:IFF<0ORF>5
0THEN6502
6503 F(F)=F
6504 PRINT:PRINT:PRINT "NAME ";:GOSUB60000:F1$(F)=
CM$
6506 PRINT "ADDRESS 1 ";:GOSUB60000:F2$(F)=CM$
6508 PRINT "ADDRESS 2 ";:GOSUB60000:F3$(F)=CM$
6510 PRINT "ADDRESS 3 ";:GOSUB60000:F4$(F)=CM$
6512 PRINT "ADDRESS 4 ";:GOSUB60000:F5$(F)=CM$
6514 PRINT "POSTCODE ";:GOSUB60000:F6$(F)=CM$
6516 PRINT "TELEPHONE NUMBER ";:GOSUB60000:F7$(F)=
CM$
6518 PRINT "DATE OF BIRTH (DD/MM/YY) ";:GOSUB60000
:F8$(F)=CM$
6520 PRINT:PRINT:PRINT"RECORD ADDED.":FORI=1TO2000
:NEXT
6999 GOTO5000
```

Explanation

Line 6500 - print up a message.

Line 6502 - input the number of the file to be added, and check that the user doesn't enter a number less than 1 or greater than 50

Line 6503 - set our file record to equal the number typed in.

Line 6504 - print up the message 'name', and go to the input routine starting at line 60000. On returning, the input string CM$ is assigned to the first field for this new file.

Lines 6506-6518 - ditto for the other seven fields in our file.

Line 6520 - print up appropriate message, and set up a loop to give the user time to read it.

Line 6999 - back to the main menu again.

Removing a file

This routine is called whenever a record is to be removed. This just sets every field of that record to be a null one, and thus when searching through it or attempting to sort it this record will be treated by the program as if it no longer existed.

```
7000 PRINT"[CLR,YEL]REMOVE FILE"
7002 PRINT:PRINT:INPUT "FILE NUMBER TO BE REMOVED"
;F:IFF<0ORF>50THEN7002
7004 PRINT:PRINT:PRINT"ARE YOU SURE (Y OR N)?"
7006 GETRE$:IFRE$="Y"THEN7012
7008 IFRE$="N"THEN5000
7010 GOTO7006
7012 F1$(F)="":F2$(F)="":F3$(F)="":F4$(F)=""
7014 F5$(F)="":F6$(F)="":F7$(F)="":F8$(F)=""
7016 PRINT:PRINT:PRINT"RECORD REMOVED."
7018 FORI=1TO2000:NEXT
7020 GOTO5000
```

Explanation

Line 7000 - print up suitable message.

Line 7002 - ask the user for the file number to be removed, and check that he enters a number greater than 0 and less than 51.

Line 7004 - check that he really wants to remove this file.

Line 7006 - he does, so carry on with the routine by going to line 7012.

Line 7008 - he chickens out, so revert back to our familiar main menu again.

Line 7010 - nothing's been pressed, so back to line 7006 again.

Lines 7012-7014 - nullify every field on the Fth card.

Lines 7016-7018 - print out a suitable message and set up a loop to give the user time to read it.

Line 7020 - revert back to the main menu again.

Examining a file

This menu option is included to give the user the chance to look at selected files simply by inputting a file number. The program will then display that file on the screen, before going back to the menu again at the press of a suitable key.

```
7500 PRINT"[CLR,YEL]EXAMINE FILE"
7502 PRINT:PRINT:INPUT"FILE NUMBER TO BE EXAMINED"
;F:PRINT:PRINT
7503 IFF<0ORF>50THEN7502
7504 PRINT "NAME             ";F1$(F)
7506 PRINT "ADDRESS 1        ";F2$(F)
7508 PRINT "ADDRESS 2        ";F3$(F)
7510 PRINT "ADDRESS 3        ";F4$(F)
7512 PRINT "ADDRESS 4        ";F5$(F)
7514 PRINT "POSTCODE         ";F6$(F)
7516 PRINT "TELEPHONE NUMBER ";F7$(F)
7518 PRINT "DATE OF BIRTH    ";F8$(F)
7520 PRINT:PRINT:PRINT"PRESS SPACE BAR TO CONTINUE
."
7522 GETANY$:IFANY$<>" "THEN7522
7526 GOTO5000
```

Explanation

Line 7500 - a message telling you what's going on.

Line 7502 - get the user to input the number of the file he wants to examine.

Line 7503 - check that the file number ian't less than zero or greater than 50.

Lines 7504-7518 - display all the fields for the file number entered.

Line 7520 - tell the user to press the space bar to continue.

Line 7522 - and wait until he does.

Line 7526 - back to the menu again.

Amending a file

This routine comes into play when the user wants to amend an existing file (if someone moves or changes their telephone number perhaps). It can also be used to add a file.

If a field is to remain as it was, pressing the RETURN key will take the user onto the next one. To change it, just use the delete key and the input routine at 60000 onwards takes care of the rest.

```
8000 PRINT"[CLR,YEL]AMEND FILE"
8002 PRINT:PRINT:INPUT "FILE NUMBER TO BE AMENDED"
;F
8003 IFF<0ORF>50THEN8002
8004 PRINT:PRINT"HIT RETURN TO LEAVE A FIELD UNALT
ERED."
8006 PRINT:PRINT:PRINT"NAME ";:CM$=F1$(F):PRINTCM$
;:GOSUB60004:F1$(F)=CM$
8008 PRINT"ADDRESS 1 ";:CM$=F2$(F):PRINTCM$;:GOSUB
60004:F2$(F)=CM$
8010 PRINT"ADDRESS 2 ";:CM$=F3$(F):PRINTCM$;:GOSUB
60004:F3$(F)=CM$
8012 PRINT"ADDRESS 3 ";:CM$=F4$(F):PRINTCM$;:GOSUB
60004:F4$(F)=CM$
8014 PRINT"ADDRESS 4 ";:CM$=F5$(F):PRINTCM$;:GOSUB
60004:F5$(F)=CM$
8016 PRINT"POSTCODE ";:CM$=F6$(F):PRINTCM$;:GOSUB6
0004:F6$(F)=CM$
8018 PRINT "TELEPHONE NUMBER ";:CM$=F7$(F):PRINTCM
$;:GOSUB60004:F7$(F)=CM$
8020 PRINT "DATE OF BIRTH ";:CM$=F8$(F):PRINTCM$;:
GOSUB60004:F8$(F)=CM$
8022 PRINT:PRINT:PRINT"RECORD AMENDED."
8024 FORI=1TO2000:NEXT
8026 GOTO5000
```

Explanation

Line 8000 - a simple message to let the user know what he's let himself in for.

Line 8002 - get the file number to be amended.

Line 8003 - and make sure it falls within the legal range.

Line 8004 - inform the user that hitting RETURN will leave a field as it was.

Line 8006 - print up the field to be altered, and put the field description (F1$(F))into the input string CM$. Print CM$ so that the user knows what he's changing, and go to the routine starting at line 60004 this time, so that the input string doesn't get set back to be a null one again. On returning from the routine, define F1$(F) to be whatever the input string now contains.

Lines 8008-8020 - as above, for the other seven fields of this file.

Line 8022 - tell the user the good (or bad?) news.

Line 8024 - set up a loop to give time to read the message.

Line 8026 - off to the menu again.

Displaying all the files

This routine is used to allow the user to flick through every file in turn, until he's either gone through all 50 of them, or gets bored and presses the 'Q' key to exit from the routine.

This is useful when scanning for some information that you know is in there somewhere, but for the life of you you can't remember where.

```
8500 PRINT"[CLR,YEL]DISPLAY ALL FILES"
8502 PRINT:PRINT:FORI=1TO50
8503 PRINT"FILE NUMBER ";I:PRINT
8504 PRINT "NAME             ";F1$(I)
8506 PRINT "ADDRESS 1        ";F2$(I)
8508 PRINT "ADDRESS 2        ";F3$(I)
8510 PRINT "ADDRESS 3        ";F4$(I)
8512 PRINT "ADDRESS 4        ";F5$(I)
8514 PRINT "POSTCODE         ";F6$(I)
8516 PRINT "TELEPHONE NUMBER ";F7$(I)
8518 PRINT "DATE OF BIRTH    ";F8$(I)
8520 PRINT:PRINT:PRINT"PRESS 'C' FOR NEXT RECORD
8522 PRINT"OR 'Q' TO RETURN TO MENU
8524 GETNR$:IFNR$="C"THENPRINT"[CLR]DISPLAY ALL FI
LES":PRINT:GOTO8530
8526 IFNR$="Q"THEN5000
8528 GOTO8524
8530 NEXTI
8532 NR$="Q":GOTO8526
```

Explanation

Line 8500 - print up a straightforward message.

Line 8502 - set up a loop that can be performed 50 times.

Line 8503 - tell the user what file number he's currently looking at.

Lines 8504-8518 - display all the fields for that file.

Lines 8520-8522 - instructions for proceeding.

Line 8524 - check for a key press, and if he presses 'C' then it's off to the next file by going to line 8530 and taking the next step through the loop.

Line 8526 - the user wants to quit, so revert back to the menu.

Line 8528 - nothing's been pressed, so wait until it is.

Line 8530 - next step through loop.

Line 8532 - the loop's finished, so fool the machine into thinking that a 'Q' has been pressed and go to line 8526 to finish everything off.

Searching through a file

This allows the user to search through any field, for any item that may be contained within that field.

For instance, a search on field 5 (usually the county in the person's address) for SHIRE, would pick out LANCASHIRE, BERKSHIRE, HAMPSHIRE, and so on. A search on field 1 for MI would pick out MIKE, MICHAEL, EMILY, etc. A powerful, and reasonably fast, routine.

```
9000 PRINT"[CLR,YEL]SEARCH FILE":PRINT:PRINT
9002 PRINT"WHAT FIELD DO YOU WANT TO SEARCH ON?":P
RINT:PRINT
9004 PRINT"NAME             (FIELD 1)
9006 PRINT"ADDRESS 1        (FIELD 2)
9008 PRINT"ADDRESS 2        (FIELD 3)
9010 PRINT"ADDRESS 3        (FIELD 4)
9012 PRINT"ADDRESS 4        (FIELD 5)
9014 PRINT"POSTCODE         (FIELD 6)
9016 PRINT"TELEPHONE NUMBER (FIELD 7)
9018 PRINT"DATE OF BIRTH    (FIELD 8)
9022 PRINT:PRINT:PRINT"PRESS APPROPRIATE NUMERIC K
EY"
9024 GETFS$:IFFS$=""THEN9024
9026 FS=VAL(FS$):IFFS=0ORFS>8THEN9024
9028 PRINT:PRINT:PRINT"FIELD NUMBER ";FS
9030 FORI=1TO2000:NEXT
9032 PRINT"[CLR]SEARCH FILE":PRINT:PRINT
9034 IFFS=8THEN9080
9040 PRINT"TEXT TO SEARCH FOR ? ";:GOSUB60000:TX$=
CM$
9042 ONFSGOTO9050,9053,9056,9059,9062,9065,9068,90
71
9050 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F1$(I))
9051 IFTX$=MID$(F1$(I),J,TX)THEN9112
9052 NEXTJ,I:GOTO9142
9053 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F2$(I))
9054 IFTX$=MID$(F2$(I),J,TX)THEN9112
9055 NEXTJ,I:GOTO9142
9056 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F3$(I))
9057 IFTX$=MID$(F3$(I),J,TX)THEN9112
9058 NEXTJ,I:GOTO9142
9059 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F4$(I))
9060 IFTX$=MID$(F4$(I),J,TX)THEN9112
9061 NEXTJ,I:GOTO9142
9062 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F5$(I))
9063 IFTX$=MID$(F5$(I),J,TX)THEN9112
9064 NEXTJ,I:GOTO9142
```

```
9065 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F6$(I))
9066 IFTX$=MID$(F6$(I),J,TX)THEN9112
9067 NEXTJ,I:GOTO9142
9068 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F7$(I))
9069 IFTX$=MID$(F7$(I),J,TX)THEN9112
9070 NEXTJ,I:GOTO9142
9071 TX=ZL:FORI=1TO50:FORJ=1TOLEN(F8$(I))
9072 IFTX$=MID$(F8$(I),J,TX)THEN9112
9073 NEXTJ,I:GOTO9142
9080 PRINT"SEARCH ON DAY(D), MONTH(M) OR YEAR(Y)?"
9082 GETSR$:IFSR$="D"THENL=1:GOTO9100
9084 IFSR$="M"THENL=4:GOTO9100
9086 IFSR$="Y"THENL=7:GOTO9100
9088 GOTO9082
9100 PRINT:PRINT:PRINT"INPUT NUMBER TO SEARCH FOR
";:GOSUB60000:IFZL>2THEN9100
9104 NS=VAL(CM$)
9106 FORI=1TO50
9108 IFNS=VAL(MID$(F8$(I),L,2))THEN9112
9110 NEXTI:GOTO9142
9112 PRINT "[CLR]RECORD NUMBER ";I:PRINT:PRINT
9114 PRINT "NAME                ";F1$(I)
9116 PRINT "ADDRESS 1           ";F2$(I)
9118 PRINT "ADDRESS 2           ";F3$(I)
9120 PRINT "ADDRESS 3           ";F4$(I)
9122 PRINT "ADDRESS 4           ";F5$(I)
9124 PRINT "POSTCODE            ";F6$(I)
9126 PRINT "TELEPHONE NUMBER    ";F7$(I)
9128 PRINT "DATE OF BIRTH       ";F8$(I)
9130 PRINT:PRINT"PRESS 'C' FOR NEXT RECORD
9132 PRINT"OR 'Q' TO RETURN TO MENU
9134 GETNR$:IFNR$="C"THENPRINT"[CLR]SEARCH FILES":
PRINT:GOTO9110
9136 IFNR$="Q"THEN5000
9138 GOTO9134
9142 PRINT:PRINT:PRINT"SEARCH CONCLUDED."
9144 FORI=1TO2000:NEXT
9146 GOTO5000
```

Explanation

Line 9000 - message about what's going on.

Line 9002 - message to choose field to search through.

Lines 9004-9022 - which keys to press for what, and telling the user to press one of them.

Lines 9024-9026 - get a key press, and if it isn't one of the numbers 1 to 8 then go back and wait until it is.

Line 9028 - inform the user which field he's chosen to search through.

Line 9030 - and give him time to read it.

Lines 9031-9032 - clear screen and print message.

Line 9034 - if he's searching for a date (e.g. everyone whose birthday falls in August), then go off to line 9080, since this routine is handled differently from the rest.

Line 9040 - get the text to search on using the input routine at 60000.

Line 9042 - depending on the field to be searched, jump to the correct part of the program.

Line 9050 - set TX to equal the length of the string that we're looking for. Set up a loop to go through all 50 files, and set a loop to check through the entire length of the field.

Line 9051 - if a match is found for the search string anywhere in the Ith field then go to line 9112 to print everything out.

Line 9052 - nothing's been found, so continue the search. When it's all over, trot off to line 9142.

Lines 9053-9073 - ditto for all the other fields.

Line 9080 - ask the user if he wants to search on a day, a month or a year.

Lines 9082-9088 - get an input and set the variable L accordingly. L indicates at which point in the date string we're going to start looking.

Line 9100 - input the number to look for by using the routine at line 60000 onwards. If the user attempts to search for a string of more than two characters, forget it, and go back again to input a number.

Line 9104 - set NS to equal the VALue of the number.

Line 9106 - set up a loop to go through all 50 fields.

Line 9108 - if a match is found, then go to line 9112 to print everything up.

Line 9110 - carry on through all the fields, then trot off to line 9142.

Lines 9111-9128 - display the field where the match has been found.

Lines 9130-9134 - see if the user wants to look for another string, or wants to return to the menu. If he decides to quit, it's back to the menu at line 5000.

Lines 9142-9146 - end of search and back to the menu.

Sorting through the files

This set of routines allows the user to sort the files into order depending on the contents of any of the first seven fields. The sort is reasonably quick on a low number of fields, but if you've got a full file you might as well go and make a cup of tea and settle down in front of the television.

```
9500 PRINT"[CLR,YEL]FILE SORT":PRINT:PRINT
9502 PRINT"WHAT FIELD DO YOU WANT TO SORT ON?":PRI
NT:PRINT
9504 PRINT"NAME                (FIELD 1)
9506 PRINT"ADDRESS 1           (FIELD 2)
9508 PRINT"ADDRESS 2           (FIELD 3)
9510 PRINT"ADDRESS 3           (FIELD 4)
9512 PRINT"ADDRESS 4           (FIELD 5)
9514 PRINT"POSTCODE            (FIELD 6)
9516 PRINT"TELEPHONE NUMBER (FIELD 7)
9522 PRINT:PRINT:PRINT"PRESS APPROPRIATE NUMERIC K
EY"
9524 GETFF$:IFFF$=""THEN9524
9526 FF=VAL(FF$):IFFF=0ORFF>7THEN9524
9528 PRINT:PRINT:PRINT"FIELD NUMBER ";FF
9530 FORI=1TO2000:NEXT
9532 PRINT"[CLR]FILE SORT":PRINT:PRINT
9533 FORJ=1TO48:PRINT"[HOME,10CR]"J
9534 ONFFGOTO9550,9560,9570,9580,9590,9600,9610
9550 FORI=1TO49:IFLEFT$(F1$(I+1),1)=" "THEN9641
9552 IFF1$(I)>F1$(I+1)THEN9632
9554 NEXTI
9556 GOTO 9641
9560 FORI=1TO49:IFLEFT$(F2$(I+1),1)=" "THEN9641
9562 IFF2$(I)>F2$(I+1)THEN9632
9564 NEXTI
9566 GOTO 9641
9570 FORI=1TO49:IFLEFT$(F3$(I+1),1)=" "THEN9641
9572 IFF3$(I)>F3$(I+1)THEN9632
9574 NEXTI
9576 GOTO 9641
9580 FORI=1TO49:IFLEFT$(F4$(I+1),1)=" "THEN9641
9582 IFF4$(I)>F4$(I+1)THEN9632
9584 NEXTI
9586 GOTO 9641
9590 FORI=1TO49:IFLEFT$(F5$(I+1),1)=" "THEN9641
9592 IFF5$(I)>F5$(I+1)THEN9632
9594 NEXTI
9596 GOTO 9641
9600 FORI=1TO49:IFLEFT$(F6$(I+1),1)=" "THEN9641
9602 IFF6$(I)>F6$(I+1)THEN9632
9604 NEXTI
```

```
9606 GOTO 9641
9610 FORI=1TO49:IFLEFT$(F7$(I+1),1)=" "THEN9641
9612 IFF7$(I)>F7$(I+1)THEN9632
9614 NEXTI
9616 GOTO 9641
9632 S$=F1$(I):F1$(I)=F1$(I+1):F1$(I+1)=S$
9633 S$=F2$(I):F2$(I)=F2$(I+1):F2$(I+1)=S$:S$=F3$(
I):F3$(I)=F3$(I+1):F3$(I+1)=S$
9634 S$=F4$(I):F4$(I)=F4$(I+1):F4$(I+1)=S$:S$=F5$(
I):F5$(I)=F5$(I+1):F5$(I+1)=S$
9635 S$=F6$(I):F6$(I)=F6$(I+1):F6$(I+1)=S$:S$=F7$(
I):F7$(I)=F7$(I+1):F7$(I+1)=S$
9636 S$=F8$(I):F8$(I)=F8$(I+1):F8$(I+1)=S$:GOTO955
4
9641 NEXTJ
9642 PRINT:PRINT:PRINT"SORT CONCLUDED.":FORI=1TO20
00:NEXT
9644 GOTO5000
```

Explanation

Line 9500 - print up a message.

Lines 9504-9522 - display list of options and get the user to choose one.

Lines 9524-9530 - wait for a key to be pressed, and check that it fits into our chosen categories. If it doesn't, loop back until the user presses something that does, and if it does then tell him what field he's going to sort on, give him time to read the message, and zoom on to the next part of the program.

Lines 9531-9532 - just for clarity.

Line 9533 - start of the grand loop to ripple through the sort for each field. If an empty field is found then that's it for that field, so go to line 9641 to take the next step in the loop.

Line 9534 - go to the correct part of the program, dependent on which field we're looking through.

Line 9550 - go through each field in turn, and if a null string is found then go to line 9641 to take the next step in the grand J loop.

Line 9552 - compare the Ith field with its neighbour, and if a change has to be made go to line 9632 to swop everything over.

Line 9554 - next step in the I loop.

Line 9556 - and off for the next step in the J loop.

Lines 9560-9616 - as above for the other six fields.

Lines 9632-9636 - swop everything over. We may only be sorting on one field, but all the other records have to be altered as well!

Line 9641 - next step in the J loop.

Line 9642 - end of sort, so print a message and give the user time to read it.

Line 9644 : back to the menu at line 5000.

The QUIT routine

This routine simply switches everything off, but does allow the user the chance to return to the menu if he has a change of heart e.g. if he hasn't yet saved all the data to disk or tape.

```
10000 PRINT"[CLR,YEL]ARE YOU SURE (Y OR N)?"
10005 GETSURE$:IFSURE$="Y"THENPRINT"[CLR]":END
10010 IFSURE$="N"THEN5000
10015 GOTO10005
```

Explanation

Line 10000 - check that the user is convinced.

Line 10005 - he is, so end the program.

Line 10010 - he's not, so back to the menu again.

Line 10015 - nothing suitable has been pressed yet, so back to line 10005.

Conclusion

This program is certainly not the world's most amazing database, although it certainly works and could easily be amended if necessary to file items other than names, addresses, birthdays and telephone numbers. Stock control is one application that springs to mind.

It serves its purpose in that it shows how what at first sight may seem a relatively complicated program can be simply built up in a series of stages, or modules. The prospect is not as daunting as you might at first think. All you've got to do is amend it!

Now let's take a look at adventure programs.

11

Getting Adventurous

You might be forgiven for thinking that writing an adventure game on the C16 is an impossible task. Just 12K to play with (no, we're not going to use high-resolution graphics!), surely not enough to create a worthwhile game? Well, you'd be wrong, as the listings in this chapter testify. Over 50 locations to explore, a reasonable vocabulary, and a collection of tough little puzzles to solve. It may not be Valhalla or The Hobbit, but it certainly stands up to some of Scott Adams' early work. For those of you who may not be familiar with the adventure 'genre', here is a brief explanation.

Adventures: a brief history

Adventure games have been played on computers for many years, and are one of the most popular of all types of computer games, if not *the* most popular.

It is sometimes difficult to describe exactly what an adventure game consists of. You're in a magical world of the writer's imagination, doing battle with unknown and often unseen problems that sometimes appear to defy all logical solution. You can be placed underground, underwater, in outer space, in colossal caves, or just about anywhere within the known universe, but the ultimate objectives of all the games are usually the same: to survive and collect all the treasure that is rumoured to exist in these weird and wonderful games.

I must admit that, personally, I'm fascinated by adventure games, and my interest in the games is shared by countless other people around the world, who have made this one of the most popular of computer diversions.

It is hard to explain this popularity to a non-addict. Peculiar looks and pitying stares are the usual response when it is revealed that you spend hours at the keyboard, glued to happenings in an imaginary world.

On the other hand, joining one of the many adventure user groups will place you among many like-minded people who fully understand the frustration at trying to solve a particular problem. 'What do you do with the platinum pyramid?' no longer evokes a 'What on earth are you on about now' attitude, instead you're more likely to get 101 hints and tips on solving the problem of the platinum pyramid.

Adventure enthusiasts even have their own Agony Aunt now in Tony Bridges, who writes a regular weekly column for the microcomputer magazine *Popular Computing Weekly.* Every week he'll take a look at an aspect of adventure playing, or a particular problem in one of the more popular games, and you're welcome to contact him over any problems that you might be experiencing in your own adventure game.

How it all began

Although most adventure programs these days seem to be written in Basic, which is the style of writing that we'll be showing you in this chapter, or machine code, the very first one was written in Fortran, not a language known for its string handling capabilities. Which language you choose is very much up to you, bearing in mind the restrictions of the computer in front of you.

Basic is usually chosen because it's easier than anything else, most Basics have a good set of commands for manipulating strings, and there is no great requirement for speed in this type of game. The essence of these games should always be that you have to think, not act in the frantic fashion of a good arcade game, and because of that we don't have to program everything to happen at lightning speed.

Some adventure games are written in machine code - Zork is a classic example - but the writing of a game like that is beyond our present aims. It is a vast program, usually supplied on three different disks, such is its size.

In Zork, speed is required because of the many and varied ways it can accept the inputting of information from you, the player. Most adventures are restricted to the TAKE STAFF style of commands: one verb and one object, but Zork goes beyond that to the level where you can say something like BURN ALL THE BOOKS EXCEPT THE BLACK ONE, and other complicated instructions.

The first adventure game, often known simply as Adventure, used the simple GO NORTH style of instructions: for that game, and for just about everything that's appeared since, credit has to go to Willie Crowther and Don Woods, who wrote the program on a DEC (Digital Equipment Corporation) PDP-10, in, as we've seen, Fortran. The program required about 300K of computer memory to play it: a great deal more than you get on the C16!

Abridged versions have appeared since then for most of the popular home computers, and it was the work of Jim Butterfield that led to the version now available for most of the Commodore range of computers.

Since then, a version has appeared for the IBM Personal Computer, but for some reason it is being marketed commercially. Odd, since it is available free of charge from most user groups!

If you have never played this Adventure, I would strongly suggest that you do so. Not only is it one of the best adventure games ever written, it is also the origin of every other adventure game. Without it, people like Scott Adams and Greg Hassell would probably never have written their own series of (very good) adventure games.

Adventure is sometimes called the Colossal Cave Adventure, for the opening scenario goes like this:

'Somewhere nearby is colossal cave, where others have found fortunes in treasure, though it is rumoured that some who enter are never seen again. Magic is said to work in the cave. I will be your eyes and hands. Direct me with commands of 1 or 2 words. I should warn you that I only look at the first five letters of each word, so you'll have to enter "northeast" as "ne" to distinguish it from north.'

All of this was developed on a mainframe computer with 300K of memory. So how did they get to appear on the microcomputers that we know today?

The transition to microcomputers

The first person to think of putting an adventure on to a small microcomputer was Scott Adams (or at least he was the first person who succeeded), an American who is commonly acknowledged to be the father of adventure games on small computers.

His story makes interesting reading, and you can find it in the December 1980 edition of the American magazine BYTE, in which there was a special feature on adventure games, and Scott Adams related the story of how it all began.

For the benefit of those who haven't got access to the magazine, here's a brief synopsis:

Scott Adams' first game was written on a Radio Shack TRS-80 level II computer, and came about after he'd already written a few other, non-adventure, games for it.

At the time he was working as a systems programmer for Stromberg Carlson, and he'd been introduced to the original Adventure by a friend. After apparently playing the game for ten days he managed to solve the whole thing, having been totally addicted from that opening scenario given earlier.

However, he realised that not everyone could afford a DEC PDP-10! So, the quest was on to produce a reasonable adventure on a much smaller computer: in his case the TRS-80.

The idea came to him of producing an adventure interpreter. This would allow him to write many different adventures, but at the same time cram an awful lot of information into a very small area of memory.

The program at the end of this chapter works along similar lines, in that routines exist within it to move from room to room, store the room descriptions, handle the input of data, and so on, and these routines are common to every listing given. This makes it possible to create adventures with a minimum amount of work from the writer, but at the same time they can be different enough to keep people occupied trying to solve them for many, many hours.

Possibly the most difficult part of writing an adventure, once the actual program structure has been grasped and understood, is getting the original idea in the first place, and working it through as a strong idea that doesn't rely on the impossible happening before the adventure can be solved.

The idea for Scott Adams' first adventure, generally reckoned to be his best, was not particularly brilliant, in that one was doing the usual treasure seeking and problem solving. Nevertheless, it did fit into 16K as opposed to 300K!

After six months of testing his adventure, and of course the interpreter that was driving it all, this first program (called Adventureland) was released through The Software Exchange of Milford, New Hampshire, and Creative Computing Software. Thus, as he says in his own article, the Scott Adams series of adventures was born.

Apparently it almost died there and then, since his wife was taking great exception to him spending six months locked in a room writing programs! However, all was solved when she decided to write an adventure, and came up with the idea for Pirate Adventure, the second best adventure program he's ever marketed.

In this one the idea is different, in that you have to do slightly more than merely collect treasures and solve problems. You have to build a pirate's ship, and not many people start off with the knowledge to do that.

And so the transition to microcomputers was complete. It was possible to write an adventure with only a minimal amount of memory, and the market suddenly begain to explode into the situation we have today, where the number of adventure programs (and programmers) is legion.

Why they are so successful

It is easy to analyse the success of, say, arcade games. The sound effects, the stunning graphics, are obviously pleasing to the human ear and eye, and our society seems to be depressingly heading into a more violent era. Thus the chance to annihilate a few more aliens for a mere 20 pence is not one to be missed.

But adventure games have none of this. There is usually no graphical display, although the arguments about that have raged far and wide in recent times. Generally, there is no sound being generated by the computer either, although again there are exceptions.

Finally, there is no 'shoot-em-up and zap-em-down' approach to adventures. They are games for the thinker, rather than the person of action.

Perhaps this is part of the secret of their great success. To solve a good adventure like the original Crowther and Woods game requires a lot of logical throught, to say nothing of a lot of time. The first people to start playing the game were computer programmers themselves, and one survey in the States showed that, when an implementation of Adventure appeared on the office computer, an estimated two weeks work would be lost due to staff playing the game in their free, and not so free, time.

Obviously people tried to put a stop to this, and started restricting access to the game, but it was generally reckoned that whatever a company tried to do, nothing would stop its employees from playing the game. Better to let them have their way for a couple of weeks, and see them emerge contented at having attained the goal of master adventurer.

The same is true for most people who start playing an adventure game. Once you've started, it is virtually impossible to rest until you've completely solved the puzzle.

How can I get past the troll without losing treasure? How do I open the clam? How do I open the treasure chest in the pirate's maze? All these questions have to be solved before attaining the magical status of master adventurer. Sometimes you're setting yourself an impossible task, but that won't prevent people from taking hours trying to solve it, until they give up in disgust.

It becomes a question of pride: 'I am not going to be beaten by a stupid computer!' is the usual response. Also, pride comes in when you hear of someone else talking about a room, or a particular problem, that you haven't encountered. The desire to find that room, or solve that problem, drives many people back to the keyboard again.

And, strangely enough, you will very rarely get a direct answer when you ask someone how to solve a certain problem. You'll usually get a cryptic hint, but nothing more. So you're back to your own logic again, and few people will admit to not being able to solve something.

Finally, adventure games usually have a sense of fun. Take the classic Adventure. The version by Jim Butterfield produces some lovely responses at times. Like this:

```
FEED BIRD

THE BIRD IS NOT HUNGRY, HE IS MERELY PINING FOR THE
FJORDS!
```

Shades of John Cleese and ex-parrots. If you try typing in the inevitable rude statements, requesting the snake to do the anatomically impossible, again a variety of replies can be generated.

So a combination of problem solving, pride and fun have all contributed to making adventure games required playing for most people.

So that's how they all started. The big question is, how do YOU start?

Getting started

Possibly the most difficult part of writing an adventure game is outlining the story. In effect it will have to be a miniature novel, involving (relatively) realistic concepts, although an ingredient of most adventure worlds is always that little touch of magic that sets them aside from the real world.

The example in this chapter, Castle Adventure, is set (reasonably enough) inside a castle, where all manner of unusual things await the intrepid adventurer.

The plot, just as in a good novel, must flow smoothly from one stage to the next, with no totally unexpected, inexplicable events. One adventure I know suddenly has a sword that you've been happily carrying along turn into a snake in your hands, which then bites you and kills you off.

This is totally inexcusable, and shouldn't find a home in any real adventure. The impossible happens quite often in these games, but at least there should be a warning that it's going to happen, and it should not be sufficient to kill off the character.

So if we're going to have magic, let's keep it on a fairly reasonable level, and stick to iron staffs being waved and causing a bridge to appear over the chasm.

Events that kill off the hero, like crossing a rickety bridge with a heavy bear in tow, should generally be as expected as possible, and only be the fault of the adventurer. In real life, would you expect a rickety bridge to support the weight of a heavy, lumbering bear?

So anything that happens in the game must have a remote basis in reality, and the inexplicable shouldn't really happen without at least being safe to the player.

The plot

As we've said, this is possibly the most difficult part of all. Many adventures have now been written, and coming up with an original scenario each time is getting gradually harder and harder.One tried and trusted idea is by dipping into a few books such as *Lord of the Rings,* in which there are a multitude of possible plots which could be turned into very reasonable games. However, as in all implementations of this sort one has to be very careful about the laws of copyright, as you may have seen with the Hitch Hiker's Guide to the Galaxy game, so you'll probably have to change a lot of names to protect the innocent, i.e. you!

The traditional thud and blunder adventure, steeped in Gothic names and ancient runes, has been done by many authors, although obviously the scope here is vast for doing variations on a theme.

One possible answer might be to read a few science fiction novels (bearing in mind the author's copyright), such as the works of Michael Moorcock, and obtain a few ideas from there.

To the beginner though it must seem that just about every possible idea has been tried before, including exploring ancient tombs and crypts, jungle adventures that pit you against various natives and native problems, cowboy adventures, outer space adventures, underwater adventures, and the like, and that it would be impossible to come up with a new and original plot-line for your story.

But remember that there have been many more novels written than there have been computer adventures, and people still keep managing to come up with original themes for those, so the ideas are always there: it's just a question of thinking them up.

Visitors from outer space, detective adventures, psychological adventures, biblical adventures, are all relatively new areas, and perhaps combining one of these new ideas with a character playing role could pave the way for a whole new set of computer games.

The work is up to you though, and your plot, whatever it consists of, must ring true throughout, and keep the player of the game constantly entertained, forever pitting him against new challenges, new tasks, and keeping the interest by finding out just that little bit extra with each game.

The hazards

One of the most important parts of any adventure game will be the constant search for new problems to set the player, new tasks that have to be accomplished before you can proceed further, and making those hazards solvable, but (preferably) as difficult as possible.

The number of problems set will always vary from game to game, and should to some extent depend on the number of rooms in the game. Perhaps on a 1 to 6 ratio, with a new task to be solved every half dozen rooms or so?

But no matter how you spread things out, you'll need a good solid set of problems which give the player plenty to chew over, along with a reasonable set of constant events that can also give cause for worry.

Whatever the kind of problem, be it in a set place or occurring at random, one golden rule of programming this type of game remains the same: if the player solves the problem, make sure the program checks for this and adjusts its variables accordingly. There is nothing worse for a player than, having spent hours achieving one goal, to throw away the relevant object which has enabled him to do this (or perhaps have it taken away by the program once it has fulfilled its duty), and then to see a bug in the program causing the problem to re-appear!

In other words, don't make your adventures impossible, which is always a problem when you're manipulating a lot of objects. Just placing one of them in the wrong room could cause the program to become insoluble: a cardinal sin.

One of the more common constant problems is that of a torch. If you're deep underground it's fairly safe to assume that you won't be able to see very much, and so a torch becomes vital. To light the torch you will also need some matches, and these must also be hidden in the game somewhere. Finding the torch and lighting it is usually no problem, but keeping it lit often is. A sudden gust of wind perhaps, or a swim through some water would do the trick. If you go through water, you would also get the matches wet, so how do you light the torch again?

A torch carries with it another problem. There is usually a limit on how much you can carry at a time, and certain objects will always have to be with you, like torches, axes, and so on, and so the problem becomes what to carry at the same time.

Dropping things often breaks them (e.g. bottles), so you'll have to make your adventure as devious as possible, to ensure the maximum amount of thinking for the person who will ultimately play it.

All these problems will have to take place in some kind of land or other, so let's draw a map.

Drawing the map

We'll assume you have worked out some rough kind of plot line, and you want to draw a map to see what it all looks like.

It's tempting at first to take the largest bit of paper you can find, and draw the map on that, but this is not always a good idea. My own original maps tend to be done on extremely small sheets of paper that then get stuck together to form one large map of the whole game. Using just one large sheet to start with usually produces a considerable distortion of scale by the time the map's finished.

By using several smaller sheets, each one devoted to a particular part of the adventure, it is possible to see every part of it clearly. This sheet is a maze, this sheet is a collection of interlocking tunnels, and so on. An adventure game is not that easy to write, so you'll want to keep life as simple as possible for yourself.

For now, we'll assume that you've drawn your map, however rough it may be, you've got some idea of the general plot for the whole story, and you know (again roughly) where all the hazards are going to present themselves.

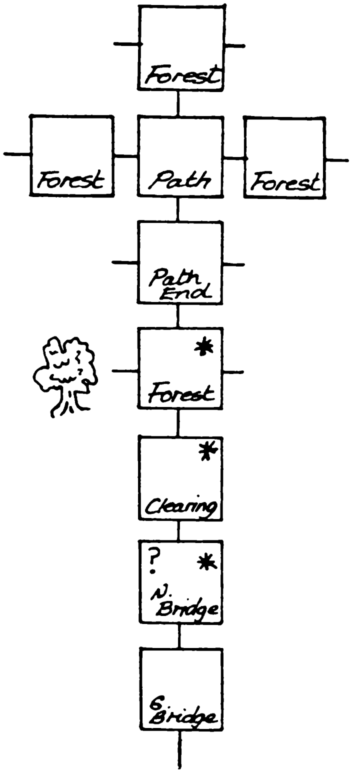
Moving from paper to computer

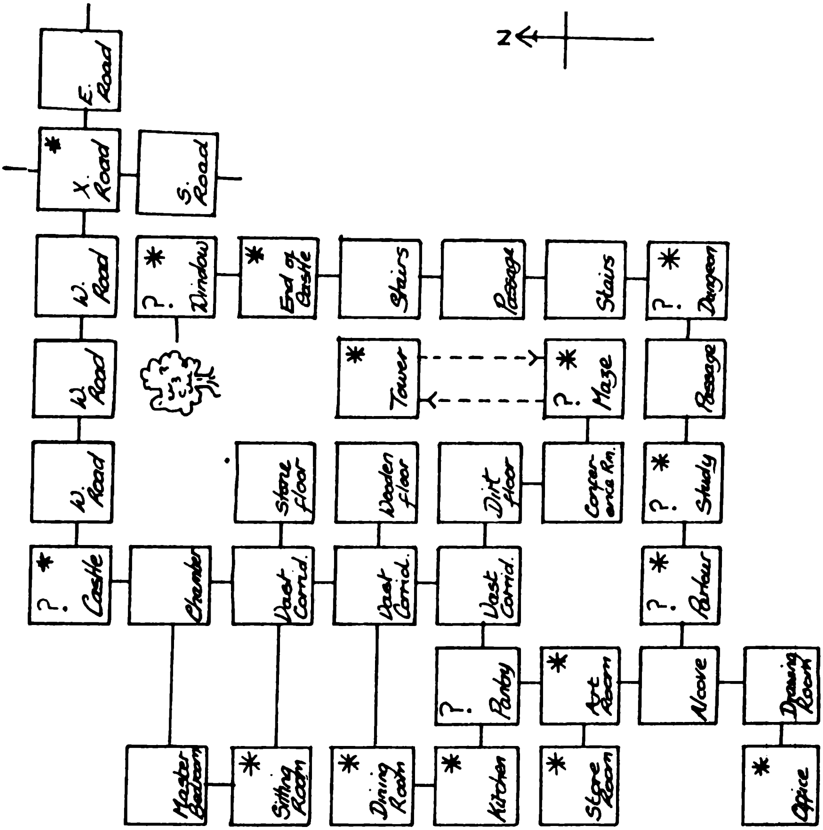
One of the first steps is to draw a much more sensible looking map, as shown on pp. 174-5.

Castlemaze.

?— problem to solve

✱— useful objects/treasures




N
E. Road
X. Road
S. Road
W. Road
W. Road
W. Road
Castle
Window
Chamber
End of Castle
Stone floor
Vast Corrid.
Stairs
Tower
Wooden floor
Vast Corrid.
Passage
Dirt floor
Vast Corrid.
Stairs
Maze
Conference Rm.
Dungeon
Passage
Study
Parlour
Master Bedroom
Sitting Room
Dining Room
Kitchen
Pantry
Art Room
Store Room
Alcove
Drawing Room
Office

This can be labelled with all the hazards, and we'll then have to come up with our list of objects to be used. So the next step is to look at the list of hazards as you originally drew them up, and decide what the solution would be to each hazard, bearing in mind that you can only move on to the next part of the adventure after you've solved the problem. In other words, don't put the solution further into the game than the problem!

A list of solutions will give you a healthy list of objects, and these will then form the basis of the list that we'll type into our program later.

With our program set up as it is, although obviously you could modify it if you want to, the routine that checks your data entry only looks at the first three letters of each word. Thus if you had a TRACK and a TRAM in your adventure the program listing would interpret them to be the same object, and you would get some very strange displays being shown up on the screen! So, be careful.

This list of objects will have to be extended beyond a simple list of those generated by the problems and their solutions. We haven't mentioned lamps, or anything like that, so you'll have to have words for LAMP. What happens if you drop a bottle? If you're going to have it break, you'll also need to have an object something like A PILE OF BROKEN GLASS.

These and other problems will all have to be thought of before we start typing anything in, and inevitably we'll have to add objects to our list as we go along developing the program, but in Basic that is no great difficulty.

And on to verbs

As well as our list of nouns, or objects, the other great list in any adventure games, and the list that to a large extent dictates how good a game it is, is the list of verbs.

Some adventures have many more verbs than others, and as we've seen Zork can handle around a hundred of them, but Castle Adventure confines itself to a mere 30, although this could easily have been extended by another dozen or so.

A lot of verbs is a good idea, and your original starting list should always be the first ten verbs listed in the program. These are all standard verbs, like GET, LOOK, HELP, GO, and so on, that should occur in every adventure, and the routines for handling these same verbs from game to game do not vary very much. Obviously they will change a little as the needs of the different games change, but it's a healthy and encouraging start when you see your initial list of forty (or whatever) verbs almost immediately whittled down to thirty.

The rest of the verbs are very much up to you, but again they will to a large extent be dictated by the problems that have to be solved.

There is no point in having a can of fly spray to kill the giant fly if the verb SPRAY is not included in the vocabulary. KILL is too woolly a word, and could produce the wrong response if the spray was not being held.

Additional verbs should also be there, just to encourage diversification of response from the computer, and keep the player's interest. A good idea is to give bizarre ideas on the part of the player equally bizarre responses from the computer. It all adds to the humour of playing this type of game.

Amazing

Most adventures have a maze of one sort or another, and having got our verbs and nouns, it makes sense to put a maze somewhere.

Hard mazes are very easy to construct, simply by giving every one of (say) six rooms the same description, so the player always thinks he's in the same room, and if he makes a move in any one of the three directions you don't want him to move in, why, send him back to the start!

Some general rules

Although we've been looking at specifics for the last few pages, for the next few we'll turn our attention to some general rules when writing these games, and concentrate on five of the most important parts of every adventure game:

1. Movement of characters

2. Responses to inputs

3. Screen displays

4. Picking things up and putting them down

5. Problem solving

Movement

As your character moves around his wonderful adventure world, there are obviously certain rooms he will and will not be able to go into straight away. Some rooms will be purely east-west or north-south corridors, in which case it would be rather silly to tell your character that he could move north/south and east/west respectively.

You may or may not display which directions he can move in at all. Certainly the original Adventure didn't, and you were left to your own devices to find every possible direction out of a room, hence the need to draw a map. That game was additionally complicated by having up and down as well as the four cardinal compass points, and also having north-east, south-west and so on.

In Castle Adventure we've stuck purely to the four cardinal directions, with up and down movements being handled in specific problem areas. If you don't want to display the possible directions it will certainly prompt the player into drawing a map, and it might well annoy him considerably to be told over and over again 'YOU CAN'T GO THAT WAY', although interest could be sustained by the addition of the little word YET, thus making him think 'Aha! perhaps I can go along there later'.

Personally, I'm in favour of displaying the available choice of directions, as it speeds up the playing process, but if necessary you can just resort to hints like 'A VAGUE TRACK HEADS OFF TO THE SOUTH', and the like.

It's up to you, but whatever style you pick, make sure that you stick to it throughout the game.

Screen responses

This is obviously the factor that is most important in keeping the interest and attention of the player throughout the game, and if you want to resort to sound, colour and graphics that's up to you.

In designing and writing your adventure there is an important factor to bear in mind whenever you're planning the responses to the statements typed in by the player in response to the WHAT NOW prompts, and that is that people playing adventures will never, ever type in what you want them to.

You may have a situation where a player comes to a halt in front of a gate that he can't climb over because the top of it is riddled with barbed wire (an escape from Colditz type adventure?), until he gets hold of a set of wire cutters. You have programmed all your responses to GET GATE, GET WIRE, and so on, and are waiting for the player to get the cutters and type CUT WIRE.

What if he types CUT GATE? What happens then? Or what about something typed in in sheer desperation, as people do, like EAT GATE? Does the gate get swallowed up in a display of apparent relish?Anticipating people's lines of enquiry is one of the most difficult things to allow for, and will take up an awful lot of program code that will probably never be used.

Still, even if it is used only once at least you'll have the satisfaction of knowing that someone out there will consider that the game that he's playing is an extremely robust, well thought-out adventure.

Always try to anticipate the impossible. You'll never manage all of it, of course, and will have to rely on some stock I DON'T UNDERSTAND type responses, but a few of those mixed up and one picked out at random will keep the interest from flagging.

And never forget the use of the word YET. It will keep a player trying long after the more straightforward 'YOU CAN'T OPEN THE GATE' will.

So the golden rule here must be to keep it interesting, and try to anticipate everything that the player might type in. You won't get them all, but at least you can conjure up some different responses.

Also, a large list of verbs is a great help here: even if the responses are only short and sweet, at least the player will be seeing something different on the screen.

Screen displays

To a small extent we've covered this one already, but it's worth going over some of the ground again.

The use of graphics has been deplored often enough before now to render any comment here redundant, although you might think the odd display of a sword or amulet every now and again might liven things up a little. But nothing can beat the written word.

Sound is a different question, and the arguments concerning this are almost as legion as those concerning the use of graphics. My own view is that if you're going to use sound, it must be done extremely well, as the computer is capable of a very complex series of sound outputs. If you're only going to give a little beep every now and again, it's hardly worth the effort of putting it in there in the first place, and you'll soon have people racing for the volume control and a blessed silence.

If done well, it can greatly enhance a game, as people who have played the Temple of Apshai on a Commodore 64 will know: the use of sound is very good here, and the whole atmosphere of moody, omnipresent danger is well presented. On the other hand, all their programming efforts are wasted if somebody turns the volume down. Have sound in your programs if you wish, but don't be disappointed if everyone immediately opts to play out the game in silence.

The words that are displayed on the screen are obviously dictated by the responses you've allowed for, but an overall attractive layout is desirable, usually using lower case, since most people seem to prefer that for some reason. Perhaps it's more restful on the eyes as you do battle against a giant troll!

Silly little things can so easily spoil a game in this area - if your room descriptions overlap the edge of the screen so that words are split up, or an inventory list causes some of the objects to be displaced against each other, or even if your output is riddled with spelling errors.

It doesn't take too long to check all these things, and the results are well worth the trouble. A neat adventure is more likely to be played than a badly spelt, badly laid out one.

The golden rule here? Keep it simple, but keep it tidy.

Picking things up and putting them down

Two of the most important words in the adventurer's catalogue are GET and DROP.

Obviously, in any game there will be a number of things that you can pick up, and a number that you can't, with the former probably far outweighing the latter. Nevertheless, all possible occurrences must be taken into account, and just because you know that the BARRED GATE is too heavy to carry, that won't stop virtually every player who comes along from attempting to pick it up and walk off with it.

Another annoying thing to find in any adventure program is a description that might read something like 'YOUR PROGRESS IS HALTED BY A SOLID WALL OF ROCK', and when you type in GET WALL, the only response is 'I CAN'T SEE ANY WALL HERE', or 'I DON'T KNOW WHAT A WALL IS'.

Look out for that one, for although it can be covered by a blanket response of NO!, that is not very good practice and will certainly not produce an excellent adventure game. Far better to have a response actually geared to the request like 'THE WALL CANNOT BE CARRIED', or something like that.

Some things in a game are only meant to be carried after certain actions have taken place, in which case you'll need a number of variables to flag the progress of the adventurer, and you'll also have to use the word YET to keep the level of interest there. 'YOU CAN'T CARRY IT YET', will have someone attempting to carry whatever IT is until the cows come home, even if they never can carry it.

When dropping things, a subtle level of difficulty comes into the game. Dropping bottles is usually a good one, since you can have them break on your adventurer, thus rendering them useless for the rest of the game. The original Adventure had as one of its treasures a Ming Vase, but dropping it caused it to smash into delicate little pieces, unless (of course!) you'd taken the precaution of placing a pillow underneath it.

GET and DROP are fun, and don't confuse GET with TAKE. The two words are not the same! For instance, people talk about TAKEing medicine, not GETting it!

Problem solving

The key to any adventure is how good and how complicated the problems may be in a game, but don't make it too complicated to get started, or your adventurer might give up in disgust and never play an adventure game again.

Encourage people by at least letting them get started, and then pile the problems on, preferably making the first few lean towards the easy side, and have them get harder as the game gradually progresses.

The Scott Adams games are particularly good here, as it is always possible to get somewhere at a first sitting, even if that somewhere isn't very far, and you can gradually improve your progress just about every time you play the game.

Problems usually have to be solved in a set order too, in that solving one leads you to another, which gives you a clue to an earlier hazard you were puzzling over, which in turn sets you off somewhere else, and so on.

The number of problems in a game is obviously up to the writer of the game, but too many will soon discourage people. A problem every room will become totally boring after only a short playing session, but the intervention of a few rooms between hazards will soon perk up the player, even if he does walk into another one almost immediately.

So keep up the interest, and let the player get a little further each time. And above all, don't make it an unsolvable adventure!

But enough of all this theory. Let's get typing.

Castle Adventure

The Datamaker

The listing for Castle Adventure comes in two parts, which was the only way of getting it all to fit into the C16. The first part generates all the data for the game, and creates a series of files on disk, which the second part then reads. This data contains our list of nouns and verbs, the room descriptions, and the directions in which a player can move from room to room.

```
1 NP=53:LO=35:NN=31:NV=30:DIMP%(NP,3),P$(NP),VB$(N
V),NO$(NN),OB%(LO,1),OB$(LO)
2 P$(1)="OUTSIDE AN OLD MEDIEVAL CASTLE."
4 DATA0,8,4,0,53,7,3,6,0,0,3,2
5 P$(2)="AT A CROSS ROAD.":P$(3)="ON THE GREAT EAS
T ROAD."
6 P$(4)="ON THE GREAT WEST ROAD.":DATA0,0,2,1,0,0,
2,4
7 P$(5)=P$(4):P$(6)=P$(4):DATA0,0,2,5
8 P$(7)="ON THE GREAT SOUTH ROAD.":DATA2,7,0,0
9 P$(8)="IN A SPLENDID CHAMBER 30 FEET    HIGH.":D
ATA1,11,0,10
10 P$(9)="IN A COSY SITTING ROOM.":DATA10,0,11,0
11 P$(10)="IN THE MASTER BEDROOM.":DATA0,9,8,0
12 P$="IN A VAST CORRIDOR STRETCHING OUTOF SIGHT T
O THE ":P$(11)=P$+"SOUTH."
13 DATA 8,12,14,9
14 P$(12)=P$+"NORTH AND SOUTH.":DATA11,13,15,25
15 P$(13)=P$+"NORTH.":DATA12,0,16,17
16 P$="IN A BEDROOM WITH A ":P$(14)=P$+"STONE FLOO
R.":P$(15)=P$+"WOODEN FLOOR
17 P$(16)=P$+"DIRT FLOOR.":DATA0,0,0,11,0,0,0,12,0
,33,0,13
18 P$(17)="IN A DUSTY PANTRY.
19 DATA 0,18,13,24
20 P$(18)="IN A PRIVATE ART GALLERY.":DATA17,26,0,
19
21 P$(19)="IN A MUSTY STORE ROOM WITH MUSTY COBWEB
S EVERYWHERE."
22 P$(20)="AT THE TOP OF A BIG TREE."
23 P$(21)=P$(20)
24 P$(22)=P$(20)
25 P$(23)=P$(20)
26 DATA0,0,18,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
27 P$(24)="IN THE KITCHEN.":DATA25,0,17,0,0,24,12,
0,18,28,29,0
28 P$(25)="IN THE DINING ROOM.":P$(26)="IN A SHADO
WY ALCOVE."
29 P$(27)="IN AN AUSTERE OFFICE.":DATA0,0,28,0
30 P$(28)="IN THE DRAWING ROOM.":DATA26,0,0,27,0,0
,30,26,0,0,0,29
31 P$(29)="IN THE PARLOR.":P$(30)="IN THE STUDY."
32 P$(31)="IN A DAMP STONE PASSAGE."
33 P$(32)="IN THE DUNGEON.":DATA0,0,32,30,41,0,0,3
1
34 FORI=1TONP:FORJ=0TO3:READP%(I,J):NEXTJ:NEXTI
35 P$(33)="IN AN ANCIENT CONFERENCE ROOM.":DATA16,
0,35,0
36 P$(34)="IN A TOWER WHICH OVERLOOKS A HUGE"
37 P$(34)=P$(34)+"KINGDOM DOWN A MONSTROUS MOUNTAI
N.
38 DATA40,0,0,0
39 P$(35)="IN A MAZE OF TWISTY LITTLE      PASSAG
ES ALL ALIKE."
40 FORI=36TO40:P$(I)=P$(35):NEXTI:DATA36,36,36,33,
37,35,35,35,36,36,38,36
41 DATA36,36,36,39,40,36,36,36,36,34,36,36,42,32,0
,0
42 P$(41)="ON A LONG FLIGHT OF STAIRS DOWN."
43 P$(43)="ON A LONG FLIGHT OF STAIRS UP."

44 P$(42)="IN A MILE-LONG PASSAGE WITH     STAGNA
NT WATER COVERING YOUR FEET.
45 P$(44)="AT THE END OF THE CASTLE AND CAN SEE A
FOREST OUT A SMALL WINDOW."
46 DATA43,41,0,0,44,42,0,0,0,43,0,0
47 P$(45)="IN A DENSE DARK FOREST."
48 FORI=46TO48:P$(I)=P$(45):NEXTI:DATA45,49,46,48,
45,47,46,49,50,51,46,48
49 DATA45,47,49,48
50 P$(49)="ON AN OLD PATH MADE BY HORSES."
51 P$(51)="IN THE MIDDLE OF A CLEARING WITH AN OLD
 BRIDGE LYING TO THE SOUTH.
52 P$(52)="ON THE NORTH SIDE OF THE BRIDGE."
53 P$(53)="ON THE SOUTH SIDE OF THE BRIDGE."
54 DATA 45,50,46,48,49,47,46,48,47,52,0,0,51,0,0,0
,0,2,0,0
55 DATABOW,KEY,BOO,ARR,BRO,SAP,LIQ,SWO,PAI,WHE,SIL
,PEN,PIL,SCE
56 DATA BAR,VAU,KNI,VAS,SIG,SOR,SPI,NOR,SOU,EAS,WE
S,N,S,E,W,DOO,WIN
57 DATAGO,GET,LOO,INV,SCO,DRO,HEL,CLI,DIG,QUI,SAY,
DRI,OPE,TAK,KIL,REA,SHO,ATT
58 DATACLO,LOC,UNL,GIV,USE,VAL,SWI,SHA,EXA,THR,FIN
,JUM
59 FORI=1TONN:READNO$(I):NEXTI:FORI=1TONV:READVB$(
I):NEXTI
60 DATA A LONG BOW,-1,0,A BRONZE KEY,0,0,A LEATHER
-BOUND BOOK,30,0
61 DATA A SILVER ARROW,22,10,A BROKEN ARROW,0,10,A
 GIGANTIC SAPPHIRE,0,10
62 DATAA VIAL OF AMBER LIQUID,24,0,A GOLDEN SWORD,
34,10
63 DATAA LARGE REMBRANDT PAINTING,18,20
64 DATA A WHETSTONE,19,0,A SET OF SILVERWARE,25,10
,A PLATINUM PEN,27,10
65 DATA A VELVET PILLOW,44,0,THE SORCERER'S SCEPTE
```

```
R,0,10,A GOLD BAR,0,10
66 DATA A VAULT IN THE WALL,0,0,A DEAD KNIGHT,0,0,
A MING VASE,9,10
67 DATA"A SIGN SAYING: [RVS]DIABOLICAL MAZE[OFF]",
35,0,A WICKED SORCERER,32,0
68 DATAA DEAD SPIDER,0,0
69 FORI=1TO21:READOB$(I),OB%(I,0),OB%(I,1):NEXTI
70 DATA NORTH,SOUTH,EAST,WEST
71 FORI=0TO3:READD$(I):NEXTI
72 P$(50)="AT THE END OF A PATH WITH FOREST SURROU
NDING YOU IN ALL DIRECTIONS."
73 PRINT"PRESS 'SPACE' WHEN READY."
74 GETA$:IFA$<>" "THEN74
75 OPEN2,8,2,"0:DATA,S,W"
76 X$=CHR$(13)
77 FORI=1TO53:PRINT#2,P$(I);X$;:NEXT
78 FORI=1TO53:FORJ=0TO3:PRINT#2,P%(I,J);X$;:NEXTJ,
I
79 FORI=1TONN:PRINT#2,NO$(I);X$;:NEXT:FORI=1TONV:P
RINT#2,VB$(I);X$;:NEXT
80 FORI=1TO21:PRINT#2,OB$(I);X$;OB%(I,0);X$;OB%(I,
1);X$;:NEXT
81 FORI=0TO3:PRINT#2,D$(I);X$;:NEXT
82 CLOSE2
```

Looking at the first line tells us just about everything we need to know about the program. NP is the number of places in the game, LO is the number of objects that we'll have, NN is the number of nouns and NV is the number of verbs.

The room direction data is in the order North, South, East, West. The data following the object descriptions refers to a) where it starts in the game, and b) its score value. Thus in line 64 we see that the platinum pen starts the game at location 27, and has a score value of 10, whereas the whetstone isn't worth anything, although it does have a useful purpose to fulfil.

Finally, lines 75 to 82 store all this data onto disk. If you're using a tape system, don't despair. Change line 75 to read:

75 OPEN 1,1,1,"DATA"

and change all the PRINT#2s to PRINT#1s. Alter line 82 to read CLOSE1 and there you are.

Once you've got that up and running (he said casually) we can then concentrate on the second part of the listing, which looks after all this data and allows you to play the game.

The Driver program

```
1 PRINTCHR$(14):COLOR0,8,6:COLOR4,10,4
2 GB$="A GOLD BAR FALLS OUT!"
3 D1$="YOU ARE SHRINKING!
4 DI$="YOU HAVEN'T GOT IT.":B1$="YOU HAVE NO BOW."
:B2$="YOU HAVE NO ARROW."
5 CM$=CHR$(44)
6 X=0:ZZ=1:PI=12
7 QT$=CHR$(34)
8 VT$="BEHIND THE SIGN IS A VAULT IN THE WALL. THE
 VAULT IS LOCKED.
9 REM
10 DB$="YOU MUST SUPPLY A DIRECT OBJECT."
11 REM
12 CP=49:S1$="I DON'T SEE IT HERE.":S2$="DON'T BE
RIDICULOUS.":GOTO185
13 GOTO159
14 IFCP=52ANDKN=0THEN128
15 IFOB%(2,0)=-1ANDPI=CPTHEN136
16 IFCP=29ANDSP=0THEN139
17 T=T+1:GOSUB148:IFVB$="3.1"THENP%(30,2)=31:GOTO1
3
18 IFVB=-1AND(NO>21ANDNO<30)THENVB=1
19 IFVB$="CRO"THENIF(CP=52ORCP=53)THENCP=52+ABS(CP
-53):GOTO13
20 IFVB<>30AND(VB>10ORVB=20ORVB=6)ANDNO$=""THENPRI
NTDB$:GOTO17
21 IFVB=30THEN117
22 IFVB=-1ANDNO<>0AND(NO<22ORNO>29)THENPRINT"YOU M
UST SUPPLY A VERB.":GOTO17
23 IFVB<1ANDNO=0THEN54
24 IFNO=0ANDVB>10THEN54
25 ONVBGOTO27,36,13,43,46,50,89,55,80,147,58,59,62
,36,71,72,76
26 ONVB-17GOTO71,81,81,62,50,54,90,91,100,104,50,1
08,117
27 IF(NO<22ORNO>29)ANDNO$<>""THENPRINT"I DON'T KNO
W HOW TO DO THAT.":GOTO17
28 IFNO$=""THENPRINT"WHERE?":GOTO17
29 IFNO>25THENNO=NO-4
30 NO=NO-22:IFP%(CP,NO)=0THENPRINT"THERE IS NO WAY
 TO GO IN THAT DIRECTION.":GOTO17
31 IFCP=1ANDNO=1ANDDF=0THENPRINT"THE CASTLE DOOR I
S LOCKED.":GOTO17
32 IFCP=17ANDNO=1ANDCF=0THENPRINT"THE CRACK IS TOO
 SMALL FOR YOU.":GOTO17
```

```
33 IFCP=18ANDNO=0ANDOB%(9,0)=-1THENPRINT"THE PAINT
ING IS TOO BIG FOR THE CRACK.":GOTO17
34 IFNO=0ANDOB%(20,0)=CPTHENPRINT"THE SORCERER TUR
NS YOU INTO A FROG.":GOTO135
35 CP=P%(CP,NO):GOTO13
36 IFOB%(NO,0)=-1THENPRINT"YOU'VE ALREADY GOT IT!"
:GOTO17
37 IFNO=0THENPRINT"WHAT'S A "N1$"?":GOTO17
38 IFOB%(NO,0)<>CPTHENPRINTS1$:GOTO17
39 IFNO=17ORNO=21ORNO=20ORNO=16THENPRINTS2$:GOTO17
40 IFZZ>4THENPRINT"YOU'RE CARRYING TOO MUCH.":GOTO
17
41 IFNO=19ANDPF=0THENPRINTVT$:PF=1:OB%(16,0)=CP:OB
%(19,0)=-1:ZZ=ZZ+1:GOTO17
42 PRINT"OK":ZZ=ZZ+1:OB%(NO,0)=-1:GOTO17
43 IFZZ=0THENPRINT"YOU ARE EMPTYHANDED.":GOTO17
44 PRINT"YOU ARE HOLDING:":FORI=1TOLO:IFOB%(I,0)=-
1THENPRINTOB$(I)
45 NEXTI:PRINT:GOTO17
46 GOSUB47:GOTO17
47 J=0:FORI=1TOLO:IFOB%(I,0)=1THENJ=J+OB%(I,1)
48 NEXT:PRINT"YOU HAVE SCORED"J"POINTS OUT OF 100.
":IFJ<100THENRETURN
49 PRINT "[CD,RVS]WELL DONE![OFF]":END
50 IFNO<>0ANDOB%(NO,0)<>-1THEN100
51 IFNO=0THENPRINT"I'VE NEVER HEARD OF A "N1$".":G
OTO17
52 IFNO=18ANDOB%(13,0)<>CPTHENOB$(18)="A SHATTERED
 VASE":OB%(18,1)=0
53 OB%(NO,0)=CP:PRINT"OK":ZZ=ZZ-1:GOTO17
54 PRINT"TRY RE-PHRASING THAT.":GOTO17
55 IFCP<49ANDCP>44THENCP=CP-25:GOTO13
56 IFCP<24ANDCP>19THENCP=CP+25:GOTO13
57 PRINT"THAT IS NOT POSSIBLE.":GOTO14
58 PRINT"ALRIGHT...";N1$:GOTO17
59 IFOB%(NO,0)<>-1THENPRINTDI$:GOTO17
60 IFNO<>7THENPRINTS2$:GOTO17
61 PRINTD1$:ZZ=ZZ-1:OB%(7,0)=0:CF=1:GOTO17
62 IFNO<>31ANDNO<>16ANDNO<>30THENPRINT"I DON'T KNO
W HOW TO.":GOTO17
63 IFNO=16ANDOB%(16,0)<>CPTHENPRINT"WHAT VAULT?":G
OTO17
64 IFNO=16ANDOB%(2,0)<>-1THENPRINT"YOU DON'T HAVE
THE KEY.":GOTO17
65 IFNO=16THENPRINT"THE VAULT IS OPEN":VF=1:IFOB%(
15,0)=0THENPRINTGB$:OB%(15,0)=CP
66 IFNO=16THEN17
67 IFNO=31THEN124
68 IFCP<>1THENPRINT"WHAT DOOR?":GOTO17
69 IFOB%(2,0)<>-1THENPRINT"YOU DON'T SEEM TO HAVE
THE KEY.":GOTO17
```

```
70 PRINT"THE DOOR IS OPEN.":DF=1:GOTO17
71 PRINT"HOW?":GOTO17
72 IFOB%(NO,0)<>-1THEN91
73 IFNO<>3THENPRINT"HOW DO YOU EXPECT TO READ ";OB
$(NO);"?":GOTO17
74 PRINT"IT SAYS:  [RVS]A SECRET PASSAGE LIES NEAR
BY"
75 PRINT"[RVS]IT OPENS IF YOU NUMBER PI    ":GOTO17
76 IFOB%(NO,0)<>-1ANDOB%(NO,0)<>CPTHENPRINTS1$:GOT
O17
77 IFOB%(1,0)<>-1THENPRINTB1$:GOTO17
78 IFOB%(4,0)<>-1THENPRINTB2$:GOTO17
79 ZZ=ZZ-1:OB%(4,0)=CP:GOTO17
80 PRINT"YOU NEED A TOOL.":GOTO17
81 IFNO=16ORNO=30ORNO=31THEN84
82 IFOB%(NO,0)<>-1THEN80
83 PRINT"I DON'T KNOW HOW TO.":GOTO17
84 IFNO=16ANDOB%(16,0)<>CPTHENPRINT"WHAT VAULT?":G
OTO17
85 IFNO=16THENPRINT"THE VAULT IS CLOSED AND LOCKED
.":VF=0:GOTO17
86 IFNO=31THEN121
87 IFCP<>1THENPRINT"WHAT DOOR?":GOTO17
88 PRINT"THE DOOR IS CLOSED AND LOCKED.":DF=0:GOTO
17
89 PRINT"YOU MUST BE JOKING!":GOTO17
90 PRINT"ITS VALUE IS"OB%(NO,1)"POINTS.":GOTO17
91 IFOB%(NO,0)<>-1THENPRINTDI$:GOTO17
92 IFNO<>14THEN95
93 FORI=1TO19:IFOB%(I,0)=-1THENOB%(I,0)=CP
94 NEXT:ZZ=0:CP=23:GOTO13
95 IFNO<>8THENPRINT"WOW, THIS IS FUN!":GOTO17
96 IFOB%(20,0)<>CPTHENPRINT"WHOOSH!":GOTO17
97 IFSH=0THENPRINT"THE SWORD BOUNCES OFF THE SORCE
RER AND  HITS YOU.":GOTO135
98 PRINT"THE SHARP SWORD SLICES THE SORCERER.":SH=
SH+1:IFSH<4THEN14
99 OB%(20,0)=0:OB%(14,0)=CP:PRINT"[CD]THE SORCERER
 DISAPPEARS.":GOTO14
100 IFOB%(NO,0)<>-1THENPRINTS1$:GOTO17
101 IFNO<>8THENPRINTS2$:GOTO17
102 IFOB%(10,0)<>-1THEN80
103 PRINT"THE SWORD IS NOW RAZOR SHARP.":SH=1:GOTO
17
104 IFOB%(NO,0)<>-1ANDOB%(NO,0)<>CPTHENPRINTS1$:GO
TO17
105 IFNO=17ANDOB%(2,0)=0THENOB%(2,0)=CP:PRINT"SOME
THING'S IN HIS POCKET!":GOTO17
106 IFNO=21ANDOB%(6,0)=0THENOB%(6,0)=CP:PRINT"SOME
THING'S IN THE STOMACH!":GOTO17
107 PRINT"IT'S JUST "OB$(NO)".":GOTO17
```

```
108 IFNO=4ANDOB%(4,0)=22THENPRINT"LOOK FOR IT IN T
HE FOREST.":GOTO17
109 IFNO=2ANDOB%(2,0)=0THENPRINT"KEEP TRYING.":GOT
O17
110 IFOB%(NO,0)=-1THENPRINT"YOU'RE HOLDING IT, STU
PID!":GOTO17
111 IFOB%(NO,0)=CPTHENPRINT"IT'S RIGHT IN FRONT OF
 YOU, STUPID!":GOTO17
112 IFNO<>20THENPRINT"PULL YOUR FINGER OUT AND LOO
K FOR IT!":GOTO17
113 PRINT"[CLR]YOU'RE IN THE SORCERER'S TORTURE CH
AMBER-- HE HAS A WHITE-HOT POKER";
114 PRINT" AND HE IS   COMING TOWARD YOU!":FORJ=1T
O3:GOSUB148
115 IFVB=25ANDNO=14ANDOB%(14,0)=-1THEN91
116 PRINT"THE SORCERER THRUSTS THE POKER AT YOU.":
NEXT:GOTO135
117 IF(CP>19ANDCP<24)ORCP=34THENPRINT"DOWN[2CD]DOW
N[2CD]DOWN[2CD]":GOTO135
118 IFCP<>44THENPRINT"WHEEEE!":GOTO17
119 IFWF=0THENCP=43:GOTO118
120 PRINT"YOU LAND SAFELY IN THE TREE'S BRANCHES."
:CP=21:GOTO17
121 IFCP<>44THENPRINT"I DON'T SEE A WINDOW.":GOTO1
7
122 IFWF=0THENPRINT"IT IS ALREADY CLOSED.":GOTO17
123 PRINT"IT'S STUCK.":GOTO17
124 IFCP<>44THEN121
125 IFWF=1THENPRINT"IT IS ALREADY OPEN.":GOTO17
126 PRINT"IT'S NOT EASY, BUT YOU MANAGE TO GET THE
WINDOW OPEN.  YOU SEE A";
127 PRINT" BIG LEAFY TREE  ABOUT 2 METERS BELOW TH
E WINDOW.":WF=1:GOTO17
128 PRINT"A BLACK KNIGHT IS RIDING ACROSS THE
BRIDGE TOWARD YOU!":GOSUB148
129 IFVB<>17ORNO<>17THEN134
130 IFOB%(1,0)<>-1THENPRINTB1$:GOTO134
131 IFOB%(4,0)<>-1THENPRINTB2$:GOTO134
132 PRINT"THE ARROW FINDS A CHINK IN THE KNIGHT'S
ARMOUR.  HE FALLS.
133 KN=1:ZZ=ZZ-1:OB%(4,0)=52:OB%(17,0)=52:GOTO17
134 PRINT"THE KNIGHT SKEWERS YOU WITH HIS LANCE."
135 FORI=1TO2500:NEXT:PRINT"[RVS]YOU ARE DEAD[OFF]
.":FORI=1TO2500:NEXT:GOTO147
136 PRINT"A PIRATE SNEAKS UP ON YOU AND STEALS THE
KEY.  [RVS]HAR HAR HAR";:PI=0
137 PRINT" HE CHORTLES:":PRINT"[RVS]I'LL HIDE THIS
 DEEP IN THE MAZE!
138 OB%(2,0)=34:ZZ=ZZ-1:GOTO17
139 PRINT"A GIANT SPIDER DROPS FROM THE CEILING!
140 PRINT"IT IS MOVING TOWARD YOU!":GOSUB148
```

```
141 IFVB<>17ORNO<>21THEN146
142 IFOB%(1,0)<>-1THENPRINTB1$:GOTO146
143 IFOB%(4,0)<>-1THENPRINTB2$:GOTO146
144 PRINT"THE ARROW RIPS INTO THE SPIDER.":SP=1:ZZ
=ZZ-1
145 OB%(21,0)=29:OB%(4,0)=0:OB%(5,0)=29:GOTO17
146 PRINT"THE SPIDER POUNCES ON YOU AND SINKS ITS
FANGS INTO YOUR NECK.":GOTO135
147 PRINT"[CLR]":GOSUB47:END
148 PRINT"[CD,RVS,BRN]WHAT NOW?[OFF,BLK] ";:GOSUB1
76
149 PRINT:N1$="":V1$="":NO=0:VB=0:NO$="":VB$=""
150 CM=LEN(CM$):FORI=1TOCM:IFMID$(CM$,I,1)<>" "THE
NV1$=V1$+MID$(CM$,I,1):NEXTI
151 VB$=LEFT$(V1$,3):FORI=1TONV:IFVB$(I)=VB$THENVB
=I:GOTO154
152 NEXTI
153 VB=-1:N1$=V1$:GOTO156
154 IFLEN(V1$)+1>LEN(CM$)THENNO=0:RETURN
155 N1$=RIGHT$(CM$,LEN(CM$)-1-LEN(V1$))
156 NO$=LEFT$(N1$,3):FORI=1TONN:IFNO$(I)=NO$THENNO
=I:RETURN
157 NEXTI
158 RETURN
159 PRINT"[CLR]YOU'RE ";P$(CP):PRINT:IFCP=1THENPRI
NT"[RVS]LEAVE ALL TREASURES HERE!
160 FORI=1TOLO:IFOB%(I,0)=CPTHENPRINT"[RVS,GRN]YOU
 SEE[OFF,BLK] ";OB$(I)
161 NEXTI
162 IFCP=1ANDDF=0THENPRINT"THE DOOR IS LOCKED."
163 IFCP=18ANDVF=0ANDOB%(16,0)=18THENPRINT"THE VAU
LT IS LOCKED."
164 IFCP=17ANDCF=0THENPRINT"A NARROW CRACK LEADS S
OUTHWARD."
165 IFCP=1ANDDF=1THENPRINT"THE DOOR IS OPEN."
166 IFCP=35ANDVF=1ANDOB%(16,0)=35THENPRINT"THE VAU
LT IS OPEN."
167 IFCP=17ANDCF=1THENPRINT"A WIDE CRACK LEADS SOU
THWARD."
168 IFCF=0THENP%(17,1)=0
169 IFCP=44ANDWF=1THENPRINT"THE WINDOW IS OPEN.  A
 TREE IS 2 METERS BELOW.
170 K=0:PRINT"[CD,RVS,RED]YOU CAN GO[OFF,BLK] ";:F
ORI=0TO3:IFP%(CP,I)=0THEN173
171 IFK=1THENPRINT", ";
172 PRINTD$(I);:K=1
173 NEXTI:IFK=0THENPRINT"NOWHERE."
174 IFK=1THENPRINT
175 PRINT:P%(17,1)=18:GOTO14
176 CM$=""
177 PRINT"[RVS]*[OFF,CL]";
```

```
178 GETZ$:IFZ$=""THEN178
179 Z=ASC(Z$):IFZ>95THEN178
180 ZL=LEN(CM$):IFZL>25THEN182
181 IFZ>31THENCM$=CM$+Z$:PRINTZ$;:GOTO177
182 IFZ=13ANDZLTHENPRINT" ":RETURN
183 IFZ=20ANDZLTHENCM$=LEFT$(CM$,ZL-1):PRINTZ$;
184 GOTO178
185 NP=53:LO=35:NN=31:NV=30:DIMP%(NP,3),P$(NP),VB$
(NV),NO$(NN),OB%(LO,1),OB$(LO)
186 PRINT"[BLK]PRESS 'SPACE' WHEN READY."
187 GETA$:IFA$<>" "THEN187
188 OPEN2,8,2,"O:DATA,S,R"
189 FORI=1TO53:INPUT#2,P$(I):NEXT
190 FORI=1TO53:FORJ=0TO3:INPUT#2,P%(I,J):NEXTJ,I
191 FORI=1TONN:INPUT#2,NO$(I):NEXT:FORI=1TONV:INPU
T#2,VB$(I):NEXT
192 FORI=1TO21:INPUT#2,OB$(I),OB%(I,0),OB%(I,1):NE
XT
193 FORI=0TO3:INPUT#2,D$(I):NEXT
194 CLOSE2
195 GOTO13
```

First of all, for any cassette users, a few changes will have to be made. Line 188 will need to read:

188 OPEN 1,1,0,"DATA"

As before, change all the INPUT#2s to INPUT#1s, and alter the CLOSE2 in line 194 to read CLOSE1. And there you are, home and dry.

The routine in lines 176 to 178 is there to check on what you've typed in, and to make sure that you can't use the cursor keys (among others). Typing RETURN takes us back to the main program again.

Lines 159 to 175 tell you where you are, what you can see there, what directions you can go in, and one or two other things depending on the state of play of the game.

The most important routine of all lies in lines 148 to 158, as it is this which decodes what you've typed in, and turns your VERB NOUN statement into a couple of numbers. It's only a lot of string handling, so you should be able to work out how it does what it does.

Given that we now have a verb number and a noun number, we can turn to lines 14 to 26, which check to see that what you've typed in makes sense, and if it does either print a message or fall through to lines 25 and 26, which send program execution off to the correct place.

Thus if your verb number was number 1 (in this case, the word GO), the program would go to line 27. If you've typed in verb 2, the word GET, the program would go to line 36, and so on. Each verb has its own little routine, although some of these can be extremely short, and in this way we can make the writing of the program that much easier.

By going through some of these routines you should be able to see how the program comes together to form a whole, functioning unit.

Conclusion

Castle is a fairly simple adventure, but then we've all got to start somewhere. Either just play the game and (I hope) enjoy it, or go on to write your own games. Just change a few things to begin with, like room descriptions and object descriptions, to see how that affects the game. Then, start putting a few objects in different places, which will mean that different problems now have to be solved. Before you know it you'll have written an adventure!

12

Introduction to Machine Code

Most of you will probably have seen some machine code routines in magazines and other publications, but won't have the faintest idea what they mean. This is reasonable, since most machine-code listings look unintelligible enough to the best of people. However, by the end of this chapter you'll be able to pick out most of what's going on in a listing, as most machine-code programs make extensive use of a very limited set of commands.

Machine code on the C16 only allows us to use 56 commands anyway, although it's fair to point out that most of these commands can operate in a variety of different ways. Only about half of these commands are commonly used, and only about half of that number are used very extensively. If you were asked to learn 28 different words in, say, an obscure African dialect, I'm sure that most of us could manage that in a few days. We are going to concentrate on those most commonly used instructions, even if we do cover the whole lot somewhere in this chapter.

Most introductions to machine code start off by telling you why you should learn about it. 'Programs can be written that operate at lightning speeds', is the main reason given. Personally I regard those sort of arguments as a waste of space. You know why you want to learn all about machine code, or presumably you wouldn't be reading this chapter in the first place.

To assemble, or not to assemble

Most people seem to think that you need some kind of an assembler to program in machine code. An assembler is a program that translates all the various symbols and numbers that you're going to be typing in over the course of the next few pages, into something that looks a bit more sensible and intelligible to you. Good assemblers can go a lot further than that, but this is an introduction to machine-code programming, so we'll ignore the more complicated stuff for now.

Luckily the C16 is blessed with a reasonably competent monitor, or assembler, built into it, so without further ado we'll type a program in and get it working for us.

Simple border routine

The following program is a reasonably short machine-code listing for drawing a border around the screen. It serves to show the speed of machine code when compared to the speed of Basic (and there's a more dramatic demonstration to follow later), and since it uses a grand total of 15 different commands, there won't be too much difficulty in following what's going on.

```
B*
   PC  SR AC XR YR SP
.;8FEB 33 00 D3 00 F6
.
3800 A9 66        LDA #$66
3802 A2 27        LDX #$27
3804 A0 00        LDY #$00
3806 9D 00 0C     STA $0C00,X
3809 9D 00 08     STA $0800,X
380C 9D C0 0F     STA $0FC0,X
380F 9D C0 0B     STA $0BC0,X
3812 CA           DEX
3813 D0 F1        BNE $3806
3815 8D 00 0C     STA $0C00
3818 8D 00 08     STA $0800
381B 8D C0 0F     STA $0FC0
381E 8D C0 0B     STA $0BC0
3821 A2 F0        LDX #$F0
3823 9D 00 0C     STA $0C00,X
3826 9D 00 08     STA $0800,X
3829 9D 27 0C     STA $0C27,X
382C 9D 27 08     STA $0827,X
382F 9D F0 0C     STA $0CF0,X
3832 9D F0 08     STA $08F0,X
3835 9D 17 0D     STA $0D17,X
3838 9D 17 09     STA $0917,X
383B 9D E0 0D     STA $0DE0,X
383E 9D E0 09     STA $09E0,X
3841 9D 07 0E     STA $0E07,X
3844 9D 07 0A     STA $0A07,X
3847 9D D0 0E     STA $0ED0,X
384A 9D D0 0A     STA $0AD0,X
384D 9D F7 0E     STA $0EF7,X
```

```
3850 9D F7 0A      STA $0AF7,X
3853 8D 00 39      STA $3900
3856 8A            TXA
3857 38            SEC
3858 E9 28         SBC #$28
385A AA            TAX
385B AD 00 39      LDA $3900
385E E0 00         CPX #$00
3860 D0 C1         BNE $3823
3862 60            RTS
3863 00            BRK
.
.
```

Now for the difficult part: typing it in. First of all, in normal Basic mode, enter POKE 52,55:POKE 56,55 <RETURN>. This sets aside a couple of kilobytes of memory for us to use before we start doing some typing. Since we haven't yet explained what any of the symbols are, you're going to have to enter it like a parrot. Enter the monitor with the MONITOR command, and type in the command A 3800 LDA #$66 <RETURN>. The computer will magically transform that line into a piece of code for you. Carry on with the next line (A 3802 LDX #$27 <RETURN>), and so on, until you've finished the whole thing. The numbers you're entering in are in hexadecimal, and if you're confused about hexadecimal and decimal then it might be an idea to return to Chapter 5 and find out all about them there.

Going back to our earlier command A 3800, 3800 is a hexadecimal number. This is equal to 3 times 16 to the power 3, plus 8 times 16 to the power 2, or 14336. This happens to be the start of the spare 2K of memory that we set aside with our earlier POKE commands.

Let's get back to our program. If you look at the listing, you'll see that the first line consists of:

3800 A9 66 LDA #$66

This tells the computer (and us) that at memory location $3800 (or 14336) sits the hexadecimal number A9, equivalent to the decimal number:

9 times 16 to the power 0, plus
10 times 16 to the power 1

which is equal to the decimal number 169. At memory location 14337, one further on, sits the hexadecimal number 66, equivalent to the decimal number:

6 times 16 to the power 0, plus
6 times 16 to the power 1

which is equal to the decimal number 102. Further on we come to the strange word LDA, followed by #$66. If you look at Appendix 4 at the back of this book, and track down the machine-code instruction headed LDA, you'll see that those letters stand for LoaD the Accumulator. You don't need to worry what an accumulator is just yet, we'll be finding out more about that later. The next lot of symbols, the #$66, tells us that we're LoaDing the Accumulator with the hexadecimal number 66, or the decimal number 102. You can treat this as reasonably analogous to the Basic statement:

```
LET A=66
```

In other words, we're assigning a value to this mysterious object the accumulator. Looking again through the listing, you'll see that the first number is always a memory location, the next little lot (a collection of one, two or three numbers) is all hexadecimal numbers, and the third lot is a collection of mnemonics and numbers.

To compare it to Basic again for a moment, you can regard it as LINE NUMBER followed by STATEMENT followed by a REM statement explaining what's going on. In other words, the assembly listings as presented throughout this chapter are all fairly similar to Basic listings, although what they achieve is way, way beyond what Basic can ever hope to do.

Back to the listing. Type it all in slowly and carefully, and save it onto tape with the command:

```
S "BORDER",01,3800,3863
```

or onto disk with the command:

```
S "0:BORDER",08,3800,3863
```

The S stands for SAVE, the name in quotes follows the usual basic rules for filenames, the first number after the quotes represents the device number that we want to save our program onto, and the third and fourth numbers represent the first memory location that we want to save and the last memory location respectively.

When you're sure that everything is correct, you can run the program by coming out of the disassembler by typing X and a carriage return. Change the background colour to white so that we can see what's happening by typing in:

```
COLOR 0,2,7
```

and then type SYS 14336, followed by RETURN. A border will instantly be drawn around the screen, provided, of course, that you've typed the program in correctly. Control should then be returned to you again, and the usual READY message with the flashing cursor underneath should confidently be displayed on the screen.

Type SYS 14336 a few times (clear the screen before you do, otherwise nothing will appear to be happening) just to get a feel for the power of machine code. You don't yet understand how the program is doing what it is doing, but at least you can now look at machine code listings without feeling too daunted. Most of the commands used in our border routine are in that common group mentioned earlier that get used all the time, and they will, over the next few pages, become very familiar friends indeed.

Starting to count

Now that you're getting a bit more familiar with what a machine code program looks like, how it's built up and how to enter it into memory, the following set of programs should serve to convince you of how fast this language really is.

These programs were first presented aeons ago by Mike Gross-Niklaus for the Commodore PET, but a quick dusting off and translation for the C16 will give a remarkable demonstration. Enter and run the following Basic program:

'MILLION COUNT BASIC'

```
10 COLOR 0,1:COLOR 4,1
15 PRINT "[CLR,RVS,YEL,6SP]"
20 SC=3072:N=5
30 FORI=SCTOSC+5
40 POKEI,48
50 NEXT
60 D=N
70 A=PEEK(SC+D):A=A+1
```

```
80 IFA<58THENPOKESC+D,A:GOTO60
90 POKESC+D,48
100 D=D-1
110 IFD=-1THENEND
120 A=PEEK(SC+D):A=A+1:GOTO80
```

The idea of the program is that it counts up to a million in the top left-hand corner of the screen, by the rightmost digit all the time (when that reaches 9, the one next to it is updated and the rightmost one set to zero again). If the next digit in reaches the total of nine when it comes round to updating, that is set to zero as well and the one next to that updated. The program goes on (and on, and on) until the count finally reaches a million. Don't bother waiting for it, as it takes around 7 ½ hours!

Yes, I know it can be done faster, but the program is meant to be a direct comparison with the machine-code one.

So, enter the Monitor as usual and type in the following program:

```
B*
   PC  SR AC XR YR SP
.;6F9F 33 00 87 00 F6
.
3800 A2 05       LDX #$05
3802 A9 30       LDA #$30
3804 9D 00 0C    STA $0C00,X
3807 CA          DEX
3808 10 FA       BPL $3804
380A A2 05       LDX #$05
380C BD 00 0C    LDA $0C00,X
380F 18          CLC
3810 69 01       ADC #$01
3812 C9 3A       CMP #$3A
3814 F0 06       BEQ $381C
3816 9D 00 0C    STA $0C00,X
3819 4C 0A 38    JMP $380A
381C A9 30       LDA #$30
381E 9D 00 0C    STA $0C00,X
3821 CA          DEX
3822 10 01       BPL $3825
3824 60          RTS
3825 BD 00 0C    LDA $0C00,X
3828 18          CLC
3829 69 01       ADC #$01
382B 4C 12 38    JMP $3812
382E 00          BRK
382F 00          BRK
3830 00          BRK
.
.
```

Save it to tape or disk when you've finished, and then exit the Monitor as usual. Then enter the following short program:

```
10 COLOR 0,1:COLOR 4,1
15 PRINT "[CLR,RVS,YEL,6SP]"
20 T=TI
30 SYS14336
40 PRINT:PRINT:PRINT"TIME TAKEN = ";(TI-T)/60;" SECONDS"
50 END
```

When you're satisfied with it, RUN it, and if you can watch the digits changing you're a better man than I. The count this time takes less than 30 seconds, as compared to about 7 ½ hours. Quite an improvement. And in case you're thinking that the program isn't really counting to a million, it is, as we shall see later on.

You may by now be wondering what is the point of typing in these programs when you haven't a clue what's really happening. Well, the object is to get you used to entering a listing, enable you to see how fast machine code is, and perhaps whet your appetite for more.

By now you should be au fait with entering machine code listings, saving programs onto tape or disk, and generally getting an idea of how powerful this language is. (As a by-line, if you want to load the programs back from tape into the computer when you're not in the monitor, you'll have to LOAD "BORDER",1,1 instead of the usual LOAD "BORDER". This tells the computer NOT to load the program in at the stat of Basic, where it would normally go, but at the place where it was saved from: in our case, starting at memory location 14336 decimal, 3800 hexadecimal.)

It is convenient, for the purposes of this chapter, to have these programs typed in and working by the time we get to explaining precisely how they work. If you have them on the screen in front of you while we go through changing a few things, explaining how this command works, how that one operates, and so on, it's a lot easier to grasp when the programs are already up and running than it is when you're still trying to type them in. You can see what's happening, having already overcome the horrors of typing them in.

First lessons

It's important, in these early days of learning how to program in machine code, to get certain facts very, very clear indeed. A little time spent now going over what to some will be fairly simple stuff, will save a lot of time for all of us later on.

First of all, what is a byte? Put simply, a byte is the equivalent of one character, such as the letter A, and when we speak of a computer having X thousand bytes of usable memory, then that is just another way of saying that that computer is capable of storing X thousand characters in its memory. A byte is not the smallest amount of information that a computer can concern itself with, however. Bytes are split up (on the C16 anyway) into eight bits, with four bits combining to form a nibble. Each bit has a value associated with it, and we can talk in terms of the 2nd bit in the 4th byte of memory, or whatever.

To put it pictorially:

Bit No : 7 6 5 4 3 2 1 0
Value : 128 64 32 16 8 4 2 1

If you add all those numbers up you'll see that they total 255, which is why the largest value that can be stored in any byte is 255: you just can't get in any number higher than that.

To alter the value stored in any particular byte, you're probably well aware of the commands POKE and PEEK.

If we, say, POKE 14336 with 169, what precisely are we doing? Look at the number 169, and try to figure out what set of values in the above combinations add up to 169, remembering of course that you can't have two bits set to equal 64: only one bit can be set to have that value in it, and that is the seventh one (cunningly referred to as bit 6 just to confuse things).

The number 169 is in fact made up of bits 7 (128), 5 (32), 3 (8) and 0 (1). So, the result of POKEing 14336 with 169 is to turn bits 7, 5, 3 and 0 on, and turn bits 6, 4, 2 and 1 off.

To turn bits 0, 3, 5 and 7 on without affecting the status of bits 1, 2, 4 and 6, which may already be on or off, we have to use the following statement:

```
POKE 14336,PEEK(14336)OR169
```

Okay, so that's bits and bytes sorted out, and we can already cope with hexadecimal and decimal, so let's take a look at some of the more commonly encountered commands in machine code.

To take the example of a simple Basic program first of all, we'll add two numbers together: the numbers 4 and 6. In Basic, this might be written as:

```
10 A=4
20 B=6
30 C=A+B
40 PRINTC
```

The machine-code equivalent might go something like this:

```
3800   A9   04        LDA  #$04
3802   A2   06        LDX  #$06
3804   8E   00   0C   STX  $0C00
3807   6D   00   0C   ADC  $0C00
380A   8D   02   0C   STA  $0C02
380D   60             RTS
```

What does all this mean? Line by line, we are:

Loading the accumulator with the hexadecimal number 4.

Loading the X register with the hexadecimal number 6.

Storing the X register at memory location 0C00 (hexadecimal), or 3072 (decimal).

Adding to the accumulator (which contains the value 6) the contents of memory location 0C00 (which now contains 4, because we've just put it there).

Storing the new value in the accumulator (4 + 6 = 10) at memory location 0C02 (hexadecimal) or 3074 (decimal).

Returning from this subroutine.

Well, that makes a bit more sense, but probably not much. What is the accumulator, this mysterious X register that has cropped up from nowhere, why should storing something in a memory location be the same as printing the result of adding 4 and 6?

The accumulator is the heart of the processor that looks after the C16, and it is the accumulator that does most of the work in all machine-code programs. In itself, it is nothing more exciting than an 8-bit storage area, and as such it can store any number up to 255, as we've seen.

But why put a four into it?

To revert back to a Basic way of thinking for a while, we're all used to assigning values to variables in Basic. A=4, B=6 and so on. In machine code there is no such thing as a variable in this sense. What we have to do is to use the accumulator, the X register and the Y register to store and retrieve numbers. These are the only three things that are capable of storing and retrieving numbers, and all three come into extensive use in most machine-code programs.

Thus, the result of putting a 4 into the accumulator can be thought of as being reasonably equivalent to assigning the value of 4 to a variable. Similarly, putting a 6 into the X register (another 8-bit storage area) is similar to assigning the value of 6 to another variable. Finally, if we'd wanted to, we could have assigned a value to the Y register (our third and final 8-bit storage area), which would again have been similar to giving a Basic variable a value.

With only three 'variables' to play with, you might well imagine that life can get pretty hairy at times, and you are so, so right. Fortunately, since machine-code programs operate at amazingly fast speeds, this apparent limitation doesn't really worry us, as long as we remember where everything is stored.

Storing values in memory

To go back to the last little machine-code program, you'll see that we stored the value in the X register at a certain memory location. In machine code you put values into registers and store them all over the place (remembering of course where you've put them), retrieving them when the need arises. On the C16 memory location 0C00 happens to be on the screen, and storing the content of the X register at location 0C00 is equivalent to POKEing a number (whatever happens to be in the X register) into location 0C00 in hexadecimal, or 3072 in decimal. This, of course, is the start of screen memory, since the screen is just another set of memory locations like everything else.

As we're dealing with the C16, life is never as easy as we'd like it to be. Just as in Basic POKE3072,6 doesn't achieve very much, nor does storing the X register at location 0C00. This is because we also need to put some colour there so we can see what's happening. In Basic, you might POKE 2048,0 to get a black character appearing at the top of the screen (a black letter D if we've got a four there). In machine code, we'd have to:

```
380D   A2   01        LDX #$01
380F   8E   00   08   STX $0800
3812   8E   02   08   STX $0802
3815   60             RTS
```

In other words, load the X register with a 1, and store it at locations 0800 and 0802, the hexadecimal equivalents of colour memory locations 2048 and 2050. You'll note our RTS has been moved down a bit to accommodate these new instructions. Without having an RTS at the end of a routine, unlike Basic which grinds to a halt when there are no more statements to execute, machine code merrily trundles on through memory until it finds something to do: and that something may not be very pleasant for the machine. You won't damage the computer, but you may well cause it to crash, losing the program that you've so lovingly typed in.

So loading a register (or the accumulator) is the equivalent of defining a variable, and storing that register somewhere in memory is the equivalent of POKEing a variable into memory.

Incidentally, to run that little program given earlier you can type the command SYS 14336, which transfers program execution to location 14336, and only returns to Basic Ready mode when it encounters an RTS (or something that causes the machine to throw a wobbler. A technical term, that one).

Some formal definitions

We'll give some formal definitions of the commands used so far:

STA

STore the contents of the Accumulator at the memory location specified.

LDA

LoaD the Accumulator with the numeric value specified.

LDX

LoaD the X register with the numeric value specified.

STX

STore the contents of the X register at the memory location specified.

ADC

ADd to the accumulator with Carry the contents of the specified memory location.

Don't worry about the 'with Carry' part. We'll be hearing a lot more about that later.

All the definitions given above are basic ones, as most of those commands have variations on the way that they've been used so far. We'll take care of those variations as and when we get to them.

More simple programs

To accustom you to entering simple machine-code programs, help you think in machine code rather than in Basic, here are a couple of simple programs.

One popular program places a heart (or diamond, or whatever) on the screen. Not very exciting perhaps, but it is something. What you're not usually told is why the program works. Why should I put a 65 into the accumulator? Read on...

```
3800   A9   41        LDA  #$41
3802   8D   00   0C   STA  $0C00
3805   A2   00        LDX  #$00
3807   8E   00   08   STX  $0800
380A   60             RTS
```

Here we're putting the value 65 into the accumulator. Why? The screen code for a heart is 65, so if, in Basic, you typed:

POKE 3072,65

A heart would appear in the top left-hand corner of the screen (if you could see it of course). That POKE command is the equivalent of the first 5 bytes of that little machine-code program above. Load the accumulator with the code for a heart, and store it at memory location 0C00 hexadecimal, or 3072 decimal.

Now, what about the next part? Loading the X register with a 0 (we might just as easily have loaded the accumulator with a 0 instead), and storing it at location 0800, is the equivalent of:

```
POKE 2048,0
```

which turns our little heart into a black heart.

Finally, we return from the subroutine and end up in Basic Ready mode again.

An example like this doesn't really show any great advantage over Basic, so we'll print out 5 rows of the things (200 hearts in all) by introducing a few new machine-code statements. But first, a possible Basic equivalent.

```
10 FORI=0TO199
20 POKE3072+I,65
30 POKE2048+I,0
40 NEXTI
```

This will put 200 little black hearts at the top of the screen. In machine code, this might be written as:

```
3800   A2   C8        LDX #$C8
3802   A9   41        LDA #$41
3804   9D   FF   0B   STA ($0BFF),X
3807   A9   00        LDA #$00
3809   9D   FF   07   STA ($07FF),X
380C   CA             DEX
380D   D0   F3        BNE 02 38
380F   60             RTS
```

There are a few new commands here. Before considering them in detail, we'll explain what the program is doing in plain English.

First of all, put a value of C8 hexadecimal, or 200 decimal, into the X register.

Put a value of 41 hexadecimal, or 65 decimal, into the accumulator.

Store it at memory location 0BFF offset with the X register. In other words (working in decimal for a while), the first time around the loop the X register contains 200, so we're going to store the contents of the accumulator at location 3071 offset with 200, or 3271.

Put a value of zero into the accumulator.

Store it at memory location 07FF offset with the X register.

Decrease the content of the X register by 1. That is, the first time around decrease it to 199, then 198, and so on.

If the result of doing that isn't equal to zero (i.e. the X register doesn't contain a zero), then branch back to location C002 and load the accumulator with a 41, or decimal 65 again and repeat as before.

We finally get here when the X register does contain a zero and program execution comes to a halt. So, plenty of new commands, and if you run this program with a SYS 14336, you'll be amazed at the difference between the Basic version and this machine-code one. You can see Basic performing, but this just seems to happen instantaneously.

One final point may be puzzling you. Why are numbers seemingly reversed when entered into the disassembler? For example, if you look at the line starting at memory location 3804, why do we have FF 0B in one part, and 0BFF in the mnemonic part? As you know, a number greater than 255 cannot be stored in one single byte, and so when we're talking about numbers as large as 0BFF hexadecimal, or 3071 decimal, this number has to be spread over two bytes.

Later on we'll be talking about something called the stack, a place for storing useful information. The stack, like seemingly everything else in the computer world, stores numbers on a 'last-in, first out' basis. That is, the number it was last told to remember is the first number that it will retrieve. Like a stack of plates, the last one to be put on top of the pile is the first one to be taken off it. So the first number the computer sees when it's looking at two bytes storing a large number is the last one that was put in there: in this example, the 0B. Then it sees the next one, the FF, and remembers the number as 0BFF.

New commands explained

What we've seen in that last program are the commands BNE, DEX and STA offset with X. In order then:

BNE

Branch if the result of the previous instruction does not yield a value of zero. In other words, is Not Equal to zero.

DEX

DEcrement the content of the X register by one.

STA offset

STore the content of the Accumulator at a specified memory location, offset with the contents of the X (or Y) register.

We're already beginning to see some variations on the earlier command theme that we gave you (STA offset is a very different animal to STA), and there'll be plenty more to come.

And some more

JSR

Jump to the SubRoutine at the specified memory location.

CMP

CoMPare the contents of the accumulator with a specified number. We can also compare them with the contents of a specified memory location, among other things.

BEQ

Branch if the result of the previous instruction is EQual to zero: similar in operation to the BNE instruction met earlier. That one was, if you remember, branch if the result of the previous instruction does not give a value of zero.

How do we decide how far to branch? Obviously that will depend on where you want program execution to continue, but you should note that with all these branching commands there is a limit to how far we can go, and that limit is either 127 memory locations further on in the program or 128 memory locations further back in the program.

Sticking with decimal numbers for a while, if we could have a command such as:

BEQ 30

program execution would jump forward 30 bytes if the result of an operation was zero.

If we had something like:

BEQ 210

program execution would jump back (256-210) 46 bytes if the result of an operation was zero.

Out on the border

We'll now go back to those earlier programs to examine how they worked, starting with the border one, so get the Monitor in there, load the Border routine (which you did save, didn't you?), and let's take a proper look through the listing.

As you can see, this relies quite heavily on the STA offset feature. This apparent inconvenience of having to type out endless lists of STA instructions doesn't really matter, since machine code whizzes along at a fair old rate of knots. The equivalent performance in Basic would take an eternity ... well, almost.

Starting with the very first line of the program, you can see that we load the accumulator with a value of 66 hexadecimal, or 102 decimal.

This determines not only what character is going to form our border, but also what colour that character will be displayed in, since we use the same value for both things.

Then, load the X register with a 27 hexadecimal, or 39 decimal (one less than the screen width), and the Y register with a zero. Looking back on the program I can't for the life of me remember why I did that, but I'm sure there must have been a reason at the time! It certainly doesn't affect this program.

Then we store the value in the accumulator at memory location 0C00 (the start of the screen) offset with the value in X. Since this is 27 hexadecimal to start with, that's where the value 66 goes: at 0C27. The equivalent of POKE 3111,102. We similarly put the value of 66 into the colour memory for that screen location.

The next two lines do the same for the bottom row of the screen, before decrementing the X register and seeing if we've reached a value of zero yet. If we haven't, then trot back to location 3806 and go through the whole performance again, remembering that this time around the X register will have the value 26 hexadecimal in it, and so we now alter screen location 0C26 and colour memory location 0826.

This continues until the X register finally has a zero in it, when we colour in the top left corner of the screen, and the leftmost character on the bottom line of the screen. This has to be done since, when the the contents of the X register reached zero, we never went round the loop again to store the accumulator at 0C00 offset with zero, i.e. 0C00.

Now for the complicated part. The X register is loaded with a value of F0 hexadecimal, or 240 decimal. This is the equivalent of six lines of the screen (40 columns per line), and allows us to put four characters down the left-hand side of the screen and four down the right, remembering to put some colour there as well, of course.

Then we temporarily store the value held in the accumulator at memory location 3900: just somewhere to store the value where it won't come to any harm. We then transfer the contents of the X register to the accumulator, because although we can perform mathematical operations on the accumulator, we can't on the X register, and we want to do a bit of subtraction. The SEC command simply tells the computer that there's a bit of subtraction coming up, and the next line (SBC #$28) subtracts the hexadecimal value of 28, or the decimal value of 40, from the value in the accumulator. The new value in the accumulator is now transferred back to the X register, and the old value of the accumulator (102, from way back at the start of the program) is picked up from where it was dropped off in memory location 3900: its temporary storage position.

The value in the X register is now compared with 0, and if the result of this operation doesn't give us a zero we branch back to memory location 3823 and start this whole performance again. Remember that now the X register contains a value 40 less than it previously did, and so our STA offset command affects one line further up the screen.

This continues until we've filled in all the edges of the screen, and the program comes to a halt with the RTS command at location 3862.

To go through the four new commands encountered in this program:

TXA

Transfer the contents of the X register to the contents of the Accumulator, leaving the contents of the X register unaffected.

SEC

SEt the Carry flag. In other words, tell the computer that there's some mathematics coming up which involves subtraction.

SBC

SuBtract from the accumulator with Carry the specified number, or we can also subtract from the accumulator the value stored in a specified memory location.

TAX

Transfer the contents of the Accumulator to the X register, leaving the contents of the accumulator unaffected.

Well, that's one reasonably complicated program explained in great detail, so we might as well get the other one out of the way now, and go step by step through the Million Count routine. Again, it'll be a great help if you can get this one up on the screen and follow it through as we explain what's happening.

If I had a million

Again, we'll go through this one instruction at a time, so that you can get a clear understanding of what is happening. To start, the first two instructions are reasonably straightforward: load the X register with the value of 5, and the accumulator with a value of 30 hexadecimal, or 48 decimal. Decimal 48 is the value associated with the numeric figure 0.

Then we store our accumulator value (the figure 0) at memory location 0C00 offset with the value of the X register. Decrement the X register, and check to see that it is either positive or zero. If it is, go back to memory location 3804 and store the accumulator at 0C00 offset with the new value of X. This continues until X becomes negative and we then have 6 zeroes up on the screen. The program has begun.

The X register is re-loaded with 5 again, and the accumulator in the next line uses the LDA offset command. Analogous to the STA offset, only this time we're receiving a value, not placing one. CLC stands for CLear the Carry flag, and tells the computer that there's going to be some addition going on in the very near future.

We then add 1 to the value stored in the accumulator, and compare that with 3A hexadecimal. This is one way of checking to see if the accumulator is displaying the code for the number 9 (hexadecimal 39) or has gone over that limit. This is checked for in the next line, because if the accumulator does contain 3A program execution continues at memory location 381C. If it doesn't, then we store this updated value on the screen and JuMP back to memory location 380A.

Now, if the accumulator contains a value greater than nine, we need to switch this back to zero, and update and check the next digit on the left. LDA # $30 at location 381C puts the numeric code for zero into the accumulator, and stores it on the screen. The value in the X register is then decreased by one, and if the result of doing this is negative (Branch on PLus checks for a number being either positive or equal to zero) then we've reached a million and the program ends.

If, however, the X register still contains a positive number or zero we branch to memory location 3825 and carry on. This loads the accumulator with the content of memory location 0C00 offset with X, but remember we have now decremented X and so are looking at the character to the left of the one that we've just set to zero.

Another CLC command heralds a further bit of addition coming up, and one is added to the content of the accumulator. Then program execution jumps to location 3812 to start the whole series of checks off all over again.

As with the Border routine, we'll give you some formal definitions of the new instructions encountered in this program.

BPL

Branch if the result of an operation is either positive or zero. As long as it isn't negative, that's fine by us. It actually stands for Branch on PLus: for once, a confusing mnemonic.

CLC

CLear the Carry flag. In other words, prepare for some addition.

ADC

ADd with Carry the contents of the accumulator to the number specified. This can also be used to add the contents of the accumulator to the contents of a specified memory location.

JMP

JuMP to a specified memory location. Equivalent to GOTO in Basic really, and it's as good an idea to try to avoid it in machine code as it is in Basic. This is because we will often want to change a couple of bytes of a program, which will involve juggling a few things around. If the location that we're jumping to happens to move, the machine-code program will not change where we've specified to jump to, and program execution will undoubtedly end up in no-man's land.

LDA offset

LoaD the Accumulator with the content of a specified memory location, offset with the value of the X register, or Y register for that matter.

Flags, registers and other wonders

We mentioned in the last section such things as STA offset, an alternative version of the STA command, instructions such as ADC (ADd with Carry), setting and clearing the carry flag, and we offered dark hints about registers such as the X and Y register. Before we encounter the horrors of double precision arithmetic it's about time to get formal again and explain what all these things really mean.

To start, we'll take a look at the way the 7501 microprocessor at the heart of the C16 really executes a program.

How the 7501 executes a program

Rather like a program written in Basic, the 7501 operates quite simply by fetching an instruction from memory, acting on and executing that instruction, and then going to fetch another one. But how does it know which instruction is the next one? A special register is set aside to control all this, and this register is known as the program counter. It is this register that informs the 7501 which instruction it has to execute next.

Fortunately for us, this program counter is automatically incremented after every instruction, and as a programmer you don't have to worry about updating it yourself. This program counter register is actually two bytes stuck together (rather than the one byte of the accumulator, the X register, and the Y register), and as such it comprises 16 bits. This enables it to look at any location within the 64K of memory (a mixture of RAM and ROM) that the C16 contains.

You may remember our earlier dissection of a byte, which looked like this:

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

Well, the program counter, being two bytes long, looks something like this:

Bit no.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Adding all those values up gives us a grand total of 65535, which is the largest number that the program counter can accomodate. Since 64K is the equivalent of (64 times 1024) bytes, or 65536 bytes, the program counter can readily access any byte within those 65536 (0 to 65535 being 65536 numbers).

As you've seen from the listings given earlier, machine code instructions can consist of one, two or three bytes. The first one is always the machine code equivalent of the operation code (known for short as the op-code), as a glance at Appendix 4 will reveal. This byte is directed off to the instruction register, to find out what it means, and then routed to something known as the instruction decode logic. This sends out appropriate signals to all the other elements of the microprocessor, warning them all that something is about to happen.

The second byte, and third if appropriate, are then pushed off into the data bus buffer, and from there sent either to the arithmetic logic unit (known as the ALU) if they merely contain data, or to our old friend the program counter if they contain the address of a memory location.

Now aren't you glad that all this sort of stuff is handled automatically, and you don't have to worry about it?

More registers

We've already had extensive experience of looking at the X and Y registers, and the accumulator. Since the accumulator is the only register on which arithmetical and logical operations can be performed, it is the most powerful register within the 7501. It is also the only one on which logical operations can be carried out. The program counter is another register that we've discussed.

But there are other registers, and the first one that we'll come to is known as the processor status register.

Processor status register

This is, as usual, an 8 bit register, although we can in fact only get useful information from seven of these bits. In diagrammatic form, the register looks like this:

Bit	7	6	5	4	3	2	1	0
Flag	N	V	-	B	D	I	Z	C

Flags are something that we encountered earlier, but never really explained properly. The status of these flags is tested by a number of machine code instructions, and some of them we've already met: BNE, BPL, SEC and so on. We'll go through each of these flags in turn, starting with one that we've already met.

The carry flag (C)

This is the one handled by byte zero of the status register. It really comes into play when we want to deal with numbers that are greater than 255. As we've already seen, 255 is the largest number that can be stored in any one byte and if, as with the program counter, we want to handle numbers that are larger than that, we have to use two bytes that are next to each other.

To link two bytes together there has to be something that joins the first one to the second, and this is the purpose of the carry flag. Think of it as a number carrying over from one byte to the next, if you like. Rather like the way you did addition at primary school: if a number is bigger than 10, say 18, then you put the 8 down and carry 1.

There are four important instructions for using and acting on the carry flag, and these are:

BCC: Branch on Carry Clear.

This instructs the program to branch off somewhere else if the carry flag is clear, i.e. it isn't set, and the value stored in it is currently zero. To make sure that this doesn't happen accidentally, but only when we want it to, the next instruction clears the carry flag before we perform any mathematical calculations.

CLC: CLear the Carry flag.

This sets the carry flag to zero, and thus ensures that nothing untoward happens to our programs.

BCS: Branch on Carry Set.

The opposite of BCC. If the carry flag is not clear, i.e. it has been set and the value currently stored in it is a one, then branch to whatever part of the program we wish to go to. Again, there is an analogous command to CLC to make sure that this doesn't happen accidentally, although this is usually only used when we want to perform some subtraction.

SEC: SEt the Carry flag.

This sets the carry flag to one, i.e. it is no longer clear.

To illustrate this, here is a short program:

```
. 3800   18          CLC
. 3801   A9 01       LDA #$01
. 3803   69 01       ADC #$01
. 3805   B0 03       BCS $380A
. 3807   4C 03 C0    JMP $3803
. 380A   8D 00 0C    STA $0C00
. 380D   8D 00 08    STA $0800
. 3810   60          RTS
```

This first of all clears the carry flag, the loads the accumulator with the number one. The next instruction adds one to the value currently in the accumulator and then checks to see if a carry has been set. That is, has the number in the accumulator exceeded 255 and flipped around to 0 again? If it has, we go off to 380A and perform the instructions there, but if it hasn't we go back to 3803 and add another one to the accumulator.

The instructions STA $0C00 and STA $0800 then store the content of the accumulator (a zero now, since it's flipped over from 255 to zero as the carry flag was set) in the top left-hand corner of the screen. The result is that an '@' sign appears.

The three other flags in the status register that are affected by numerical operations are the zero flag, the overflow flag, and the negative flag:

The zero flag (Z)

This is the second bit of the status register, and it is set if the result of a mathematical operation is zero. It is this flag that is being looked at in the BNE and BEQ commands we covered earlier. If the result of an operation is equal to zero, then the zero flag is set and a BEQ command, seeing that the flag is set, would then send program execution off somewhere else.

The overflow flag (V)

This is the seventh bit of the status register, and is set when two positive or negative numbers are added together, and the result exceeds either +$7F or −$80. Hexadecimal numbers, to avoid writing the word out thousands of times, are usually prefixed with a dollar sign, as in the above example. We will use this convention throughout the rest of this chapter.

The negative flag (N)

The 8th bit of the status register, this is set when two signed numbers, if added together, give a negative result. If they don't, it isn't set.

The three flags that we haven't looked at yet are the B, D and I flags, or respectively the BRK command, Decimal mode, and IRQ disable flags. Briefly, the first one indicates whether an interrupt request to the 7501 was caused by a 'break' instruction (op-code 00) or by some externally generated interrupt, the second one determines whether the ALU will operate in binary mode or decimal mode, and the final one shuts out all interrupts to the 7501, should you decide not to let anything interrupt you for a while.

From flags and registers, we'll move on to another important concept, which you must grasp before you can start to use machine code at all seriously. This is concerned with what are termed modes of addressing. We said earlier that many of the commands that we've used (LDA, STX and so on) have variations on a main theme, and these variations are governed by which addressing mode we are currently operating in.

Modes of addressing

In all, there are some thirteen different modes of addressing, and these can be summarised as:

MODE	OPERAND FORMAT
Immediate	#aa
Absolute	aaaa
Zero page	aa
Implied	
Indirect absolute	(aaaa)
Absolute indexed,X	aaaa,X

Absolute indexed,Y	aaaa,Y
Zero page indexed,X	aa,X
Zero page indexed,Y	aa,Y
Indexed indirect	(aa,X)
Indirect indexed	(aa),Y
Relative	aa or aaaa
Accumulator	A

In the table, the letter 'a' represents a hexadecimal digit, so aaaa is a four digit hexadecimal number, such as $4F2C. X and Y refer to the X and Y registers.

We'll give a brief rundown on each of these modes over the next few pages.

Immediate addressing

This form of addressing allows you to specify a single byte constant as the operand (i.e. the thing to be operated with). Thus any number between 0 and 255 can act as the operand, and it has to be prefixed with a hash sign. Thus the instruction:

```
LDX #$5B
```

loads the hexadecimal number $5B (or decimal 92) into the X register. Instructions using immediate addressing are always 2 bytes long, and the second byte is always the operand.

Absolute addressing

This allows us to address any one of the 65536 memory locations in the 7501, since, as we've seen, two bytes put together allow us to look at such large numbers. The table above shows us that absolute addressing uses two consecutive bytes. Since the first byte in a machine-code instruction is always the op-code, it follows that an instruction using absolute addressing will always take up three bytes of memory, with the second byte being the lower part of the operand address and the third byte being the higher part. For instance:

```
AD 00 0C LDA $0C00
```

will take the content of memory location $0C00 and store it in the accumulator.

Zero page addressing

This is an alternative form of absolute addressing, in which the operand now consists of a single byte. This means that we can only cover the first 256 bytes of memory, and this is referred as page zero. Since we're only using a single byte for the operand, it follows that zero page addressing instructions will always consist of two bytes: the op-code and the operand. For instance:

```
LDA $1B
```

will load the content of memory location $1B into the accumulator.

Apart from JSR and JMP, any command that can use absolute addressing can also use zero page addressing. What are the advantages of using zero page? Well, since we're using two-byte instructions instead of three-byte, we're taking up less memory, and this two-byte instruction takes one cycle less to process than the normal three-byte. Zero page is also useful as a temporary storage area for data values, since there is a reasonable amount of empty space in there.

However, be very careful when using zero page. Since it is faster to do this, and also takes up less memory, the 7501 wants to use it as much as you do, so watch out that your two interests don't run into conflict.

Implied addressing

About half the instructions that the 7501 is capable of processing do no more than clear registers, transfer data from one register to another, increment or decrement registers, and so on. These commands need no operand, since the 7501 receives enough information from the op-code: the rest is implied, hence the name for this mode of addressing.

Some examples include DEX for decrementing the content of the X register, TXA for transferring the content of the X register to the accumulator, and so on. All implied commands take up just one byte of memory.

Indirect absolute addressing

There is only one instruction in the entire repertoire of the 7501 that can use this mode of addressing, and that is the JMP instruction. Using this mode, the JMP command causes the program counter to be loaded with a new address from which the 7501 is to fetch its next instruction. When used in absolute mode, the JMP command simply puts the destination address into the program counter, as in:

JMP $3821

This causes the memory location 3821 to be stored in the program counter, hence program execution will continue at that location. However, when used in implied mode, something totally different happens. For instance, the command:

JMP ($01F0)

causes the program counter to be loaded with the low-order address stored in location $01F0, and the high order address stored in location $01F1. It's probably easier to see what's going on by way of an illustration, rather than talking about it:

Location	Content
3800	$6C
3801	$F0
3802	$01
:	:
:	:
:	:
01F0	$21
01F1	$38

Thus what gets stored in the program counter is the content of memory location $01F0 and $01F1. The program counter would now contain the memory address $3821, and program execution would now continue at this point. Isn't all this a bit complicated, when you could probably use an ordinary JMP instead? Well, most of the time you would use an ordinary JMP, but there are times when this sort of indirect addressing is most useful.

For example, there are a lot of systems now available that allow several keyboards to be connected up to one main terminal. That main terminal then has to act according to which keyboard is currently being used to access it. Using indirect addressing, we could then start acting upon different parts of the program in the main terminal according to which keyboard was currently in use, as the 7501 would change the contents of some of its memory locations as different devices were attached to it (or in this case, as different keyboards came into play). Thus the indirect jump to those locations that change would cause program execution to continue at the correct part of the program for that keyboard.

Absolute indexed addressing

We've referred to this in the past as STA offset. To put it more formally, the address of the operand is computed by taking the absolute address implied in the instruction and adding the content of the X or Y register to it, depending on which is being used. This sort of addressing makes for a three-byte machine-code instruction. For example:

```
LDA $3820,X
```

will load the accumulator with the content of memory location $3820 plus X. That is, if the X register contains a 5, then the accumulator will be loaded with the content of memory location $3825. This sort of addressing is useful in many ways, whether for accessing long lists of data, providing some animation on the screen, and so on. It was used effectively in the Border routine, for example.

Zero page indexed addressing

This is the two-byte equivalent of absolute indexed addressing, and since we only use one byte for the operand we are restricted to accessing the first 256 bytes of memory: zero page again. As with absolute indexed, the operand is found by adding the content of the X or Y register to the zero page address specified in the second byte of the instruction. For example:

```
LDA $57,X
```

If the X register contains a 7, then the accumulator will be loaded with the content of memory location $57 + $7, or $5E. As with other zero page equivalents, this command takes up one less byte of memory than its absolute equivalent, and it will usually run one cycle faster per instruction as well.

However, a word of warning. Since we are only using one byte for the operand, we are restricted to just those first 256 bytes. But what about if the operand plus the X register exceeds 256? The answer is that it just wraps around, so that the command:

```
LDA $FA,X
```

would be okay as long as the X register didn't contain anything greater than 5, but if it contained 6, then the effective address thus produced ($FA plus 69) would be one more than 255, so it would wrap around to zero again, causing chaos and havoc in your programs.

Indexed indirect addressing

If you're a drinking man, now's the time to reach for the whisky, because this is going to get a mite hairy.

This addressing mode is a combination of two that we've already discussed: indexed addressing and indirect addressing. If you re-read those sections now, you'll remember that indexed addressing required us to add a value to the operand to get the effective address for our data, and indirect addressing required us to go to the address of the first of the two memory locations that contained the data, rather than acting on the data itself.

By combining these two, we can use a two-byte instruction to access all 64K of memory in the C16. Swings and roundabouts of course, since this command now takes 6 cycles to operate rather than the usual 3 or 4 of the ordinary zero page indexed and absolute indexed.

This is how it works. The value in the X register is added to the zero page operand specified in the instruction, to produce an indirect zero page address. So far, fairly similar to our ordinary indexed addressing mode. Then, and this is where the indirect part comes in, the effective memory address comes from the first byte of the indirect zero page address (the low-order byte) and the second byte of the indirect zero page address (the high-order byte). The content of this effective address is then stored in the accumulator.

Let's take a look at an example:

LDA ($1B,X)

If the X register contains, say, $40, then the 7501 will compute an address of $5B ($1B offset with $40). It then looks at location $5B to get the low-order address and $5C to get the high-order address of the effective memory address. If $5B contained $21 and $5C contained $38, then the effective memory address would be $3821. The content of $3821 would then be loaded into the accumulator.

It's best to play around with scraps of paper and little diagrams to figure out this one!

Indirect indexed addressing

If you thought that one was bad...

This combines the same two addressing modes as indexed indirect, but uses them in reverse order. For example:

LDA ($1B),Y

Say the Y register contained $40. The base address would be fetched from location $1B (low-order) and $1C (high-order). So if $1B contained $21, and $1C contained $38, then the base address would be $3821. Our effective address is found by adding the content of the Y register to that address, to give an effective address of $3861. The content of that address is then stored in the accumulator.

Note that these last modes are the only real difference between the X and the Y registers. You have to use the X register with indexed indirect, and you have to use the Y register with indirect indexed.

Relative addressing

This is the mode used by all the branching instructions, some of which we've already seen, such as BNE, BEQ and BPL. The address to transfer program execution to (or, to put it another way, the value that is to be stored in the program counter), depends on the operand after the op-code.

To give an example:

BNE $05

will cause program execution to continue 5 bytes further on if the result of the last operation was not equal to zero. The command:

BNE $F9

would cause program execution to branch backwards 6 bytes. As we saw earlier on, we can only branch forward a maximum of 127 bytes, and backwards a maximum of 128 bytes, since all these branching instructions have a single byte for their operand.

Accumulator addressing

This is a collection of four instructions that affect the content of the accumulator, and we'll be looking at all of them later.

The scroll routines

To break things up a little, a short routine. Well, two of them in fact, occupying locations $3800 to $382E and $3830 to $385E. The first one is used to scroll the screen to the left, and the second scrolls it to the right. Some interesting techniques are involved in these two routines, although both of them are really quite similar. We'll take the first one and go through that fairly exhaustively. By then you should be able to follow the second one: it uses much the same logic.

```
B*
   PC  SR AC XR YR SP
.;7FC5 33 00 AD 00 F6
.
3800 A9 28       LDA #$28
3802 A2 18       LDX #$18
3804 85 57       STA $57
3806 A9 0C       LDA #$0C
3808 85 58       STA $58
380A A0 00       LDY #$00
380C B1 57       LDA ($57),Y
380E 85 59       STA $59
3810 C8          INY
3811 B1 57       LDA ($57),Y
3813 88          DEY
3814 91 57       STA ($57),Y
3816 C8          INY
```

```
3817 98            TYA
3818 C9 27         CMP #$27
381A D0 F4         BNE $3810
381C A5 59         LDA $59
381E 91 57         STA ($57),Y
3820 A5 57         LDA $57
3822 18            CLC
3823 69 28         ADC #$28
3825 85 57         STA $57
3827 90 02         BCC $382B
3829 E6 58         INC $58
382B CA            DEX
382C D0 DC         BNE $380A
382E 60            RTS
382F 00            BRK
3830 A9 28         LDA #$28
3832 A2 18         LDX #$18
3834 85 57         STA $57
3836 A9 0C         LDA #$0C
3838 85 58         STA $58
383A A0 27         LDY #$27
383C B1 57         LDA ($57),Y
383E 85 59         STA $59
3840 88            DEY
3841 B1 57         LDA ($57),Y
3843 C8            INY
3844 91 57         STA ($57),Y
3846 88            DEY
3847 98            TYA
3848 C9 00         CMP #$00
384A D0 F4         BNE $3840
384C A5 59         LDA $59
384E 91 57         STA ($57),Y
3850 A5 57         LDA $57
3852 18            CLC
3853 69 28         ADC #$28
3855 85 57         STA $57
3857 90 02         BCC $385B
3859 E6 58         INC $58
385B CA            DEX
385C D0 DC         BNE $383A
385E 60            RTS
385F 00            BRK
3860 D0 C1         BNE $3823
3862 60            RTS
3863 00            BRK
.
.
```

First, load the accumulator with $28 (or decimal 40, the number of characters in a screen line) and load the X register with $18 (or decimal 24, since we're only going to scroll the bottom 24 lines of the screen and leave the top line stationary for any messages you might want to put there. Altering this value alters how many lines will be scrolled). Store the accumulator at location $57, the first spare byte in page zero.

The accumulator is then loaded with the value $0C, and stored at location $58, the second spare value. Loading the Y register with a zero and loading the accumulator with the content of location $57 offset with Y has the effect of putting a $0C (decimal 12) in the accumulator the first time around the loop, and this value is then stored in location $59. The program then looks at every screen memory location in turn, and stores whatever happens to be there in the location immediately to the left of it. Remember the value $0C which we originally loaded into the accumulator at $3806? $0C00 happens to be the start of the screen memory, and so what is happening here is that in the order low-byte high-byte everything is successively moved one location to the left.

At location $3823 $28 (or decimal 40) is added to the value held in the accumulator, which has the effect of stepping us down onto the next screen line. At locations $382B to $382D we check to see if all 24 lines on the screen have been covered. If they have, then return from this routine, and if they haven't, go back and move another line along one character.

To test this, you might try the following short program:

```
FORI=1TO40:SYS14336:NEXT
```

and watch what happens.

Logical operators and the accumulator

In machine code, the equivalent of commands like 'greater than', 'less than', and so on, have to be broken down into the result of a comparison being either positive or negative.

However, AND and OR operate in exactly the same way as our examples with those binary numbers quoted in the early chapters of this book. The truth tables function in the same way, and if you're having problems grasping what the result of an AND or an OR should be, then the tables should be able to sort things out for you.

There are another two machine-code expressions for logically comparing numbers, and the first that we'll look at is the 'exclusive or' function. It is probably the most difficult to understand at first, and as with the other two the only way to get a proper grasp of what it's really doing is to look at the truth table.

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

Thus, if A is false and B is false, then C will also be false. If A is false and B is true, then C will true. Similarly, if A is true but B is false, C will again be true. Finally, if both A and B are true, then C will be false. In other words, C will only be true if A and B are different, i.e. one is true and the other is false.

The fourth command that allows us to compare whatever happens to be with the accumulator with a number is the BIT command, which operates in much the same way as the AND command, but with one major difference. ANDing something with the accumulator alters the value kept in the accumulator, just as ORing or EORing (Exclusive OR) something does. BIT, however, leaves the accumulator and whatever you're BITting it with unaltered, but does affect the flags in the status register.

Thus, the following will happen:

The Z flag is set if the result of the equivalent AND instruction would have been zero.

The N flag is loaded with the original contents of the seventh bit being tested.

The V flag is loaded with the original contents of the sixth bit being tested.

All these logical operators really come into their own when talking about binary arithmetic and multiplication, but that must wait a little.

Now for the four instructions that can shift each individual bit in the accumulator, or in a memory location, to the left or the right.

Rotating and shifting

For all these four instructions, we have also got to consider the carry flag, since it effectively acts as the 'ninth bit' for the operand, or the number that we're working on. In either of the two shift operations, the bit position at the opposite end of the byte from the one that gets moved off is reset to zero. In the two rotate operations, the bit position at the opposite end of the byte from the one that gets moved off is given the value that was stored in the carry flag before the operation took place. The symbols used to represent these instructions are as follows:

ASL : Accumulator Shift Left
LSR : Logical Shift Right
ROL : ROtate Left
ROR : ROtate Right

These operations can work either on a location in memory or on the accumulator. They can also be used in any of the four addressing modes: absolute, zero page, zero page offset with X, and absolute offset with X.

As well as affecting the carry flag, which we'll illustrate with a diagram in a moment, these four instructions affect two of the flags in the status register.

ASL, ROL and ROR cause the negative (N) flag to be set if bit 7 of the shifted result is set to a 1, otherwise it is reset to a zero.

LSR always causes the negative flag to be reset, as it always puts a zero into bit 7.

If the shifted result is zero, then the zero (Z) flag is set, otherwise it is reset to 1.

All of this is probably best illustrated by a diagram. We'll look at the decimal number 84 (binary 01010100), and say that the carry flag has been set to 1.

Carry Flag	Bit Position 7 6 5 4 3 2 1 0	
1	0 1 0 1 0 1 0 0	Before shift (decimal 84)
0	1 0 1 0 1 0 0 0	After ASL (decimal 168)
0	0 0 1 0 1 0 1 0	After LSR (decimal 42)
0	1 0 1 0 1 0 0 1	After ROL (decimal 169)
0	1 0 1 0 1 0 1 0	After ROR (decimal 170)

A quick glance at the numbers in the decimal column after these operations will reveal a number of interesting things. Performing a shifted left or right will either double or halve the number under consideration, which can be useful in many kinds of arithmetic, as well as in certain kinds of sort routines. With a little bit of logical thought, you should be able to see what kind of powerful uses we could put these instructions to, and we'll be looking at some of them later.

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:

```
3800 :   LDA #$01 :      Load the accumulator with 1.
3802 :   JSR $F265 :     Jump to internal subroutine to check
                         stop key.
3805 :   TAX :           Transfer contents of accumulator into
                         X register.
```

Step by step, here's what happens:

(1) Find address of next instruction, i.e. 3802, and put this in the program counter.

(2) Execute instruction, and get back address for next one from program counter (i.e. 3802)

(3) Fetch next instruction, i.e. JSR $F265

(4) Find address for next instruction (3805), and put this onto stack.

(5) Put next vacant position (509, as 3805 occupies two bytes) into stack pointer.

(6) Put F265 into program counter, and jump to subroutine at $F265.

(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. 3805) and put it into program counter.

(9) Go and execute instruction at 3805.

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 just about everything, and two of the commands which do this concern the accumulator. These are:

PHA : Push contents of accumulator onto stack, but don't change the value in the accumulator.
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.

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.

Let's put this into a program.

Example program

Our code could look something like this:

```
. 3800  18         CLC
. 3801  D8         CLD
. 3802  A9  86     LDA  #$86
. 3804  69  86     ADC  #$86
. 3806  8D  02 0C  STA  $0C02
. 3809  A9  07     LDA  #$07
. 380B  69  07     ADC  #$07
. 380D  8D  00 0C  STA  $0C00
. 3810  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 14336 (since this is the decimal equivalent of 3800), 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 7501 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 $0C02.
Load the accumulator with $07.
Add with carry $07.
Store the result in memory location $0C00.

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.

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 (some we've seen already, some we haven't), and these are:

CPX : ComPare the contents of a memory location with the contents of the X register.

Syntax : CPX $0C04

CPY : ComPare the contents of a memory location with the contents of the Y register.

Syntax : CPY $0C02

CMP : CoMPare the contents of a memory location with the contents of the accumulator.

Syntax : CMP $0C00

When encountering the compare command, for example CPX $0C04, the program reads the contents of memory location $0C04, 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:

```
. 3800  A9  53      LDA  #$53
. 3802  8D  04  0C  STA  $0C04
. 3805  A2  01      LDX  #$01
. 3807  E8          INX
. 3808  EC  04  0C  CPX  $0C04
. 380B  F0  03      BEQ  $3810
. 380D  4C  07  38  JMP  $3807
. 3810  8E  00  0C  STX  $0C00
. 3813  60          RTS
```

This program loads a 'heart' symbol into the accumulator, then stores it at memory location $0C04 (the screen). The X register is then loaded with a 1, and incremented.

Next, we compare location $0C04 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 appears to be heading) for machine code programming on the C16 is to produce some amazing games.

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 ½ lines worth.

You may have to change the background colour to be able to see them, but the point to note is the sheer speed with which things happen.

```
. 3800  A9  53      LDA  #$53
. 3802  A2  01      LDX  #$01
. 3804  E8          INX
. 3805  F0  06      BEQ  $380D
. 3807  9D  00  0C  STA  $0C00,X
. 380A  40  04  38  JMP  $3804
. 380D  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 $0C00, offset with X, and we jump back to increase X again.

You may have been puzzled by the different natures of 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.

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!

Later on we'll look at program timing, and having said very early on that one of the chief advantages of machine code is its speed of operation, we'll also look at ways of slowing programs down!

```
. 3800  A9  53     LDA  #$53
. 3802  A2  00     LDX  #$00
. 3804  A0  20     LDY  #$20
. 3806  8C  25 38  STY  $3825
. 3809  8D  26 38  STA  $3826
. 380C  9D  00 0C  STA  $0C00,X
. 380F  98         TYA
. 3810  9D  FF 0B  STA  $OBFF,X
. 3813  E8         INX
. 3814  EA         NOP
. 3815  EA         NOP
. 3816  EA         NOP
. 3817  EA         NOP
. 3818  EA         NOP
. 3819  EA         NOP
. 3820  D0  ED     BNE  $380C
. 3822  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 $3825, and the contents of the accumulator in $3826: both safely out of the way.

The accumulator contents are then stored at location $0C00, offset with X, and the contents of the Y register transferred to the accumulator and then stored at $0BFF (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

All the mnemonics used in machine code are listed for you in Appendix 4 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 INX 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 goes back and increments 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 7501 for precise timing of instruments, program control or whatever, it is obviously better to use this timer than the jiffy clock. Also, every microsecond counts: some operations take longer under different circumstances.

Charget and the interrupt

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 the C16: in fact it sits in locations 1139 to 1156. 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, so 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 remembered in the accumulator, and 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 1139 to 1156, allows you to do this, but be careful: charget is operating all the time, so any changes will have to make sense to it.

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 7501 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 7501 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 C16 makes a good attempt at keeping everything as it was before the interrupt happened.

BRK: BReaK.

This starts up another interrupt sequence, and when used 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.

More mathematical operations

Multiplication

We saw earlier on how to handle addition and subtraction in machine code, and you could be forgiven for thinking that, just like the commands SBC and ADC for those mathematical operations, there would be a couple of commands to handle multiplication and division.

Unfortunately there aren't, and so multiplication has to be handled as a series of additions. To multiply, say, 4 by 5, you have to perform the calculation 4+4+4+4+4. In other words, add 4 to 0 five times, which can easily be achieved by a simple looping procedure.

For example:

```
3800   A0  05        LDY    #$05
3802   A9  00        LDA    #$00
3804   69  04        ADC    #$04
3806   88            DEY
3807   D0  FB        BNE    $3804
3809   8D  00   0C   STA    $0C00
380C   A9  00        LDA    #$00
380E   8D  00   08   STA    $0800
3811   60            RTS
```

Step by step then:

Load the Y register with 5.

Load the accumulator with 0.

Add 4 to the contents of the accumulator.

Decrement the Y register.

If the result of decrementing the Y register is not equal to zero then branch back and add another 4 to the accumulator.

Store the accumulator at memory location $0C00 (the top left-hand corner of the screen).

Load the accumulator with 0.

Store it at the colour memory location that handles the top left-hand corner of the screen.

Return from this subroutine.

Now this is all very well if we don't wish to add up numbers greater than 255. If we do, however, the accumulator will keep flipping back to zero and a carry will be registered each time our sum exceeds 255, and thus 256 will be lost from our answer each time that happens. One way to check for this is, if a carry is set, to add one to the memory location that will be handling the high order value of the number, which can be done using the INC command. The INC command adds one to the contents of a specified memory location. If the answer is going to exceed 256 times 256, or 65536, then yet another memory location will be needed to check for another carry. For now, we'll stick to smaller numbers.

This small program multiplies 25 by 24, and displays the result on the screen. Of course, since the screen codes used for displaying characters are never the same as the character to be represented (that is, the number 16 doesn't have a screen code of 16), instead of numbers being printed up we get letters appearing instead. Oh well.

```
3800    A2  00      LDX    #$00
3802    8E  25  38  STX    $3825
3805    A2  19      LDX    #$19
3807    A9  00      LDA    #$00
3809    18          CLC
380A    69  18      ADC    #$18
380C    90  03      BCC    $3811
380E    EE  25  38  INC    $3825
3811    D0  F6      BNE    $3809
3813    8D  02  0C  STA    $0C02
3816    AD  25  38  LDA    $C025
3819    8D  00  0C  STA    $0C00
381C    A9  00      LDA    #$00
381E    8D  02  08  STA    $0802
3821    8D  00  08  STA    $0802
3824    60          RTS
```

As usual, we'll go through this one instruction at a time.

Load the X register with a zero.

Store it in location $3825, which is where we'll handle the high-order value of the result of multiplying 25 by 24.

Load the X register with $19, or decimal 25, which is the number of times we'll perform this addition.

Load the accumulator with zero to get it ready for receiving the low-order value.

Clear the carry flag before addition.

Add 24 to the value held in the accumulator.

If a carry isn't set, then we Branch on Carry Clear forward three locations.

Otherwise, increment the high order value at $3825.

Decrease the value in the X register by one.

If the result of that isn't equal to zero, then branch back and add another 24 to the accumulator.

Finally, store the result on the screen by placing the value in the accumulator at location $0C02, picking up the high-order value from $3825, and placing that at $0C00. Put some colour into the locations and retreat from this routine.

In the above example, it doesn't really matter whether we multiply the numbers in the order 25 by 24, or 24 by 25, since we'll still be making roughly the same number of passes around our main routine. However, if you wanted to multiply 5 by 1000 it would obviously be a lot quicker to do it one way than another. You'd probably have to write a short routine to make sure that the quickest route was taken, although the time required to perform that routine might well outweigh the advantages of running it in the first place.

Division

Just as multiplication has to be treated as a series of additions, so division has to be treated as a series of subtractions. In the multiplication program above, we had to clear the carry flag before adding the numbers up. With subtraction, we have to set the carry flag before subtracting them. In the following example, we'll divide 86 by 4.

```
3800    A0  00      LDY    #$00
3802    A2  04      LDX    #$04
3804    AE  40  38  STX    $3840
3807    A9  56      LDA    #$56
3809    38          SEC
380A    E9  04      SBC    #$04
380C    C8          INY
380D    CD  40  38  CMP    $3840
3810    B0  F7      BCS    $3809
3812    8C  00  0C  STY    $0C00
3815    8D  02  0C  STA    $0C02
3818    A9  00      LDA    #$00
381A    8D  00  08  STA    $0800
381D    8D  02  08  STA    $0802
3820    60          RTS
```

One line at a time:

Load the Y register with a zero.

Load the X register with 4, since this is what we want to divide by.

Store the X register at location $3840, so we can keep an eye on it.

Load the accumulator with $56, or decimal 86, since this is what we want to divide.

Set the carry flag before subtraction.

Subtract 4 from the value held in the accumulator.

Increment Y, since this is holding the quotient.

Compare the accumulator with the value previously stored at $3840.

If the carry flag is still set then branch back and take four away again.

It isn't set, so we can store the quotient on the screen, and the remainder is the value held in the accumulator. We finally colour our result on the screen in black so that you can see it, and then exit from the program.

There is a further way of carrying out mathematical operations, which involves something known as Binary Coded Decimal arithmetic, or BCD for short, which is switched on or off with the commands:

SED : SEt Decimal mode of operation.

CLD : CLear the Decimal flag.

You may recall the decimal flag (D) from our earlier discussion on the status register. When this flag is set, the 7501 is all ready to handle BCD. You must clear the flag after you've finished, otherwise dire consequences will ensue.

BCD is a strange animal. The main difference between this and ordinary arithmetic, is that a carry occurs after each half-byte (or four bits) exceeds 9. What possible use is this, you may wonder. Well, for one thing this notation acts as a link between binary (which the machine likes to work in) and decimal (which we like to work in). For instance, if the answer to an addition sum was decimal 23, in ordinary binary this would have been stored as:

0 0 0 1 0 1 1 1

In other words, 23. However, in BCD this is looked at not as one byte, but as two nibbles, like this:

0 0 0 1   0 1 1 1

Which represents 1 times 10 from the left hand nibble, plus 7 from the right hand nibble, which of course equals 17.

Binary multiplication

We've already seen that there are a number of commands available to us for manipulating the bits within a byte: ROL, ASL and so on. Hence it would seem to make sense to use these commands in some kind of sensible fashion. One of their most powerful uses comes into effect when we consider the topic of binary multiplication. Up until now, we've been handling our arithmetical calculations in the kind of way that we've been used to doing ever since leaving school. However, the 7501 doesn't particularly like this manner of dealing with numbers, and so the techniques involved for handling multiplication and division in a conventional way can, at times, get very complicated.

The 7501 prefers to work in binary, and using the aforementioned shifting and rotating instructions, arithmetical operations become much more straightforward.

Let us consider the way we would normally multiply two numbers together, say 123 and 89.

```
  123
   89
--------
 1107     - known as partial product 1.
0984      - known as partial product 2.
--------
10947
--------
```

You can see that, as we multiply by each number, we move along one row of digits. More correctly, we are increasing the power of ten by 1 each one each time we produce a new partial product. Binary arithmetic follows much the same rules, and using the same numbers our sum now becomes, in binary:

```
       01111011 - 123
       01011001 - 89
---------------
       01111011
      00000000
     00000000
    01111011
   01111011
  00000000
 01111011
00000000
---------------
010101011000011 - 10947
```

The only difference here is that each time we move the partial product to the left, we're increasing by a power of two, not a power of ten as in ordinary decimal arithmetic.

However, there are two fundamental differences in the way that the computer performs this operation and the way that we've just done it.

(1) It keeps a running total as it goes along, which is updated after each partial product has been calculated, whereas 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 power of ten or two depending on what system we're working in. On the 7501 it's far easier to rotate the current partial product one row to the left (as we would) or to the right. This latter option is known as high order to low order multiplication.

Such sums as the one we covered above come under the heading of 16 bit multiplication, since the answer is a number way in excess of 255. For the time being we'll just look at 8 bit multiplication (since it's easier to understand!) and multiply together the two numbers 11 and 8.

```
3800   A2  08       LDX   #$08
3802   A0  0B       LDY   #$0B
3804   8E  40  38   STX   $3840
3807   8C  41  38   STY   $3841
380A   A0  08       LDY   #$08
380C   A9  00       LDA   #$00
380D   18           CLC
380E   4E  41  38   LSR   $3841
3811   90  04       BCC   $3818
3814   18           CLC
3815   6D  40  38   ADC   $3840
3818   0E  40  38   ASL   $3840
381B   88           DEY
381C   D0  F0       BNE   $380E
381E   8D  00  0C   STA   $0C00
3821   A9  00       LDA   #$00
3823   8D  00  08   STA   $0800
3826   60           RTS
```

In our usual way, we'll examine this listing one line at a time.

In the first line, we load the X register with an 8.

Then, load the Y register with $0B, or decimal 11. These are the two numbers that we'll be multiplying.

Store the X register at a 'safe' location, $3840.

Ditto for the Y register, at location $3841.

Load the Y register with an 8. Since we're dealing with 8 bit numbers only, we're going to pass through this loop 8 times. Unfortunately, 16 bit numbers cannot be handled purely by loading this register with a 16, as we shall see.

Load the accumulator with a zero.

Clear the carry flag.

Shift the content of location $3841 logically one place to the right (re-read the section on the shift and rotate commands if you've forgotten the result of doing this).

If no carry has been set, then branch forward four bytes.

Clear the carry flag again.

Add to the accumulator, with carry, the content of memory location $3840.

Shift every bit in the accumulator one bit to the left.

Decrease the Y register.

If we haven't done all 8 passes, then switch back to location $380E.

We have finished, so store the result on the screen and give it some colour so that we can actually see it.

Unfortunately this routine will not work on two numbers that, when multiplied together, give an answer that is greater than 255. The reason for this lies in the way that the ASL command works. Each time we progress through the main program loop, every bit is shifted one bit to the right, and by the eighth time around the bit that originally started off at the right-hand edge of the byte will have 'fallen off' the left-hand edge.

```
THEN J=J+40
3008 IF W$="R"THENJ=J+1:IF INT((J-40)/40)=(J-40)/4
0 THEN J=J-40
3009 IF PEEK(3072+J)=46 THEN POKE 3072+J,42:POKE 2
048+J,2:S=0:RETURN
3010 GOTO 3500
3012 PRINT"[CLR,BLK]GRRR...":H=H+1:GOTO 4000
3500 P$=W$
3501 IFP$="U"THENW$="D":J=Q:GOTO3002
3502 IFP$="D"THENW$="L":J=Q:GOTO3002
3503 IFP$="L"THENW$="R":J=Q:GOTO3002
3504 IFP$="R"THENS=S+1
3506 IFS=3THENS=0:GOTO3012
3507 W$="U":J=Q:GOTO3002
4000 REM
4005 FORI=1TO10:GETF$:NEXT
4010 PRINT"[2CD]SCORE NOW STANDS AT YOU [RVS]";H:P
RINT"AND THE COMPUTER [RVS]";C"
4020 PRINT"[CD]ANOTHER GAME (Y OR N)"
4030 GET F$:IFF$="" THEN 4030
4040 IF F$="Y"THENGOSUB5070:GOTO10
4050 IFF$="N" THEN PRINT"[CLR]BYE ...":END
4060 GOTO 4030
5000 PRINT"[CLR,BLK]WELCOME TO THE GAME OF TRAP!"
5010 PRINT"[CD]THE OBJECT OF THE GAME IS TO TRAP T
HE "
5020 PRINT"COMPUTER SO THAT IT CAN'T MOVE"
5030 PRINT"[CD]OF COURSE, IT IS TRYING TO DO THE S
AME"
5040 PRINT"TO YOU!!"
5050 PRINT"[CD]PRESS M TO MOVE DOWN, A LEFT, D RIG
HT, "
5060 PRINT"AND I UP"
5070 PRINT"[CD]DO YOU WANT A FAST, MEDIUM OR SLOW
GAME"
5080 PRINT"PRESS F, M, OR S"
5090 GETD$: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:POKE4071,46:POKE 3047,0
5140 RETURN
```

Quick wits are required as you try to block the computer in. Unfortunately, it's trying to do the same to you as well, so watch out.

Arrow

```
90 COLOR 0,8,7:COLOR 4,3,4
100 PRINT"[CLR,BLK,CD,RVS]ARROW  "
110 PRINT"[CD]INSTRUCTIONS (Y OR N)?"
112 GETZ$:IFZ$="Y"THEN115
113 IFZ$="N"THEN190
114 GOTO112
115 PRINT"[CD]OKAY, THEN "
120 PRINT"[CD]GUIDE THE MOVING 'SNAKE' WITH THE CU
RSORMOVEMENT KEYS.
140 PRINT"[CD]DON'T HIT THE BOUNDARY (OR YOURSELF)
;"
150 PRINT"..TRY TO HIT THE BOXES FOR POINTS."
160 PRINT"[CD]YOU HAVE 60 SECONDS OF PLAY. GOOD LU
CK!"
170 PRINT"[CD,RVS]HIT ANY KEY TO START"
180 GETZ$:IFZ$=""GOTO180
190 DIMP(255),D(3),V(8),H(8),T(8),R(8):K=.1
200 D(0)=22:D(1)=60:D(2)=62:D(3)=30
210 T9=3072:T6=3599:C1=2048
220 REMSET SCREEN UP
230 PRINT"[CLR,BLK]   SCORE: 0":PRINT"A"
240 FORJ=0TO81:IFPEEK(T9+J)<>1THENNEXTJ
250 L=J:FORJ=T9+LTOT9+2*L-1:POKEJ,81:POKEJ+23*L,81
:POKEJ-1024,2:POKEJ+23*L-1024,2
255 NEXTJ
260 FORJ=T9+2*LTOT9+24*LSTEPL:POKEJ,81:POKEJ+L-1,8
1:POKEJ-1024,2:POKEJ+L-1-1024,2
265 NEXTJ
270 V=5:H=5:V1=0:H1=1:P2=10:D1=2
280 TI$="000000"
290 PRINT"[HOME]";RIGHT$(TI$,2):IFTI>T6GOTO620
300 GETZ$:IFZ$=""GOTO330
303 Z=ASC(Z$):IFZ<>17ANDZ<>157ANDZ<>29ANDZ<>145THE
N330
305 IFZ=17THENZ=0
306 IFZ=157THENZ=1
308 IFZ=29THENZ=2
310 IFZ=145THENZ=3
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:POKEP1-1
024,7
380 POKEP,D(D1):P1=P(R6):IFP1<>0THENPOKEP1,81:POKE
```

```
P1-1024,6
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:POKEP3+H3-1024,4
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:POKEH3-1024,7:NEXTH3,V3:V(V%)=0:POKE
R(V%),32:POKER(V%)-1024,7:GOTO290
600 FORJ=1TO1000:NEXT
620 PRINT"[HOME,CD,RVS,BLK]ANOTHER GAME?[OFF] (Y O
R N)?
625 REM
630 GETZ$:IFZ$=""GOTO630
640 IFZ$="Y"THENCLR:GOTO190
650 IFZ$<>"N"GOTO630
660 PRINT"[CLR]BYE ...":END
```

Guide the moving snake around the screen to bump into boxes and score points. But mind you don't bump into yourself, because the snake grows the longer the game progresses (and the more points you get).

Response

```
5 VOL8:POKE1344,0
10 COLOR 0,8,7:COLOR 4,9,5:PRINTCHR$(14):PRINT"[CL
R,BLK]REACTION TIMER"
20 PRINT"[14CBMU]"
30 PRINT"[CD]A SIMPLE GAME TO TEST YOUR REACTIONS
...
40 PRINT"[CD]A BOX WILL BE DISPLAYED IN THE MIDDLE
45 PRINT"OF THE SCREEN.  IT'LL START OFF WHITE,
50 PRINT"AND WHEN IT IS FILLED IN RED YOU MUST
```

```
55 PRINT"PRESS ANY KEY AS FAST AS YOU CAN.
60 PRINT"[CD]YOU'LL HAVE TEN GOES, AND THEN THE
65 PRINT"RESULTS WILL BE DISPLAYED.
70 GOSUB2000
80 PRINT"[CLR]PRESS ANY KEY WHEN YOU'RE READY TO
   START."
81 SOUND1,0,0
85 POKE239,0
86 GETB$:IFB$=""THEN86
87 POKE239,0
89 J=J+1
90 PRINT"[CLR]ATTEMPT NUMBER "J
91 PRINT"[18CBMU]"
92 PRINT"[10CD]"
93 D$="[CBMN,CU,6CBMP,CD,CBMH,7CL,CD,6CBMY]
94 B$="[CBMN,CU,CL,CBMN,CU,3CBMP,CD,CBMH,CD,CL,CBM
H,CD,4CL,3CBMY]"
95 B$=B$+"[2CU,3CL,RVS,RED,3SP,CD,3CL,3SP,OFF,BLK]
"
96 B$=B$+"[2CU,3CL,RVS,WHT,3SP,CD,3CL,3SP,OFF,BLK]
"
98 POKE239,0:POKE198,64
99 SOUND1,400,3120
100 PRINTTAB(17)B$
110 FORI=1TOINT(RND(.5)*30+10)
112 IFPEEK(198)<>64THENPRINT"[2CD]JUMPED THE GUN!"
:J=J-1:SOUND1,0,0:FORK=1TO2000:NEXTK:GOTO80
113 SOUND2,600,10
114 NEXT:SOUND1,0,0:SOUND2,0,0:PRINT"[CU]"TAB(17)C
$
115 TI$="000000":D=TI
125 GETA$:IFA$=""THEN125
130 D1=D1+TI-D:D1$=STR$(D1/(J*60)*1000)
131 PRINT"[2CD]AVERAGE TIME ="D1$:PRINT"THOUSANDTH
S OF A SEC."
132 FORM=1TO1000:NEXT
135 IFJ=10THEN140
136 GOTO87
140 PRINT"[CLR]AVERAGE RESPONSE TIME = ";D1/600*10
00
142 PRINT"THOUSANDS OF A SECOND
144 PRINT"[2CD]ANOTHER GO (Y OR N)?"
146 GETAG$:IFAG$="Y"THENRUN
147 IFAG$="N"THENPRINT"[CLR]BYE ...":END
148 GOTO146
2000 PRINT"[CD]PRESS 'SPACE' TO CONTINUE.
2002 GETA$:IFA$<>" "THEN2002
2003 FORI=1TO10:GETA$:NEXT
2004 RETURN
```

Just a short program to work out your reaction time in thousands of a second. I got down to about 195 thousands of a second, on average, so perhaps you can do better?

Conclusion

None of these programs should take you too long to type in but, as with all listings from books or magazines, do save the program before running it. I will not be responsible for hair-loss if you tear it all out while watching a program eliminate itself before your very eyes when you haven't got a backup copy of it.

If the thought of typing them all in puts you off, however, you may be pleased to hear that a cassette of all seven programs is available direct from the publishers. Also on the tape are the main listings from Chapters 7, 8, 9, 10, 11 and 12. Some sixteen prorams in all for the C16 (good job this book isn't for the C128!).

None of these games are what you might call finished products, and all have plenty of room left in memory for you to improve and enhance them. That way you'll learn an awful lot more about the C16 than merely sitting down and typing in a bunch of listings. Who knows, you might change them so much that you could send them off to a magazine and get paid for them!

14

Peripherals

Cassette

The humble cassette deck is the storage device that most people will be using with their C16, 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 C16 without relocating it.

(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.

Disk

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
```

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

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.

```
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.

Okay, that's some simple stuff out of the way, let's start getting a little more complicated!

Peripheral support

There are a number of different ways in which data can be stored on tape or disk, and the first file type is one that we've already seen, the simple program:

```
10 PRINT "[CLR]HELLO THERE, I'M A C16."
```

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 cassette 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 monitor requires a totally different syntax, as we saw in Chapter 5.

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 exactly the same as usual, 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 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

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.

When operating with sequential files either on tape or on disk, it is important to remember a few rules.

Principal among 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 see how to avoid them.

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
56 REM ONE 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!
```

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:A(I)=I
55 REM READ IN EACH ITEM OF DATA, AND STORE IT
56 REM IN THE ARRAY A(
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 to use instead of INPUT is 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.

When using disk drives, the same sort of syntax is required, and the following program, heavily REMmed to make it easy to follow, 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 COLOR 0,8,7:COLOR 4,10,4:PRINTCHR$(14):PRINT"[C
LR,BLK]"
40 DIM A$(25),B(25)
45 OPEN15,8,15:REM OPEN COMMAND CHANNEL
```

```
50 PRINT#15,"IO":REM INITIALISE DRIVE
55 GOSUB245:REM CHECK ERROR CHANNEL
60 CR$=CHR$(13)
65 PRINT"[RVS]WRITE TEST FILE"
95 OPEN2,8,2,"@0:TEST FILE,S,W":REM OPEN FILE
105 GOSUB245
110 INPUT"A$,B";A$,B:REM GET DATA FROM USER
115 IFA$="END"THEN135
120 PRINT#2,A$","STR$(B)CR$;:REM WRITE DATA
125 GOSUB245
130 GOTO110
135 CLOSE2:REM CLOSE CHANNEL
170 OPEN2,8,2,"0:TEST FILE,S,R":REM READ FILE
175 GOSUB245
180 INPUT#2,A$(I),B(I):REM INPUT DATA AGAIN
185 RS=ST:REM STORE DISK STATUS
190 GOSUB245
195 PRINTA$(I),B(I)
200 IFRS=64THEN220:REM CHECK END OF FILE
205 IFRS<>0THEN230:REM CHECK FOR ERROR IN FILE STA
TUS
210 I=I+1
215 GOTO180
220 CLOSE2
225 END
230 PRINT"[RVS]BAD DISK STATUS IS"RS
235 CLOE2
240 END
245 INPUT#15,EN$,EM$,ES$,ET$:REM GET ERROR
285 IFEN$="00"THENRETURN
290 PRINT"[RVS]ERROR ON DISK ";EM$
295 CLOSE15:CLOSE2
300 END
```

Relative files, and how to write one

This obviously applies only 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.

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 takes 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 characters, 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. 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 in which sector in which track and starting at which character a particular item of information is to go. 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, 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. This makes it easier to keep track of where all the data is, but 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 acces file are as follows:

(1) Open up a route from the C16 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.

Reading 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 C16.
(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 C16, and channel 15 is the route for commands to the ROM in the disk and messages coming back from it.

We must also 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, while the disk drive tidies everything up) it's as well to keep them all open while 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 to open 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 is:

```
TIMM...............DENIS.....01 234 5678.........
```

and the part to go to the buffer (as the keyword need not be part of the direct access record), is stored in the variable A$ thus:

```
A$="DENIS.....01 234 5678"
```

it can 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 is:

```
PRINT#15,"B-P";C;P
```

where C is the data channel number and P is the pointer position required.

So our command is:

```
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 looks like this:

```
PRINT#15,"B-A";0;1;0
```

which attempts to allocate drive 0, track 1, sector 0. Then:

```
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: now do you see why it should be alphabetical? 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$

Conclusion

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!

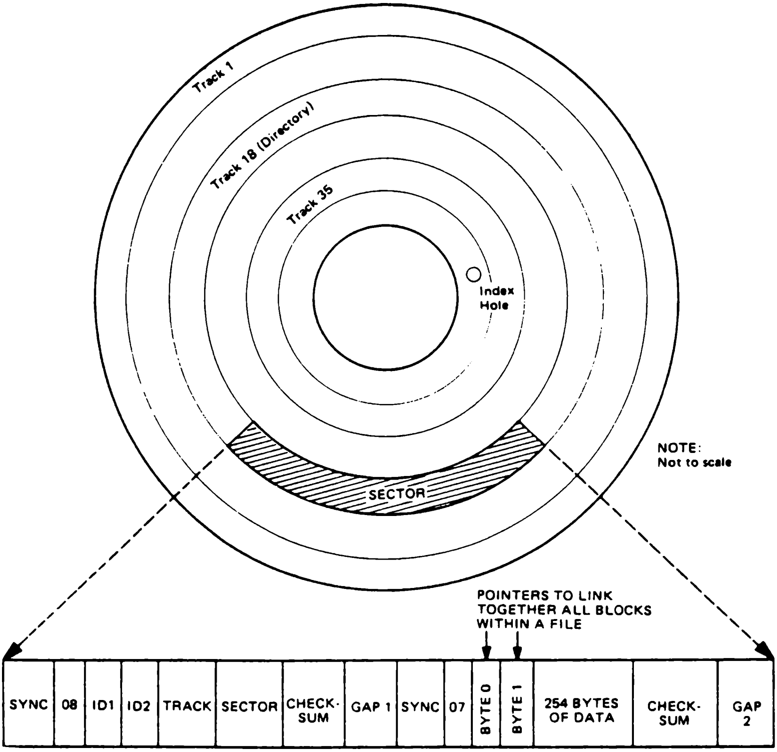
Don't put blank strings as part of the disk information: convert them into shifted spaces or something.

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.

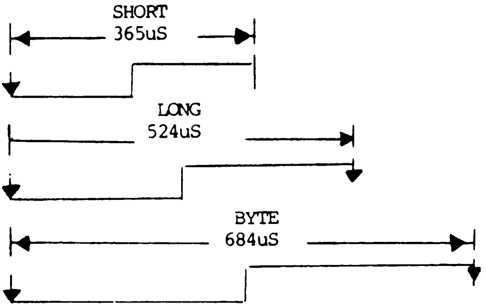
Thanks to Mike-Gross Niklaus for the random access information.

How data is stored on tape

A different matter altogether, as you might imagine.

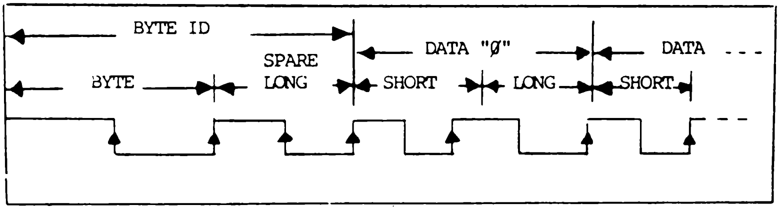
Martin Maynard takes up the story:

"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 C16. The C16 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 C16 tape played through your stereo is quite revealing, and can give an interesting check on how well the cassette deck (the C16 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!

Some disk utilities

A quick couple of utilities for you to play about with.

The first simply reads the name of a disk and the name of every file on it, and store that in an array in memory, while the second is a very short one for telling you how much room is left on a disk. You can get this in direct mode by just using the DIRECTORY command, but try doing it from within a program! These two utilities provide the answer.

```
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
```

```
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
```

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 ones chosen by Commodore. They tend to be slow and extremely noisy, and the quality of the output sometimes leaves a lot to be desired.

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 strongly influences the amount of work done in it. 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 is probably still the Epson FX80. 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 C16 doesn't support that, so somehow you've got to get from one to the other.

Data manipulation

Whatever printer you get, the following few programs should all work quite happily on it. 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

```
10 OPEN6,4,6:PRINT#6,CHR$(18):REM CLOSE LINE SPACING
11 REM THIS MIGHT NOT WORK ON YOUR PRINTER!
15 OPEN 4,4:CMD4
20 FORI=0TO24:FORJ=0TO39
30 A=PEEK(1024+I*40+J)
40 GOSUB60000
50 PRINTA$;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
```

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 voilà! 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 C16 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.

However, 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 C16: 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. Then when you switch from screen to printer you won't be 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 C16 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, or 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);
```

would do. 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 it in to 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)
1095 IF ABS(A1)>1THEN
B$=MID$(B1$,2,LG+1)+"."+RIGHT$(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.

Of course, you could just use PRINT USING!

Looking at a listing

As it stands, the following program 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 rather than in the section on disk drives.

There is another thing to watch for when using this with a printer: the program checks, using PEEK(202), 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(202) 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.

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.

One word of warning: the strange symbol in line 1030 (the third one) is meant to be the up-arrow key.

```
10 DIMA$(126):FORI=0TO126:READA$(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$(0)):IFP=0THENPRINT:GOTO2
5
40 IF(PEEK(203)<>0)OR(P<128)THENPRINTCHR$(P);:GOTO
50
45 PRINTA$(P-128);
50 IF(A$=":"ORA$=",")AND(PEEK(202)>65)THEN65
55 IFPEEK(202)>75THEN65
60 GOTO37
```

```
65 PRINT:PRINTN;:SL=SL+1:GOTO37
500 INPUT#15,EN$,EM$,ES$,ET$:IFEN$="00"THEN RETURN
502 PRINT"*** DISK ERROR *** [RVS]";EM$
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,RGR
1050 DATA RCLR,RLUM,JOY,RDOT,DEC,HEX$,ERR$,INSTR,E
LSE,RESUME,TRAP
1060 DATA TRON,TROFF,SOUND,VOL,AUTO,PUDEF,GRAPHIC,
PAINT,CHAR,BOX,CIRCLE,GSHAPE
1070 DATA SSHAPE,DRAW,LOCATE,COLOR,SCN,CLR,SCALE,H
ELP,DO,LOOP,EXIT,DIRECTORY
1080 DATA DSAVE,DLOAD,HEADER,SCRATCH,COLLECT,COPY,
RENAME,BACKUP,DELETE,KEY
1090 DATA RENUMBER,MONITOR,USING,UNTIL,WHILE
```

Appendix 1
C16 Fundamental memory map

0000	0	Chip directional register
0001	1	Chip I/O & serial bus & tape control
0002	2	Token value of search
0003 - 0006	3 - 4	Renumber parameters
0007	7	Search character for end of line
0008	8	Scan-quotes flag
0009	9	TAB column save
000A	10	Flag; 0 = load; 1 = verify
000B	11	BASIC input buffer pointer/subscript no.
000C	12	Default DIM flag
000D	13	Variable flag; FF = string; 00=numeric
000E	14	Numeric flag: 80 = integer; 00 = floating point
000F	15	Flag; DATA scan; LIST quote; memory
0010	16	Flag; Subscript - FNx

0011	17	Flag; 0 = INPUT; 152 = READ; 64 = GET
0012	18	Flag; ATN sign - comparison evaluation
0013	19	Current I/O prompt flag (1 = prompt off)
0014 - 0015	20 - 21	BASIC stores integer values here
0016	22	Pointer; temporary string stack
0017 - 0018	23 - 24	Last temporary string vector
0019 - 0021	25 - 33	Stack for temporary string vector
0022 - 0025	34 - 37	Utility pointer area
0026 - 002A	38 - 42	Product area for multiplication
002B - 002C	43 - 44	Pointer; start of BASIC program
002D - 002E	45 - 46	Pointer; start of BASIC variables - end of current BASIC program
002F - 0030	47 - 48	Pointer; start of arrays - end of variables
0031 - 0032	49 - 50	Pointer; end of arrays
0033 - 0034	51 - 52	Pointer; start of string storage (moves down from from top of available memory to arrays and OUT OF MEMORY)
0035 - 0036	53 - 54	Pointer; end of string storage
0037 - 0038	55 - 56	Pointer; to top of current RAM available to BASIC
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	Pointer; current DATA item
0043 - 0044	67 - 68	Vector; jump for INPUT statement
0045 - 0046	69 - 70	Current variable name
0047 - 0048	71 - 72	Current variable address
0049 - 004A	73 - 74	Variable pointer for FOR - NEXT statement
004B - 004C	75 - 76	Y-save; operator-save; BASIC pointer-save
004D	77	Comparison symbol
004E - 004F	78 - 79	Work area; function definition pointer
0050 - 0051	80 - 81	Work area; string descriptor pointer
0052	82	Length of string
0053	83	Garbage collect use
0054 - 0056	84 - 86	Jump vector for functions
0057 - 0060	87 - 96	Numeric work area
0061	97	Accumulator #1; exponent
0062 - 0065	98 - 101	Accumulator #1; mantissa
0066	102	Accumulator #1; sign
0067	103	Series evaluation constant pointer

0068	104	Accumulator #1; hi-order (overflow)
0069 - 006E	105 - 110	Accumulator #2; floating point
006F	111	Sign comparison; Accumulator 1 - Accumulator 2
0070	112	Accumulator #2; lo-order (rounding)
0071 - 0072	113 - 114	Cassette buffer length - series pointer
0073 - 0074	115 - 116	Line increment value for auto
0075	117	10K Hi-res; Flag
0076 - 0078	118 - 123	Misc work values
0079 - 007B	121 - 123	DS$ descriptor
007C - 007D	124 - 125	Top of run-time stack
007E - 007F	126 - 127	Music, tone and volume; temps
0083	131	Current Graphic mode
0084	132	Current colour
0085	133	Multi-colour one
0086	134	Foreground colour
0087	135	Maximum No. of columns
0088	136	Maximum No. of rows
0089	137	Paint-left; Flag
008A	138	Paint-right; Flag
008B	139	Paint-stop if not same as background colour

0090	144	Kernal I/O status word;ST
0091	145	STOP/RVS key; Flag
0092	146	Temp
0093	147	Flag; 0 = Load. 1 = Verify
0094	148	Flag; serial bus
0095	149	Buffered character for serial bus
0096	150	Temp for BASIC
0097	151	No. of open files, index for file table
0098	152	Default input device (normally 0)
0099	153	Default output device (normally 3)
009A	154	Flag; $80 = direct mode, $00 = program mode
009B	155	Tape pass 1 error log
009C	156	Tape pass 2 error log
009F - 00A0	159 - 160	Temp. data area
00A1 - 00A2	160 - 161	Monitor working vector
00A3 - 00A5	163 - 165	Real time jiffy clock
00A6	166	Serial bus use
00A7	167	Read/write byte from tape
00A8	168	Temp for serial routine
00A9	169	Temporary colour vector
00AA	170	Countdown, tape write/bit count

00AB	171	Current filename length
00AC	172	Current logical file number
00AD	173	Current secondary address
00AE	174	Current device number
00AF - 00B0	175 - 176	Pointer; current filename
00B2 - 00B3	178 - 179	I/O start address (low byte, high byte)
00B4 - 00B5	180 - 181	Load RAM base
00B6 - 00B7	182 - 183	Pointer to cassette base
00BA - 00BB	186 - 187	Pointer to data, tape write
00BE - 00BF	190 - 191	Pointer; Bank fetch
00C0 - 00C1	192 - 193	Temp; scrolling
00C2	194	Flag; RVS/OFF
00C3	195	End of line for Input pointer
00C4 - 00C5	196 - 197	Input cursor log (row, column)
00C6	198	Which key: 64 if no key
00C7	199	Flag; INPUT or GET from keyboard
00C8 - 00C9	200 - 201	Pointer; current screen line address
00CA	202	Cursor position on above line
00CB	203	0=direct else programmed
00CC	204	Current screen line length
00CD	205	Current cursor physical line number

00CE	206	Last I/O character
00CF	207	Flag; insert mode
00D0 - 00D7	208 - 215	Used by speech software
00D8 - 00E8	216 - 232	Used by application software
00E9	233	Screen line link table
00EA - 00EB	234 - 235	Screen editor colour
00EC - 00EE	236 - 238	Screen work values
00EF	239	Number of characters in keyboard buffer
00F0	240	Flag; Pause
00F1 - 00F4	241 -244	Monitor; storage
00F5	245	Cassette checksum
00F6	246	Monitor work value
00F7 - 00F8	247 - 248	Cassette work values
00F9	249	DMA control mask
00FA	250	Work byte
00FB	251	Current ROM bank
0100 - 010A	256 - 257	Processor stack area
0200 - 0258	512 - 600	BASIC input / Monitor buffer
0259 - 025C	601 - 604	BASIC storage
025D - 02AC	605 - 684	BASIC/DOS interface area
02AD - 02B0	685 - 688	Current x,y position
02B1 - 02B4	689 - 692	Destination for X and Y coordinates

02B5 - 02CB	693 - 715	Graphics work area
02CC - 02E8	716 - 744	Print using, Graphics work area
02E9	745	Temp screen row number
02EA	746	String length
02EB	747	Trace on =255
02EC - 02EE	748 - 750	Directory work area
02EF	751	Graphics work area
02F0	752	Number of graphics parameters
02F1	753	Parameter relative or absolute (1 - 0)
02F2 - 02F3	754 - 755	Vector; Float-fixed
02F4 - 02F5	756 - 757	Vector; Fixed-float
02F6 - 02FD	758 - 765	Unused
02FE - 02FF	766 - 767	Vector; for function cartridge users
0300 - 0301	768 - 769	Error message link[8686]
0302 - 0303	770 - 771	Warm start link [8712]
0304 - 0305	772 - 773	Crunch Basic Tokens link[8956]
0306 - 0307	774 - 775	Char. list link [8B6E]
0308 - 0309	776 - 777	Char. dispatch[8BD6]
030A - 030B	778 - 779	Token evaluation [9417]
030C - 030D	780 - 781	Crunch hook vector[896A]
030E - 030F	782 - 783	List hook vector [8B88]
0310 - 0311	784 - 785	Execute hook vector[8C8B]

0312 - 0313	786 - 787	Interrupt link[CE42]
0314 - 0315	788 - 789	Hardware interrupt vector [CE0E]
0316 - 0317	790 - 791	BRK interrupt vector [F44C]
0318 - 0319	792 - 793	OPEN vector [EF53]
031A - 031B	794 - 795	CLOSE vector[EE5D]
031C - 031D	796 - 797	Set-input vector [ED18]
031E - 031F	798 - 799	Set-output vector[ED60]
0320 - 0321	800 - 801	Restore I/O vector[EF0C]
0322 - 0323	802 - 803	INPUT vector[EBE8]
0324 - 0325	804 - 805	Output vector[EC4B]
0326 - 0327	806 - 807	Test-STOP vector [F265]
0328 - 0329	808 - 809	GET vector [EBD9]
032A - 032B	810 - 811	Abort I/O vector [EF08]
032C - 032D	812 - 813	User defined vector[F44C]
032E - 032F	814 - 815	LOAD link [F04A]
0330 - 0331	816 - 817	SAVE link [F1A4]
0333 - 03F2	819 - 1010	Cassette buffer
03F3 - 03F4	1011 - 1012	Data length to tape (write)
03F5 - 03F6	1013 - 1014	Data length to tape (read)
03F7 - 0436	1015 - 1078	RS232 input buffer
0437 - 0472	1079 - 1138	Tape error log
0473 - 0478	1139 - 1144	CHRGET
0479 - 0484	1145 - 1156	CHRGOT

04E7 - 04EA	1255 - 1258	PUT characters
04EB - 04EE	1259 - 1262	String work area
04EF - 04F6	1263 - 1270	TRAP and error logs
04F7	1271	Stack pointer for error trap
04F8 - 04FB	1272 - 1275	DO loop work area
04FC - 04FF	1276 - 1279	Sound work area
0500 - 0502	1280 - 1282	USR program jump
0503 - 0508	1283 - 1288	RND seed value
0509 - 0512	1289 - 1298	Logical file number
0513 - 051C	1299 - 1308	Device number
051D - 0526	1309 - 1318	Secondary address
0527 - 0530	1319 - 1328	Keyboard buffer
0531 - 0532	1329 - 1330	Start of memory
0533 - 0534	1331 - 1332	Top of memory
0535 - 0536	1333 - 1334	Timeout/end flags
0537 - 0538	1335 - 1336	Tape buffer counts
0539	1337	Tape buffer pointer
053A	1338	Tape file type
053B	1339	Character (colour) attribute
053C	1340	Flash flag
053D	1341	Unused
053E	1342	Base location of screen top
053F	1343	Keyboard buffer size

0540	1344	Flag; key repeat; 128=all - 64=none
0541 - 0542	1345 - 1346	Key repeat counters
0543	1347	Key shift flag
0544	1348	Last shift pattern
0545 - 0546	1349 - 1350	Keyboard table set-up vector
0548	1352	Flag; Auto scroll
0549 - 054A	1353 - 1354	Screen work values
054B - 055C	1355 - 1372	MLM work locations
055D	1373	FN key pending count
055E	1374	FN key pointer
055F - 05E6	1375 - 1510	Key definition area
05E7 - 05EB	1511 - 1515	DMA work locations
05EC - 06EB	1516 - 1771	Banking routine page
05F0 - 05F1	1520 - 1521	Long jump vector
05F2 - 05F4	1522 - 1524	Long jump registers
05F5 - 06EB	1525 - 1791	Reserved RAM for extra ROMs
06EC - 07AF	1792 - 1967	BASIC pseudo stack
07B0 - 07CC	1968 - 1996	Tape working values
07CD - 07D0	1997 - 2000	RS232 working values
07D1	2001	RS232 in pointer
07D2	2002	RS232 read pointer
07D3	2003	RS232 input counter

07D4 - 07D8	2004 - 2008	RS232 work values
07D9 - 07E4	2009 - 2020	Character load program
07E5	2021	Current screen bottom margin
07E6	2022	Current screen top margin
07E7	2023	Current screen left margin
07E8	2024	Current screen right margin
07E9	2025	0 = scrolling enabled
07EA	2026	255 = auto insert enable
07EB	2027	Previous character printed
07EC - 07ED	2028 - 2029	Current colour attribute
07EE - 07F1	2030 - 2033	Screen line wrap table
07F2	2034	SYS A-reg save
07F3	2035	SYS X-reg save
07F4	2036	SYS Y-reg save
07F5	2037	SYS status reg save
07F6	2038	New key detect
07F7	2039	Lockout CTRL-S
07F8	2040	Monitor read
07F9	2041	Colour decode switch
07FA	2042	Split screen bit mask
07FB	2043	Split screen video base
07FC	2044	Tape motor interlock
0800 - 0BFF	2048 - 3071	Colour memory

0C00 - 0FFF	3072 - 4095	Screen memory
1000	4096	Start of BASIC RAM (in lo-res)
2000	8192	Start of BASIC RAM (in hi-res)
8000 - FFFF	32768 - 65535	BASIC ROM
D000 - D7FF	53248 - 55295	Character sets in ROM
FD10 - FD1F	64784 - 64799	Parallel port-6529
FDD0 - FDDF	64976 - 64991	ROM bank select
FE00 - FEFF	65024 - 65279	DMA disk interface
FF00 - FF1F	65280 - 65311	TED I/O control chip

Appendix 2
C16 ROM memory map

8003; Warm restart

8000; Module or Cold start

8003; Warm restart

8019; Initialize

802E; Initialize Basic

80CF; Power up message

8105; Vectors for $300

8117; Initialize vectors

8123; CHRGET

8155; Error message vectors

818E; Keywords

8383; Keyword Action vectors

841B; Function vectors

8455; Operator vectors

8471; Error messages

867E; Error routine

8681; 'out of memory'

86F1; Break Entry

86FB; 'ready.'

8706; Ready for Basic

872E; Handle new line

8818; Re-chain lines

885A; Receive input line

886E; Scan stack FOR/GOSUB

88C0; Move memory

8905; Check stack depth

8923; Check memory space

8953; Crunch tokens

8A3D; Find new Basic line

8A79; Perform [NEW]

8A98; Perform [CLR]

8AED; Back up text pointer

8AFF; Perform [LIST]
8BBC; Perform [RUN]
8C25; Execute statement
8C9A; Perform [RESTORE]
8CC0; Break
8CD8; Perform [STOP]
8CDA; Perform [END]
8D03; Perform [CONT]
8D2C; Perform [GOSUB]
8D4D; Perform [GOTO]
8D83; Perform [RETURN]
8DB0; Perform [DATA]
8DBE; Scan for next statement
8DE1; Perform [IF]
8E0B; Perform [REM/ELSE]
8E1B; Perform [ON]
8E3E; Get fixed point number
8E7C; Perform [LET]
8FE0; Perform [PRINT#]
8FE6; Perform [CMD]
9000; Perform [PRINT]
9088; Print string from (y.a)
90A6; Print format character
90B8; Perform [GET]
90EE; Perform [INPUT#]
9108; Perform [INPUT]
9142; Prompt and input
914F; Perform [READ]
920B; Input error messages
9294; Perform [NEXT]
9314; Type match check
932C; Evaluate expression
9439; Constant pi
9485; Evaluate within brackets
948B; ')'
9493; comma
94A1; Syntax error
94AD; Search for variable
9561; Set up FN reference
95FB; Perform [OR]
9602; Perform [AND]
9628; Compare
969B; Perform [DIM]
96A5; Locate variable
973A; Check alphabetic
9744; Create variable

985B; Array pointer subroutine
986C; Value 32768
9879; Float-fixed
989B; Set up array
991C; 'bad subscript'
9921; Illegal quantity
9A2F; Compute array size
9A62; Perform [FRE]
9A76; Fix-float
9A7D; Perform [POS]
9A86; Check direct
9A9D; Perform [DEF]
9ACB; Check fn syntax
9ADE; Perform [FN]
9B54; Calculate string vector
9B66; Perform [STR$]
9B74; Set up string
9BDA; Concatenate
9C1B; Build string to memory
9C4B; Discard unwanted string
9C52; Make room for string
9CAA; Clean descriptor stack
9CBB; Perform [CHR$]
9CCF; Perform [LEFT$]
9D03; Perform [RIGHT$]
9D15; Perform [MID$]
9D46; Pull string parameters
9D61; Perform [LEN]
9D67; Exit string mode
9D70; Perform [ASC]
9D81; Input byte parameter
9D93; Perform [VAL]
9DCF; Parameters for POKE/WAIT
9DDE; Parameters for SOUND
9DE4; Float-fixed
9DFA; Perform [PEEK]
9E12; Perform [POKE]
9E1B; Perform [DEC]
9E6A; Perform [WAIT]
9E87; Perform [subtract]
9E9E; Perform [add]
9F7B; Complement FAC#1
9FB2; 'overflow'
9FB7; Multiply by zero byte
A01E; Perform [LOG]
A07B; Perform [multiply]

A0A9; Multiply-a-bit

A0DC; Memory to FAC#2

A107; Adjust FAC#1/2

A154; Underflow/overflow

A162; Multiply by 10

A179; + 10 in floating point

A17E; Divide by 10

A197; Perform [divide]

A21F; Memory to FAC#1

A24C; FAC#1 to memory

A281; FAC#2 to FAC#1

A291; FAC#1 to FAC#2

A2A0; Round FAC#1

A2B0; Get sign

A2BE; Perform [SGN]

A2CE; Fixed-float

A2DD; Perform [ABS]

A2E0; Compare FAC#1 to mem

A320; Float-fixed

A358; Perform [int]

A37F; String to FAC

A40A; Get ASCII digit

A453; Print 'IN..'

A45F; Print line number

A46F; Float to ASCII

A5A3; Decimal constants

A5CC; TI constants

A5E4; Perform [SQR]

A5EE; Perform [power]

A627; Perform [negative]

A660; Perform [EXP]

A6B3; Series evaluation 1.

A6C9; Series evaluation 2.

A707; Perform [RND]

A760; Save Basic stack

A769; Restore Basic stack

A772; Trim Basic stack

A77D; Kernal calls

A7B5; Perform [SYS]

A7DE; Perform [SAVE]

A7F0; Perform [VERIFY]

A7F3; Perform [LOAD]

A84D; Perform [OPEN]

A85A; Perform [CLOSE]

A86B; Parameters for LOAD/SAVE

A89D; Check default parameters

A8A5; Check for comma

A8B0; Parameters for open/close

A906; Make room for string

A954; Garbage collection

AA57; Calculate end of string

AA70; Perform [COS]

AA77; Perform [SIN]

AAC0; Perform [TAN]

AB1A; Perform [ATN]

AB8F; Perform [RENUMBER]

ADCA; Perform [FOR]

AE5A; Perform [DELETE]

AEF7; Print using

B42B; Perform [TRAP]

B440; Perform [RESUME]

B4BE; Perform [ERR$]

B507; Perform [HEX$]

B544; Perform [PUDEF]

B557; Perform [DO]

B5AC; Perform [EXIT]

B603; Perform [LOOP]

B652; Perform [TRON]

B655; Perform [TROFF]

B6CD; Perform [AUTO]

B6E8; Perform [HELP]

B729; Perform [KEY]

B7C2; Key redefinition

B849; Perform [SOUND]

B8BD; Perform [VOL]

B8D1; Perform [PAINT]

B9D4; Perform [CHAR]

BAE2; Perform [BOX]

BD35; Perform [GSHAPE]

BE29; Perform [SSHAPE]

BF79; Perform [RGR]

BF85; Perform [RCLR]

BF87; Perform [RLUM]

BFC1; Perform [RJOY]

BFFD; Perform [RDOT]

C01E; Perform [CIRCLE]

C37B; Set graphics cursor

C3F7; Parse graphics commands

C48F; Get graphics parameters

C4D9; Perform [DRAW]

C50F; Perform [LOCATE]

C51A; Perform [COLOR]

C567; Perform [SCNCLR]
C5B8; Perform [SCALE]
C5C3; Perform [GRAPHIC]
C7BF; Confirm graphics
C8BC; Perform [DIRECTORY]
C941; Perform [DSAVE]
C951; Perform [DLOAD]
C968; Perform [HEADER]
C99C; Perform [SCRATCH]
C9CC; Perform [COLLECT]
C9DA; Perform [COPY]
C9F4; Perform [RENAME]
CA00; Perform [BACKUP]
CB1F; Parse DOS command
CDAB; List of names!
CE00; Interrupt entry
CE0E; IRQ sequence
CE60; Do split screen
CEF0; Kernal UDTIM
CF26; Get time
CF2D; Set time
CF66; Print monitor messages'
CF8A; Get colour code
CF96; Fetch memory
CFBF; Handle tape motor
D80B; Screen in address low
D834; Get screen size
D839; Put/get row/column
D849; Get I/O address
D84E; Initialize I/O
D888; ESC-n normal screen
D891; Clear screen
D89A; Home cursor
D8A1; Set screen pointers
D8D4; Input from keyboard
D965; Input from screen
D9BA; Quote test
D9C7; Setup screen print
DA21; Check line increment
DB11; Read keyboard
DC49; Output to screen
DCDA; Graphics/text control
DCE6; Set colour code
DD0D; Check line decrement
DD2E; Lock shift
DD35; Unlock shift

DE06; Decode ESCapes

DE1A; ESC vectors

DE48; ESC-R; reduce screen

DE5E; ESC-T; top of window

DE60; ESC-B; bottom of window

DE8B; ESC-I; insert line

DEA0; ESC-D; delete line

DECB; ESC-Q; erase to end

DEE1; ESC-P; erase from start

DEF6; ESC-V; scroll up

DF04; ESC-W; scroll down

DF1D; ESC-L; scroll enable

DF20; ESC-M; scroll disable

DF26; ESC-C; cancel insert

DF29; ESC-A; auto insert

DF39; Check screen line wrap

DF46; Break screen wrap

DF59; Make screen wrap

DF66; Calculate screen wrap mask

DF82; ESC-J; start-line

DF95; ESC-K; end-line

E01E; Keyboard select vectors

E026; Keyboard 1 - unshifted

E067; Keyboard 2 - shifted

E0A8; Keyboard 3 - 'comm'

E0E9; Keyboard 4

E12A; Shift/run equivalent

E133; Colour code table

E158; Send 'listen'

E181; Send to serial bus

E1EE; Serial timeout

E1F7; Send listen SA

E1FC; Clear ATN

E203; Send talk SA

E208; Wait for clock

E21D; Send serial deferred

E22F; Send 'untalk'

E242; Send 'unlisten

E252; Receive from serial bus

E2B8; Serial clock on

E2CD; Serial clock off

E2D4; Get serial in clock

E311; Delay 1 ms.

E319; 'press record

E31B; 'press play'

E327; Tape message

E364; Blank screen
E378; Screen on
E38D; Start tape
E3A2; Copyright message
E3B0; Kill motor
E535; Write tape header
E5B0; Initiate tape write
E8F3; Initiate tape read
E9CC; Find any tape header
EA21; Find specific header
EB58; I/O error
EB73; Tape messages...
EBC6; Print if direct
EBD9; Get...
EBE8; Input
EC4B; Output
ED18; Set input device
ED60; Set output device
EDFA; Send 'talk'
EE5D; Close file
EEF8; Set file values
EF08; Abort all files
EF0C; Restore default I/O
EF53; Do open file
F005; Send SA
F043; Load program
F064; From serial
F0F0; From tape
F160; 'searching'
F172; Print filename
F189; Print 'loading/verifying'
F194; Save program
F1A4; Save link
F1B5; Save to serial
F206; Bump r/w pointer
F228; Print 'saving'
F234; Save to tape
F265; Check stop key
F273; Output error messages
F2A4; Power reset entry
F2CE; Kernal reset
F2D3; Kernal move
F2EB; IRQ Vectors
F30B; Initialize I/O
F352; Initialize system consts.
F3D0; Function key commands

F40C; Save filename data
F413; Save file details
F41A; Flag status
F41C; Get status
F41E; Set status
F423; Set timeout
F427; Read/set top of memory
F429; Read top of memory
F42F; Set top of memory
F436; Read/set bottom of memory
F445; Perform [MONITOR]
F44C; BRK/USR entry
F478; Perform ['R']
F4D7; Perform ['M']
F529; Perform ['>']
F54B; Perform ['G']
F570; Monitor commands
F582; Addresses for Monitor commands
F5CE; Perform ['C']
F5D1; Perform ['T']
F60E; Perform ['H']
F6CA; Print '?'
F6E0; Perform ['S']
F6F1; Perform ['L']
F6F3; Perform ['V']
F70A; Perform ['F']
F724; Perform ['D']
F91F; Perform ['A']
FBD8; Output messages
FC19; Get I/O address
FC1E; Check 8-rom
FC56; 8-rom mask
FC59; Phoenix routine
FC7F; Long fetch routine
FC89; Long jump routine
FCB3; IRQ entry
FCB8; Long IRQ routine
FCD1; ROM banking routines
FCF4; Phoenix entry
FCF7; Long fetch entry
FCFA; Long jump entry
FCFD; Long IRQ entry
FD00; ACIA
FD10; 6529 Parallel port
FDD0; Cartridge bank port

FE00; DMA disk system

FF30; Ted chip

FF49; Jump table

FF81; Jumbo jump table

FFF6; Hardware vectors

Appendix 3
Hyperbolic 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 COTANGENT	ARCCOTH(X)=LOG((X+1)/(x−1))/2

Appendix 4
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: $C \leftarrow \boxed{7}\boxed{6}\boxed{5}\boxed{4}\boxed{3}\boxed{2}\boxed{1}\boxed{0} \leftarrow 0$

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.

BEQ

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 ↑

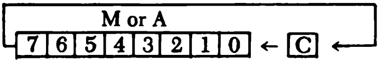
N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL

Rotate Left

Operation:



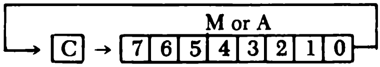
N Z C I D V
√ √ √ – – –

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

ROR

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: $A - M - \overline{C} \rightarrow A$
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

Index

USING THE COMMODORE 16

The longer programs in this book are available on one cassette at £7.95 (post free) direct from Duckworth. Send a cheque/postal order or order by phone with your Access/Barclaycard.

We also publish THE COMPLETE COMMODORE 16 ROM DISASSEMBLY by Peter Gerrard & Kevin Bergin at £6.95. This is an essential reference book for the serious machine code programmer, giving a full disassembly of all the C16 ROM routines.

DUCKWORTH
The Old Piano Factory
43 Gloucester Crescent
London NW1 7DY
Tel: 01-485 3484

Duckworth Home Computing

USING THE COMMODORE 16
by Peter Gerrard

This is an essential book for any C16 user. Starting with a refresher course in Basic programming, it moves on to explore the more sophisticated facilities available on the C16, including windows, graphics and sound commands, disk commands and the use of the built-in machine code monitor.

With the aid of numerous examples the book shows you how to master everything from a simple game in Basic to machine code programming. A full-blown database, a complete adventure game and several other amusing and instructive programs are also included. Each program is accompanied by detailed notes enabling you not only to understand its structure but also to modify it to meet your own needs. A chapter is devoted to peripherals, including the creation of sequential and relative files, and finally the Appendixes give details of ROM and RAM memory maps, hyperbolic functions and the complete machine code instruction set.

Peter Gerrard, former editor of *Commodore Computing International,* is the author of *Using the Commodore 64* and co-author of *The Complete Commodore 16 ROM Disassembly.*

ISBN 0-7156-2003-7

9 780715 620038

Duckworth
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 2003 7
IN UK ONLY £9.95 NET