The Commodore 64 Puzzle Book
Brian Boyde-Shaw
Sigma Technical Press
CONTENTS

How to Use This Book 1

1. In The Clear - a simple problem to get you started 4

2. Screen Printing - print what you like, where you like!
   Eleven puzzles using GET, LEN, ASC, MID$, RND, POKE, INT, READ and DATA. 10

3. Strings and Things - names, messages, all sorts of data,
   with LEFT$, OR, AND, DIM, RESTORE and CHR$. 30

4. Playtime - designing your own games 46

5. Alphanumerics - more on names and numbers, and
   switching from one to the other. 62

6. Number Crunching - arithmetic the easy way - seven
   mathematical puzzles. 73

7. Miscellany - Time, Sound and Graphics - using RIGHT$,
   STR$, TI$ and TI. Turn your 64 into a stop watch, POKE
   your way to music and super graphics. 88

8. Tell Me the Answer - how to solve the extra - tough puzzles. 117
“Life is full of problems”. How many times have we heard this said?

Life’s problems will always be with us, it’s only the solutions that come and go with each generation. So it is with computing.

One man’s way of solving a particular problem, either in real life, or in the fictitious world of the computer, may not be the same as another’s. The end result will often be the same, but the method used will differ in one or more ways. That’s what makes life so interesting, and computer programming so addictive.

So, as in real life, we are called on to solve our own problems in the end, or at least receive guidance on how to go about it, so it is with computers. It’s only the method that’s employed that will differ from programmer to programmer.

Problem solving in life is an everyday occurrence, from the moment we get up, to the moment we close our eyes, and sometimes even after that. If we can get some practice in solving problems without putting ourselves into problematic situations, then maybe we shall be better placed for solving the day to day problems of life.

A micro computer is an ideal companion to help us here.

It’s portable; so we can carry it around from place to place, usually we can GOTO where we like to use it.

It’s easily powered; a mains supply is usually available at most places we are likely to go, but IF we venture out into the sandy wastes, and that is highly unlikely for the majority of us, THEN perhaps we’ll have an Extra problem.

It doesn’t eat a lot, use much current that is; so it’s cheap to RUN.

Admittedly it’s a good idea, most of the time, to have a means of PRINTing or displaying what we are doing, so a TV or monitor must be available on site, so to speak. IF there isn’t THEN there is usually someone around who could LET us borrow one.

More than one power point is also a great advantage, as a power INPUT to all the extras we need is greatly to be desired, as long as we don’t end up with a Christmas-tree like adaptor, which is definitely not recommended.

It’s patient, at least the Commodore 64 is; so it’s not always pestering you FOR What TO do NEXT. So we can take as long as we like to solve our problems.

Neither is it going to keep worrying us to hurry up and GET on with it.

It’s independent of user; so any one can join in.
AND finally it’s logical; which is always a good tactic OR method to use, when planning your problem solving strategy.

Also quite important too, is the fact that a computer is NOT naturally hostile. It only tells us when we are wrong, when we ask it to do something it has not been designed to do. If your programming technique is OK, then it's the friendliest person in town.

You may not get your problem solved, but you will certainly get some reaction to everything you do, even if it only turns out to be dumb insolence.

You must learn to talk to your Commodore 64, and discuss things with it. Verbalising your thoughts is a great way of understanding just what it is you are trying to do. You don’t need an American accent either.

I am asked time and time again, by enthusiastic home computer users, once they have had their Commodore 64 a few months, and have RUN out of games to play or money to buy them with, “What on earth can I do with the thing, now I’ve got it.” And they sit there, tickling the keys from time to time, with a thoughtful disturbed air. Some months previously, they forked out £200 odd, for their computer system, and are now beginning to think that maybe they’ve wasted their money.

But help is at hand, in the shape of this book ..... It will hopefully keep you engrossed for hours, puzzling out how to solve one problem after another, until at the end, you’ll be so surprised at your new-found abilities, that you’ll rush right out and buy the next book in the series.

So how, then, does this book work?

There are forty plus problems, divided unequally into six sets.

Typical of life, isn’t it, some always get more than others. Each set, or chapter, dealing in general with specific points of the programming language BASIC, though you’ll use many points over and over again.

In addition there are fourteen Extra problems, just to liven things up a little, that make the problem you’ve just solved, seem like kid’s stuff, and bugs to find as well.

I have assumed that you already have a minimum working knowledge of Commodore 64 BASIC, and have been PRINTing and LETting for some time now. Suggestions are made on how to go about solving each problem.

But because, in computer programming, there is no one single answer, only good ones and not so good ones, alternatives are sometimes suggested. You will usually be given a complete listing for one method of solving the problem you’re dealing with, but not always.

All the problems require you to produce some effect on the TV screen, from simple words to complicated random words and numbers, and collections of words and
numbers and graphics.

At the beginning of each chapter is a list of those commands, statements and functions that are used in solving that chapter's problems, where they are used for the first time. Throughout the book, suggestions are made for further problems too.

The main aim of this book, and those that will follow in the series, is NOT to teach you computer programming in Basic, but to allow you to develop your ability to solve problems in a logical manner, to a stage where, eventually, you could perhaps become a competent programmer, using whatever language you might prefer.

Basic, as we all know, is not the only programming language.

There are a goodly number of good programming books in every book shop, which teach you how to program, but not necessarily how to solve problems. The Commodore 64, in it's unexpanded form, is not such an easy computer to program, without using a lot of POKE statements. I have tried to avoid these as far as is possible, and to invent ways of overcoming the restrictions of the unexpanded Commodore BASIC language.

In a future book I shall be looking at problem solving using Simon's BASIC for example, which makes the task of the Commodore programmer that much more simple and straight-forward, and adds quite a large number of extra commands and statements to the structure of Commodore BASIC. In the meantime, the best of luck. Off you go to Chapter One and enjoy yourself. Using a computer should be fun.
CHAPTER ONE

IN THE CLEAR

PRINT; LET (assumed use); INPUT; FOR... TO...STEP; NEXT; IF...THEN; GOTO; GOSUB; RUN; LIST; CLEAR SCREEN; ASCII codes; Variables; Operators and punctuation.

This book is written for enthusiastic amateurs, hobbyists and other suitably interested Commodore 64 owners. It is not a book on how to program your computer in BASIC, but more a journey through a series of graduated problems, using a number of BASIC commands, statements and functions as you go along.

It is assumed that readers of this book will have an ongoing knowledge of simple BASIC programming, and the commands, statements and functions from this basic core of knowledge will not therefore be explained. Those required outside this core will be dealt with in detail as required. The required core of Commodore 64 BASIC language commands and statements, are as listed at the beginning of this chapter.

Most real life problem solving situations are beset by a number of difficulties themselves. For example we know that if something can go wrong, it will, regardless, so we must assume therefore the worse possible situation as a result of our actions. As a result of this distressing fact of life we must try to solve our problems in the simplest way possible. Being clever for the sake of being clever is not good programming, it is in fact quite time and memory wasting. In the end the more simple problems we solve, the more capable we become in solving the
bigger ones, as a result of this relatively easy practice. But computer programming has a further complication in that, from the moment we begin to type our program in on the keyboard, errors start creeping in. Either because we are not giving the task in hand our full attention, or we are just plain bad typists. When it comes to keying in a program, slow two finger typists are perhaps better than quick two handed ones.

There are three possible error situations that could arise as a result of typing in a program, and most if not all, will be reported to you by the Commodore 64, once your program is RUN.

1. Making typing mistakes - is the most common one, and is the mistake we are all guilty of, even competent typists. The beauty of a computer is that these are easily corrected. The 64 has an efficient editing system, though a little frustrating at times, as perhaps you’ve noticed.

2. Making simple programming errors - for example, leaving out one set of inverted commas within a PRINT statement, full stops instead of commas, leaving out the NEXT or RETURN in loops or subroutines. These can only be corrected by intelligently reading the reported errors, and searching the lines around the error for an explanation. One of the most common faults made by beginners, is failing to press the RETURN key at the end of a line, ending up with multiple errors, instead of multiple statements, so to speak. In the programs in this book multiple statements will be used as far as is possible. If the 64 does report an error on a line, say 100, then check that line first, by typing in LIST 100, and seeing how many lines you get for your money. You may be surprised, I’ve seen as many as a whole short program!

3. Making the computer confused - this comes about by bad planning, or just plain user error. For example, you GOTO a particular line to repeat some part of a program. Then you edit your program, and in doing so you lose a line. But it happens to be the line that the computer will eventually be told to GOTO. It’s not there any more, so the computer is now unable to make sense of your program, or you can’t make sense of what the computer is doing. But if you’ve planned your program properly, and have it all written out before you start to type it in, then that sort of error should be relatively easy to find.

But let’s first have a look at why I think planning is so important in programming, and why I think problem solving is an extremely useful activity, if ever you are going to learn to program efficiently.

When presented with a problem, the average human being arrives at a solution by a complex series of thought processes, that might or might not have had a logical sequence. Man’s ability to be logical in his actions is born of two things. One, his inherited thought processes, and two, those learnt during early life, from parents and his early infant teachers. Gifted children are, I’m afraid, few and far between, but not as scarce as you might think.

So if we assume that the average person is basically illogical and arrives at a
personally acceptable outcome, as a result of judging some of the possibilities presented to him, based on all his knowledge and experience gathered to date, then we must resort to a suitable planning strategy to avoid any unacceptable results.

For example if you happened to be a member of a panel judging a beauty contest at a holiday camp, you could be involved in a great deal of discussion and possible argument at the time of the parade, in order to declare which of the many outwardly beautiful contestants was indeed the ‘most beautiful’. Beauty is in the eye of the beholder, we’re told. And that eye has been brainwashed by everything it has experienced since the first moment it opened and viewed the world for the very first time.

If, therefore, our fictitious beauty panel is to arrive at a considered and agreeable conclusion, there must have been a modicum of planning behind their decision. There would have to have been a list of attributes to look for, together with a table of points that could be allocated to each attribute. Big, small and medium is not a good list of points or anything else. One can always ask, ‘what is big’ and ‘what is small’, which again relies on past experience, and which for each of us, is quite different.

Therefore the big, small and medium must be explained, and guidance given on how to view it and allocate those aforementioned points, so that in the final analysis the result will have been arrived at in a logical fashion.

For example our beauty contest, if judged purely on the outward shape of the contestants, will have big, small and medium allocated to those three well known important areas of the female anatomy. And therefore the three areas will have to judged and considered in isolation, and points afterwards awarded by each judge closely following the rules laid down for each particular attribute. Someone has to lay down these rules in the first place of course, and in the case of the beauty contest, it could well have been the contest organiser or sponsor or both.

Likewise with problem solving on a computer. Everyone knows a computer is ‘stupid’ and will only do what it is told. Therefore all instructions must be logically written, following the rules laid down by those who were responsible for writing Commodore BASIC for the 64 in the first place.

Each problem situation must be broken down into small individual tasks, so that we are able to solve a series of minor problems, in turn and in the correct sequence.

Also let me point out that there are three steps to be taken in solving a computer problem, namely:

1. It’s creation and design.
2. Typing it in.
3. Making sure it works by running it,
and correcting all the mistakes.

Which could be entitled, big, small and medium, so to speak.

To be sure the most difficult of these three is the first one, creating and designing a solution. A badly designed program will never do it’s job properly, and will probably crash at the first opportunity. It will not be the fault of the computer, it will be yours for getting the facts right and the planning wrong. The wrong girl will have won the beauty contest!

This will be the procedure followed in all the problem solving situations in the following chapters, including this one.

In the meantime here is your induction problem, one that I had to solve, before starting to write this book.

THE SCENARIO.

I have a number of computers, the Commodore 64 included, and two dot matrix printers, neither of which is a Commodore printer.

To clear the screen on the Commodore 64 you just press the SHIFT and CLR HOME keys at the same time. To clear the screen in a program, this instruction has to be placed on a program line. When writing your program you have to put this instruction to clear the screen in a PRINT statement, in inverted commas:

```
10 PRINT "SHIFT CLR HOME"
```

When you do this on the keyboard, a little heart shape appears between the inverted commas on the TV screen, and if this information is sent to a Commodore printer, you indeed get a little heart shape in your listing. But with a non-Commodore printer you won’t, unless you do some elegant character definition for the printer.

So we, or at least I, had a problem, which is as follows:

How do we write a program to ‘clear the screen’ using a Commodore 64 with a non-Commodore printer, so that what you see on the screen also appears on the printer when you list it?

If you literally type 10 PRINT “SHIFT CLR HOME”, (i.e. the words S-H-I-F-T and C-L-R H-O-M-E), the 64 will indeed print the words out, but the screen will not, of course clear. If you use the little heart shape the printer won’t print it, but the 64 will clear the TV screen.
How, then, can we clear the screen and print the instruction on the printer at one and the same time, with the same statement? Think about it, before you move on to the solution.

THE SOLUTION

From our knowledge of the computer we know that we must press a key to give an instruction to the computer to clear the screen. We also know that the printer will quite happily print all letters, numbers and punctuation.

We know that when a key is pressed, the computer receives a coded instruction - i.e. a number representing a character. It then looks this up in it's memory to find out what the number means, and carries out the instruction. So:

A=6
PRINT A
RETURN

will produce a 6 on the screen, because the computer knows that A is a temporary label for 6.

Likewise
PRINT"A"
RETURN

will produce an A on the screen, because anything in inverted commas is printed literally.

In the first two of the above three lines of program, it found the code for A and did the necessary. As it also read the inverted commas in the third line, then it also knew that A was an alphanumeric and not a number.

Knowing that every key has an equivalent character number we can substitute a character number for A:

PRINT CHR$(65)
RETURN

assuming we know about ASCII codes.

Now the computer will print A again, because CHR$(65) means A to the computer, and A literally, no need for inverted commas now.
So we can say

CHR$(65) is equivalent to "A"

All these CHR$(X) equivalents will be found on pages 135 to 137 of your
Commodore 64 User Manual. And, by the way, CHR$(54) = 6.

If we look now for the CHR$ equivalent for SHIFT CLR HOME, we find that it is 147.

It is logical to assume therefore that:

\[ \text{PRINT CHR$(147) = PRINT"SHIFT CLR HOME"}\]

If we type this in, then lo and behold the screen magically clears.

Go on, try it.

PRINT CHR$(147)

RETURN

My printer will of course respond to this instruction and print it out, because it only contains letters, numbers and punctuation, (which includes brackets). Also if we RUN a program containing a line stating PRINT CHR$(147), the screen will automatically clear, when the computer reads that particular line.

Our problem is therefore solved.

All the programs in this book will use CHR$(147) to clear the screen, which also means we could have used a typewriter or simple word processor to have typed out the programs in this book.

Typewriters do not have little heart shapes, inverse or otherwise. By the way the little heart shape is in reverse, or inverse, video, and

PRINT CHR$(18);CHR$(115)

RETURN

will produce this on the screen, but won’t clear it!

Using CHR$(X) in a program can be very useful, especially if you want to press a key that you can’t program, such as REVERSE ON and OFF.

Remember that!
CHAPTER TWO

SCREEN PRINTING

GET X$; LEN(X$); ASC(X$); MIDS(X$,S,X); RND(X); INT(X); POKE X,Y; READ X; DATA

In any problem solving situation, it is always best to start off with simple procedures, which, once learned, can be used like building blocks to solve, later on, more complex and difficult problems.

Scatter periods of revision in the path every now and then, and you have a reasonable learning situation. This is the procedure we will follow in this second chapter, which deals with printing information in various places on the TV screen.

Problem One

Write a program to print your name on a completely clear screen.

The problem is to write your name, which hopefully you should know, on the TV screen. This could be placed in memory as the program RUNs. But the simplest way to do it is with a PRINT statement.

You must, therefore, clear the screen first, with whatever clear screen statement you have decided to use, then print your name, but stop the program from ending. If it ends in the normal fashion, the word READY will appear, together with its flashing cursor, and this will not be a completely clear screen.
We have now broken the problem down into four clear steps:

1. Clearing the screen.
2. Telling the computer what your name is.
3. Printing it.
4. Keeping the screen clear.

Steps two and three could be done at the same time, as I inferred earlier.

To stop a program from going any further once it has carried out a particular command or statement, you must keep it stationary on the line the computer is reading at the time.

To do this we use the GET statement.

For example: GET K$

The computer will scan the keyboard, looking for something to put in a memory location labelled K$. If it finds nothing, meaning no one has pressed a key, it will of course plod on and finish the program, or do whatever is there on the next line.

So we must add something to GET K$ to keep the computer on the look out for any key that may have been pressed, before it's allowed to plod on. For example, using line 30:

30 GET K$: IF K$="" THEN 30

The computer will now scan the keyboard, and if it finds that A KEY HAS NOT BEEN PRESSED, which is what K$="" means, an empty string, as it is called, will be returned, and the computer carries out the next STEP in that program line, which in our case is THEN 30. Then it does it all again, until a key is pressed.

What has happened is that line 30 was found to be true, and the THEN part was therefore carried out. If line 30 were found to be false, then the THEN part would not have been carried out, and the computer would automatically move on to the next LINE.

This is our first bit of computer LOGIC, there will be a lot more to follow. True and false are the basis of logic, and are an integral part of decision making in computer programming.

So, our program will look like this:

```
10 PRINT CHR$(147)
20 PRINT "YOUR NAME"
```
30 GET K$: IF K$ = "" THEN 30

which means:

10 clear the screen
20 print your name on the screen
30 wait until a key is pressed, before ending the program and returning to the command mode.

Type the program in, and see what happens, if you haven’t solved it already, or are just reading this part to be polite and to check your answer.

Once typed in and RUN, the program will not stop until a key is pressed, when it will print READY and present you with the flashing cursor.

The program could also have been written:

```
10 PRINT CHR$(147)
20 N$="YOUR NAME"
30 PRINT N$
40 GET K$: IF K$ = "" THEN 40
```

Where N$ is filled with your name, and could be used as often as you wanted in your program, if it were any longer.

It’s also a good idea to label all the variables you use, in such a way that they are easily recognised. For example, in the first program:

```
N$ = NAME STRING
K$ = KEY STRING
```

Labelling the variables used in a program is very important, not only from the point of view of making them easily recognised, but to make your program easy to understand when someone else has to deal with it.

The program could also have been written on two lines, for example:

```
10 PRINT CHR$(147):N$="YOUR NAME":PRINT N$
20 GET K$: IF K$ = "" THEN 20
```

Line 20 could not have been placed on line 10, as this would have made ‘your name’ flash on and off the screen repeatedly, and you weren’t asked to do that!

Go on, try it, but don’t forget to change the THEN 20 to THEN 10, or the program will crash. If you understand that, remember my remarks in an earlier chapter regarding making mistakes when rewriting or editing a program.
The second problem extends the first somewhat.

************************************************************************

* Problem Two *
* Write a program to print your name ten times on a clear screen. *
************************************************************************

You know now how to write your name on a clear screen, but now we must repeat it ten times. The name writing part must now be placed into a repeat loop, or more technically a FOR ... NEXT loop.

Again choose a variable for the sequence that means something.

The main problem can again be broken down into four steps:

1. Filling the memory.
2. Clearing the screen.
3. Printing your name ten times.
4. Keeping the screen clear, except for your ten names.

So our program, in longhand so to speak, will look something like this:

```
10 N$="YOUR NAME"
20 PRINT CHR$(147)
30 FOR R=1 TO 10
40 PRINT N$
50 NEXT R
60 GET K$: IF K$="" THEN 60
```

where

10. puts your name into the memory, under the label N$
20. clears the screen
30. starts the repeat sequence
40. prints your name
50. sends the computer back to line 30, until R=10
60. waits for a key to be pressed.

This program has six lines, and therefore is wasteful of memory, in that you are allocating instructions to more line numbers than you need to. So we can rewrite it:

```
10 N$="YOUR NAME":PRINTCHR$(147):FOR R=1 TO 10:PRINT N$:NEXT R
20 GET K$: IF K$="" THEN 20
```
Here R stands for Repeat.

Your name will now appear ten times down the top left hand side of the TV screen, with the top line or row of the screen blank. We will find out later on how to fill that top line, and how to print information elsewhere too.

***************

* Problem Three

* Write a small program to ask for someone's name, then to tell them how many letters it has in it, together with a suitable message.

This particular problem can be broken down into five parts:

1. Get someone to type in their name.
2. Store the name, so it can be used.
3. Calculate how many letters it has in it.
4. Compose a suitable message.
5. Clear the screen, and print out the information.

Parts 1 and 2 can be solved together by using an INPUT statement, and allocating the received keyboard information to a string variable called N$.

To solve part 3, we use the function LEN(X$). This calculates the number of letters, spaces and so on, that the string variable has in it, so that:

X$="SOLVED IT"
L=LEN(X$)
PRINT L

causes a nine to be written on the screen, as 'SOLVED IT' has eight letters and one space in it. We can use a numeric variable for the length of the string variable we wish to know about, or the expression LEN(X$) can be used on its own in a statement, so that:

X$="SOLVED IT"
PRINT LEN(X$)

will produce a nine on the screen in the same way as the first method above.

Part 5 is easy, we can use a PRINT statement and include it in part 4, for example:

PRINT""YOUR NAME HAS"";L;""LETTERS IN IT""
or

PRINT "YOUR NAME HAS"; LEN(X$); "LETTERS IN IT"

So our solution could look like this:

10 PRINTCHR$(147)
20 PRINT "TYPE IN YOUR FIRST NAME"; INPUT N$: L = LEN(N$)
30 PRINTCHR$(147); "YOUR NAME HAS"; L; "LETTERS IN IT"

Line 20 solves parts 1, 2 and 3, and line 30 solves parts 4 and 5.

Did you notice anything new about this short program, if and when you typed it in and ran it?

Yes, the information in line 30 was written on the top line of the screen. Voila, I said I would show you how.

If you clear the screen and immediately afterwards use a PRINT statement, then you can use the top line. Any other statement will shunt the screen display down a line, but of course the PRINT statement must be on the same line as the clear screen statement; have a look at lines 10 and 20 in the last program.

As a further exercise, 'Extra one', and a simple one to start off with, rewrite the above program to PRINT both messages on the top line of the screen.

*****

That was easy wasn't it. If it wasn't then the solution will be found in chapter eight, 'Tell me the answer'.

Now that you've seen how easy it is to solve problems, see if you can do another extra problem, 'Extra two'.

Change the program for problem three so that someone tells the computer how many letters there are in his (or her) name, after it has been typed in. Then tell them if they are right or wrong.

Clue: you'll have to resort to an IF...THEN statement, and don't forget about spaces, as they aren't counted as letters.

·~**********************************************************************

15
The problem this time is to fill only half the screen, with a six letter word.

The 64's screen is twenty five rows, or lines, by forty characters, or columns, which makes a total of one thousand character positions. Half the screen therefore is five hundred characters.

But five hundred characters, although half filling the screen, puts half a line in the centre of the screen, which looks rather untidy to me, so I will use 500 minus half a line; 500-20=480 characters only.

You, of course, can use 500 if you want to. Also the top line of the screen, you will find, will not be filled because of what we have discussed before, so our half the screen will be moved down a bit.

What you do will depend on how you interpret 'half filled the screen'.

The next step is to write the six letter word over and over again, continuously, filling only 480 screen character positions. If you divide 480 by the number of letters in the word, then you will know how many times the word must be written, and the instruction to print it put into a FOR...NEXT loop.

So our program could look like this:

```
10 W$="MYWORD"
20 PRINTCHR$(147)
30 FOR R=1 TO 480/6
40 PRINT W$;
50 NEXT R
60 GET K$: IF K$="" THEN 70
```

Line 40 you will notice has a semi-colon after the W$, this has the effect of printing all the 480/6 W$'s one after the other in a row, and treats the whole screen as one continuous row. Lines 30 and 50 of course make up the repeat loop, with the information to be repeated inside on line 40.

Again this program is a little on the long side, but it can be rewritten using only three lines. By the way, the 64 will accept only two screen lines of program before it reports an error, but this does not mean 2 x 40 = 80 characters. If you use the last character position on the second row, then the cursor jumps round to the third row, and the computer will again report an error, so you have only 79 characters positions for your program line.

The three line program looks like this:

```
10 W$="MYWORD": PRINTCHR$(147)
20 FOR R=1 TO 480/6: PRINT W$;: NEXT
30 GET K$: IF K$="" THEN 30
```

Study the above program and you will see that NEXT has no variable after it, and
the program lines don't have any spaces between the statements and instructions; this, of course, saves memory space.

You could also, in line 20, have written:

```
20 FOR R=1 TO 80: PRINT W$: NEXT
```

to have done exactly the same thing.

*****

Generally speaking, the first four problems have given you some ideas on using static information in your programs; this is useful for, say, headings and instructions throughout a program.

All the programs have started printing at the left hand end of the second line of the screen, except for the short excursion onto the top line.

We will now consider other methods of getting information into memory, and of printing it in other parts of the screen as required.

****************************************************************************

Problem Five

Write a program to get someone to type their first name in on the keyboard and then to print that name twelve times on a clear screen.

*****************************************************************************

Getting someone to follow instructions in a computer program requires the use of a PRINT statement to tell them what to do, and then an INPUT statement to receive their answer or whatever, and to store it in memory. So:

```
10 PRINT "PRESS ANY ONE KEY"
20 INPUT K$
30 PRINT "YOU PRESSED ":K$
```

asks the program user to press a key and then tells them what key they pressed. Naturally, they could have pressed more than one key and all of them would have been reported, as we have used INPUT and not GET in line 20, which labels the reply K$ and then uses it in a PRINT statement.

So our program could look like this:

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

17
INPUT can of course be combined with a PRINT statement, and the message inside the inverted commas is then called a prompt, the variable follows immediately after the message with a semi-colon before it.

Naturally, an INPUT from the keyboard to be displayed on the screen, longer than forty characters would look untidy when printed. So its length must be limited at the moment, as in line 30 above. The whole instruction for dealing with this situation fits into one program line:

30 IF LEN(N$)>40 THEN PRINT"TOO LONG, TRY AGAIN":FOR D=1 TO 2000:NEXT:GOTO 10

It first checks the length of the keyboard input N$, if its length is forty characters or less then the IF...THEN statement is false and the computer moves on to the next line of the program. If true, it carries out the rest of the line, printing the message, and leaving it on the screen for 2000 time units, then going back to line 10 to start the sequence all over again.

Line 40 prints N$ twelve times on twelve separate lines or rows, because, this time, a semi-colon has not been used after N$.

Line 50 waits as usual for its keyboard input.

Naturally, you could get the program to RUN continuously by changing line 50 to:

50 GET K$: IF K$="" THEN 10

but then of course you would have to press the RUNSTOP key to end the program.

Problem Six

Write a program to display on the central line of the screen any chosen word typed in on the keyboard.

Having obtained information from the keyboard, and printed it where the computer wants to put it, we can now try to place it exactly where we want to put it.

The computer will print on any line on the screen, if you tell it to ignore those lines that go before where you want to put the information. It will also print information in any column you want, if you use the PRINTTAB statement. TAB moves the
cursor across the screen similar to the action of the space bar on a typewriter.

So we could write:

```
10 PRINTCHR$(147):FOR L=1 TO 10:PRINT:NEXT
20 PRINT"THE ELEVENTH LINE IS HERE"
30 PRINTTAB(10);"THIS STARTS AT THE ELEVENTH COLUMN"
```

So, using these two facilities, we can start our information off from nearly every character position on the screen.

For example:

```
10 FOR L=1 TO 11:PRINT:NEXT:PRINTTAB(19);"XX"
```

puts two X's in the centre of the screen, but the trouble is that if four X's were used they would not be truly central, and it would be the same for any number of X's other than two.

So, a way must be devised to make sure that characters can, if required, be placed exactly in the centre of the screen.

If the length of the word to be written is found and then divided by two, and the result taken away from half the length of a screen line, which is 20, then the centre of the word will be dead centre, providing there is an even number of letters in the word. An odd number of letters will have the word displaced by one character space.

So:

```
PRINTTAB(20-LEN(W$)/2)
```

is the formula to be used, because this method ensures that the two halves of your word appear either side of screen centre.

For example, if the word is ten letters long, dividing it by 2 gives 5, and 20–5 = 15. The word will start at column 16, and finish at column 25, making fifteen character spaces each side of the word.

DON'T FORGET THE FIRST COLUMN IS NUMBERED 0 AND THE LAST ONE 39, giving 40 columns.

So, our solution program would look like this:

```
10 PRINTCHR$(147):INPUT:"TYPE IN ANY SHORT WORD ":W$
20 IF LEN(W$)>40 THEN PRINT"TOO LONG, TRY AGAIN":FOR D=1 TO
2000:NEXT:GOTO 10
```

19
30 PRINTCHR$(147):FOR R = 1 TO 11:PRINT:NEXT
40 PRINTTAB(20-LEN(W$)/2);W$
50 GET K$:IF K$="" THEN 50

Line 40 could be put on line 30 if required, making the listing only four lines long.

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

* Problem Seven
* Now write a program to accept a word typed on the keyboard, then print that
* word over and over again in completely random places.
*
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

There are two main parts to this problem. The first is getting the word typed in, and the second is printing it correctly.

Getting it in is not difficult, as we have done it before with an INPUT statement and a prompt. The word of course should be checked for length, but you may think that this is not too important. For example, if the word is printed on the screen, so that the end of it would stick out from the right hand side of the TV screen, if that were possible, then the computer breaks it in two and prints the second part on the left hand side of the next line. You have noticed this, no doubt.

It is possible for this to be catered for, so that the random locations that are chosen do not result in the word breaking up, but that’s for later on.

So, the first part of the program should be obvious:

10 PRINTCHR$(147):INPUT "TYPE IN A WORD";W$:PRINTCHR$(147)

Now we have our word, called W$, stored in memory.

It’s position on the screen must be random, so RND(1), the random number function must be used.

The trouble is printing it correctly, and this section of the problem has two further parts to it:

1. Getting an acceptable random number.
2. Putting the word on the screen.

The 64 screen is, as I have previously mentioned, divided into 1000 character
positions, a 40 by 25 matrix. Each character area or position has a memory mapped screen location. Page 63 or 38 of your 64 manual will show you this. Turn to it now.

The top left hand area, or square, is number 1024, and the bottom right hand one 2023, making a total of 1000 squares.

In order to get a character placed in one of these squares, we can use a combination of blank PRINT lines and the function TAB(X), but that method is a little tiresome. Alternatively, we can make use of the POKE statement. For example POKE 1524,65 should place a letter A in the approximate centre of the screen, a white A on a blue screen, but if your screen happened to be white, then of course you wouldn’t see the A.

So, in order to cover ourselves for all eventualities, we’ll make the A black.

To do this we use another POKE statement. The screen of the Commodore 64 can also be POKEd to change the colour of the character displayed in any character square. Look, on page 64 or 139 of your manual, at the colour memory map. The top left hand corner is memory location 55796 and the bottom right hand square 56295, again giving 1000 locations. So, for example, POKE 55796,0:POKE 1524,65 will produce a black A in the middle of the screen, (0 is colour black, the other colours are given on page 139).

Our random number, therefore, for the screen location must be in the range 1024 to 2023, and for the colour, 55296 to 56295.

Now RND(1) produces a random number between 0 and 0.999999999, which as you can see has to a goodly number of unwanted decimal places.

The function INT, standing for INTeger, meaning whole number only please, will remove the decimal places, so we have:

\[ P = \text{INT}(\text{RND}(1)) \]

where \( P \) = the variable for our random Position on the screen, but unfortunately this way of using the RND function only produces 0, whereas we want 1024 to 2023. Go on, try it if you don’t believe me!

To get the numbers in the desired range is quite easy. We could start by multiplying the random number by the difference between the smallest and largest we want, in our case 1000, and take the integer part:

\[ P = \text{INT}(\text{RND}(1) \times 1000) \]
But, we will now get a number between 0 and a 999. So, if we add the lower limit, 1024, we get:

\[ P = \text{INT}(RND(1) \times 1000) + 1024 \]

which means we now get a number between 1024, \((0 + 1024)\), and 2023, \((999 + 1024)\), which is exactly what we require.

We now have the correct random number to POKE all the character positions on the screen, but not of course for the colour memory map.

The difference between 55296 and 1024 is 54272, so the colour location or position to be used for all the locations on the screen would be 55272 + P. To colour the randomly selected A black, would need:

\[ P = \text{INT}(RND(1) \times 1000) + 1024 \]

\[ \text{POKE} \; 54272 + P, \; \text{O} : \text{POKE} \; P, \; 65 \]

By the way, do you remember why we use 65 instead of A? Yes, we are POKEing characters, not letters and numbers again.

Putting the word onto the screen presents us with two problems.

A. Positioning the word on the screen.
B. Putting the correct characters onto the screen.

To put the word on the screen starting at any random screen location requires breaking the word down into its characters, because POKE’s will only POKE one character at a time, and then printing each character in turn next to the other. But, of course, we must know the ASCII code of these characters as well, because a POKE statement can only be used with numbers.

We must separate each letter of the word in turn, and then find its ASCII code number. To separate the letters we can use the MID$ function, for example:

\[ L = \text{MID$(W$,$N,1)$} \]

where:

\[ L = \text{the Letter} \]
\[ W$ = \text{the Word, and} \]
\[ N = \text{the position Number of each letter in the word.} \]

To make this idea work, we have to place it in a loop, for example:

\[ \text{FOR N=1 TO LEN(W$)} \]
\[ L = \text{MID$(W$,$N,1)$} \]
will produce all the letters in turn, calling each one \textit{L}.

But we need the ASCII code for them, so:

\begin{verbatim}
FOR N=1 TO LEN(W$)
  L=ASC(MID$(W$,N,1))
NEXT
\end{verbatim}

will do the trick, by producing the ASCII code for each letter in turn.

Now all we have to do is POKE our random position with \textit{L}, and we can write the word, or is it?

Well no, because to complicate matters further the Commodore 64 has two sets of characters available, and when you switch on the computer you get set 1, (see pages 132 to 134 of the user manual). If the first letter of our word was \textit{A}, the code poked would naturally be 65, but this unfortunately gives a club shape, which is not much good, when we want to write \textit{A}, unless we want to write it in some sort of code. But switch to set 2 and everything will be alright. We must therefore get the correct screen code for the \textit{A} in set 1, which of course, (see page 132), is POKE 1, or 65 – 64.

Now we tell the computer that if it gets a code less than 64, it can go ahead and use it but, if not, to take 64 away from the number of the code, and then use that one instead.

Our instruction to the computer now looks like this:

\begin{verbatim}
FOR R=1 TO LEN(W$)
  L=ASC(MID$(W$,N,1))
  IF L<64 THEN POKE P,L
  IF L>64 THEN L=L-64:POKE P,L
NEXT
\end{verbatim}

The next problem is that \textit{P} does not change, so we are printing the character in the same POKEd position all the time. No good! We have to get the routine to advance one character space each time it’s POKEd. We must therefore upgrade, or increment, \textit{P} by 1 everytime we use it in the above loop.

Our final program then will look like this:

\begin{verbatim}
10 PRINTCHR$(147):INPUT¨TYPE IN A WORD¨:W$:PRINTCHR$(147)
20 P=INT(RND(1)*1000)+1024
30 FOR N=1 TO LEN(W$):L=ASC(MID$(W$,N,1))::IF L<64 THEN 50
40 IF L>64 THEN L=L–64
50 POKE P+54272,0:POKE P,L;P=P+1:NEXT
60 FOR D=1 TO 2000:NEXT:GET K$:IF K$=¨¨ THEN 20
\end{verbatim}
Line 60 allows the word to remain on the screen for 2000 time units, before, if you haven't pressed a key, sending the computer back to line 20, to pick another random number, and do it all again, as the problem asked.

For 'Extra three' how about changing the program so that the screen is cleared each time the word is printed?

This problem is based on problem seven, so a few changes to the beginning of solution seven, will produce a new solution for the first part of the problem eight.

We need two words typed in, so we can write:

```
10 PRINTCHR$(147):INPUT"TYPE IN A WORD ":W1$
20 INPUT"NOW ANOTHER ONE ":W2$
```

I've called the two words W1 $ and W2$, as an aid to reading the program, W1 $ for the first word, and W2$ for the second.

We now have two words in memory, and are required to place them both, alternately, in random positions on the screen. Next, therefore, generate two random numbers for the two different screen locations:

```
P1=INT(RND(1)*1000)+1024
P2=INT(RND(1)*1000)+1024
```

should take care of this.

We now need to write the two words in these random locations. To do this let's use the short solution from the last problem contained on lines 30, 40 and 50 of the listing. We must of course use it twice, once for each location, not forgetting to change the P number.

The full program will look like this:

```
10 PRINTCHR$(147):INPUT"TYPE IN A WORD ":W1$
20 INPUT"NOW ANOTHER ONE ":W2$
30 PRINTCHR$(147):P1=INT(RND(1)*1000)+1024:
P2=INT(RND(1)*1000)+1024
40 FOR R=1 TO LEN(W1$):L=ASC(MID$(W1$,R,1)):IF L<64 THEN 60
50 IF L>64 THEN L=L-64
```
Did you notice I’d written one word in black and the other in white? This makes the randomness easier to see, so how about experimenting with other colours, for example as ‘Extra four’, write each letter of the word in a different colour, until you run out of them, then start again.

And yet another extra for you to try:

There is one snag with this program the way it is written: both the random locations for the words could be the same, and if this happened, you would only see the second one. Change the program now so that if this occurs, the error is trapped, we’ll call this one ‘Extra five’. I told you programming generated it’s own problems.

~~~~~~~~~~~~~~~

Now to complete this chapter on screen printing, we’ll have a look at three problems dealing with numbers.

********************************************************************************

* Problem Nine *

* Write a program which stores five numbers, then prints them down the * * centre of a clear screen.

********************************************************************************

We can divide this problem into two main parts:

1. Storing the numbers.
2. Printing them.

The first part is simple, we can use READ and DATA statements, for example:

   10 READ A
   20 DATA 5

When this minute program is RUN, A will equal 5, and will be stored away in the computer’s memory. But we need to store five numbers, so lines 10 and 20 must be carried out five times, so:
10 FOR R=1 TO 5:READ N
20 NEXT
30 DATA 1,2,3,4,5

will READ the five numbers: 1,2,3,4 and 5.

But the only way, as far as I am concerned, we can PRINT these numbers on the screen in a particular place is to POKE them there. PRINT and TAB is just a little too tedious. If I want to POKE them, then I must ASCII code them first. I cannot just use 1,2,3,4,5, so I must find the ASCII codes for the numbers I wish to use first.

1 is 49; 2 is 50; 3 is 51; 4 is 52, etc., so:

10 FOR R=1 TO 5:READ N
20 NEXT
30 DATA 49,50,51,52,53

will READ the correct codes into memory for POKEing 1,2,3,4 and 5 on to the screen.

Printing them out in a column down the centre of the screen is a little more complicated.

So let’s look at the problem in more detail.

a. We must find the centre of the screen.
b. Then use five locations, one under the other, and place our five numbers in them.

The screen has twenty five rows or lines. To use the central five, we need to put numbers on lines 11 to 15, leaving ten blank at the top, and ten blank underneath. The eleventh line starts at 1424, the centre of which must be 1444 (1424+20), and for the colour, must be 55716(54272+1424+20). The next four positions under 1444 and 55716 must be forty added four times, one at a time, so both the text location and colour location must be incremented each time a number is POKEd.

The program, therefore, could look like this:

10 P=1444:C=55716:PRINTCHR$(147)
20 FOR R=1 TO 5:READ N
30 POKE C,0:POKE P,N:C=C+40:P=P+40:NEXT
40 GET K$:IF K$="" THEN 40
50 DATA 49,50,51,52,53

where P = character Position
C = character Colour position
R = Repeat loop number
and N = Number read from the DATA line.

Note that I have declared the value of the two variables, P and C, that we are using on line 10, at the start of the program.

So once again our problem is solved.

***************************************************************************
* Problem Ten
* Write a program to print a single random number in the centre of the
* screen, overprinting, after a short delay, each number with the next one
*
***************************************************************************

This particular problem is quite simple and straight forward for a change, now we know how to produce the correct numbers with a POKE statement and ASCII codes. All we need to do is generate a random number in the correct range and we're away. Since

N=INT(RND(1)*9))+48

produces random numbers between 48 and 57, we have the right ASCII codes for the real numbers 0 to 9, try it and see.

Now we must find the correct place to POKE them on to the centre of the screen.

The centre must be where the thirteenth row crosses the twentieth or twenty first column. We will choose row 13, which starts at screen character memory position 1504. The centre of this line therefore is 1504 + 20, and likewise 55776 + 20 will be the colour code centre, 54272+1504+20.

Lastly we must leave the number on the screen for a short while, so we will use a short delay routine for that:

FOR D=1 TO 200:NEXT

will be long enough, I think. We then go back to the next random number, but we must also allow for the program to be stopped from the keyboard.

For our program we could have the following:

10 PRINTCHR$(147)
20 N=INT(RND(1)*9))+48:POKE 55796,0:POKE 1524,N:FOR D=1 TO 200:NEXT
30 GET K$:IF K$="" THEN 10

27
Lines 10 and 20 could of course have been put on one line together.

Well that was easy again, wasn’t it?

So how about making it a little more difficult, by doing ‘Extra Six’:

Make the random number in the middle of the screen change its colour each time it’s printed.

Problem Eleven

We know how to put numbers in the centre of the screen now, but to make a cross from five numbers, we need three vertically, and three horizontally, which seems a bit odd, but of course the centre number is used twice, once in the vertical column, and once in the horizontal row.

We can break the problem down into four minor parts.

1. Find the centre of the screen.
2. Generate three random numbers, and print them there.
3. Generate the remaining two random numbers, and finish the cross with them.
4. Prove the program is indeed random.

The centre of the screen for the three numbers is at 1484 and 55756, so we can say P=1484 and C=55756.

To generate three random numbers, for the vertical column of three, we need to give the computer a choice of numbers from 0 to 9, or ASCII codes 48 to 57, which means a random statement like:

\[ N=\text{INT}(\text{RND}(1) \times 9) + 48 \]

Put this inside a loop that repeats three times, together with the screen location routine, and we will have put three random numbers between 0 and 9 in a straight column in the centre of the screen. So:

\[
\text{FOR} \ R=1 \ \text{TO} \ 3: \ N=\text{INT}(\text{RND}(1) \times 9) + 48 \\
\text{POKE} \ C,0: \text{POKE} \ P,N:C=C+40:P=P+40: \text{NEXT}
\]
To get the remaining two numbers in place, we only need two more now, so use:

```
FOR R=1 TO 2:N=INT(RND(1)*9)+48
```

but we need to get the two numbers either side of the centre of the three numbers, in order to form a cross. This means going back up two complete lines from our last position or square, and moving to the left, one square, that is –81 screen squares, so:

```
POKE C-81,0:POKE P-81,N:C=C+2:P=P+2:NEXT
```

will start with the left hand side of the cross, and then jump across to the right hand side with the C=C+2 and P=P+2.

To prove that our program is random, we must allow it to jump back to the random numbers part and start again, after a delay for us to read the screen, and also allow for the program to be stopped from the keyboard. So our program could look like this:

```
10 P=1484:C=55756:PRINTCHR$(147)
20 FOR R=1 TO 3:N=INT(RND(1)*9)+48
30 POKE C,0:POKE P,N:C=C+40:P=P+40:NEXT
40 FOR R=1 TO 2:N=INT(RND(1)+9)+48
50 POKE C-81,0:POKE P-81,N:C=C+2:P=P+2:NEXT
60 D=1 TO 1000:NEXT:GET K$:IF K$="" THEN 10
```

And that’s the last problem in this chapter solved. I hope you enjoyed this excursion into Problem BASIC. The next chapter, Strings and Things, deals with problems in getting the computer to play about with names and numbers again, but this time changing them before we print them out on the screen.

~~~~~~~~~~~~~
In this chapter we shall extend our use of the BASIC language, to include string functions, and other functions of a logical nature that allow us to deal with alphanumeric characters.

Problem Twelve

Write a program to ask someone for their first name, then print out their initial, with the message 'The first letter of your name is'.

Hello, my name's Eric, what's yours?
This problem has two parts: first we need to obtain the first letter of someone's name, then we print out the message with the initial, which is a mixture of capital letters and lower case. This, of course, is quite a normal requirement.

For a solution to the first part of the problem,

```
PRINTCHR$(147):INPUT "TYPE IN YOUR FIRST NAME"; N$
```

will obtain a name from the keyboard, labelled as N$.

To get the first letter or initial we need to use the string function LEFT$(N$, X), where X will be the number of characters we want to take from N$, starting at the left hand end of N$.

So, to get the initial I$, which is the first letter of N$, calling it I$ to suit, we can use:

```
I$ = LEFT$(N$, 1)
```

Now we need to print it out.

To get both capitals and lower case together in one string, we need first to engage lower case mode before typing in the string, so that we can use the shift key to put in the capital T in The.

So the next thing to do is press the Commodore and Shift keys together and then type:

```
print "The first letter of your name is "; I$
```

Naturally we could have made the problem a whole lot easier by printing out the final message all in capitals or upper case, but this is a book of logic problems after all.

In future we will only use upper case mode, now you know how to use both. Both upper and lower case letters are essential when writing educational programs for the primary area.

Our program could therefore, be:

```
10 PRINTCHR$(147); INPUT "Type in your first name"; n$
20 I$ = LEFT$(n$, 1)
30 PRINTCHR$(147); PRINT "The first letter of your name is "; I$
```

If the response to the INPUT request is Fred, you would get a capital F for I$. Also you could have selected lower case mode before you started to type in the program, and the program would still have looked the same.

But, if you change back to upper case mode before you RUN the program, you will
find that the capital T has now turned into the shifted graphic on the T key. This of course will also occur if a program containing this routine were loaded and RUN after switching on the computer.

So we now have another problem. How can we write programs with both upper and lower case letters in them, which run correctly when loaded?

The answer of course is to tell the computer to switch to lower case mode, as soon as the program starts. To do this we must include at the beginning of the program the necessary statement.

I told you CHR$ would turn out to be important. This is the statement we use, tied to the required ASCII or CHR$ code. On page 135 of your 64 manual, you will find that this code is 14, so we must include the required extra line at the beginning of our program, for example:

```
5 print chr$(14)
```

Problem Thirteen

Write a program to ask someone for their first name, and then create a nickname for them, by taking the first four letters and adding a suitable ending, together with a suitable message.

This problem is reasonably straightforward, in that it is an extension of problem twelve.

The only difference is that this time we require to take the first four letters of the input name N$. So:

```
I$=LEFT$(N$,4)
```

should suffice to do this.

Adding a suitable ending, can be achieved in a couple of ways, for example:

```
NN$=I$+"LY"
PRINT NN$
```

where LY is a suitable ending, or

```
PRINT I$;"LY"
```
the choice is up to you.

So the program is:

```
10 PRINTCHR$(147):PRINT"TYPE IN YOUR FIRST NAME":INPUT N$
20 I$=LEFT$(N$,4):NN$=I$+"LY"
30 PRINTCHR$(147):PRINT N$;"YOUR NICKNAME IS ";NN$
```

where NN$ stands for ‘nickname’.

You could of course dress up the final screen a little better, by placing the nickname and message centrally on the screen, by using empty PRINT lines and a PRINTTAB statement. I’ll leave that up to you.

Problem Fourteen

Now write a program to ask someone for their first name, then spell their name backwards on the screen, together with a suitable message.

After getting the name into memory under N$, we find we have two problems to solve.

Reading each letter of the name into memory, and then printing them out. We’ll limit the length of name to ten letters, which is a useful dodge to stop users of the program from entering a continuous string of letters.

To read all the letters of N$ into memory, we need to read each one in turn, using another form of string slicing, as LEFT$(N$,X) always starts at the left hand side, and reads the letters up to position X.

The function we need is, of course, MID$(N$,Y,X), where Y is the position in the word where you want to begin, and X is the number of letters you need, as before. For example,

```
MID$(N$,1,4)=LEFT$(N$,4)
```

But, of course, we need to change the value of Y in MID$(N$,Y,X) after each letter is read into memory. To do this, the routine must be placed in a loop, and each letter given a subscript, for example L$(1)$, L$(2)$, etc.

There are two ways of realising this: One way is to use a loop plus an increment, for example:

```
P=1:FOR R=1 TO L:L$(R)=MID$(N$,P,1):P=P+1:NEXT
```
With this routine each letter will be placed into memory, providing we have used 
$L = \text{LEN}(\text{N$})$, to get a value for $L$. Each time the loop runs, $P$ is incremented by 1, 
until $L$ is exhausted, so that for example, for the name FRED, we would have:

$$\begin{align*}
L$(1) &= F \\
L$(2) &= R \\
L$(3) &= E \\
L$(4) &= D
\end{align*}$$

To achieve the same thing without incrementing $P$, we can use:

$$\text{FOR } R = 1 \text{ TO } L: L$(R) = \text{MID$(N$, R, 1): NEXT}$$

To print the name backwards, we need to take the letters, starting at the end of the 
array, that is $L$(4) or D in the above example, and PRINT them out.

To do this, we use another loop:

$$\text{FOR } R = L \text{ TO } 1 \text{ STEP -1: PRINT } L$(R): NEXT}$$

This will print out the letters on the screen; notice the ; after $L$(R) to tell the 
computer to print each letter on the same line.

Our complete program, using the two loops, could look like this:

```
10 PRINTCHR$(147): INPUT"TYPE IN YOUR FIRST NAME": N$
20 L = \text{LEN}(N$): IF L > 10 THEN GOSUB 100: GOTO 10
30 FOR R = 1 TO L: L$(R) = \text{MID$(N$, R, 1): NEXT
40 PRINTCHR$(147)
50 PRINT N$
60 PRINT"YOUR NAME SPELT BACKWARDS IS": FOR R = L TO 1 STEP -1
70 PRINT L$(R): NEXT
80 GET K$: IF K$ = "": THEN 80
90 GOTO 10
100 PRINT: PRINT "YOUR NAME IS TOO LONG"
110 PRINT: PRINT "I ONLY WANT TEN LETTERS"
120 FOR D = 1 TO 2000: NEXT: RETURN
```

Line 80 allows the final screen to remain until any key is pressed, whereupon the 
program will repeat.

Line 20 takes care of the occasions when the inputted name is longer than ten 
letters. By the way leave the IF...THEN part of line 20 out of the program, and you 
could get a bad subscript command, because arrays of more than ten elements 
require the array to be dimensioned.

So I killed two birds with one stone.
Now write a program to ask someone for their name, but this time print out only the vowels from it, on a clear screen, and with a suitable message.

Our problem this time is quite similar to problem fourteen, in that we have to read each letter in turn, but then look at it to see if it happens to be a vowel or not, and if it is, print it out at the right time.

To check whether each letter is a vowel or not is our main problem.

To do this we must check each letter in turn, and compare it with the letters A, E, I, O and U:

\[
\text{IF } V(R) = "A" \text{ OR } V(R) = "E" \text{ OR } V(R) = "I" \text{ OR } V(R) = "O" \text{ OR } V(R) = "U" \text{ THEN PRINT } V(R) \]

Will take care of this, as the IF...THEN statement will check \( V(R) \), where \( V = \text{vowel} \), against each vowel, and only carry out a PRINT statement if any of the comparisons are true, making full use of the OR function.

Our program could be:

```plaintext
10 DIM NAME$(40):P=1
20 PRINTCHR$(147):PRINT:"PLEASE TYPE IN YOUR NAME"
30 INPUT NAME$: P=1: L=LEN(NAME$)
40 IF L>40 THEN 20
50 FOR R=1 TO L: V$(R)=MID$(NAME$,P,1): P=P+1: NEXT
60 PRINTCHR$(147):PRINT:"HELLO";NAME$:PRINT
70 PRINT:"YOUR NAME HAS THE FOLLOWING VOWELS:";
80 PRINT: FOR R=1 TO L
90 IF V$(R) = "A" OR V$(R) = "E" OR V$(R) = "I" OR V$(R) = "O" OR V$(R) = "U" THEN PRINT V$(R); " , ";
100 NEXT
110 GET K$: IF K$="": THEN 110
```

You will notice here that I have decided to dimension the string A$ for a full screen width of 40 characters, (see line 10 – DIM NAME$(40)), as you perhaps would not get many vowels in ten letters.

Also, just to show that the incrementing routine works, discussed in the previous problem, I have used it again in line 50.

Line 90 is a bit squashed, as I wanted to get all the comparisons in on one line to avoid five IF...THEN statements: that’s why I left out all of the spaces around the OR’s.
Problem Sixteen

Read a list of n pairs of words into memory, choose one pair randomly, and display the first member of that pair on the screen. Ask the user to type in the corresponding member of the pair, repeat until the user wishes to stop, and keep a score.

One of the biggest difficulties encountered, when solving problems, is understanding what the problem actually requires you to do. An incorrect interpretation at this stage can result in the wrong problem being solved and failure. Beautiful programs, but you still have your problem to solve.

To illustrate this potential pitfall, problem sixteen asks you to read a list of words into memory: this is a good start, as it naturally leads you to think of a READ and DATA statement.

The problem could have said, ‘from a list of words in memory’, which would have allowed you to use LET statements instead, and have made the solution a little easier. But it didn’t, so you can’t.

So, the first thing we have to do is compose a READ and DATA statement or statements to get our words into memory, together with their labels, so, for example, using just four words:

```
FOR R=1 TO 4
READ W$(R)
NEXT
DATA DOG,CAT,BOY,GIRL
```

would put the four words into memory with labels W$(1) to W$(4), so that:

- W$(1) = DOG
- W$(2) = CAT, etc.

we could then generate a random number between 1 and 4 and use it to select a random word from the list above.

But we need to have random pairs of words, so that the word inputted from the keyboard can be checked against the second word of the pair, after the first one has been shown on the screen. The program could be changed to:

```
FOR R=1 TO 2
READ W1$(R),W2$(R)
NEXT
DATA DOG,CAT,BOY,GIRL
```
This now pairs DOG with CAT using W1$(1)$ and W2$(1)$, and BOY with GIRL.

Now W1$(1)$, or whatever string was chosen by the random number that was selected, could be displayed and the answer compared with W2$(1)$, but this would mean reading all the words from the DATA statements into memory each time the program was RUN, and choosing a random one from all the pairs.

A second way of solving this problem, would be to read a random quantity of pairs of words and then printing one from the last pair, for example:

\[
N=\text{INT}(\text{RND}(1)*2))+1\\
\text{FOR } R=1 \text{ TO } N:\text{READ } W1$,W2$\\
\text{NEXT}\\
\text{DATA DOG,CAT,BOY,GIRL}\\
\text{PRINT } W1$
\]

would print either DOG or BOY on the screen, and then we could have the answer compared with W2$, which is CAT or GIRL This, then is the main part of the problem solved.

You are asked to keep score, so everytime the answer is correct, the score must be incremented, remembering too of course, that you were asked to keep the program running until the user wanted to stop.

The trouble is that if the program is to run continuously, then the computer will eventually run out of words to read from the DATA statements, and you’ll be presented with a DATA error. So we must include a RESTORE statement in the program, which makes the next READ take its input as the first member of the DATA statement (DOG in our case).

Each time the program is RUN anew, the score must be made zero, so our program could look like this:

\[
10 \text{ S}%=0\\
20 \text{ RESTORE:N=INT(RND(1)*10)+1}\\
30 \text{ FOR } R=1 \text{ TO } N:\text{READ } W1$,W2$:\text{NEXT}\\
40 \text{ PRINTCHR$(147)\text{PRINT}\\"WHAT WORD GOES WITH }";W1$," ?\text{"}\\
50 \text{ PRINT:INPUT A$:IF A$=W2$ THEN PRINT:PRINT\\"YES, WELL DONE, IT'S }";W2$;S%=S%+1:GOTO 70\\
60 \text{ PRINT:PRINT\\"SORRY, THE WORD YOU WANT IS }";W2$\\
70 \text{ GET K$:IF K$="E" \text{ THEN GOTO 20:REM E TO END}\\
80 \text{ IF K$<"E" \text{ THEN GOTO 20:REM E TO END}\\
90 \text{ PRINT:PRINT\\"BYE, YOUR SCORE IS";S%:END}\\
100 \text{ DATA BAT,BALL,DOG,CAT,HORSE,CART,WHITE,BLACK,SPADE, BUCKET}\\
110 \text{ DATA OUT,IN,DOWN,UP,COLD,HOT,FRONT,BACK,BUTTER,BREAD}
\]

You can of make your own choice of words for the DATA statements, and make
them as difficult as you like. You could also increase the number of pairs, by increasing the range of your random number generator, and the number of pairs of words to match in lines 100 and 110.

We now come to the ‘and things’ part of this chapter.

* ~*****************************************************************************
* Problem Seventeen
* ~*********************************************************************

Initially, the problem we have to solve is asking how old someone is, and then to calculate what year they were born in.

If you find out their age, then you could subtract this from the calendar year, and give them the answer, for example:

```
10 INPUT"TYPE IN YOUR AGE";AGE
20 BORN=1984-AGE
30 PRINT"YOU WERE BORN IN";BORN
```

But you will soon realise that you could be a whole year out, providing you tested your program fully by running it with a number of different friends and relations.

**WHY?**

Well, consider whether the friend or relation has yet had a birthday this year or not.

If they have, they will be a calendar year older than if they haven’t.

For example; say you were born in 1952, and had a birthday in November, but the present month is April 1984. You would be 31, but the program as it stands would tell you you were born in 1953. Run the program again when December comes round and you’ll find you were born in 1952, which is correct of course.

So we realise that whether or not you’ve had a birthday is important to the correct running of the program, and must therefore be allowed for.

Also, say you ran the program in 1985, you would again get a wrong answer, because you would be year older, 33, in December 1985, but the program would not have allowed for this. When RUN it would show you having been born a whole year earlier than you actually were, 1951, that is 1984-33. So this must be
allowed for in the program too.

A suitable solution therefore would be:

```plaintext
10 PRINTCHR$(147)
20 INPUT "TYPE IN YOUR NAME"; NAME$: INPUT "TYPE IN THE YEAR"; YEAR
30 INPUT "TYPE IN YOUR AGE"; AGE
40 PRINT "HAVE YOU HAD A BIRTHDAY YET THIS YEAR?"; INPUT "ANSWER YES OR NO"; ANS$
50 IF ANS$= "NO" THEN 80
60 IF ANS$= "YES" THEN 90
70 IF ANS$< "NO" THEN PRINTCHR$(147): GOTO 40
80 PRINTCHR$(147): BORN=(YEAR-AGE)-1: PRINT "HELLO "
    NAME$: "YOU WERE BORN IN"; BORN: GOTO 100
90 PRINTCHR$(147): BORN=YEAR-AGE: PRINT "HELLO "
    NAME$: "YOU WERE BORN IN"; BORN
100 GET K$: IF K$="" THEN 100
110 IF K$<>"" THEN 10
```

In the above program, any key will restart it, and to stop it, use the RUN/STOP key.

Line 80 allows for the time when the user hasn’t had a birthday yet, by subtracting one year from the eventual answer.

You will also note that I have used full names for the variables, NAME, YEAR, AGE, etc.; using full names does make your program easier to understand, but of course uses more memory. I find that using an abbreviation, N for name, A for age for example, is a reasonable middle course to take.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This next problem is a simple use of LEN$(X).

**************************************************************************
*                        Problem Eighteen                          *
*                                                                 *
* Write a program to ask someone for a short word, then print out that word a *
* number of times equal to the number of letters in the word, in a rectangular block *
* on the screen.                                                  *
**************************************************************************

This problem needs hardly any solving at all, I just thought you might be in need of a break, but couldn’t bring yourself to leave the computer.

Once you have the word in memory, its length can be found, and used to operate a repeat loop.
The only snag is that if anyone disregarded the word 'short', and typed in a word longer than the number of available lines on the screen, then the printing would scroll up the screen and spoil the effect. In my solution I decided to limit the word to ten characters to avoid this happening, but you can please yourself what limit to set.

The important lines required, which do all the work are:

\[
L = \text{LEN}(W$) \\
\text{FOR } R=1 \text{ TO } L: \text{PRINT } W$: \text{NEXT}
\]

which prints \(W\), the word typed in, \(L\) number of times, where \(L\) is also the number of character in \(W\), the ':' after \(W\$\) of course forces a new line every time through the loop, and you end up with a block of words.

So the solution could be as follows:

\[
10 \text{ PRINTCHR$(147)$: INPUT"TYPE IN A SHORT WORD": } W$: L = \text{LEN}(W$) \\
20 \text{ IF } L > 10 \text{ THEN PRINT"TOO LONG, TRY AGAIN": FOR } D=1 \text{ TO } 1000: \text{NEXT:GOTO 10} \\
30 \text{ PRINTCHR$(147)$: FOR } R=1 \text{ TO } L: \text{PRINT } W$: \text{NEXT} \\
40 \text{ PRINT:PRINT"ANY KEY TO GO AGAIN"} \\
50 \text{ GET } K$: \text{IF } K$="" \text{ THEN 50} \\
60 \text{ GOTO 10}
\]

But now as an extra problem, 'Extra seven', ask for a five letter word and then print that word diagonally across the centre of a clear screen.

Good luck.

- - - - - - - - - - - - - - - -

This next problem should give you a bit of help in case you had any difficulty with 'Extra seven.'

*******************************************************************************

Problem Nineteen

Write a program to ask someone their surname, then to print it centrally down the screen, starting at the third screen row. The number of times it is to be printed must equal the number of letters there are in the name, limiting the number to nineteen.

*******************************************************************************

The problem here, once you have your word or surname, is to make sure that you
get it printed centrally down the screen, and starting at the third screen row.

To start at the third row, we must print blank lines on the first two, for example:

```
FOR R=1 TO 2:PRINT:NEXT
```

or just:

```
PRINT:PRINT
```

Then we have to get, not only the name printed out the correct number of times, but in a central position as well.

To do this we must get the middle letters of the name in the centre of the screen.

So if we split the name in half, subtract the result from half the line, and then start printing at this column position, the word should appear centrally placed. If we happen to get an odd number of letters typed in, we shall end up one column out, but this can’t be helped.

You can’t print half a character/column, like you can on some typewriters, not yet anyway, so:

```
PRINTTAB(20-L/2)S$
```

should work, where L=LEN(S$), and S$ is the word or surname typed in when the program is RUN.

```
10 PRINTCHR$(147):PRINT:PRINT:”TYPE IN YOUR SURNAME”: PRINT:INPUT S$
20 L=LEN(S$):IF L<20 THEN 40
30 PRINT:PRINT”YOUR NAME IS TOO LONG, TRY AGAIN”: FOR D=1 TO 2000:NEXT:GOTO 10
40 PRINTCHR$(147):FOR R=1 TO 2:PRINT:NEXT
50 FOR R=1 TO L:PRINTTAB(20-L/2)S$:NEXT
```

should do the trick.

******************************************************************************

* Problem Twenty *

* Now write a program to ask someone their full name, then to print that name a *
* number of times on the screen, equal to the number of letters in the name, but *
* each time take away one letter, ending with just the initial of their first name. *
*
******************************************************************************

In this problem we must get the name into memory, find its length, restrict it if
necessary, then print it out as required.

You are requested to take one letter away each time you print the name, which means somewhere in your program the name must be incremented by -1, or if you prefer, decremented by 1.

But we cannot say PRINT N$-1 or N$-2, etc., it won’t work. You can, of course, use L-1 or L-2, so you could therefore break the name down into individual letters and reprint them again and again, reducing the number of letters by one each time.

So we could say:

```plaintext
L = LEN(N1$)
FOR R = 1 TO L:
    N2$(R) = MID$(N1$, R, 1):
NEXT
```
or

```plaintext
L = LEN(N1$)
P = 1:
FOR R = 1 TO L:
    N2$(R) = MID$(N1$, P, 1):
    P = P + 1:
NEXT
```
to break the name down into individual letters or character strings.

Then print it out again with:

```plaintext
FOR R = 1 TO L:
    PRINT N2$(R): NEXT
```

but you would only get the complete name.

We want the name repeated, and each time reduced by one letter, so it is better to try:

```plaintext
FOR R1 = 1 TO L:
    FOR R2 = 1 TO L:
        PRINT N2$(R2):
    NEXT
    L = L - 1:
NEXT
```

This will print the name L number of times, but each time it is printed, it will be printed one letter less, because of the L = L - 1.

So, our listing would look like this:

```plaintext
10 DIM N2$(21)
20 PRINTCHR$(147): PRINT: PRINT"TYPE IN YOUR FULL NAME":
    PRINT: INPUT N1$
30 L = LEN(N1$): IF L < 22 THEN 50
40 PRINT: PRINT"YOUR NAME IS TOO LONG, TRY AGAIN":
    FOR D = 1 TO 2000: NEXT: GOTO 20
50 PRINTCHR$(147): P = 1:
    FOR R = 1 TO L:
        N2$(R) = MID$(N1$, P, 1): P = P + 1:
    NEXT
60 FOR R1 = 1 TO L:
    FOR R2 = 1 TO L:
        PRINT N2$(R2):
    NEXT
70 L = L - 1: PRINT: NEXT
```

42
In line 70 I’ve put in an extra empty PRINT statement, to separate the names as they reduce in length down to the initial.

And now, the basis of many a good educational alphabet game.

In this problem the first thing we have to do is generate a random letter from the possible twenty six available.

Generating a random number between 1 and 26 is easy enough, but of course we can’t print it directly on the screen.

But, we can generate a random number that does equate to a letter that can be printed, so long as we remember that the numbers start at 65 not 1.

So, if we use:

\[ X1=\text{INT}(\text{RND}(1) \times 25) + 65 \]

which will give us a number between 65 and 90,

and

\[ X2=X1-64 \]

which will give us corresponding numbers between 1 and 26, we will then get a number corresponding to an ASCII code letter, \( X1 \), which can be printed, and a number corresponding to its position in the alphabet, \( X2 \).

We cannot, of course, compare any letter \( L\) typed in on the keyboard, with a number, as there would immediately be a mismatch reported, so we must find the character corresponding to \( X1 \), with \( \text{CHR}\$(X1) \), and compare it to \( L\):

\[ \text{IF} \ L\ = \ \text{CHR}\$(X1) \]

or

\[ \text{IF} \ L\ <> \ \text{CHR}\$(X1) \]
and we have solved the most difficult part of the problem.

The second part of the problem, to give users of the program two tries before telling them they're wrong, requires the number of times the user gives an answer to be recorded, starting at zero, and when the second try has been recorded, to proceed to the next part of the program.

So let's have a look at the following program, and explain it as we go:

```
10 X1=INT(RND(1)*25)+65:X2=X1-64:T=0
This generates a random number, X1, between 65 and 90, corresponding to the ASCII codes for A to Z. X2 is then the position of the letter in the alphabet, because if A=65, then 65-64=1, and A, we all know, is the first letter of the alphabet.

20 PRINTCHR$(147):PRINT"WHAT LETTER IS NUMBER";X2;"IN THE ALPHABET":INPUT L$
asks the questions and receives the letter in reply.

30 IF L$<>CHR$(X1) THEN 100
sends the computer to a part of the program where any wrong answer is dealt with, after comparing the INPUT with the character corresponding to X1.

40 PRINT:PRINT"WELL DONE, YOU'RE RIGHT"
50 PRINT:PRINT"YOU'RE CLEVER, AREN'T YOU"
60 PRINT:PRINT"PRESS ANY KEY TO END":PRINT"OR SPACE BAR TO CONTINUE"
deals with the correct answer, and asks if another go is wanted or not.

70 GET K$:IF K$=CHR$(32) THEN 10
checks the keyboard, and if the spacebar, CHR$(32), has been pressed, the program is repeated.

80 IF K$<>"" THEN END
allows the program to END if any other key is pressed.

90 IF K$="" THEN 70
makes the computer wait in a loop, until a key is pressed.

100 T=T+1:IF T>1 THEN 130
110 PRINT:PRINT"SORRY, YOU'RE WRONG, TRY AGAIN":FOR D=1 TO 2000: NEXT
```
120 GOTO 20

increments the tries, and if it’s the second try, sends the computer to line 130 to deal with this situation.

Line 110 tells the user that a wrong answer has been typed in, waits awhile so that there's enough time for the screen to be read, then runs through the program again, but with the same random number as before.

130 PRINT:PRINT"WRONG AGAIN, DO YOU WANT ANOTHER NUMBER?":
     PRINT"ANSWER YES OR NO"
140 PRINT:INPUT A$
150 IF A$="NO" THEN END
160 IF A$="YES" THEN 10
170 IF A$<>"NO" THEN PRINTCHR$(147):GOTO 130
180 END

In this part of the program, which is only used after the second wrong try, the user can choose whether to try again or not.

I have used INPUT here instead of GET for a good reason. If, during a previous part of the program the user had pressed a key more than was required, the keyboard buffer might be already contain a character, and if GET K$ had been used on line 140, strange things might have happened, depending on which key had been pressed.

You can always improve the program somewhat, by telling the user what the correct answer should be on line 130, but I'll leave that up to you to arrange.
PEEK(X)

In this chapter we shall be dealing with basic games routines, using the RND function to produce pure games of chance. Remember, life is full of chances and coincidences, and it’s always a good idea to have some knowledge of how these random chances and coincidences come about, and just how random they indeed are. Once you have solved these problems I recommend that you let each one run on for a good while, just to see how random the computer’s random generator is. For example, if you generate 1000 whole numbers from 1 to 5 you might expect to get 200 of each, but I doubt it. You are more likely to get something like the following:

<table>
<thead>
<tr>
<th>Number</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>198</td>
</tr>
<tr>
<td>2</td>
<td>209</td>
</tr>
<tr>
<td>3</td>
<td>189</td>
</tr>
<tr>
<td>4</td>
<td>213</td>
</tr>
<tr>
<td>5</td>
<td>191</td>
</tr>
</tbody>
</table>

but experiment, and see what you get. Of course the bigger a sample you choose, the longer it takes, but the more indicative of true randomness it will seem. The computer does not have a true random generator inside it, but a psuedo one, that puts out a pre-planned series of numbers, but in such a way that it’s extremely hard to see any pattern.

Let’s hope that this chapter will create some interest in randomness and a feeling for the science of statistics, and at the same time allow you a bit of fun.
Problem Twenty Two

Write a program that produces two random numbers between 1 and 4, then, if they are the same, prints out a prize depending on the random numbers: the bigger the number, the bigger the prize; also keep the program running, and keep score.

The whole problem centres around the need to produce, or generate, two random numbers between 1 and 4, then if they happen to be the same, dish out a prize depending on the size of the two numbers. So, that means every time we have generated two random numbers, we must compare them, like this:

\[
N1 = \text{INT}(\text{RND}(1) \times 4) + 1 \\
N2 = \text{INT}(\text{RND}(1) \times 4) + 1
\]
will produce two numbers between 1 and 4, and

   IF N1=N2 THEN go on and do something else

will compare them.

Now for the subsidiary part of the solution, providing prizes to fit the occasion.

The bigger the numbers, the bigger the prize is what we need to program for.

We can judge the size of the numbers by looking at N1 each time, (or N2 of course), and we can also create some subroutine to go to, depending on the size of N1, (or N2), for example:

   IF N<2 THEN GOSUB 1000

Line 1000 could tell the winner the size of his prize, as would the other subroutines for N<3, N<4 and N=4, providing you keep these comparisons in size order.

You could of course use: N=1, N=2 and N=3 instead of N<2, N<3 and N<4, it's up to you.

To save on programming space and memory, we can create one small subroutine to draw out the prize, if we are actually going to give one, and the problem does say, 'print out a prize', not just tell the winner what he's won.

For prizes we can draw some coloured graphics characters, the number that we draw, depending on the size of the prize.

So starting at a particular location on the screen we can draw the prize, which can be a block of the particular characters we've picked.

In order to make the screen display spacing even, it's also a good idea to alter the gap between the writing, which tells them what size prize they've won.

So, let's make a start with the following routine:

1. S=1287-P
2. FOR L=1 TO P
3. FOR R=1 TO P
4. POKE S+54272,8:POKE S,C
5. S=S+1
6. NEXT
7. S=S+40-P
8. NEXT

and the following explanations:
1. positions the prize, starting at 1 28 7, minus a value depending on the size of the prize,
2. and 3. set up two loops to draw the character we’ve selected for the prize across the screen P times, and with P number of lines,
4. POKEs the character, the required number of times and colours it,
5. moves the POKE position one space to the right,
6. completes the row loop,
7. moves the POKE position to the beginning of the next row, 40–P,
8. completes the loop for the block of characters.

This routine can now be placed in a subroutine, all on a couple of lines. I’ve used 5000 and 5010 in the final program.

We must also keep the program running and keep score, which must be zero at the beginning of the program, and then incremented each time the player wins, but by a different value depending on the size of the prize.

We can use GET K$ to keep the program running.

I’ve dressed this program up a little with an initial start screen, you of course can do a lot more with it if you wish. You could in fact make it a lot more interactive by not having N2 as a random number, but a number inputted from the keyboard by the player, using the required INPUT statement. I’ll leave this up to you, but it’s simply done, but for once you’re on your own. Sometimes in life, that’s the way the cookie crumbles.

My solution program looks like this:

```
10 PRINTCHR$(147);TAB(3)"PRESS SPACE BAR FOR YOUR FIRST GO"
20 GET K$:IF K$=CHR$(32) THEN 50
30 IF K$<>"" THEN 20
40 IF K$="" THEN 20
50 S%=0
60 N1=INT(RND(1)*4)+1:N2=INT(RND(1)*4)+1: PRINT:PRINTTAB(14) "I’M THINKING"
70 FOR D=1 TO 1500:NEXT
80 IF N1=N2 THEN 200
90 PRINT:PRINTTAB(15)"YOU LOST!"
100 PRINT:PRINTTAB(12)"YOUR SCORE IS";S%:
110 PRINT:PRINTTAB(11)"OR ANY KEY TO END"
120 GET K$:IF K$=CHR$(32) THEN PRINTCHR$(147):GOTO 60
130 IF K$="" THEN 130
140 IF K$=CHR$(32) THEN PRINTCHR$(147):GOTO 60
150 IF K$="" THEN 130
200 IF N1<2 THEN GOSUB 1000:GOTO 60
210 IF N1<3 THEN GOSUB 2000:GOTO 60
220 IF N1<4 THEN GOSUB 3000:GOTO 60
230 IF N1=4 THEN GOSUB 4000:GOTO 60
```

49
1000 FOR R=1 TO 6:PRINT:NEXT
1010 PRINTTAB(5)"YOU ARE AWARDED A LITTLE PRIZE"
1020 P=2:C=83:S%=S%+1:GOSUB 5000:RETURN

2000 FOR R=1 TO 9:PRINT:NEXT
2010 PRINTTAB(5)"YOU ARE AWARDED A MEDIUM PRIZE"
2020 P=4:C=83:S%=S%+2:GOSUB 5000:RETURN

3000 FOR R=1 TO 12:PRINT:NEXT
3010 PRINTTAB(6)"YOU ARE AWARDED A BIG PRIZE"
3020 P=6:C=83:S%=S%+3:GOSUB 5000:RETURN

4000 FOR R=1 TO 13:PRINT:NEXT
4010 PRINTTAB(5)"YOU ARE AWARDED A GRAND PRIZE"
4020 P=6:C=90:S%=S%+4:GOSUB 5000:RETURN

5000 S=1287-P:FOR L=1 TO P:FOR R=1 TO P:POKE S+54272,8:
POKE S,C:S=S+1:NEXT
5010 S=S+40-P:NEXT
5020 FOR D=1 TO 2000:NEXT:PRINT:PRINTTAB(15)"ANOTHER GO?"
5030 PRINT:PRINTTAB(12)"SPACE BAR FOR YES"
5040 PRINT:PRINTTAB(13)"ANY KEY FOR NO"
5050 GET K$:IF K$=CHR$(32)THEN PRINTCHR$(147):RETURN
5060 IF K$="" THEN 5050
5070 IF K$<"" THEN PRINT:PRINTTAB(12)"YOUR SCORE WAS";S%:END

Lines 10 to 50 only come into play when you initially type RUN
Lines 60 to 80 find the two numbers and compare them, we also have a ‘fun delay’
the computer is not really thinking!
Lines 90 to 150 is the ‘lose’ routine.
Lines 200 to 230 decide what to do with a ‘win’, and send the computer back to
play the game again, when asked to do so by line 5050.
Lines 1000 to 4020 set up the parameters for the various prizes, depending on
the result of 200 to 230, and print out a winning message.
Lines 5000 to 5070 draw the prize on the screen and asks if the player wants
another go, and then prints out his score if not.
You can always change the prizes to suit yourself, but be careful not to go outside
the range of the screen POKE values 1024 to 2023 and 55296 to 56295, as
strange things will happen to your program, which can be quite frustrating at
times, and may even mean you might have to type your program in all over again.
POKEing memory locations outside the screen ranges, can change your program
drastically, as most of this memory area is used by the computer to store your
program, and RUN it.
• Problem Twenty Three

• write a program that will continuously produce two random numbers between 1 and 10, give the first number to one player and the second to another, the first player to reach 21 loses. Also, print out all the numbers on the screen in two columns, players’ names at the top of the columns.

Again, a problem needing to produce two random numbers, but we should know how to do that by now.

Once we’ve got the two numbers, we must give the first to player one and the second to player two, and then produce two more numbers, which add to the first two, and so on, until one total reaches 21. We must also print the numbers in columns on the screen, with the players names at the top, so:

```
FOR R=1 TO 2:PRINT:NEXT
PRINTTAB(5)"PLAYER ONE";TAB(25)"PLAYER TWO"
PRINTTAB(5)P1$;TAB(25)P2$
N1=INT(RND(1)*10)+1:N2=INT(RND(1)*10)+1
```

will set up the columns with the names, once we’ve got P1$ and P2$ from the keyboard, then:

```
PRINT:PRINTTAB(9)N1:T1=T1+N1
IF T1>20 THEN GOSUB 1000
```

will check the totals and print out each number for player one, and we can have a similar routine for player two, changing N1 for N2, and T1 for T2.

On GOSUB 1000 we can write the totals and who has won or lost, and ask if another go is required. Naturally, by this time we should be dressing up our simple programs to produce a reasonable screen presentation too, so, for example:

```
10 PRINTCHR$(147):PRINT:"THIS GAME IS FOR TWO PLAYERS"
20 PRINT:INPUT"PLAYER ONE, NAME PLEASE ";P1$
30 PRINT:INPUT"PLAYER TWO, NAME PLEASE ";P2$
40 PRINTCHR$(147)
50 T1=0:T2=0
60 FOR R=1 TO 2:PRINT:NEXT:PRINTTAB(5)"PLAYER ONE"; TAB(25) "PLAYER TWO"
70 PRINTTAB(5)P1$;TAB(25)P2$
80 N1=INT(RND(1)*10)+1:N2=INT(RND(1)*10)+1
90 PRINT:PRINTTAB(9)N1;;T1=T1+N1
100 IF T1>20 THEN GOSUB 1000:GOTO 2000
110 PRINTTAB(29)N2:T2=T2+N2
120 IF T2>20 THEN GOSUB 1000:GOTO 2000
```
130 GOTO 80
1000 PRINT: PRINT: PRINTTAB(3)"TOTAL"T1; TAB(23)"TOTAL"T2
1010 IF T1>20 THEN S$="ONE": GOTO 1030
1020 IF T2>20 THEN S$="TWO"
1030 PRINT: PRINTTAB(8)"PLAYER "; S$" YOU HAVE LOST"
1040 RETURN
2000 PRINT: PRINTTAB(9)"ANOTHER GO, YES OR NO?"
2010 GET K$: IF K$="Y" THEN PRINTCHR$(147): GOTO 50
2020 IF K$="N" THEN END
2030 IF K$>"" THEN 2010
2040 IF K$="" THEN 2010

could be our program, with lines:

10 to 30 getting the two players' names and storing them as P1$ and P2$, 50 setting the totals to zero, 60 and 70 heading up the two columns, 80 generating the two random numbers we need, 90 to 120 checking the totals for being more than 20 individually, because the problem does say the first player to reach 21. If we checked both the totals, T1 and T2, at the same time, then both players might have on screen totals of greater than 20, 130 sending the computer back to get two more random numbers, if either total is less then 21, 1000 to 1020 printing out the totals when one of them is greater than 20, and allocating to S$ the number of the player who has lost, 1030 to 1040 printing out which player has lost, and returning then to the main program.

You could have course have written:

    IF T1>20 THEN S$=P1$

and

    IF T2>20 THEN S$=P2$

which would have printed out the loser's name instead of his number.

but I'll leave that up to you to decide.

2000 to 2040 allows for another go, if the players want to continue with playing this very simple form of pontoon.

In problem twenty two it was suggested that you could make the game more interesting and interactive, if you allowed some keyboard input. Well, problem twenty four does just that.
Problem Twenty Four

Write a program that asks a player to input two numbers from the keyboard, then generate nine random numbers between 1 and 5 and print them on the screen in a column; if the random numbers are the same at the positions indicated by the keyboard numbers, then the player has won. Keep the program running.

To solve this problem we need to generate nine random numbers and print them on the screen in a column. Generating the numbers should be easy by now, and so is printing them out, if we use a PRINT statement. But this isn’t very sophisticated, so perhaps POKE statements would be a better way of doing it, especially when you consider that you must also indicate which rows have been chosen.

With a PRINT routine, this could lead to far too many complications. For example:

P=1314 (you’ve got to choose somewhere to start, this is as good a place as any)

FOR R=1 TO 9
    N=INT(RND(1)*5)+48

    (if we’re going to use POKE, then we need ASCII codes, not just the numbers 1 to 5), this instruction will produce numbers between 48 and 52.

    POKE P+54272,0:POKE P,N:P=P+40:NEXT

    (this POKEs random numbers between 1 and 5, in nine positions in one column).

Now we have to check which rows were chosen by our fictitious player, and whether they hold the same number.

We know the numbers chosen for the rows, and each row has forty character positions, so if we start from 1314–40, (the row above our first random number), and multiply this row number by forty, we will be able to check each character position used with a PEEK statement.

But, we also want to indicate the row number on the screen, so we can POKE an indicator alongside the random number at the same time. So let’s change 1314–40 to 1317–40, which moves the character position over three places.

Also we have to check if these two random numbers are the same, so as

1317–40=1277, then:
P=1277:POKE P+(40*N1)+54272,1:POKE P+(40*N1),81
P=1277:POKE P+(40*N2)+54272,1:POKE P+(40*N2),81
IF PEEK(P+(N1*40)-3)=PEEK(P+(N2*40)-3) THEN they are the same, will do the trick. Commodore code 81 is a full circle or ball in set 1, or a Q in set 2, where the POKE numbers correspond to the ASCII codes.

The third line uses PEEK(X), which tells the computer to look in the memory location P+(N1*40)-3, compare it with what is in the other location, P+(N2*40)-3, and if the contents are the same, (that is, both random numbers placed there by the POKE loop are the same), then the player has won. If they are not the same, then the player has lost.

To indicate this we can use:

```
FOR R=1 TO 12:PRINT:NEXT
PRINT"WELL DONE, YOU'RE RIGHT, ANOTHER GO?"
```

or

```
FOR R=1 TO 12:PRINT:NEXT
PRINT"WRONG, DO YOU WANT TO TRY AGAIN?"
```

and then we will need a routine to decide on the next action to be taken.

The PEEK function is very useful, especially in programs with a games element, it looks out for collisions and coincidences, and possible near misses.

Our solution now will look like this:

```
10 PRINTCHR$(147)
20 PRINT:PRINT:"TYPE IN TWO DIFFERENT ROW NUMBERS BETWEEN 1 AND 9":PRINT:INPUT N1:INPUT N2
30 IF N1=N2 THEN 10
40 IF N1>9 OR N2>9 THEN 10
50 IF N1<1 OR N2<2 THEN 10
60 P=1314:FOR R=1 TO 9:N=INT(RND(1)*9)+48:POKE P+54272,0:POKE P,N
70 P=P+40:NEXT
80 P=1277:POKE P+(40*N1)+54272,1:POKE P+(40+N1),81
90 P=1277:POKE P+(40*N2)+54272,1:POKE P+(40*N2),81
100 IF PEEK(P+(N1+40)-3)=PEEK(P+(N2*40)-3) THEN 120
110 FOR R=1 TO 12:PRINT:NEXT:PRINT"WRONG, DO YOU WANT TO TRY AGAIN?":GOTO 200
120 FOR R=1 TO 12:PRINT:NEXT:PRINT"WELL DONE, YOU'RE RIGHT, ANOTHER GO?":GOTO 200
200 REM GO AGAIN ROUTINE
210 PRINT:PRINT"PRESS SPACE BAR TO GO AGAIN"
220 PRINT:PRINT"ANY KEY TO END"
```
We continue now with our random games problems, but this time deal with letters instead of numbers.

The first part of the problem, which shouldn’t be a problem by now, is to generate three random letters, L1, L2 and L3, (actually numbers with the correct ASCII codes). But we are asked that the letters generated must be all different, so we must incorporate a comparison routine to eliminate unwanted repeats, so:

IF L1=L2 OR L1=L3 OR L2=L3 THEN do it again

should deal with that. Notice how we use the logic operator OR to compare the three variables.

The player has to input one letter from the keyboard, and this is then compared with L1, L2 and L3. But L1, L2 and L3 will be numbers that you have generated, and the keyboard input will be a letter, so in our comparison routine we must turn L1, L2 and L3 into the same type of string as the input.

To do this we can use:

L$=CHR$(L1) etc.

which returns the ASCII code for L1 as a string.

The player is allowed three chances to get the correct letter, so to save space we can use the OR function again, and use another variable for checking, so:

IF L$=CHR$(L1) OR L$=CHR$(L2) OR L$=CHR$(L3) THEN R=1 will suffice.

We are also asked to keep score, and a record of the number of tries. This is quite straightforward, by putting S for the score, and T for the tries to zero,
at the beginning of the program, and incrementing them as required in the check routine.

Our program should look something like this:

```
10  S=0:T=0:R=0
20  L1=INT(RND(1)*25)+65:L2=INT(RND(1)*25)+65:  L3=INT(RND(1)*25)+65
30  IF L1=L2 OR L1=L3 OR L2=L3 THEN 20
40  PRINTCHR$(147):PRINT'GUESS THE LETTER I HAVE IN MY MEMORY':PRINT:INPUT L$
50  GOSUB 1000:IF R=1 THEN PRINT:GOTO 70
60  PRINT:PRINT"HARD LUCK, YOU'RE WRONG":PRINT:PRINT"DO YOU WANT TO TRY AGAIN?":GOTO 80
70  PRINT'WELL DONE, YOU HAVE WON':PRINT:PRINT"DO YOU WANT TO TRY AGAIN?"
80  PRINT:PRINT"TYPE 1 FOR YES OR 2 FOR NO"
90  GET K$:IF K$=CHR$(49) THEN 20
100 IF K$=CHR$(50) THEN 130
110 IF K$="" THEN 90
120 IF K$<>"" THEN 90
130 PRINT:PRINT"YOU'RE SCORE WAS";S
140 PRINT:PRINT"AND YOU HAD";T;"TRIES"
150 PRINT:PRINT"GOODBYE"
160 END
1000 IF L$=CHR$(L1) OR L$=CHR$(L2) OR L$=CHR$(L3) THEN
   S=S+1:R=1:T=T+1:RETURN
1010 R=2:T=T+1:RETURN
```

We do not have to say yes or no for continuing or otherwise with our program, in fact we can use any key, but a normal alternative is 1 and 2, as in line 80. But of course the IF...THEN statements associated with it would be different, as in lines 90 and 100, I have used CHR$(49) for 1, and CHR$(50) for 2; look them up in your ASCII code tables.

~~~~~~~~~~~~

We now move on to simple animated graphics.

* Problem Twenty Six *

* Now write a program that fires a missile from different places at the bottom of the screen, to a randomly placed target at the top, keep score, and keep the program running. *
Here we must produce a randomly placed target and a randomly placed ‘gun’, the former at the top of the screen, and the latter at the bottom. So we need two random numbers relevant to those two areas of the screen.

You should have no trouble with this except that perhaps your limits may be different to mine. I chose only four possible firing positions, and four possible target positions, with:

\[
\begin{align*}
T &= \text{INT} (\text{RND}(1) \times 4) + 1082 \\
G &= \text{INT} (\text{RND}(1) \times 4) + 1922
\end{align*}
\]

T = target
\[
G = \text{gun, (and shell)}
\]

T in the centre area of the top of the screen, and G in the centre area at the bottom.

You need also to choose a character for the target, I choose a spade, 65, and a ball, 81, for the shell.

Now all you have to do is fire the gun, and hope it hits the target, your chances are one in sixteen. To fire the gun we can use POKE statements, so that the shell moves vertically up the screen, so G=40 is the increment to use, and of course we do not want the shell straying down outside the screen memory area, so we must trap it before it does, with: IF G>1103 let it carry on moving.

If we POKE the shell in all the positions up the screen, we will of course have a complete line of shells, which, while pretty, is not realistic. We must therefore remove each shell once it’s moved on up the screen. To do this use POKE G,32, a blank space to cover it up, or more correctly remove it.
Our program could look like this:

```
10 PRINTCHR$(147):S%=81:S%=0
20 T=INT(RND(1)*4)+1082:POKE T+54272.0:POKE T.65
30 G=INT(RND(1)*4)+1922
40 POKE G+54272.1:POKE G,B
50 IF G>1103 THEN D=1 TO 50:NEXT:POKE G,32:G=G–40:GOTO 40
60 IF PEEK(T)=81 THEN S%=S%+1:GOTO 90
70 FOR D=1 TO 500:NEXT:PRINT:"OH DEAR, YOU MISSED ME"
80 PRINT:PRINT:"YOUR SCORE IS";S%;GOTO 120
90 PRINTTAB(T-1066)"CRASH";FOR D=1 TO 1000:NEXT
100 PRINTCHR$(147):PRINT"WELL DONE, YOU HIT ME"
110 PRINT:PRINT:"YOUR SCORE IS";S%
120 PRINT:PRINT"ANOTHER GO?
130 GET K$:IF K$="Y" THEN 10
140 IF K$="N" THEN 170
150 IF K$="" THEN 130
160 IF K$<>"" THEN 130
170 PRINTCHR$(147):PRINT"GOODBYE, YOUR SCORE WAS";S%
180 FOR D=1 TO 2000:NEXT:END
```

On line 90 I have made the word CRASH appear when you score a hit, this makes for an interesting sideline to the program, and I’ve written it so that the A of CRASH appears where the collision took place, this centralises the word each time. Can you follow the reasoning for PRINTTAB(T–1066)?

Also, as we are now at the start of the second half of the problems, I have introduced an extra problem. See if you can find it when you run the solution, it’s not too obvious, then when you’ve found it, correct it.

The answer is in the last chapter.

```
To continue on this theme, and to end this chapter, you now have to extend problem twenty six.
```

Problem Twenty Seven

Now extend solution twenty six so that the target fires back if the shell misses.

To do this we must extend that part of the program that registers a miss, and also leave something behind at the gun area for the target to fire at.
The beginning of the program can remain the same, but we need to have two POKE characters, one for the extra missile and one for the gun G. So, when we generate our random number for the gun, it can also be the extra missile P. We can say therefore:

\[ G = \text{INT}(RND(1) \times 4) + 1922 \]

and

\[ P = G: \text{POKE } G+54272,7: \text{POKE } G,88 \]

using 88, a club, as the gun, allowing P to run up the screen, leaving G behind at 'home'.

We can check for a hit in the normal PEEK(X) fashion, and then have another separate routine for a miss, which fires a projectile back again.

But if the target justs fires back from its position at the start of the run, then it is bound to miss the gun, if the gun missed the target! So what must we do to allow for this? One solution is to move the target to a new position, and then let it fire back.

But if the target happens to be at the right hand end of the limits we imposed for its position, it must be moved to the left, and vice versa.

So, the new number we generate must allow for this. We generated the original random positions, namely 1082, 1083, 1084 and 1085, so:

\[ \text{IF } T > 1084 \]

will allow us to do something for the right hand side, and:

\[ N = \text{INT}(RND(1) \times 2) + 1 \]

\[ \text{IF } T > 1084 \text{ THEN } T = T - N \]

will move it back towards the left, and

\[ T = T + N \]

will move it to the right, when T is not greater then 1084. We can now POKE our spade or target into this new position, and fire back.

Of course, we must first blank out the old target and the old projectile, before doing so, so a subroutine could look like:

```
1000 POKE P,32:POKE T,32:N=INT(RND(1)*2)+1
1005 IF T>1084 THEN T=T-N: GOTO 1015
1010 T=T+N
```

59
1015 POKE T+54272.0:POKE T,65
1020 P=T+40
1025 POKE P+54272.0:POKE P,66

which puts the target in a new position, and then starts to increment its projectile down the screen; 66 is a rocket.

We must then take note of if and when the gun is hit, and whether the projectile is moving outside the screen memory area by:

1030 IF P>1886 THEN 1040
1035 POKE P,32:P=P+40:GOTO 1025
1040 IF P=G THEN POKE P,86:S=S-1:PRINT"HARD LUCK, YOU'RE DEAD!"
1045 IF P=G THEN PRINT:PRINT"YOUR SCORE IS";S:GOTO 2000
1050 POKE P,32:PRINT"YOU'RE LUCKY, I MISSED YOU":PRINT:PRINT"YOUR SCORE IS";S

Line 2000 can be another 'another go?' routine, following on straight after 1050.

Line 1035 completes the travel of the target's projectile down the screen.

My full solution looks like this:

10 PRINTCHR$(147)
20 T=INT(RND(1)*4)+1082:POKE T+54272.0:POKE T,65
30 B=81:P=INT(RND(1)*4)+1922
40 G=P:POKE G+54272.7:POKE G,88:P=P-40
50 POKE P+54272.1:POKE P,B
60 IF P>1103 THEN FOR D=1 TO 50:NEXT:POKE P,32:P=P-40:GOTO 50
70 IF Peek(T)=81 THEN S=S+1:GOTO 90
80 GOTO 1000
90 PRINTTAB(T-1066)"CRASH":FOR D=1 TO 500:NEXT
100 PRINTCHR$(147):POKE G+54272.7:POKE G,88:PRINT"WELL DONE, YOU HIT ME"
110 PRINT:PRINT"YOUR SCORE IS";S:GOTO 2000
120 PRINTCHR$(147):PRINT"GOODBYE, YOUR SCORE WAS";S
130 END

1000 POKE P,32:N=INT(RND(1)*2)+1:POKE T,32
1005 IF T>1084 THEN T=T-N:GOTO 1015
1010 T=T+N
1015 POKE T+54272.0:POKE T,65
1020 P=T+40
1025 POKE P+54272.0:POKE P,66
1030 IF P>1886 THEN 1040
1035 POKE P,32:P=P+40:GOTO 1025
1040 IF P=G THEN POKE P,86:S=S-1:PRINT"HARD LUCK, YOU'RE DEAD!"
1045 IF P=G THEN PRINT:PRINT"YOUR SCORE IS";S:GOTO 2000
1050 POKE P,32:PRINT"YOU'RE LUCKY, I MISSED YOU":PRINT:PRINT"YOUR SCORE IS";S

2000 PRINT:PRINT"ANOTHER GO?"
2005 GET K$:IF K$="Y" THEN 10
2010 IF K$="N" THEN 120
2015 IF K$="" THEN 2005
2020 IF K$<>"" THEN 2005
2025 END
CHAPTER FIVE

ALPHANUMERIC

SPC(X)

This chapter, as it's title implies, deals with letters and numbers, either on their own or mixed. It's purpose is to practice solving problems in changing from letters to numbers and vice versa. We shall also be practicing placing letters and numbers on the screen, using both PRINT and POKE statements, and will also be looking at a different form of TAB(X), namely SPC(X).

But now to the first problem.

******************************************************************************~*

* Problem Twenty Eight
* *
* Write a program to ask someone to input a number between 1 and 26 from the *
* keyboard, and to print out, on screen, the alphabet letter corresponding to that *
* number.
*
******************************************************************************~*

As the problem states 'print out, on screen', but not in a particular place, then we can, without fear of solving the problem incorrectly, use a PRINT statement and not a POKE statement.

Now the main part of the problem is to change the imputed number, which we shall call N, to the variable that corresponds to it's letter in the alphabet.
Our knowledge of ASCII and CHRS codes tells us that the number 1 is not A, but something entirely different, and likewise 26 is not Z. The alphabet, starting at letter A, in ASCII code, starts at 65, and Z finishes it at 90; $65 + 25 = 90$.

So the first thing we must do is add the difference between 1 and 65 to the inputted number, to get the correct code to use in a print statement. So:

```
PRINTCHRS(N+64)
```

will print out the required letter, the difference between 1 and 65 is 64.

So try direct or command mode, no line numbers!

```
N=10
PRINTCHRS(N+64)
```

and what do you get? Do you get J, you should, J is the tenth letter of the alphabet.

Now try in program mode:

```
10 FOR N=1 TO 26
20 PRINTCHRS(N+64)
30 NEXT
RUN
```

which produces the whole alphabet, but of course, as there are less than twenty six lines on the screen, you lose the beginning.

So try:

```
10 FOR N=1 TO 26
20 PRINTCHRS(N+64);
30 NEXT
RUN
```

which prints them out across the screen, did you notice the difference? Remember a ';' in a PRINT statement tells the computer to print out the separate bits of information next to each other on the same line.

To put a space between each letter you could change line 20:

```
20 PRINTCHRS(N+64);" ";
```

or to print the letters in columns, try:

```
10 PRINTCHRS(147):FOR N=1 TO 26:PRINTCHRS(N+64):NEXT
```

All this of course is by way of revision of how to place characters on the screen.
using just PRINT statements and punctuation.

We can of course use PRINTTAB(X)

Try this:

```
10 PRINTCHR$(147):FOR N=1 TO 20:PRINTTAB(N)CHR$(N+64):NEXT
```

which prints the first twenty letters of the alphabet diagonally across the screen, or

```
10 PRINTCHR$(147):FOR N=7 TO 26:PRINTTAB(N)CHR$(N+64):NEXT
```

which prints the last twenty, but starting at the eighth column, because the TAB starts at N, and N is equal to 7, at the start of the loop.

To get back to the left hand side of the screen we can use:

```
10 PRINTCHR$(147):FOR N=7 TO 26:PRINTTAB(N-6)CHR$(N+64):NEXT
```

Finally to get characters placed in particular positions on the screen, we can use another PRINT statement, PRINTSPC(X), as:

```
10 PRINTCHR$(147):FOR N=7 TO 26:PRINTSPC(N-6)CHR$(N+64):NEXT
```

shows that it produces the same effect as TAB(X), when each TAB or SPC statement starts on a new line. But with more than one TAB or SPC statement on a line, TAB counts from the lefthand side of the screen, and SPC counts from the last SPC statement. As in:

```
20 PRINTTAB(2)"THAT'S";TAB(9)"ALL";TAB(13)"FOLKS"
30 PRINTSPC(2)"THAT'S";SPC(9)"ALL";SPC(13)"FOLKS"
```

Did you notice the difference?

It can be used on it's own, as:

```
40 PRINTSPC(12)"THAT'S ALL FOLKS"
```

ends our alphabet sequence in the middle of the screen.

So back to the problem, we can now print our alphabet letter anywhere on the screen, once we're given it's position in the alphabet.

So a program like the following will solve our problem:

```
10 PRINTCHR$(147)
20 PRINTTAB(12)"GIVE ME A NUMBER";PRINT:PRINTTAB(12)
   "BETWEEN 1 AND 26"
```
30 PRINT:PRINTSPC(18):INPUT N
40 IF N<1 OR N>26 THEN 10
50 FOR R=1 TO 6:PRINT:NEXT
60 PRINTSPC(8):"THE LETTER CORRESPONDING":PRINT:PRINTSPC(11)
   "TO YOUR NUMBER IS:"
70 PRINT:PRINTSPC(19):CHR$(N+64)
80 FOR D=1 TO 5000:NEXT:GOTO 10

Line 30 puts our INPUT request in the middle of the screen, as you can’t have
INPUTTAB(X), but don’t forget you must have a semi-colon after the spaces
statement, otherwise a following colon would force a new line, and the INPUT
would be again at the beginning of that new line.

This is another use of SPC(X).

Line 40 checks if someone has asked for the twenty seventh letter of the alphabet
or more, or if they’re being extra inventive, and wanting the zero’th letter of the
alphabet.
Line 50 moves the next print line down the screen, it’s called ‘window dressing’!
Line 60 makes full use of the SPC(X) function.
Line 70 prints out the required letter centrally on the screen.
Line 80 of course produces a reasonable delay before doing it all again, you can
have any number you like here of course. The trick is to pick on a number that
leaves something on the screen long enough for it to be read by a slow reader, but
not too long to irritate or bore the user.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Now the mirror image of the last problem.

****************************************************************************************************

* Problem Twenty Nine
*
* Write a program to ask someone to input a letter of the alphabet, then to print out *
* the number, between 1 and 26, that corresponds to that letter, put your request in *
* upper case and the answer in lower case *
****************************************************************************************************

Our main problem here is to change from letters to numbers, and of course
allowing a letter to enter the program via INPUT.

Naturally we must use INPUT L$, as we are asked to INPUT a letter.

Then find it’s ASCII code and check if it is within the correct range for a letter of the
alphabet, that is 65 to 90, A to Z. So:

```
L=ASC(L$):IF L<65 THEN ..... 
```

and

```
IF L>90 THEN ..... 
```

will suit our purposes.

Now we have a number, but it's always too big for the correct answer.

So we must subtract 64 from it to get the right number:

```
PRINT L-64 
```

will do.

The remaining part of the problem, is to do the upper and lower case changes as required.

`PRINTCHR$(14)` puts all into lower case, and to go back again we can use `PRINTCHR$(142)`, remember?

So our program looks like this:

```
10 PRINTCHR$(147);CHR$(14)
20 PRINTTAB(12)"GIVE ME A LETTER":PRINT:PRINTTAB(11)"FROM YOUR ALPHABET"
30 PRINT:PRINTSPC(18);INPUT L$
40 L=ASC(L$):IF L<65 THEN 10
50 IF L>90 THEN 10
60 FOR R=1 TO 6:PRINT:NEXT
70 PRINTSPC(9)CHR$(142):"YOUR LETTER IS NUMBER:" 
80 PRINT:PRINTSPC(18)L-64
90 PRINT:PRINTSPC(12)"IN THE ALPHABET" 
100 FOR D=1 TO 5000:NEXT:GOTO 10
```

As an extra problem, Extra eight, try the following:

change the solution to program 29, so that you use GET K$ instead of INPUT, and at the same time print out on the screen the letter of the alphabet chosen, similar to the INPUT statement. Also when the answer appears, make it appear in inverse or reverse video.

```
```

In the next few problems, and in a few Extras as well, I am going to ask you to put letters and numbers in definite screen positions, and then to do exercises with
them to test your powers of logic.

*Problem Thirty*

And now a program to print four different random letters, one in each of the four corners of the screen, then after a short delay, print four more to show that the program really is random.

Knowing all we do now, this is quite a simple problem. Generating four letters requires first four numbers to be generated, as the 64's random number generator only produces numbers, as it's name implies.

Of course the number produced must be within the correct range, and of the required value to produce letters when operated upon, so:

\[ N(A) = \text{INT}(\text{RND}(1) \times 25) + 65 \]

will produce numbers in the range 65 to 90, for the letters A to Z, when put into the statement:

\[ \text{PRINTCHR}\$(N(A)) \]

We only need four letters at the one time, so:

\[
\begin{align*}
\text{FOR } R & = 1 \text{ TO } 4 \\
N(A) & = \text{INT}(\text{RND}(1) \times 25) + 65 \\
\text{NEXT}
\end{align*}
\]

will produce these as: N(1), N(2), N(3) and N(4), and as we need only four, there is no need to Dimension or DIM.

To position our letters in the corner of the screen, is the next part of the problem.

For this we can use \text{PRINTSPC}(X), for the two top positions and by experiment, or theory, if you prefer to do it that way, I found that on my equipment:

\[
\begin{align*}
\text{PRINTSPC}(2) \text{ and PRINTSPC}(34)
\end{align*}
\]

presented the best position, if we put a couple of blank print lines at the top of the screen as well.

For the bottom two corners, obviously the same \text{PRINTSPC}(X) routine, but we must print nigh on a screenfull of blank lines, to get there, with our old friend:

\[
\begin{align*}
\text{FOR } R & = 1 \text{ TO } 19: \text{PRINT} : \text{NEXT}
\end{align*}
\]
So our solution looks like this:

10 FOR R=1 TO 4:N(R)=INT(RND(1)*25)+65:NEXT  
20 PRINTCHR$(147):PRINT  
30 PRINTSPC(2);CHR$(N(1));SPC(34);CHR$(N(2))  
40 FOR R=1 TO 19:PRINT:NEXT  
50 PRINTSPC(2);CHR$(N(3));SPC(34);CHR$(N(4))  
60 FOR D=1 TO 1000:NEXT  
70 GOTO 10

Lines 20 and 30 print the two letters at the top of the screen, lines 40 and 50 print the two at the bottom, and lines 60 and 70 allow you to view the result of your actions, before doing it all again. which, as was asked for, shows that the program is indeed random.

Now as Extra nine, use POKE statements instead of PRINT statements, to put the letters in the four corners of the screen as before.

Problem Thirty One

Write a program to print four single random numbers in a square format, centrally on the screen. After a short delay, if a number is odd, print an ‘O’ in the nearest screen corner, or if even, an ‘E’, then after another short delay do it all again.

This particular problem has two main parts, the first being to generate four random numbers; it doesn’t say they have to be different, so that makes the problem quite straightforward, thank goodness.

We can use a repeat loop to do it:

FOR R=1 TO 4:N(R)=INT(RND(1)*9)+1:NEXT

which gives us four numbers between 1 and 9.

If we now use a PRINT statement to do this, we find that when we come to print out the odd and even characters in the corners, at the top and the bottom of the screen, we find that we can’t without the whole screen display scrolling up, or clearing the screen first.

So we must think of another way. We could, of course, print it all out at once, but then we wouldn’t have solved the problem, as it asks for a short delay, (so it looks like the computer is thinking about what it has to do, I suppose), between the central display and the corner one.
The answer, of course, is to use POKE statements all the time.

But POKE statements require screen characters to be used, and in power up mode on the 64, we get Set 1, and, in Set 1, CHR$(1) = A$, which isn’t exactly what we want.

1 is CHR$(49)$, and 9 is CHR$(57)$, etc. So our revised repeat loop to suit POKE statements is:

```
FOR R=1 TO 4:N(R)=INT(RND(1)*9)+49:NEXT
```

which gives us the required range of numbers we need, 49 to 57.

We can now POKE the central area of the screen with the numbers and display them in a square format there.

We could use:

```
POKE 55714,1:POKE 1442,N(1)
POKE 55717,1:POKE 1445,N(2)
POKE 55834,1:POKE 1562,N(3)
POKE 55837,1:POKE 1565,N(4)
```

which POKEs N(1), N(2), N(3) and N(4) into the centre of the screen, and colours them white at the same time. You could of course colour each number a different colour.

Now comes the second part of the problem, deciding if the numbers generated are in fact odd or even. One way to do this is to check if there is any remainder when the number in question is divided by 2, for example:

- 6 divided by 2 equals 3, and
- 7 divided by 2 equals 3.5.

Also if you want to remove the decimal part of a number, you use INT(X), as we did when generating random whole numbers. So:

```
INT(7/2)=3, whereas:
7/2 =3.5
```

So, to check if a number is even, we could say:

```
IF N(1)/2=INT(N(1)/2) THEN ......, it is even, because if
N(1)=8; N(1)/2=4, and so would INT(N(1)/2)=4.
```

But if: N(2)=7; N(2)/2=3.5, and INT(N(2)/2)=3.

Therefore N(2) is odd!
Got it?

Which could result in a routine like this, for example:

```
70 IF N(1)/2=INT(N(1)/2) THEN POKE 55378,1:POKE 1106,5
80 POKE 55378,1:POKE 1106,15
```

using POKE 5, which equals E, for even, and

POKE 15, which equals O, for odd.

We could now repeat this short routine four times, to check each number, and POKE the required screen position with either E or O, depending on the result.

In the following solution I have coloured each number differently, and repeated that colour for the E and the O.

You can, if you wish, of course, POKE the numbers and letters where you will, providing you solve the problem, by putting them somewhere in the centre and the corners.

```
10 FOR R=1 TO 4:N(R)=INT(RND(1)*9)+49:NEXT:PRINTCHR$(147)
20 POKE 55714,1:POKE 1442,N(1)
30 POKE 55717,8:POKE 1445,N(2)
40 POKE 55834,7:POKE 1562,N(3)
50 POKE 55837,0:POKE 1565,N(4)
60 FOR D=1 TO 800:NEXT
70 IF N(1)/2=INT(N(1)/2) THEN POKE 55378,1:POKE 1106,5:GOTO 90
80 POKE 55378,1:POKE 1106,15
90 IF N(2)/2=INT(N(2)/2) THEN POKE 55413,8:POKE 1141,5:GOTO 110
100 POKE 55413,1:POKE 1141,15
110 IF N(3)/2=INT(N(3)/2) THEN POKE 56178,7:POKE 1906,5:GOTO 130
120 POKE 56178,7:POKE 1906,15
130 IF N(4)/2=INT(N(4)/2) THEN POKE 56213,0:POKE 1941,5:GOTO 150
140 POKE 56213,0:POKE 1941,15
150 FOR D=1 TO 3000:NEXT:GOTO 10
```

Now for `Extra ten`:

Change this program solution so that you get a random heart, club, diamond or spade in each of the four corners, but only one in each of course, and if any two opposite corners are the same, the player has won. Then allow another go, if required, or end the game.

~~~~~~~~~~~~~~~
This next problem will give you some help if you’re stuck with ‘Extra ten’.

Problem Thirty Two

Now write a program to generate four different random numbers, and print them, one in each corner of the screen. Then, print a total of these four numbers in the centre of the screen.

We know by now how to generate four different random numbers, so that they can be POKE’d to the screen, with:

```
FOR R=1 to 4:N1(R)=INT(RND(1)*9)+49:NEXT
```

We also know where to POKE them, once we’ve got them, in order to put them in the four corners of the screen.

But the main problem here is to add them all together before we print out the total in the centre of the screen.

We have generated four numbers between 49 and 57, and if we add them together, we will get a number somewhere between 196 and 228, which of course is not the true total. This is because ‘1’ is CHR$(49) and so on. The true total should be somewhere between 4 and 36, which means that we are a total of 192 out each time. As we have four numbers, each number must be 192 divided by 4 too large, or 48.

This ties in with the difference between the ASCII codes for the numbers 49 to 57, and the real numbers they stand for, 1 to 9.

So the next thing to do is to subtract 48 from each coded number and then add them together, which now gives us a total between 4 and 36, as required.

So we can say:

```
FOR R=1 TO 4:N2(R)=N1(R)-48:NEXT
```

and

```
T=N2(1)+N2(2)+N2(3)+N2(4)
```

where T stands for the variable Total, the total of the four numbers.

Now we have to print out this total, which can be done with a PRINTTAB statement.
to position it in the centre of the screen. Even small programs should be 'window
dressed' you know. So:

```
    FOR R=1 TO 11:PRINT:NEXT:PRINTTAB(18)T
```

will do the trick.

Our program then should look something like this, providing you agree with my
POKE positions:

```
10 FOR R=1 TO 4:N1(R)=INT(RND(1)*9)+49:NEXT 
20 PRINTCHR$(147):POKE 55378,0:POKE 1106,N1(1) 
30 POKE 55413,0:POKE 1141,N1(2) 
40 POKE 56178,0:POKE 1906,N1(3) 
50 POKE 56213,0:POKE 1941,N1(4) 
60 FOR D=1 TO 600:NEXT 
70 FOR R=1 TO 4:N2(R)=N1(R)-48:NEXT 
80 T=N2(1)+N2(2)+N2(3)+N2(4) 
90 FOR R=1 TO 11:PRINT:NEXT:PRINTTAB(18)T 
100 FOR D=1 TO 3000:NEXT:GET K$:IF K$=""" THEN 10 
110 PRINTCHR$(147):END 
```

I've put a GET K$ line in at the end of the solution to allow the program to RUN
again, and again, or not, as required. No one asked me to prove the program was
random, too often in life we waste time solving problems that aren't there!

As an extra problem, 'Extra eleven', now POKE the total to the centre of the screen,
then flash it on and off a number of times, before going to the GET K$ statement.
CHAPTER SIX

NUMBER CRUNCHING

I have called this chapter ‘number crunching’ because all the problems deal with numbers and operations on them. Not all of them use numbers in the normal mathematical fashion, but I have included three simple problems for those amongst you who are mathematically inclined.

Problem Thirty Three

Generate a set of ten random numbers, then check whether each one is odd or even, print out the whole set in one column, the even numbers in another and the odd ones in a third.

This first problem on number crunching is quite simple to solve, but any session dealing with mathematics should always, in my opinion, start simple.

We must first generate ten random numbers, and to do this we can use a loop. We must also decide how large our numbers should be. I’ve chosen from 1 to 99, so:

```
FOR R=1 TO 10:N1(R)=INT(RND(1)*99)+1:NEXT
```

should produce ten random numbers between 1 and 99.

We must now check to see if a number produced is odd or even, and as we did this before in the last chapter, this should be straightforward too.....so:

```
IF N1(R)/2=INT(N1(R)/2) THEN N2(R)=N1(R) deals with the even numbers.
```

For the odd ones, we can change N1(R), if the foregoing IF....THEN statement
It turns out to be false, to:

\[ N_3(R) = N_1(R) \]

so that all the even numbers are coded \( N_2(R) \), all the odd numbers are coded \( N_3(R) \), and all the numbers are initially coded \( N_1(R) \).

We have also to arrange the numbers in three columns, such that \( N_1(R) \) is on the left, \( N_2(R) \) is in the middle, and \( N_3(R) \) on the right.

We must, therefore, divide the screen up as evenly as possible into three columns, and label each column. I know the problem didn’t ask us to, but it helps if you make your screen as self explanatory as possible, so:

\[
\text{PRINTTAB}(7)"\text{NUMBER}";\text{TAB(18)}"\text{EVEN}";\text{TAB(28)}"\text{ODD}" \\
\text{and}
\]

\[
\text{PRINTTAB}(8)N1(R);\text{TAB(18)}N2(R);\text{TAB(28)}N3(R) \]

should line them all up, and now our complete solution should look like this:

10 PRINTCHR$(147) \\
20 PRINTTAB(7)"NUMBER";TAB(18)"EVEN";TAB(28)"ODD":PRINT \\
30 FOR R=1 TO 10:N1(R)=INT(RND(1)*99)+1 \\
40 IF \frac{N1(R)}{2}=INT(\frac{N1(R)}{2}) \text{ THEN } N2(R)=N1(R):GOTO 60 \\
50 N3(R)=N1(R) \\
60 PRINTTAB(8)N1(R) \\
70 IF N2(R)<>0 \text{ THEN PRINTTAB}(18)N2(R) \\
80 IF N3(R)<>0 \text{ THEN PRINTTAB}(28)N3(R) \\
90 NEXT

You will notice that each number is dealt with as it is generated, until the loop runs out. It is now that the more experienced amongst you will ask, “Why bother storing them in an array, if you are printing them out immediately?”.

Well if you removed all the \( (R) \)'s, the screen display would look quite different, and not quite the solution to the problem.

Each time the computer printed out the number, in lines 70 and 80, it would print out a number regardless of whether it was a NEW number or not. This is because \( N_2 \) and \( N_3 \) would both have a value carried over from the previous time they were printed. Remember we only want a combined total of ten numbers in the odd and even columns. If you are in any doubt, delete all the \( (R) \)'s from my program, to prove it. There are, of course, other ways of doing this, and you might have selected one. I thought this method was simple and straightforward enough to use.

As an extra thought, though, what more would you have to do if you choose more
than ten numbers to play around with?

Try it by changing line 30 to:

```
FOR R=1 TO 15.N1(R)=INT(RND(1)*99)+1
```

and seeing what happens. The answer should be obvious.

Also, try changing the PRINTTAB’s to PRINTSPC’s, but keep the screen format the same.

```
N=INT(RND(1)*5)+1
```

This next problem is a natural follow on from the previous one, but that might not be so obvious at first.

```
* Problem Thirty Four *
* Write a program to generate random numbers between 1 and 5, then place them *
* in columns one to five across the screen, so that the 1’s are in the first column, etc. *
* and all the columns are in different colours, also make the program run *
* continuously. *
```

The problem does not state how many numbers we have to generate while the program is running, only that it must run continuously, so we can’t use a FOR....NEXT loop, we shall have to use GOTO instead, as a means of going back to the beginning of the program to produce the next random number, so:

```
N=INT(RND(1)*5)+1
```

will give us random numbers between 1 and 5.

We now have to check what value each number has and place it in its own column on the screen, which also means dividing up the screen into convenient columns with PRINTTTAB(X) statements. I found by experiment, (you see - it’s not all theory), that TAB(4), (11), (18), (25) and (32) looked best, so my solution is:

```
10 PRINTCHR$(147)
20 N=INT(RND(1)*5)+1
30 IF N=1 THEN PRINTTAB(4)CHR$(144)N
40 IF N=2 THEN PRINTTAB(11)CHR$(158)N
50 IF N=3 THEN PRINTTAB(18)CHR$(5)N
60 IF N=4 THEN PRINTTAB(25)CHR$(156)N
```
70 IF N=5 THEN PRINTTAB(32)CHR$(30)N
80 GOTO 20

This program will, of course, go on forever. It prints each number in a different colour by using CHR$(X), and the ASCII or CHR$ codes given on page 135 and 136 of your 64 manual. We can’t use the colour codes given on page 139, as these are only for POKE statements, used with POKE 55296 to 56295.

As an extra problem, ‘Extra twelve’, make the program print out the total for each column, the grand total of all the numbers, and the average of all the numbers, for a total of one thousand numbers. I told you that this chapter was on number crunching. It is also a useful exercise in judging the randomness of a computer’s random number generator, as mentioned in a previous chapter.

Again, the next problem is an extension of the previous one, and it also gives you a few clues on how to solve this last extra problem, in case you need some help.

Problem Thirty Five

Extend the solution to the last problem to indicate on screen when one column is full with a total of ten numbers, with a suitable end of program message, and an option to do it all again.

We have already, in the previous solution, produced the necessary numbers and judged their value, now we have to judge when any one of them has been generated a total of ten times. This means we have to increment each number by one each time it is produced, so:

IF N=1 THEN PRINTTAB(4)CHR$(144)N:A=A+1

will do this for 1, and four similar lines would do for 2, 3, 4 and 5, providing you don’t use variable A on each line.

We must of course make A, etc., equal to zero before the program starts.

To check when any number has been generated ten times, again an IF....THEN statement can be used:

IF A=10 THEN PRINT"FULL" is useful.

Naturally, we must write FULL in the correct place, at the bottom of the full column, and after each check, go back to the beginning again to generate the next random number, until one column has ten numbers.
We are also asked to print out a suitable message.

This we can do all on the same screen, or on a new screen, after clearing the first. I opted for the latter, which means it’s a good idea to allow a short time delay for the player to read the word FULL, and to check that the column has in fact got ten numbers in it. It may not be possible, and is indeed quite unlikely that one would be able to count all the numbers. Each number occupies one screen line, and, although the minimum number of lines required to display ten numbers may be ten, it’s doubtful that it will be. A random number generator should not produce ten numbers the same, one after the other; three or four maybe, or even on occasion five, but not ten.

The maximum repeats to get one set of ten is forty six, four times nine, plus one times ten. Usually the count will be between twenty and thirty five, so clearing the screen will be a good idea, in order to tidy everything up, before printing out the message.

The message must state which column was full, and therefore some record must be kept of this, so:

```plaintext
IF A=10 THEN PRINT"FULL":X=1
```

will if we use X as the code for the the full column, allow us to print the column number which was full.

So my solution ended up looking like this:

``` plaintext
10 PRINTCHR$(147):A=0:B=0:C=0:D=0:E=0
20 N=INT(RND(1)*5)+1
30 IF N=1 THEN PRINTTAB(4)CHR$(144):A=A+1
40 IF N=2 THEN PRINTTAB(11)CHR$(158):B=B+1
50 IF N=3 THEN PRINTTAB(18)CHR$(5):C=C+1
60 IF N=4 THEN PRINTTAB(25)CHR$(156):D=D+1
70 IF N=5 THEN PRINTTAB(32)CHR$(30):E=E+1
80 IF A=10 THEN PRINT:PRINTTAB(4)"FULL":X=1:GOTO 140
90 IF B=10 THEN PRINT:PRINTTAB(11)"FULL":X=2:GOTO 140
100 IF C=10 THEN PRINT:PRINTTAB(18)"FULL":X=3:GOTO 140
110 IF D=10 THEN PRINT:PRINTTAB(25)"FULL":X=4:GOTO 140
120 IF E=10 THEN PRINT:PRINTTAB(32)"FULL":X=5:GOTO 140
130 GOTO 20
140 FOR D=1 TO 1000:NEXT:PRINTCHR$(147)CHR$(156)
150 FOR R=1 TO 10:PRINT:NEXT
160 PRINTTAB(10)"COLUMN":X;"IS NOW FULL"
170 PRINT:PRINTTAB(8)"DO YOU WANT ANOTHER TRY?"
180 GET K$:IF K$="" THEN 180
190 IF K$="Y" THEN 10
200 IF K$="N" THEN END
210 IF K$<>"" THEN 180
```

I’ve included a request for another try at the end of the program, and window
dressed the final screen to suit.

So now as 'Extra thirteen', why not turn this problem into a good game of chance, by adding extra lines to the beginning and the end of the solution which allow a player to guess which column will fill up first, and then state if he or she has won or lost. You could keep score as well, you know how to do that by now, don't you?

Once solved, this extra problem is a good simple random game occupying less than forty lines, which might, if you plug away at it long enough, teach you something about the 64's randomness.

You could also extend it to print out after each session, the number of times each column was filled. I haven't in my solution, but as I have said before, sometimes you're left on your own to solve life's problems.

The next four problems that complete this number crunching chapter, are all mathematical in nature, and if you hate maths, you can skip them if you like, or in other words, GOTO chapter seven.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**Problem Thirty Six**

* Write a program to display on the screen the 'N' times table, the number 'N' being inputted freely from the keyboard.

*_____________________________*
In this problem we have to display on the screen the N times table and, usually, a times table is expressed up to twelve times the given number N. So our final screen should, where N=6, look like:

\[
\begin{align*}
1 \times 6 &= 6 \\
2 \times 6 &= 12 \\
3 \times 6 &= 18 \\
4 \times 6 &= 24 \\
5 \times 6 &= 30 \\
6 \times 6 &= 36 \\
7 \times 6 &= 42 \\
8 \times 6 &= 48 \\
9 \times 6 &= 54 \\
10 \times 6 &= 60 \\
11 \times 6 &= 66 \\
12 \times 6 &= 72 \\
\end{align*}
\]

Notice the displacement of the numbers 1 to 12, if you don’t do this the 9 and the 10 would look odd, like this:

\[
\begin{align*}
8 \times 6 &= 48 \\
9 \times 6 &= 54 \\
10 \times 6 &= 60 \\
11 \times 6 &= 66 \\
\end{align*}
\]

So, in that part of the solution for printing up the final screen, this arrangement must be taken into account, by the use of two routines. And what do these two routines look like?

Well, taking the second line of the table just given:

- 2 is a variable, (i.e. 1 to 12)
- x is not a variable
- 6 is a variable, (the N)
- = is not a variable
- 12 is a variable, (it’s 2 times N)

So our routine must contain three variables, one of which we must ask for from the user of the program, by, for example:

```
PRINT "WHICH TIMES TABLE WOULD YOU LIKE?":INPUT N
```

You should, of course, give a limit to the value for N in your program, because when it is RUN, it might well go into ‘exponentials’, which we are not covering in this book. We shall avoid this eventuality, by limiting our solution to 100, so the maximum number in any table will be 1200.

The crux of the problem though is that PRINT line that prints out the table.
To write $2 \times 6 = 12$ on the screen, where $2$, $6$ and $12$ are variables, we can use:

```
PRINT R;"x";N;"=";R*N
```

where $R$ is the changing or incrementing multiplier, in this case $2$. $N$ is the variable, in this case $6$, and $R\times N$ is the answer $R \times N$, $(2 \times 6)$.

Don’t forget the computer needs * to multiply with. We don’t, we use x.

So to get our variable $N$ multiplied by $1$ to $12$ in turn and displaced by one column after $9$ times, requires two loops to be used:

```
FOR R=1 TO 9:PRINTR;"x";N;"=";R*N:NEXT
```

and

```
FOR R=10 TO 12:PRINTR;"x";N;"=";R*N:NEXT
```

notice the second FOR....NEXT loop starts at $10$ and not $1$.

A FOR....NEXT loop can start and finish at any number, and the loop variable can be used in a PRINT statement.

My solution looks like this:

```
10 PRINTCHR$(147)
20 PRINT"PLEASE TYPE IN YOUR NAME": PRINT: INPUT N$:L=LEN(N$)
     :IF L>32 THEN 10
30 PRINTCHR$(147):PRINT:PRINT'WHICH TIMES TABLE WOULD YOU LIKE?'
40 PRINT:PRINT N$
50 PRINT:PRINT'YOU CAN CHOOSE BETWEEN 1 AND 100'
60 PRINT:PRINT"TYPE IN YOUR NUMBER HERE":":INPUT N: IF N<0 OR N>100 THEN 30
70 PRINTCHR$(147):FOR R=1 TO 4:PRINT:NEXT
80 FOR R=1 TO 9:PRINTTAB(12)R"x"N;"=";R*N:NEXT
90 FOR R=10 TO 12:PRINTTAB(11)R;"x";N;"=";R*N:NEXT
100 PRINT:PRINTTAB(12)"ANOTHER GO?"
110 GET K$:IF K$="" THEN 110
120 IF K$="Y" THEN 30
130 IF K$="N" THEN PRINTCHR$(147):GOTO 150
140 IF K$<>"" THEN 110
150 FOR R=1 TO 11:PRINT:NEXT
160 PRINTTAB((20-L/2)-4)"GOODBYE ":N$
170 FOR D=1 TO 4000:NEXT:PRINTCHR$(147):END
```

You will have noticed that I have window dressed the program a little, and put in the error trapping routine for the $100$ limit, in line 60.
Something else you might notice in line 80 and 90, is that I have left out the semicolons in the PRINT statement in line 80.

I have written:

\[
\text{PRINTTAB(12)R'x'N='R*N in line 80}
\]

and

\[
\text{PRINTTAB(11)R;'x';N;='';R*N in line 90, although this is not recommended as a regular procedure, as it can lead to confusion in reading your program, the computer does not get confused, and will treat anything between inverted commas as a string and then print it out verbatim. Try:}
\]

\[
\text{PRINTTAB(12)''R''x''N''='''R*N''}
\]

and see what you get instead. This is interesting from a logical point of view, so I thought it might interest you, as this is a book on logical problem solving.

The short routine on lines 160 and 170 window dress the end of the program and tidy it up considerably. If you are logical, then we also assume that you have a tidy mind!

Line 60 shows a PRINT line used with an INPUT statement, and the use of a semi-colon to get the INPUT on the same screen row. Obviously you could have used an INPUT statement together with a prompt.

Generating two random numbers now is child’s play, or should be, so we’ll not even discuss that.

Displaying them on the screen is not too difficult either:

\[
\text{PRINT N1;''+'';N2;''='';?''}
\]

will do, where N1 and N2 are our two numbers.
Naturally we need an answer from the keyboard so:

```
INPUT A
```

again will satisfy our requirements.

To check the answer for correctness we need:

```
SU=N1+N2
```

IF SU<>A THEN subroutine

IF SU=A THEN next subroutine

where SU=sum

The subroutines are a little more complicated, in that we must allow the player to have two goes, and then give the answer the second time round.

We also have to keep score and to ask new questions until he or she decides to stop.

We can put the latter into a subroutine, which can print out the final score and a goodbye message if he or she chooses to stop, or go back to the beginning of the program to carry on, with:

```
3000 REM 'ANOTHER GO' ROUTINE
3010 GET K$:IF K$="" THEN 3010
3020 IF K$="Y" THEN RETURN
3030 IF K$="N" THEN 3050
3040 IF K$<>"" THEN 3010
3050 PRINTCHR$(147):FOR R=1 TO 11:PRINT:NEXT:
   PRINTTAB((20-L/2)-1)"GOODBYE "N$
   PRINT:PRINTSPC(14)"YOUR SCORE WAS"SC
3070 FOR D=1 TO 6000:NEXT:PRINTCHR$(147):END
```

I always like to personalise a computer program dealing with individuals on a one to one basis, as this program does, just like a teacher and a pupil in a classroom or tutorial situation. Line 3050 deals with this by using the player’s name on the final screen. The trick though, is not to use it too often, or it will become a bore.

If the answer is right, which shouldn’t be too difficult, the routine can be put into another subroutine:

```
2000 REM RIGHT ROUTINE
2010 PRINT:PRINTTAB(8)"WELL DONE, THAT’S RIGHT";N$:SC=SC+1
2020 PRINT:PRINTSPC(14)"ANOTHER GO?":RETURN
```

where SC=score.
I have used the player’s name again here, just once, but you will notice when you RUN the program, that not using the N$ and TAB(X) routine, spoils the presentation. Compare line 3050, or line 1010 in the routine that follows with 2010 above.

The routine for a wrong answer will be a little more complicated in that we must keep track of how many tries the user has had, with:

```
1000 REM WRONG ROUTINE
1010 T=T+1:PRINT:PRINTTAB((20–L/2)–10)"SORRY ":N$: "YOU'RE WRONG!
1020 IF T=2 THEN 1040
1030 PRINT:PRINTSPC(14)"TRY AGAIN":FOR D=1 TO 1500:NEXT:
PRINTCHR$(147):RETURN
1040 PRINT:PRINTSPC(11)"THE ANSWER IS"SU:
PRINT:PRINTSPC(14)"ANOTHER GO?":RETURN
```

This subroutine allows the player to have one wrong answer. By using IF T=2 THEN 1040, the line saying "TRY AGAIN", is used only for the first wrong answer, because when T does equal 2, line 1040 prints out the right answer, and does not then allow another try at that particular problem.

Our main program then will have a fair sprinkling of GOSUBs to take the player in and out of the RIGHT and WRONG routines as required.

Both routines, (2000 to 2020 and 1000 to 1040 respectively), must have provision for asking the player for another go, before going to subroutine 3000. This subroutine could, of course, have had the ‘another go’ question built into it, but I choose not too. It’s all a matter of personal preference. It’s always a good idea, when programs start to get a little overlong and complicated in their structure, to include some REM, or remark statements, to tell readers of your programs just what you are trying to do, or to remind yourself. How often have you written a small program one day, put it aside, if you were able to tear yourself away from the computer, then returned to it a few days later and couldn’t make head nor tail of it? Well, REM statements go someway towards pointing your confused and searching mind in the right direction. Naturally, if you had a flow chart to go with it as well programming would be that much easier, but that’s all to be found in a book on strict programming, not in a light hearted logical look at life through the eyes of a computer programmer.

I have included REM statements in the final solution for this problem, which follows:

```
5 REM MAIN PROGRAM
10 PRINTCHR$(147):PRINT"PLEASE TYPE IN YOUR NAME":PRINT:
INPUT N$:L=LEN(N$):SC=0
20 N1=INT(RND(1)*10)+1:N2=INT(RND(1)*10)+1:T=0
30 PRINTCHR$(147):FOR R=1 TO 7:PRINT:NEXT
40 PRINTTAB((20–L/2)–10)"WHAT IS THE ANSWER ":N$:PRINT:PRINTTAB(13)N1;'+';N2=' ?'
50 PRINT:PRINTTAB(13)N1;'+';N2=' ?'
```

83
This main part of the program allows the player to have two tries.

Line 80 will send the player back to the beginning to try the same problem again, but this line will be ignored if the player gets the answer right, either first or second time. In this case line 90 sends the player to the RIGHT routine.

To finish off with, how about combining lines 50 and 60 into one line, so that the result on the screen appears on one line? You’re on your own again: we only recently discussed this option.

Here is another comparison problem, together with an extremely simple sorting requirement.

*******************************************
* Problem Thirty Eight
* *
* Write a program that generates a random integer number, and then if this number is odd, display it on the right of the screen, or if it is even, display it on the left. Make the program run continuously, until stopped from the keyboard.
*
*******************************************

In this problem we must first decide how big the numbers are to be, and then judge their evenness or otherwise, once we have produced them with our random statement.

I have chosen an upper limit of 100, and a lower limit of 1, but you can have whatever numbers you like, as the problem is not based on the size of the numbers.

You should, by now, be able to write the following with your eyes shut, or at least half open, in order to produce the random numbers needed:

\[ N = \text{INT}(\text{RND}(1) \times 100) + 1 \]

To check for odd or even, I have used as before:
IF N/2=INT(N/2) THEN of course N is even with two PRINTTAB(X) statements to place the numbers on screen. We are only required to print the evens and odds in their own columns so, here is my short solution:

10 PRINTCHR$(147)
20 N=INT(RND(1)*100)+1
30 IF N/2=INT(N/2) THEN GOSUB 100:GOTO 20
40 GOSUB 200:GOTO 20
50 END
100 REM EVEN NUMBERS
110 PRINTTAB(8)N:FOR D=1 TO 500:NEXT:RETURN
200 REM ODD NUMBERS
210 PRINTTAB(28)N:FOR D=1 TO 500:NEXT:RETURN

You could, if you wanted to, create another subroutine for the delay, and this is quite often done in a larger program, where you might require to have delays of different lengths.

You could have a delay:

FOR D=1 TO 500:NEXT

and keep going to it to build up the delay you wanted, or have a series of delays of increasing length, and each time send the computer to the one you required.

So the subroutines could have looked like this, and indeed you may have done this already.

100 REM EVEN NUMBERS
110 PRINTTAB(8)N:RETURN
200 REM ODD NUMBERS
210 PRINTTAB(28)N:RETURN
300 REM DELAY SHORT
310 FOR D=1 TO 500:NEXT:RETURN

My solution runs continuously as required, because of the GOTO 20 on lines 30 and 40, but can be stopped by pressing the RUN/STOP key.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Now for the budding mathematics amongst you, we have a problem dealing with factorials. OK, fall out all the groaners, maths is a lot more fun on a computer. You can change all your mistakes so easily, and no one minds you cheating just a little.
Problem Thirty Nine

Write a program to display the factorial value of any integer. The number must be inputted from the keyboard, and you must also allow the program to be controlled from the keyboard.

Very simply put, a factorial of any number, N, is all the numbers from 1 to N multiplied together, so that factorial 10 works out to be:

\[10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3628800\]

which is quite large. So be warned, you can end up with some pretty big numbers, even starting with just a small one.

It would be a good idea therefore to limit the number to be inputted from the keyboard. But what sort of limit?

One way would be to say not more than 100 or not more then 10, to keep to simple round numbers. But there’s not much point in doing all this with numbers up to ten: you could perhaps do that in your head!

100 produces extremely large numbers, and sends the computer into exponential values, such as:

\[2.31878584E+12\]

which means \[2.31878584 \times (10 \text{ raised to the power of } 12)\]: got it?

So another possible limiting factor could be to stop before the computer goes into \(E\) values, as these are easily read and understood by all.

The next problem, then, is to decide which number stops short of being displayed in an exponential form.

One way to do this, is to experiment by feeding one number at a time into a suitable formula in the computer, until you see that you have gone too far, and then stop at the previous number, but that method is a bit long winded, to say the least.

A better way is to use a loop, and get the computer to do all the work, that’s what they’re there for anyway. Put the experiment in a loop, for the computer to calculate and display all the results, starting at some low number, and going on up until you press the RUN/STOP key. So:

```
10 FOR R=1 TO 1000
20 F=F*R:PRINT F:NEXT
```
will produce the factorial value of all the numbers between 1 and 1000, extremely quickly, but if it’s too quick for you, then include a delay loop as well, to enable you to read each number, with:

```
10 F=1:FOR R=1 TO 1000:F=F*R
20 PRINT F:FOR D=1 TO 1000:NEXT:NEXT
```

Hopefully, if you decide to do it this way, you will eventually notice that 33 turns out to be the top limit for our requirement, so my solution looks like this:

```
10 PRINTCHR$(147)
20 PRINT"GIVE ME A WHOLE NUMBER LESS THAN 34"
30 PRINT:"AND I WILL GIVE YOU IT’S FACTORIAL VALUE"
40 PRINT:"INPUT YOU NUMBER HERE PLEASE: ";:INPUT N:IF
   N>33 THEN 10
50 PRINTCHR$(147):PRINT"YOUR NUMBER’S FACTORIAL VALUE IS:"
60 F=1:FOR R=1 TO N:F=F*R:NEXT
70 PRINT:PRINT F
80 PRINT:"ANOTHER GO?"
90 GET K$:IF K$="" THEN 90
100 IF K$="Y" THEN 10
110 IF K$="N" THEN 130
120 IF K$<>"" THEN 90
130 PRINTCHR$(147):PRINTTAB(16)"GOODBYE":FOR D=1 TO 4000:
    NEXT:PRINTCHR$(147):END
```

As usual, I have window dressed the program, nothing out of the ordinary though, and answered for you the question at the end of problem 37 (see line 40).

Well that brings us to the end of the chapter on number crunching, and only leaves us with nine more problems, in the next chapter.
CHAPTER SEVEN

MISCELLANY - Time, Sound and Graphics

\[ \text{RIGHT}\$(X\$); \text{STR}\$(X); \text{Tl}\$; \text{TI} \]

This chapter is a miscellany or mixture of problems, two of which actually deal with time, three test your powers of conversion, and three give you a chance to use the 64's music facilities. The final one is a fun program, which hopefully will give you a few headaches whilst solving it.

Although the 64 has the advantage of sprites, I don't want you to use them to solve the last problem, that would make it far too easy. I'm saving sprites for the next time we meet, in my next book in this problem solving series.

And so to the first problem in this chapter.

* Problem Forty

* Write a program that will use the screen as a stop watch, displaying seconds continuously, until stopped from the keyboard, or started again from the keyboard.
The main problem here is to get the computer to generate seconds accurately. We could of course, use a delay loop, timing it ourselves, to calculate how much is needed for a one second delay, with say:

```
FOR D=1 TO 1000:PRINT D:NEXT
```

and using a stop watch, or the sweep hand on your wristwatch, count the D’s as they are displayed, until so many seconds have elapsed.

Then adjust the variable D to suit.

A long winded method, I’ll admit, and a simpler solution is to use the Commodore 64’s own built in clock.

This internal device can be accessed either as:

```
TI$
```

which is a six number string, representing a twenty four hour clock, with HOURS MINUTES SECONDS, and

```
TI
```

which counts in fiftieths of a second, called ‘jiffies’.

Everytime you switch on the computer, TI$ is set to zero automatically, TI is automatically zeroed, as it is controlled by TI$.

TI$ can also be set from the keyboard, for any particular time you should wish, and we shall be using this facility in the last chapter.

As our solution program has to be continuous, and started and stopped from the keyboard, then TI$ must be set to zero at the beginning of each RUN of the program. Switching off the computer each time is a little too much to expect, so:

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

will do this for us, where the first two zeros are the hours, the second two the minutes, and the last two the seconds. TI$ must always be given six digits, either when set from within a program, or from the keyboard, no more, no less.

As TI counts in fiftieths of a second, we need to know when it has counted fifty of them, to equal one second, and then display the fact on the screen, so:

```
C=TI:IF C=50 THEN one second has elapsed
```

will set up a counter to count in seconds.

If we use the variable SEC for seconds, then this can be incremented by one each
time C equals fifty.

So, we can use:

```
10 PRINTCHR$(147):SEC=0
20 TI$="000000"
30 C=TI$:IF C=50 THEN 100
40 GOTO 30
100 SEC=SEC+1
```

as a start, printing SEC each time it changes its value.

The next part of the problem requires us to start and stop the counter from the keyboard, without stopping the program.

To do this we can use GET K$, twice, once for stopping the watch, and once for starting it again, with:

```
110 PRINTCHR$(147):FOR R=1 TO 11:PRINT:NEXT:PRINTTAB(18)SEC
120 GET K$:IF K$="" THEN 20
```

which prints out the seconds in the middle of the screen, and sends the computer back to line 20 again, setting everything, except SEC, the seconds, back to zero.

The two lines:

```
130 IF K$="S" THEN 150    (S for stop)
140 IF K$<>'"" THEN 120
```

also allow the stopwatch to be stopped from the keyboard. Similarly,

```
150 GET K$:IF K$="" THEN 150    (R for restart)
160 IF K$="R" THEN 10
170 IF K$<>'"" THEN 150
```

will send the computer back to the beginning of the program, setting SEC back to zero as well this time, ready to start counting again.

The program can finally be stopped by pressing the RUN/STOP key.

So, now you have a very expensive stopwatch, especially if it's all in colour, but a little difficult to carry about a sports field.

You could of course, set it to display fiftieths of a second as well, but alas that's your problem.

I haven't listed my solution here, but I did take the precaution of numbering with the necessary line numbers all the hints above, so all you have to do now is type in
each line, and away you go.

We are now going to turn the Commodore 64 into an expensive digital clock, which will be useful for counting down to something, like New Year’s Eve for example, for which you could write a graphics program, complete with music to suit, to celebrate the occasion, but that too is for the future.

Problem Forty One

Write a program that will use the TV screen as a digital clock, displaying hours, minutes and seconds. Make sure the clock keeps good time - it should for the price you’ve paid.

In this problem, instead of only having to deal with the seconds, we have to display the minutes and the hours as well.

This means that T1$, which keeps track of the passage of time since the moment you switched on the computer, must be displayed on the screen.

We could of course say:

```
PRINT T1$
```

and get the time displayed, but a digital clock, which is what we have been asked to reproduce usually has ‘:’, in between the HOURS, MINUTES and SECONDS, to separate them:

```
HOURS:MINUTES:SECONDS
```

We must therefore separate T1$ into three separate parts, one for the hours, one for the minutes, and one for the seconds.

To separate them we need to string slice T1$, using the required BASIC function.

We could use MID$(T1$), three times:

```
HOU$=MID$(T1$,1,2)
MIN$=MID$(T1$,3,2)
SEC$=MID$(T1$,5,2)
```

which will conveniently split up T1$.
Or we could use LEFT$(TI)$, MID$(TI)$, and RIGHT$(TI)$, which uses all the string slicing functions, and which may be a little more obvious to someone reading your listing, so:

\[
\begin{align*}
HOU$ &= \text{MID}(TI$,1,2) = \text{LEFT}(TI$,2) \\
MIN$ &= \text{MID}(TI$,3,2) = \text{MID}(TI$,3,2) \\
SEC$ &= \text{MID}(TI$,5,2) = \text{RIGHT}(TI$,2)
\end{align*}
\]

will suit out purposes.

Our solution therefore could look like this:

\[
\begin{align*}
10 & \text{PRINTCHR$(147)$} \\
20 & TI$ = \text{"000000"} \\
30 & \text{PRINTCHR$(147)$}:\text{FOR} \ R = 1 \ \text{TO} \ 11: \text{PRINT}: \text{NEXT} \\
40 & \text{PRINTTAB(12)}:\text{TIME ELAPSED IS:}"' \\
50 & \text{PRINT}: \text{PRINTTAB(16)} \ \text{LEFT}(TI$,2) ;"' ; \text{MID}(TI$,3,2) ;"' ; \text{RIGHT}(TI$,2)
\end{align*}
\]

which will print the digital clock in the centre of the screen, but of course will not make it tick, or should I say, increment.

TI$ is of course incrementing every second, like this:

\[
\begin{align*}
000000 \\
000001 \\
000002
\end{align*}
\]

and we want to change the display every time an increase in the seconds column occurs. Otherwise if, for example, we had a GOTO 30 line at line 60, the screen would flash much too frequently. So, we need only to go back when RIGHT$(TI$,2) changes:

\[
\begin{align*}
60 & \text{SEC$ = \text{RIGHT}(TI$,2) \\
70 & \text{IF} \ \text{RIGHT}(TI$,2) = \text{SEC$} \ \text{THEN} \ 70 \\
80 & \text{GOTO} \ 30
\end{align*}
\]

This will keep the program in suspended animation until RIGHT$(TI$,2) changes, at which time it will move on to line 80, then go back to line 30, and print out our digital clock again.

Don’t forget we must clear the screen each time we move back to line 30, or the whole display will scroll up when the printing reaches the bottom of the screen. This is because, in this program we are using PRINT, not POKE statements.

My problem solution looks like this:

\[
\begin{align*}
10 & \text{PRINTCHR$(147)$} \\
20 & TI$ = \text{"000000"} \\
30 & \text{PRINTCHR$(147)$}:\text{FOR} \ R = 1 \ \text{TO} \ 11: \text{PRINT}: \text{NEXT}
\end{align*}
\]
40 PRINTTAB(12)"TIME ELAPSED IS:"  
50 PRINT:PRINTTAB(16)LEFT$(TI$,2);"":";MID$(TI$,3,2); ";":";RIGHT$(TI$,2)  
60 GET K$:IF K$="S" THEN 120  
70 IF K$="" THEN 90  
80 IF K$<>"" THEN 90  
90 S$=RIGHT$(TI$,2)  
100 IF RIGHT$(TI$,2)=S$ THEN 100  
110 GOTO 30  
120 PRINT:PRINTTAB(11)"TO RESTART PRESS R"  
130 GET K$:IF K$="" THEN 130  
140 IF K$="R" THEN 10  
150 IF K$<>"" THEN 130  
160 END

I have also allowed the user to stop and start the program from the keyboard, so lines 60, 70 and 80 in my explanation are now lines 90, 100 and 110.

Also there is one small improvement you could carry out on this program, which would save a very small amount of memory, can you see it. If you can’t, I can assure you it’s very clear, in fact, perhaps too clear.

As an extra problem, ‘Extra fourteen’, change the above program to allow a start time to be inputted from the keyboard, which could also double as a reset time to be used before restarting the clock.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

If you’re a budding electrician or electronics buff, then the next problem may interest you. It deals with Ohms Law, V=IR, where V is the voltage in an electrical circuit, I the current flowing, and R the total resistance in the circuit.

*******************************************************************************
* Problem Forty Two *
*******************************************************************************

* Write a program that will give the value of V, I or R, when either of the other two * are known, using the formula V=IR, this formula should be displayed on the * screen, with a request to input two of the values, whereupon the computer gives * the final answer. *
Even if you are not aware of any electrical theory, the expression $V=IR$ should have no worries for you, as it is a very simple first order formula. It stands for $V$ equals $I$ multiplied by $R$, or the voltage in a circuit is the result of multiplying the current by the resistance.

The main problem is that we have to write this program, such that the computer will give an answer depending on two INPUTs, but the two INPUTs have to be any two variables from three, with the answer as the third variable.

Which means that we have to allow for a choice between $V$, $I$ and $R$, on the part of the user.

We could use GET K$, then with:

```
10 GET K$: IF K$<>"V" AND K$<>"I" AND K$<>"R" THEN 10
```

which then only allows $V$, $I$ and $R$ to be inputted, then:

```
20 IF K$="V" THEN do something
30 IF K$="I" THEN do something
40 This line to respond to an INPUT R
```

Naturally we must window dress the beginning of this program with instructions as to what the user can expect, and then ask for a value the user wishes to be calculated, from $V$, $I$ or $R$, so:
INPUT "A VALUE FOR V PLEASE"; V
INPUT "A VALUE FOR I PLEASE"; I
R = V / I: PRINT "THE RESISTANCE R ="; R; " OHMS"

will produce the necessary answer for R.

But what if the user inputted a value of zero for I? You cannot, of course, divide anything by zero as the answer would then be infinity, which isn't much use to anyone. The computer cannot deal with this situation satisfactorily, so it produces an error statement. We must therefore have an error trapping routine to deal with this possibility:

INPUT "A VALUE FOR I PLEASE"; I
IF I = 0 THEN GOSUB 2000: GOTO 10

will deal with this, where line 2000 holds the error trapping routine, and tells the user why you have stopped him or her from carrying on with the program. But, it could also happen that someone using the program could enter a negative value, which would mean that R would then have a negative value as well. So:

IF I <= 0 THEN GOSUB 2000: GOTO 10

should cater for every eventuality.

The same possibilities are also applicable to values for V, therefore:

INPUT "A VALUE FOR V PLEASE"; V
IF V <= 1 THEN GOSUB 2000: GOTO 10

will do as well.

My final solution therefore looks like this:

5 REM INSTRUCTIONS AND MAIN PROGRAM
10 PRINTCHR$(147)
20 PRINT "THE VOLTAGE V,"; PRINT "APPLIED TO AN ELECTRICAL CIRCUIT"
30 PRINT; PRINT "EQUALS THE CURRENT I,"; PRINT "FLOWING IN THAT CIRCUIT"
40 PRINT; PRINT "MULTIPLIED BY THE RESISTANCE R."
50 PRINT; PRINT "THAT IS: V = I X R"
60 PRINT; PRINT "CHOOSE WHICH VALUE YOU WANT CALCULATED,": PRINT; PRINT "THE COMPUTER WILL DO THE REST"
70 GET K$: IF K$ <> "V" AND K$ <> "I" AND K$ <> "R" THEN 70
80 IF K$ = "V" THEN 200
90 IF K$ = "I" THEN 300
100 REM ANSWER FOR RESISTANCE, R
110 PRINT; INPUT "A VALUE FOR V PLEASE"; V: IF V <= 1 THEN GOSUB
2000: GOTO 10
120 PRINT: INPUT "A VALUE FOR I PLEASE" ; I: IF I <= 1 THEN GOSUB 2000: GOTO 10
130 R = V / I: PRINT: PRINT "THE RESISTANCE R = " ; R ; " OHMS"
140 GOSUB 1000: GOTO 10
200 REM ANSWER FOR VOLTAGE, V
210 PRINT: INPUT "A VALUE FOR I PLEASE" ; I: IF I <= 1 THEN GOSUB 2000: GOTO 10
220 PRINT: INPUT "A VALUE FOR R PLEASE" ; R: IF R <= 1 THEN GOSUB 2000: GOTO 10
230 V = I * R: PRINT: PRINT "THE VOLTAGE V = " ; V ; " VOLTS"
240 GOSUB 1000: GOTO 10
300 REM ANSWER FOR CURRENT, I
310 PRINT: INPUT "A VALUE FOR V PLEASE" ; V: IF V <= 1 THEN GOSUB 2000: GOTO 10
320 PRINT: INPUT "A VALUE FOR R PLEASE" ; R: IF R <= 1 THEN GOSUB 2000: GOTO 10
330 I = V / R: PRINT: PRINT "THE CURRENT I = " ; I ; " AMPS"
1000 REM ANOTHER GO ROUTINE
1010 PRINT: PRINT "ANOTHER GO?"
1020 GET R$: IF R$ = "" THEN 1020
1030 IF R$ = "Y" THEN RETURN
1040 IF R$ = "N" THEN PRINTCHR$(147): END
1050 IF R$ <> "" THEN 1020
2000 REM ZERO INPUT TRAPPING ROUTINE
2010 PRINT: PRINT "SORRY ZERO IS NOT ALLOWED" ; PRINT: PRINT "TRY AGAIN"
2020 FOR D = 1 TO 1500: NEXT: RETURN

I have included an 'another go?' routine just for fun, and as we are just 'passing the time', I have included one small error in my solution, such that the program should crash if you choose one particular variable. Run the program and find out which one it is, and then why it crashes, then correct my program. Though, even before running the program, you may be able to see the error. A close look at the line numbers might help. The answer is in the last chapter.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The next problem will help those of you who just can't get used to the TV weathermen giving tomorrow's temperatures in degrees Centigrade.

Twenty degrees Centigrade never seems quite as warm as sixty eight degrees Fahrenheit, does it?
Problem Forty Three

Write a program to convert degrees Centigrade into degrees Fahrenheit, so that the temperature can be inputted from the keyboard, and the conversion displayed on the screen.

The main problem here is to find a formula that will change degrees Centigrade into degrees Fahrenheit. Any Physics O level text book should produce this formula for you:

\[
\text{degrees Fahrenheit} = \left( \text{degrees Centigrade} \times \frac{9}{5} \right) + 32.
\]

So our computer formula is:

\[
F = C \times \frac{9}{5} + 32.
\]

No need for a bracket, as the computer carries out multiplication and division before addition and subtraction, unless of course you put the addition and/or subtraction in a bracket, as then the computer carries out calculations in a bracket before anything else.

A subsidiary problem is to print out the answer on the screen, and to ensure that it is always placed centrally in the row being used, as the principles of window dressing demand.

The answer of course, will vary from answer to answer, from 1 number long to many. For example: 20 degrees Centigrade is 68 degrees Fahrenheit, and 22 degrees Centigrade is 71.6 degrees Fahrenheit, and 22.2 degrees Centigrade is 71.96 degrees Fahrenheit, and so on.

We must therefore make allowances for this changing length, using the PRINTTAB and LEN$(X)$ devices:

\[
L = \text{LEN}$(\text{STR}$(F))$
\]

and

\[
\text{PRINTTAB}((20-L/2)-10) \text{F} ; "\text{DEGREES FAHRENHEIT}"\]

STR$(F)$ converts the number of degrees Fahrenheit, $F$, into a string so that we can
apply the LEN$(X\$) function to it, in order to find its length.

Then we subtract half that length, in the usual fashion, from half a screen row or line, but then also subtract half of the PRINT statement from it as well, which should centralise the answer, come what may.

My solution looks like this:

10 PRINTCHR$(147):PRINT:PRINTTAB(9)"TEMPERATURE CONVERSION"
20 PRINT:PRINTTAB(8)"CENTIGRADE TO FAHRENHEIT"
30 FOR R=1 TO 5:PRINT:NEXT
40 PRINTTAB(4)"ENTER DEGREES CENTIGRADE PLEASE"
50 PRINT:PRINTSPC(18):INPUT C
60 IF C<1 THEN 10
70 F=C*9/5+32:L=LEN(STR$(F))
80 PRINT:PRINTTAB((20-L/2)-10)F;"DEGREES FAHRENHEIT"
90 PRINT:PRINTTAB(14)"ANOTHER GO?"
100 GET K$:IF K$="":THEN 100
110 IF K$="Y" THEN 10
120 IF K$="N" THEN PRINTCHR$(147):END
130 IF K$<>"":THEN 100

I have error trapped for Centigrade INPUTs of less than one, and kept the program running from the keyboard.

The next problem extends the previous one, so that either temperature scale can be inputted to provide an all round conversion program.

*******************************************************************************
*                           *
* Problem Forty Four         *
*                           *
* Now extend the program in forty two, to allow the conversion to be made in either *
* direction, with the choice to be made by the user of the program.              *
*                           *
*******************************************************************************

Problem forty three was straightforward in that the only choice the user had was to pick a value for Centigrade, and then type it in.

Now, the user has the choice of either Centigrade or Fahrenheit, plus a choice of degree value.

The first part of the problem therefore, is how to present the first choice, so:

    PRINT"PLEASE ENTER YOUR CHOICE, C OR F"
    INPUT C$   (C for Choice)
will suffice, suitably window dressed for the program.

We then have to provide two separate subroutines, one for Centigrade and one for Fahrenheit, both of which will be similar to the solution for the last problem.

We know the formula for Centigrade to Fahrenheit, but for Fahrenheit to Centigrade we need a different formula, namely:

\[ C = (F - 32) \text{multiplied by 5 and divided by 9} \]

or

\[ C = (F - 32) \times \frac{5}{9} \]

in computer language. Notice the need for a bracket with this formula.

So my solution, dressed for the occasion is:

```
5 REM INSTRUCTIONS AND MAIN PROGRAM
10 PRINTCHR$(147):PRINT:PRINTTAB(9)"TEMPERATURE CONVERSION"
20 PRINT:PRINTTAB(6)"CENTIGRADE TO FAHRENHEIT: C"
30 PRINT:PRINTTAB(6)"FAHRENHEIT TO CENTIGRADE: F"
40 PRINT:PRINTTAB(4)"PLEASE ENTER YOUR CHOICE, C OR F"
50 PRINT:PRINTSPC(18):INPUT C$
60 IF C$<>'C' AND C$<>'T' THEN 10
70 IF C$="C" THEN GOSUB 1000:GOSUB 3000:GOTO 10
80 IF C$="F" THEN GOSUB 2000:GOSUB 3000:GOTO 10
90 END
1000 REM CENTIGRADE TO FAHRENHEIT
1010 PRINT:PRINTTAB(5)"ENTER DEGREES CENTIGRADE PLEASE"
1020 PRINT:PRINTSPC(18):INPUT C
1030 IF C<1 THEN 1010
1040 F=C*9/5+32:L=LEN(STR$(F))
1050 PRINT:PRINTTAB((20-L/2)-9)F;'"DEGREES FAHRENHEIT"
1060 RETURN
2000 REM FAHRENHEIT TO CENTIGRADE
2010 PRINT:PRINTTAB(5)"ENTER DEGREES FAHRENHEIT PLEASE"
2020 PRINT:PRINTSPC(18):INPUT F
2030 IF F<1 THEN 2010
2040 C=(F-32)*5/9:L=LEN(STR$(C))
2050 PRINT:PRINTTAB((20-L/2)-9)C;'"DEGREES CENTIGRADE"
2060 RETURN
3000 REM ANOTHER GO ROUTINE
3010 PRINT:PRINTTAB(14)"ANOTHER GO?"
3020 GET K$:IF K$="" THEN 3020
3030 IF K$="Y" THEN RETURN
3040 IF K$="N" THEN PRINTCHR$(147):END
3050 IF K$<>'"" THEN 3020
```
Again I have made the program continuous, and have dressed up the screen.

Remember that any programs you write, no matter how small, are improved by being presented in an acceptable and friendly manner, so any time spent in 'window dressing the screen', is time well spent.

This program can be divided into four blocks, as the REM statements show. The first block, lines 10 to 90, is the main part of the program which tells the user what is happening, and asks him to choose between Centigrade and Fahrenheit, error traps for any other letter but C or F, and then sends the program to the relevant sub routine.

The second block, lines 1000 to 1060, calculates and displays the answer to the calculation for Fahrenheit. Likewise block three, lines 2000 to 2060, does the same for Centigrade.

Block four, lines 3000 to 3050, is the 'another go?' routine.

Our program therefore is made up of four separate blocks, which could in itself be part of a larger program on general conversion, such that when temperature conversion was required, the main program could be called up, after having been made into a subroutine itself.

Naturally, the whole program would have to be wrapped in the necessary lines to enable it to be called, (GOSUB temperature conversion routine line number), and RETURNed from, when no longer required.

~~~~~~~~~

Now for the musically inclined amongst you, here are the three sound problems.

******************************************************************************
* Problem Forty Five *
* Take the first verse of the nursery rhyme 'Little Bo-peep', and write a program to *
* print each note of the tune, together with its word, or part word, on the screen, as *
* the note is played. Only have one word, or part word, on the screen at any one time. *
******************************************************************************
The Commodore 64’s music facilities, although straightforward, are perhaps at first a little complicated. How to program the notes should therefore be the first problem to solve.

‘Little Bo-peep’ is a simple tune, and doesn’t have too many different notes. There are in fact five, namely, F,G,A,B flat and C. The tune is on page 104, and the words of the first verse are as follows:

Little Bo-peep
Has lost her sheep,
And doesn’t know where to find them.
Leave them alone
And they’ll come home,
Bringing their tails behind them.

When writing music on the 64, we need to program for a number of parameters, and in a particular order:

1. Voice used, (voice is another word for channel).
2. Volume.
3. Attack/Decay.
4. Waveform.
5. High note code.
7. Duration of note.
8. Switch off parameters at end of note.

Quite a list, but not really as complicated as you might think. Until you become
proficient at using the sound facilities of the 64, it is better to restrict yourself to using one voice or channel from the three that are available. Each channel has its own set of memory addresses, with the exception of the ‘volume’. This address is the same for all three channels.

The first thing to do is to make a list of all the required memory addresses to POKE, for POKE we must, for our selected channel. We are going to use Channel One, so:

**Channel One**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume</td>
<td>54296</td>
</tr>
<tr>
<td>Attack/Decay</td>
<td>54277</td>
</tr>
<tr>
<td>Waveform</td>
<td>54276</td>
</tr>
<tr>
<td>High note</td>
<td>54273</td>
</tr>
<tr>
<td>Low note</td>
<td>54272</td>
</tr>
</tbody>
</table>

For the length of note we use a FOR...NEXT delay loop.

Before I go any further I’ll explain a few of the parameters used.

Volume is obvious, the loudness of the note played. As you can control the volume from your TV set, we shall choose the loudest. The scale starts at zero for no sound at all, and goes up to 15 for maximum.

Attack refers to the speed at which the note rises to its full volume.

Decay the speed that it falls again. If you make the length of the FOR...NEXT loop too short, you may not hear all the note as it decays. A simple explanation is given on pages 84 and 85 of your 64 manual. To get a fuller explanation you’ll have to wait for the next book in this series.

Waveform determines what overall shape your chosen note will adopt, once it has been shaped using the attack/decay parameter. The 64 has four possible waveshapes:

1. Triangular, equally divided between rise and fall.
2. Sawtooth, slow rise and sharp decay.
3. Pulse or square, instant rise, flat top, instant decay.
In the main, for musical purposes, it is better to use the triangular and the sawtooth waveforms.

Each waveform has its own code for POKEing into the 54276 waveform address:

1. Triangular - 17.
2. Sawtooth - 33.
3. Pulse - 64.

On pages 152 to 154, in Appendix M, of the 64 manual is a list of all the high and low codes to be POKEd for each note in the eight octave range available. The words 'high' and 'low' are not really concerned with frequency, but refer to the two parts of the binary code to go to make up the number which produces the required note from the sound chip.

The procedure to be followed, therefore, to tell the computer what note is to be played, and in what manner to play it, is as follows:

1. Select the voice or channel - 1
2. Select the volume - 15
3. Select the style of Attack/Decay required - 68
4. Select the waveform - 33
5. Select the note.
6. Select the length.
7. POKE note.
8. Program delay
9. Switch it off.

I have suggested the type of attack/decay and waveform to be used, the actual choice is up to you.

I have not discussed the use of the sustain/release parameter; with a straightforward tune, this is quite unnecessary.

When writing a simple tune, it's a good idea to put all the parameters in 1 to 4 above, at the beginning of your program, and to give all the addresses their own variable letter:

V for Volume.
AD for Attack/Decay.
W for waveform.
NH for high code (Note High).
NL for low code (Note Low).

So, the beginning of the program could be:

10 V=54296:AD=54277:W=54276:NH=54273:NL=54272
20 POKE V,15:POKE AD,68:POKE W,33

103
Now for the tune.

The first three notes are all quavers, and note F, which has codes of 22 and 227 in the fourth octave, a reasonable octave to sing! The first three part words are LIT, TLE, BO.

If we are to avoid a program as long as our arm, with a line for each note and its word, then READ and DATA should be used.

As each note has to be played with each separate word displayed on the screen, then we shall have to READ the word, W$, the high and low codes, HI and LO, and the length of the note, L, from the DATA statement, for each note. So line 30 could be:

30 READ W$:READ HI:READ LO:READ L

and our DATA statement:

100 DATA LIT,22,227,300,TLE,22,227,300,BO,22,227,300

I have chosen 300 for the length of a quaver purely from experiment and experience. You can, of course, choose a length to suit yourself, or maybe you already have.
The next step is to code the remainder of the first verse, and then to place the code in DATA statements.

One snag you may have come across, is that the two words in the verse, WHERE and TAILS, have two notes each. Line 140 and 180 in my program show you how I have dealt with this.

And by the way, dotted notes increase the length of the note by a half.

When the song has finally finished playing you have to switch off the program. To do this use a code which should not produce a sound. For example, line 40 should allow for this:

40 IF HI=-1 THEN END

I decided to place each word in the centre of a clear screen, so line 50 is:

50 PRINTCHR$(147):FOR R=1 TO 11:PRINT:NEXT:
PRINTTAB(20-LEN(W$)/2)W$

Now that we have all our notes coded in DATA statements, we can start to play the tune, by POKEing the two note codes:

60 POKE NH,HI:POKE NL,LO

Then leave the note playing for the required length of time with:

70 FOR T=1 TO L:NEXT

Then switch off all the codes for the note with:

80 POKE NH,0:POKE NL,0:POKE AD,0:POKE W,0

Then do it all again:

90 GOTO 20

And to signal the end of the tune use:

200 DATA END,-1,-1,-1

In line 200, don’t forget to give each parameter some DATA, that is, use one word and three numbers, or you’ll get a DATA error.

Here, then, is my full program.

I hope your’s is something like it, but there can be many differences, namely: voice, volume, attack/decay, waveform and length.
Problem Forty Six

Now, take the same nursery rhyme, but this time print the words out as a verse. *
* Again, print the word as each note is played.
*

We already have our tune coded from the last problem.

The difficulty here is to print out the words as a verse.

So, the first thing to do is to select the first word of each line, and then tell the computer to start a new PRINT line when it detects one. Your 'new line' words may not be the same as mine, it all depends where you learnt your nursery rhymes.

We will have to change the PRINT routine, starting at line 50, with:

50 IF W$="HAS" OR W$="AND" OR W$="LEAVE" OR W$="BRING"
   THEN PRINT
55 PRINT W$;

10 V=54296:AD=54277:W=54276:NH=54273:NL=54272
20 POKE V,15:POKE AD,68:POKE W,33
30 READ W$:READ HI:READ LO:READ L
40 IF HI=-1 THEN END
50 PRINTCHR$(147):FOR R=1 TO 11:PRINT:NEXT:
60 POKE NH,HI:POKE NL,LO
70 FOR T=1 TO L:NEXT
80 POKE NH,0:POKE NL,0:POKE AD,0:POKE W,0
90 GOTO 20
100 DATA LIT,22,227,300,TLE,22,227,300,B0,22,227,300
110 DATA PEEP,22,227,600,HAS,22,227,300,LOST,25,177,600
120 DATA HER,25,177,300,SHEEP,25,177,600,AND,25,177,300
130 DATA DOES,28,214,300,NOT,30,141,300,KNOW,34,75,300
140 DATA WHERE,34,75,300,WHERE,30,141,300,TO,28,214,300
150 DATA FIND,28,241,900,THEM,25,177,900,LEAVE,34,75,300,
    THEM,28,214,300
160 DATA A,28,214,300,LONE,28,214,600,AND,28,214,300,THEY"LL,
   30,141,600
170 DATA COME,25,177,300,HOME,25,177,900,BRING,28,214,300,
   ING,30,141,300
180 DATA THEIR,28,214,300,TAILS,25,177,300,TAILS,22,227,300
190 DATA BE,25,177,300,HIND,22,227,900,THEM,22,227,900
200 DATA END,−1,−1,−1
This will print out each line of my verse, with a line space between them.

We must also adapt the DATA statements to suit the new requirements, by adding spaces as necessary. But, of course, the words with added spaces must now be placed in inverted commas, or the spaces will be ignored when the DATA lines are READ. Also, we can’t write ‘WHERE’ and ‘TAILS’ twice in our verse, so I’ve allowed for that too, see lines 140 and 180.

Here then, is my amended program:

```
10 V=54296:AD=54277:W=54276:NH=54273:NL=54272
11 PRINTCHR$(147)
20 POKE V,15:POKE AD,68:POKE W,33
30 READ W$:READ HI:READ LO:READ L
40 IF HI=-1 THEN PRINT:END
50 IF W$="HAS" OR W$="AND" OR W$="LEAVE" OR W$="BRING"
     THEN PRINT
55 PRINT W$;
60 POKE NH,HI:POKE NL,LO
70 FOR T=1 TO L:NEXT
80 POKE NH,O:POKE NL,O:POKE AD,O:POKE W,O
90 GOTO 20
100 DATA "LIT",22,227,300,"TLE ",22,227,300,BO,22,227,300
110 DATA "PEEP",22,227,600,HAS,22,227,300," LOST",25,177,600,
     " HER",25,177,300
120 DATA " SHEEP","25,177,600,AND,25,177,300," DOES,28,214,300
130 DATA "N'T,30,141,300," KNOW",34,75,300," WHERE",34,75,300
140 DATA ",",30,141,300,TO,28,241,300," FIND",28,214,900
150 DATA " THEM",25,177,900,LEAVE,34,75,300," THEM",28,214,300
160 DATA ",A",28,214,300,LONE,28,214,600,AND,28,214,300,
     " THEY'LL",30,141,600
170 DATA ",COME",25,177,300,"HOME.",25,177,900,
     BRING,28,214,300,ING,30,141,300
180 DATA ", THEIR",28,214,300,TAILS,25,177,300," ",22,227,300
190 DATA BE,25,177,3300,HIND,22,227,900," THEM",22,227,900
200 DATA END,-1,-1,-1
```

You will notice that I have also changed line 40, this moves the READY down one line, for extra neatness.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

* Problem Forty Seven

* Now adjust Problem Forty Six, so that various choices of speed can be inputted
* from the keyboard, for example: quick, medium and slow.

*
The whole point of this problem is to allow you the opportunity to see the result of various simple changes that you can make to the sound produced by the 64. Changing the codes for the Attack/Decay and Waveform parameters, and altering the length of the notes, can make interesting variations to your tune.

As this part of the program must be controlled from the keyboard, I have ‘window dressed’ it to suit, with:

```
11 PRINTCHR$(147):PRINTTAB(3)"I WILL PLAY YOU THE NURSERY RHYME"
12 PRINT:PRINTTAB(12)"LITTLE BO-PEEP"
13 PRINT:PRINTTAB(4)"HOW FAST DO YOU WANT IT PLAYED?":PRINT
14 PRINTTAB(6)"SLOW:1, MEDIUM:2, QUICK:3":PRINT
15 GET SP$:IF SP$="" OR VAL(SP$)<1 OR VAL(SP$)>3 THEN 15
```

When the user has selected a speed, the computer must then be told what to do to play the tune at the required tempo.

The simplest way is to use ON...GOTO:

```
16 ON VAL(SP$) GOTO 17,18,19
17 SPD=2:GOTO 20
18 SPD=1:GOTO 20
19 SPD=0.5
```

My additions therefore fit snugly between lines 10 and 20.

We must also change line 70, which contains the note length code, by incorporating the three lengths of SPD, so:

```
70 FOR T=1 TO L*SPD:NEXT
```

will allow the selected speed to take effect.

Here is my amended program, but as the DATA statements are exactly the same as those for Problem Forty Six, I have left them out of the listing:

```
1 REM NURSERY RHYMES ON THE 64
10 V=54296:AD=54277;W=54276;NH=54273;NL=54272
11 PRINTCHR$(147):PRINTTAB(3)"I WILL PLAY YOU THE NURSERY RHYME"
12 PRINT:PRINTTAB(12)"LITTLE BO-PEEP"
13 PRINT:PRINTTAB(4)"HOW FAST DO YOU WANT IT PLAYED?":PRINT
14 PRINTTAB(6)"SLOW:1, MEDIUM:2, QUICK:3":PRINT
15 GET SP$:IF SP$="" OR VAL(SP$)<1 OR VAL(SP$)>3 THEN 15
16 ON VAL(SP$) GOTO 17,18,19
17 SPD=2:GOTO 20
18 SPD=1:GOTO 20
19 SPD=0.5
```

108
At the beginning of this problem I wrote that the program would allow you to test the attack/decay and waveform codes.

Well, feel free to experiment now, by editing line 20 as regards AD and W. You could, of course, amend the program to make the AD and W codes inputted from the keyboard as well, but remember AD must always be programmed before W.

If you are going to study the sound facilities of the 64, this would be a sensible arrangement - it's where we will start in the next book, SPRITES, SOUND and SIMON'S BASIC, for the 64", also to be published by Sigma Technical Press.

So, in the meantime, try the following changes:

With W set to 17:

- AD=202 produces a swaying melody.
- AD=206 produces a gentle rock.
- AD=210 the player hasn’t quite got his breathing right.
- AD=101 sounds like an electronic guitar.
- AD=42 sounds like a xylophone.

Change W back to 33 and AD to 202, and you've got yourself a cheap trumpet, without pursing your lips.

~~~~~~~~~~~~

Our final problem in this book, with the exception of the Extras, which you know about already, is quite complicated. If you manage to solve it completely, without reference to my notes or solution, then you can be sure that your powers of logic are reasonably good.

Consider it as a kind of 'end of term test'!

But don’t give up too easily, neither plod on into the small hours, steeped in the bleary clouds of frustration, it might put you off computing for ever!
Problem Forty Eight

Without using sprites, write a program that fires one ball from the bottom of the screen, and another from the side, such that when they collide, they explode. Make the first ball bounce around the screen, the second just shoot across, and then disappear. Print, on the screen, the time it took in seconds for the balls to collide.

This problem, while being an exercise in logical planning, also requires some little knowledge of the Commodore 64's screen, and at the same time gives some insight into randomness. Just what are the chances of two balls, which could have any one of a thousand possible different locations, landing in the same place at the same time?

The diagram of the screen, showing the increments required to drive the balls around the screen, both up and down, and sideways, is given on the next page.

But now, on with the challenge.

With this final problem, there are a number of things that have to be decided each time the ball coming from the bottom of the screen changes its position.

1. Where is the ball coming from, the bottom of the screen, or the top?
2. Is it approaching the right hand side, or the left hand side of the screen?

In other words there are eight different possible approaches to the four sides of the screen, that could occur.

They are:

a. To the top, from the left.
b. To the top, from the right.
c. To the left, from below.
d. To the left, from above,
e. To the right, from below.
f. To the right, from above.
g. To the bottom, from the right.
h. To the bottom, from the left.

The program that decides on the next position for this ball must contain eight statements, one for each of the above, eight IF....THEN statements.

But they come in pairs, two for the top, two for the bottom, two for the left and two for the right. There will be four sets of IF....THEN statements therefore, each set
sensing where the ball is going to, and where it is coming from.

For example, in (a) on the previous page.

"To the top, from the left", means that the screen position will be reducing numerically in value, that is, it will be a negative quantity, and will be decreasing in steps of 39 positions. In other words in steps of −39, see the diagram.

So, the moment the ball goes outside the available screen memory, 1024, we will know that it has gone too far, and must therefore return down the screen, in a suitable rebound fashion. So, before we POKE its next position, its numerical direction must be changed.

To do this we can write an instruction similar to the following:

\[
\text{IF } P<1024 \text{ AND } R=-39 \text{ THEN } R=41
\]

where P is the screen position, and R the step or increment being made at that time.

R must now equal +41, because the ball must change direction, and rebound to the right, therefore requiring an increase in the size of the step of 2. So:
-39 times –, plus 2, equals +41

The same process can now be applied to the top and the other direction, and to the bottom of the screen, in both directions, which will give us four instructions:

- IF $P < 1024$ AND $R = -39$ THEN $R = 41$
- IF $P < 1024$ AND $R = -41$ THEN $R = 39$
- IF $P > 2023$ AND $R = 39$ THEN $R = -41$
- IF $P > 2023$ AND $R = 41$ THEN $R = -39$

This routine would now bounce the ball from the top or the bottom of the screen, in the correct direction.

To bounce it from the sides of the screen, we must first sense when it actually gets there.

If you think about it, and I hope you do, there is only one position where, if you divide by the length of the line, you will get a whole number for your answer.

For example, seventeen places in from the left of the top row, we have position 1040, $(1024 + 16)$. Divide this by 40 and we have 26, but divide 1025 by 40, and you get 26.025, likewise any other position will not produce a zero decimal.

If we use the function INT with the instruction:

- IF $\text{INT}(P)/40 = P/40$ THEN do something relevant called $X$

$X$ will only happen if $P/40$ equals a whole number, otherwise the computer would move on to the next line.

But we want to sense the side of the screen, position 1024, not somewhere inside.

So for 1024 to be a whole number when divided by 40, we must first subtract 24 from it.

Therefore if:

- IF $\text{INT}((P-24)/40) = (P-24)/40$ THEN $X$

is used, $X$ will now occur when $P$ is at the left hand side of the screen, for any value of $P$ that indicates the first position on the left hand side.

We must also sense which direction the ball is coming from at this point. That is, from the top or the bottom of the screen, again with negative or positive steps and their values.

If they are negative, then the ball must be moving up the screen, in steps, see the diagram.

If they are positive, then they must be going down, less than 40, 39.
So our two IF...THEN statements to cover this, should look like:

\[
\begin{align*}
&\text{IF } \lfloor(P-24)/40\rfloor = (P-24)/40 \text{ AND } R=-41 \text{ THEN } R=-39 \\
&\text{IF } \lfloor(P-24)/40\rfloor = (P-24)/40 \text{ AND } R=39 \text{ THEN } R=41
\end{align*}
\]

as the ball must bounce from the side of the screen at this point, therefore R, the step, or increment, must change from 41 to 39 and vice versa.

Similarly at the right hand side, we require two more IF...THEN statements:

\[
\begin{align*}
&\text{IF } \lfloor(P-23)/40\rfloor = (P-23)/40 \text{ AND } R=-39 \text{ THEN } R=-41 \\
&\text{IF } \lfloor(P-23)/40\rfloor = (P-23)/40 \text{ AND } R=41 \text{ THEN } R=39
\end{align*}
\]

We have now produced a check routine for keeping the ball inside the screen boundaries, and always on the move.

We must produce a random POKE position P, and a random step to send the ball either left or right up the screen, seeing as it is fired from the bottom of the screen.

To do this we can use:

\[
P=\lfloor\text{RND}(1)*6\rfloor+2000
\]

which will fire the ball from any one of six positions at the base of the screen, and:

\[
S=\lfloor\text{RND}(1)*3\rfloor+1: \text{IF } S=2 \text{ THEN repeat } R=-42+S
\]

which will produce a step, R, of either -39 or -41, to move the ball up the screen, either left or right. The value of S=2 is not required as this would send it straight upwards, that is P=P-40, (R=-40).

If P=P+R is included in the program, it will increment the POKE position each time the program runs.

The problem said we must have two balls on the screen, otherwise, of course, we wouldn’t get any collisions. We have one already coming from the bottom of the screen, so let’s have the other one coming from the left hand side, and select, say, five different positions that it could start from, with:

\[
P2=\lfloor\text{RND}(1)*5\rfloor+1 \\
\text{IF } P2=1 \text{ THEN } P3=1384 \\
\text{IF } P2=2 \text{ THEN } P3=1424 \\
\text{IF } P2=3 \text{ THEN } P3=1464 \\
\text{IF } P2=4 \text{ THEN } P3=1504 \\
\text{IF } P2=5 \text{ THEN } P3=1544
\]

We must also fire the second ball across the screen and then stop it, (make it disappear), before we send out another one, therefore:

\[
\text{IF } P3>1583 \text{ THEN do it again}
\]

113
will stop it at the right hand side, and:

\[ P3 = P3 + P2 \]

will make it jump across the screen, its speed depending on the value of \( P2 \), which introduces a little more randomness into the program for the second ball.

We must also check to see if and when they collide.

So a:

\[
\text{IF } P = P3 \text{ THEN do something}
\]

statement must be included in the program, plus a sequence for exploding the two balls once they've collided. For this we can create a subroutine, which can include a check routine to ask if a repeat performance is required.

Lastly we must also check how long the whole procedure leading up to the collision took, and so by including \( T$ = "000000" \), at the start of the program, and using \( T1 \) as a seconds counter, we can display this information at the end of the program.

My program then, after a great deal of experimentation, ended up like this, but don't forget, with a problem of this nature, there are many ways of arriving at a final solution.

Let's hope that you finally got there, without too much heartbreak:

```
5 REM SELECTION OF VARIABLES
10 PRINTCHR$(147); T$="000000"
100 P1=INT(RND(1)*6)+2000
110 S1=INT(RND(1)*3)+1; IF S1=2 THEN 110
120 R1=-42+S1
130 P2=INT(RND(1)*5)+1
135 REM POSITION FOR SECOND BALL
140 IF P2=1 THEN P3=1384
150 IF P2=2 THEN P3=1424
160 IF P2=3 THEN P3=1464
170 IF P2=4 THEN P3=1504
180 IF P2=5 THEN P3=1544
190 REM POSITION FOR FIRST BALL
200 P1=P1+R1; PRINTCHR$(147)
210 IF P1<1063 AND R1=-39 THEN R1=41
220 IF P1<1063 AND R1=-41 THEN R1=39
230 IF INT((P1-24)/40)=(P1-24)/40 AND R1=-41 THEN R1=-39
240 IF INT((P1-24)/40)=(P1-24)/40 AND R1=39 THEN R1=41
250 IF INT((P1-23)/40)=(P1-23)/40 AND R1=-39 THEN R1=-41
260 IF INT((P1-23)/40)=(P1-23)/40 AND R1=41 THEN R1=39
270 IF P1>1984 AND R1=39 THEN R1=-41
280 IF P1>1984 AND R1=41 THEN R1=-39
```

114
Well there we are, let’s hope you get a collision before the lights go out, and here’s the explanation of the subroutines:

Lines 1000 to 1050 produce a few stars around the point of collision, I hope you can work out all the POKEs.

Lines 1060 to 1100 give the choice of another go.

Lines 2000 to 2050 is the timing routine, which prints out how long the program took before the balls collided.

The program, although quite long, doesn’t do very much, but don’t forget it was an exercise in logical reasoning, and not a definitive core for all arcade games. But with a few extra POKE statements, the screen could be made a little more interesting. As our balls are perhaps on the move in free space?, you could include a few stars, planets, or comets, for them to collide with or glide by, as the mood took you. You could also include a time limit, or run out of energy, for example.

I will leave that part up to you, and mention that of course it’s a lot easier with sprites!
Lastly I would mention, that if you leave out the clear screen statement in line 200, you can create some interesting screen patterns. Also, changing the colours of the POKE statements in lines 290 and 300, will make the patterns more interesting, especially if you incorporate a change colour routine as well. That, I'm afraid, is again all up to you!
This chapter is concerned solely with the solutions to the extra problems you were presented with in the previous six chapters.

My solutions will not necessarily look completely like yours, it will all depend on whether you made up your own solutions to the original problems or not.

In any event I hope that you attempted the extra problems, and that my solutions will give you some ideas on how to go about solving them.

You were asked to print both messages on the top line of the screen, but not at the same time of course.

To do this, when you clear the screen, the message should form part of the PRINT statement concerned with clearing the screen, as follows:

```
10 PRINTCHR$(147);"TYPE IN YOUR FIRST NAME":INPUT N$:
    L=LEN(N$)
20 PRINTCHR$(147);"YOUR NAME HAS";L;"LETTERS IN IT"
```

~~~~~~~~~~~~~~~~~~~~~~~~~
Here you were asked to make the program ask the user to type in how many letters he had in the name he had just typed in.

Obviously therefore, there must now be an extra INPUT statement requesting a number of letters, which can then be compared with the true length of the name:

```plaintext
10 PRINTCHR$(147): INPUT"PLEASE TYPE IN YOUR NAME"; N$
20 PRINTCHR$(147): INPUT"HOW MANY LETTERS ARE IN YOUR NAME"; N
30 L = LEN(N$): IF N = L THEN 50
40 PRINT "WRONG, YOU CAN'T COUNT": END
50 PRINT "RIGHT, I KNEW YOU COULD COUNT": END
```

Here’s the amended solution, can you spot the difference?

```plaintext
10 PRINTCHR$(147): INPUT"TYPE IN A WORD"; W$
20 PRINTCHR$(147): P = INT(RND(1) * 1000) + 1024
30 FOR N = 1 TO LEN(W$): L = ASC(MID$(W$, N, 1)): IF L > THEN 50
40 IF L > 64 THEN L = L - 64
50 POKE P + 54272, 0: POKE P, L: P = P + 1: NEXT
60 FOR D = 1 TO 2000: NEXT: GET K$: IF K$ = "" THEN 20
```

You were asked in this extra to change the colour of the letters in the name, so that each one was different, within the limits of the sixteen colours available.

Here is the amended solution:

```plaintext
10 C = 0: PRINTCHR$(147): INPUT "TYPE IN A WORD "; W1$
20 INPUT "NOW ANOTHER ONE "; W2$
```
30 PRINTCHR$(147): P1=INT(RND(1)*1000)+1024:  
P2=INT(RND(1)*1000)+1024  
40 FOR R=1 TO LEN(W1$): L=ASC(MID$(W$,R,1)): IF L<64 THEN 60  
50 IF L>64 THEN L=L-64  
55 C=C+1  
60 POKE P1+54272,C: POKE P1,L: P1=P1+1: NEXT  
70 FOR R=1 TO LEN(W2$): L=ASC(MID$(W$,R,1)): IF L<64 THEN 90  
80 IF L>64 THEN L=L-64  
85 C=C+1  
90 POKE P2+54272,C: POKE P2,L: P2=P2+1: NEXT  
100 FOR D=1 TO 2000: NEXT: GET K$: IF K$="": THEN 30

Line 10 gets the variable C, used for the Colour, in the screen charactercolour: 
POKE statement, to zero. This is not strictly needed in this instance, as it doesn’t 
matter which colour you start with.

It’s always good practice to set your variables to the values required, of 
course is very often zero, at the start of a program.

Lines 55 and 85 increment C by 1 each time the two POKE lines are used, so that C 
carries on changing the colour ad infinitum. You will no doubt notice that C goes 
beyond 15, but this not matter, as the computer knows that there are really only 
sixteen colours, (0 to 15), and that 16 really means 0 again, and 17 means 1, and so on up the scale.

You could of course if you wanted to, decrement C to zero, when it reached 15, 
with an IF.... THEN statement, if you felt the need. But that’s up to you. Oh, decisions, decisions!

Now, remember, I said that problems had a way of creating new problems? Well, 
one has just appeared in this solution.

Can you find it, and more to the point, can you correct it? If you’re stuck have a 
quick look at Extra six.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

*~*~*~*~*~*~*~*~*~*~*~*~*

* Extra five

*~*~*~*~*~*~*~*~*~*~*~*~*

In this extra you had to stop the first word being overprinted by the second word, if 
their randomly generated positions happened to be the same.

Getting the computer to look at them, then compare them, and if they are the same, 
discard them, then choose two more, and repeat the procedure again and again,
until they're different, is as good a way as any.

So, for the answer to Extra four, add the next line to my solution:

```
35 IF P1=P2 THEN 30
```

which will do just this, and until P1 and P2 are different, the computer will not move on to line 40.

~~~~~~~~~~~~~~~

In this program, in line 25, I've put:

```
IF C=6 THEN 25
```

Why?

Well 6 in the colour codes is blue, and blue happens to be the normal background colour of the screen, so anything PRINTed or POKEd with colour 6, will not show up, a kind of invisible writing.

So I asked the computer to choose again when 6 is chosen as a random colour.

Here's the solution. You know all about changing the colour for each letter, if you've dealt with Extra four:

```
10 PRINTCHR$(147):C=0
20 N=INT(RND(1)*9)+48:POKE 55796,C:POKE 1524,N:
    FOR D=1 TO 200:NEXT
25 C=C+1:IF C=6 THEN 25
30 GET K$:IF K$="":THEN 10
```

~~~~~~~~~~~~~~~

Here you were asked to PRINT a five letter word diagonally across the centre of the screen.
As the problem said PRINT, I decided to use a PRINT statement to do it.

In line 20 I've error trapped for only five letter words, and on line 30, moved the start point of the PRINT statement down the screen.

Line 40 puts a PRIN TTAB statement into a loop, so that it selects each letter of W$ in turn, and places it on the next PRINT line, but at the same time moving into the next column.

I've started at column 18 in order to centralise the five letter word, with R+17, where R, at the beginning of the loop, is 1.

Here's the solution:

```
10 PRINT CHR$(147): INPUT "TYPE IN A FIVE LETTER WORD"; W$: L = LEN(W$)
20 IF L<5 OR L>5 THEN 10
30 PRINT CHR$(147): FOR R = 1 TO 9: PRINT: NEXT
40 FOR R = 1 TO 5: PRIN TTAB(R+17) MID$(W$, R, 1): NEXT
50 GET K$: IF K$ = "": THEN 50
```

You will have noticed that line 20 could have been written in another way.

Yes, that's right:

```
20 IF L<>5 THEN 10
```

Now try making the word:

a. PRINT all in one row or line,
b. PRINT backwards,
c. PRINT on the opposite diagonal
d. PRINT a word, or words totalling up to say 30 letters long.

You're on your own for these, just thought I would point out that life is one continuous procession of problems!

```
******************************************************************************
* * * Extra eight *
* * *
******************************************************************************
```

INPUT and GET are pretty much the same, in that they allow you to get information in from the keyboard while a program is running.
The difference is that the INPUT string can be more than one character long, but GET only takes one character at a time. You could of course a series of GET statements to do the job of the INPUT statement.

In this extra, you were asked to change the INPUT to GET, and as the INPUT was only supposed to be asking for one letter, this substitution should have created no extra problems, which makes a change.

So, if we change line 30 to:

```
30 GET L$:IF L$=""" THEN 30 and add:
      55 PRINT:PRINTSPC(19)L$
```

The computer will now wait at line 30 as usual, until a key is pressed.

Then it will move directly to lines 40 and 50.

At line 55, the letter of the key that was actually pressed will appear on the screen.
This just dresses up the program, window dressing again, but also reminds the user of the key he pressed.

Inverse video is produced by using CHR$(18), so that could be added to line 80.

```
* Extra nine
* Extra nine
* Extra nine
```

Here we have to POKE our letters into the correct positions on the screen, instead of PRINTing them.

So the following should do the trick:

```
10 FOR R=1 TO 4:N(R)=INT(RND(1)*25)+65:NEXT
20 C=INT(RND(1)*15)+1:IF C=6 THEN 20
30 PRINTCHR$(147):PRINTCHR$(14)
40 POKE 55378,C:POKE 1106,N(1)
50 POKE 55413,C:POKE 1141,N(2)
60 POKE 56178,C:POKE 1906,N(3)
70 POKE 56213,C:POKE 1941,N(4)
80 FOR D=1 TO 1000:NEXT
90 GOTO 10
```

```
* Extra nine
* Extra nine
* Extra nine
```
This extra problem makes the original a little more interesting, and is a form of the card game SNAP, I suppose. It’s completely random and uncontrolled, and no cheating is allowed, so perhaps, on second thoughts, it’s not quite like SNAP!

Here’s my solution, if yours is the same as mine, then you can shout SNAP!:

```
10 FOR R=1 TO 4:N(R)=INT(RND(1)*4)+1
15 IF N(R)=1 THEN N(R)=65
20 IF N(R)=2 THEN N(R)=83
25 IF N(R)=3 THEN N(R)=88
30 IF N(R)=4 THEN N(R)=90
35 NEXT
40 C=INT(RND(1)*15)+1:IF C=6 THEN 40
45 PRINTCHR$(147)
50 POKE 55378,C:POKE 1106,N(1)
55 POKE 55413,C:POKE 1141,N(2)
60 POKE 56178,C:POKE 1906,N(3)
65 POKE 56213,C:POKE 1941,N(4)
70 IF N(1)=N(4) OR N(2)=N(3) THEN GOSUB 1000
75 GOSUB 2000
80 GET K$:IF K$="":THEN 80
85 IF K$=CHR$(32) THEN 10
90 IF K$<>"":THEN 80
95 END
1000 FOR D=1 TO 800:NEXT:PRINTCHR$(147)
1010 FOR R=1 TO 11:PRINT:NEXT
1020 PRINTTAB(12)"HURRAY, YOU WON"
1030 FOR D=1 TO 2000:NEXT:RETURN
2000 FOR D=1 TO 800:NEXT:PRINTCHR$(147)
2010 FOR R=1 TO 11:PRINT:NEXT
2020 PRINTTAB(8)"SPACE BAR FOR ANOTHER GO"
2030 FOR D=1 TO 2000:NEXT:RETURN
```

```
```

Here you had to get the total, place it in the centre of the screen, then flash it on and

---

* Extra eleven

---
off a number of times, before going to GET K$.

To do this we have to break the total down into single characters, then POKE them on to the screen in the correct place.

To do this we can use:

a. \( T$ = \text{STR} (T) \)
b. \( TT$ = \text{MID} (T$, 1, 1) \) (for the tens)
c. \( TU$ = \text{MID} (T$, 2, 1) \) (for the units)
d. \( TU = \text{VAL} (TU$) \)
e. \( TT = \text{VAL} (TT$); \text{IF } TT = 0 \text{ THEN } * \)
f. \( \text{POKE 55796,0:POKE 1524,TT} \)
g. * \( \text{POKE 55997,0:POKE 1525,TU} \)

a turns T into an alphanumeric string.
b and c split T$ down into two characters, one for the tens, and one for the units; make sure you get them the right way round.
d turns TU$ back into a numeric string.
e turns TT$ back into a numeric string.
f and g POKE the two numbers onto the screen.
* By the way, if there are no tens then you don’t have to POKE them, so avoid that line as required.

I’ll leave you to fit it into your program, but it will all come after line 80 in my program.

Good luck.

********************************************************************************
* * *
* Extra twelve *
* * *
********************************************************************************

With problem thirty four, you had to put numbers into their respective columns, and with this extra problem, you had to print out the column totals, the grand total, and the average of all the numbers generated.

To do this, we add to lines 30 to 70:

\[
\begin{align*}
30 & : A=A+1:F=A*1 \\
40 & : B=B+1:G=B*2 \\
50 & : C=C+1:H=C*3 \\
60 & : D=D+1:I=D*4 \\
70 & : E=E+1:J=E*5:K=F+G+H+I+J
\end{align*}
\]
Then change line 80 to:

   80 NEXT

And add lines:

   90 PRINTCHR$(147):CHR$(156)PRINT"THERE WERE":PRINT
   100 PRINT A;"1'S"
   110 PRINT B;"2'S"
   120 PRINT C;"3'S"
   130 PRINT D;"4'S"
   140 PRINT E;"5'S":PRINT
   150 PRINT"IN THIS COUNT"
   160 PRINT:PRINT"MAKING A GRAND TOTAL OF";K
   170 PRINT:PRINT"AND AN AVERAGE OF";K/1000

And finally add line 15:

   15 FOR R=1 to 1000

to match up with the changed line 80, to complete the loop.

The loop variable in line 15, could be any value you like, or could even be inputted from the keyboard, by making:

   15 FOR R=1 TO T

and adding between lines 10 and 20, an INPUT or GET statement, to get a repeat variable, T, of the players' own choosing.

*************************************************************************
*  Extra thirteen                                               *
*************************************************************************

Turning problem thirty five into a game of chance is not too difficult.

My idea is to get the player to guess which column fills up first with ten numbers, which is catered for in lines 2 to 9 of my extra solution.

Setting the variables to zero, getting the random number and printing it out, is done in lines 10 to 70. In addition, lines 30 to 70 keep check of the number of numbers in each column.
Lines 80 to 120 check when the columns are full, and line 130 makes sure the program does it all again until one of them is.

When the computer is sent to line 140, the information is then printed out on a clear screen, by lines 140 to 160.

Lines 163 and 166 check whether the player’s guess was right or wrong, by comparing X, the column filled first, with G, the player’s guess.

Subroutines 1000 and 2000, tell the player whether he or she has won or lost, and increments the score when necessary.

Line 170 checks if he or she wants to play again, and subroutine 3000 to 3030 is the ‘GOODBYE’ screen.

Here is my full answer:

2 PRINTCHR$(147);FOR R=1 TO 8:PRINT:NEXT
4 PRINT"GUESS WHAT NUMBER GETS TEN ENTRIES FIRST"
6 PRINTTAB(11)"NUMBERS ARE 1 TO 5"
8 PRINT:PRINTTAB(8)"TYPE IN YOUR NUMBER HERE":PRINT:
            PRINTSPC(17):INPUT G
9 IF G<1 OR G>5 THEN 2
10 PRINTCHR$(147):A=0:B=0:C=0:D=0:E=0
20 N=INT(RND(1)*5)+1
30 IF N=1 THEN PRINTCHR$(144)N:A=A+1
40 IF N=2 THEN PRINTCHR$(158)N:B=B+1
50 IF N=3 THEN PRINTCHR$(156)N:C=C+1
60 IF N=4 THEN PRINTCHR$(156)N:D=D+1
70 IF N=5 THEN PRINTCHR$(30)N:E=E+1
80 IF A=10 THEN PRINT:PRINTTAB(4)"FULL":X=1:GOTO 140
90 IF B=10 THEN PRINT:PRINTTAB(11)"FULL":X=2:GOTO 140
100 IF C=10 THEN PRINT:PRINTTAB(18)"FULL":X=3:GOTO 140
110 IF D=10 THEN PRINT:PRINTTAB(25)"FULL":X=4:GOTO 140
120 IF E=10 THEN PRINT:PRINTTAB(32)"FULL":X=5:GOTO 140
130 GOTO 20
140 FOR D=1 TO 1000:NEXT:PRINTCHR$(147)CHR$(156)
150 FOR R=1 TO 10:PRINT:NEXT
160 PRINTTAB(8)"COLUMN":X:"WAS FILLED FIRST"
163 IF X=G THEN GOSUB 1000:GOTO 170
166 IF X<>G THEN GOSUB 2000
170 PRINT:PRINTTAB(9)"DO YOU WANT ANOTHER GO?"
180 GET K$:IF K$="" THEN 180
190 IF K$="Y" THEN 2
200 IF K$="N" THEN PRINTCHR$(147):GOSUB 3000:END
210 IF K$="" THEN 180
1000 PRINT:PRINTTAB(15)"YOU’VE WON":S=S+1:RETURN
2000 PRINT:PRINTTAB(16)"YOU LOST":RETURN
3000 FOR R=1 TO 11:PRINT:NEXT
I found this simple game mildly addictive, as did the other members of my family and friends.

Not a lot happens, but you could develop it a bit, by increasing the number of guesses, say one for each family member, up to a maximum of five, (or even increase the number of columns).

This then is a good basis for a horse race game: see who gets to the winning post first, by seeing whose column fills up first.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Extra fourteen

In this last extra problem, you were asked to change problem forty one, to allow for a reset or start time.

I decided to place this facility in a subroutine at lines 1000 to 1200, and to do this have included extra lines at the beginning of the solution program.

One snag in writing it was that as we have a twenty four hour clock displaying six digits, the user could have returned from the keyboard an hour figure that contained only one number. Of the six digits, two are for hours, so this would upset T1$ somewhat. The same applies to the minutes and the seconds.

So an inputted single number has to padded out with an extra zero, to change say for example: 3 to 03. This is what lines 1030, 1040 and 1050 do. as well as get the number from the keyboard:

```
IF LEN(H$)=1 only one number for hours
THEN H$="0"+H$ turns it into 03
```

Do not use H$=H$+"0", as this would give you 30 instead, which wouldn’t do at all!

Line 1060 makes sure there are only two numbers in each string.
Lines 1070 and 1080 check for the normal values of a twenty four hour clock, as regards hours, minutes and seconds, using logic statements.
Line 1090 avoids line 1100, if it is not required.
Line 1100 tells the user that a mistake has been, so try again.
Line 1200 puts together T1$ for the clock face for the start and reset times, returns

127
to the main program and sends the computer to line 30, so avoiding setting the clock again to zero, as it does on line 20.

Here's the full solution:

```
2 PRINTCHR$(147): PRINT”DO YOU WISH TO RESET THE CLOCK?”
3 GET K$: IF K$=”” THEN 3
4 IF K$=’Y’ THEN GOSUB 1000: GOTO 30
5 IF K$=’N’ THEN 10
6 IF K$<>”” THEN 3
10 PRINTCHR$(147)
20 TI$=”000000”
30 PRINTCHR$(147): FOR R=1 TO 11: PRINT: NEXT
40 PRINTTAB(12)”TIME ELAPSED IS;”
50 PRINT: PRINTTAB(16) LEFT$(TI$, 2);”;MID$(TI$, 3, 2);”;”:; RIGHT$(TI$, 2)
60 GET K$: IF K$=”S” THEN 120
70 IF K$=”” THEN 90
80 IF K$<>”” THEN 90
90 S$=RIGHT$(TI$, 2)
100 IF RIGHT$(TI$, 2)=S$ THEN 100
110 GOTO 30
120 PRINT: PRINTTAB(11)”TO RESTART, PRESS R”
130 GET K$: IF K$=”” THEN 130
140 IF K$=”R” THEN 2
150 IF K$<>”” THEN 130
160 END
1000 REM RESET ROUTINE
1010 PRINTCHR$(147): PRINT”USING 24 HOUR CLOCK”
11020 PRINT: PRINT”TYPE IN YOUR TIME, HOURS ONLY FIRST”
1030 PRINT: INPUT H$: IF LEN(H$)=1 THEN H$=’0’+H$
1040 PRINT: PRINT”MINUTES NOW”: PRINT: INPUT M$:
   IF LEN(M$)=1 THEN M$=’0’+M$
1050 PRINT: PRINT”NOW THE SECONDS”: PRINT: INPUT S$:
   IF LEN(S$)=1 THEN S$=’0’+S$
1060 H$=LEFT$(H$, 2): M$=LEFT$(M$, 2): S$=LEFT$(S$, 2)
1070 IF VAL(H$)>23 OR VAL(M$)>59 OR VAL(S$)>59 THEN 1100
1080 IF VAL(H$)=0 AND VAL(M$)=0 AND VAL(S$)=0 THEN 1100
1090 GOTO 1200
1100 PRINT: PRINT”WRONG, TRY AGAIN”: FOR D=1 TO 600: NEXT: GOTO 1010
1200 TI$=H$+M$+S$: RETURN
```

~~~~~~~~~~~~~~~

128
**correction to problem twenty six**

Line 130 in the solution to this problem, is:

```plaintext
130 GET K$: IF K$ = "Y" THEN 10
```

If the computer goes to line 10 after a Y answer, S% will be made zero again, so the score, S%, never increments.

So either make line 10:

```plaintext
10 PRINTCHR$(147)
```

and put on line 5:

```plaintext
5 B = 81: S% = 0
```

or make line 130:

```plaintext
130 GET K$: IF K$ = "Y" THEN 20
```

and make line 20:

```plaintext
20 PRINTCHR$(147): T = INT(RND(1)*4) + 1082: POKE T + 54272, 0: POKE T.65
```

**correction to problem forty two**

This is the electrician's problem. V = IR. After line 330, the computer goes directly into line 1000, the start of a subroutine, and then because of this, in line 1020, has nowhere to RETURN to!

So add line 340:

```plaintext
340 END
```

~~~
Well we are now at the END of this trip into problem solving land with the Commodore 64 computer.

You may now have a better insight into BASIC programming, but I do hope that you have a better understanding of logical problem solving. And, you have seen that most problems can be solved, if approached in a logical way, as long as you are also armed with the knowledge of where to go to get some help!

Brian Boyde-Shaw

If so, you can either continue to bemused, or let Brian Boyde-Shaw take you through a series of graded problems – each concerned with making words, numbers, sounds and pictures on your Commodore 64.

As you solve each problem (and - yes - solutions are provided) you learn a little bit more about BASIC. Slowly, you build up your programming expertise by success after success in solving these problems.

By the time you’ve solved the first few problems you’ll be confident that you can write programs that will:

- Display text on the screen, just as you want it to look.
- Carry out any type of calculation
- Play simple games.

After that, the sky’s the limit: you’ll be able to compose tunes, draw pictures, and really make your 64 do exactly as you wish.

By solving these problems - whilst learning BASIC - you will also learn to tackle all problems in a logical fashion. Success in solving a problem is used to reinforce what is already known so that the next problem is solved more easily.

So, if you’ve tired of playing games, here is a well-proven way to learn programming, and still have fun!

Brian Boyde-Shaw is an experienced teacher, and runs the "Byte Microcomputer Centre". He is writing several books for Sigma, including a companion to this book "The Acorn Electron Puzzle Book".

For a catalogue of all of our books, write to:

Sigma Technical Press
5 Alton Road
Wilmmslow
Cheshire
SK9 5DY

Price £6.95

ISBN 0 905104 78 1