TEXAS INSTRUMENTS
HOME COMPUTER

STARTER PACK 2

CASSETTE SOFTWARE
WITH MANUAL

An integrated pack containing a series of programs on cassette that develop and graphically display major ideas covered in the accompanying book. Enables any user to progressively understand and make full use of this computer.

COLLINS
MICROSOFTWARE

TEXAS INSTRUMENTS
HOME COMPUTER

Starter Pack 2
PK McBride

COLLINS
MICROSOFTWARE

Contents

The Programs

CHARDEF – helps you to define your own graphics characters. (chapter 1)

PATTERN – design your own wallpaper! This shows some of the multi-coloured screens that you can produce with the 99. (chapter 3)

SOUNDS – turns the 99's keyboard into an electric organ (with built-in chords), and a morse-buzzer-for-people-who-don't-know-morse. The program includes a selection of sound effects. (chapters 6 and 8)

ARRAYS – demonstrates some of the possible uses of arrays. (chapter 7)

HANGMAN – a string-slicing game that will also test your knowledge of BASIC.
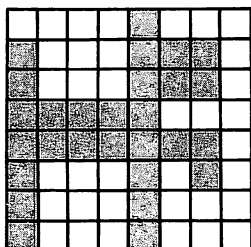
1
Creating your own characters

The characters from 128 to 159 in the 99 have been left blank so that you can use them for your own sets of special graphics.

Defining graphics characters is fiddly, but not particularly difficult. Let us have a look first at how the 99 handles characters.

Each character is made up of a set of points, 8 rows by 8 columns. Each of these points is either 'ink' or 'paper', and each row of points is seen by the 99 as one BINARY NUMBER, that is, as a set of 0's and 1's.

The character on the screen   The character as Binary Numbers

0	0	0	0	0	1	0	0
0	1	0	0	0	1	1	1
0	1	0	0	0	1	1	1
0	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1
0	1	0	0	0	1	0	1
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0

Figure 1

Note: Binary numbers

(Skip this if you understand them already.)
In normal (decimal) counting we work in base 10. This means we group numbers in 10's. 10 units make 1 ten. 10 tens make 1 hundred, and so on. Look at the number 1984.

$$1984 = \begin{aligned} & 1000 = 1 \times 10 \times 10 \times 10 \\ + & \ 900 = 9 \times 10 \times 10 \\ + & \ \ 80 = 8 \times 10 \\ + & \ \ \ \ 4 = 4 \text{ units.} \end{aligned}$$

The value of a number depends upon which column it is in.

In BINARY counting numbers are grouped in two's. 2 units make 1 two. 2 twos make 1 four, etc. This means, in practice that binary numbers are made up of 1's and 0's only. This makes them very easy for computers to handle. 1 and 0 can be translated into electric currents – 1 for on, 0 for off.

Binary									Decimal	Hexa-
8's	4's	2's	1's							decimal
0	0	0	1	=				$1\times1=$	1	1
0	0	1	0	=			1×2	$+0=$	2	2
0	0	1	1	=			$1\times2+1\times1=$		3	3
0	1	0	0	=		1×4	$+0$	$+0=$	4	4
0	1	0	1	=		1×4	$+0+1\times1=$		5	5
0	1	1	0	=		$1\times4+1\times2$	$+0=$		6	6
0	1	1	1	=		$1\times4+1\times2+1\times1=$			7	7
1	0	0	0	=	1×8	$+0$	$+0$	$+0=$	8	8
1	0	0	1	=	1×8	$+0$	$+0+1\times1=$		9	9
1	0	1	0	=	1×8	$+0+1\times2$	$+0=$		10	A
1	0	1	1	=	1×8	$+0+1\times2+1\times1=$			11	B
1	1	0	0	=	$1\times8+1\times4$	$+0$	$+0=$		12	C
1	1	0	1	=	$1\times8+1\times4$	$+0+1\times1=$			13	D
1	1	1	0	=	$1\times8+1\times4+1\times2$	$+0=$			14	E
1	1	1	1	=	$1\times8+1\times4+1\times2+1\times1=$				15	F

Figure 2

Got the hang of Binary numbers? Good, because now it's time for . . .
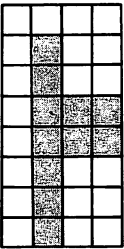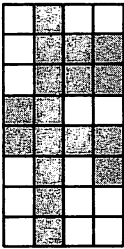
Hexadecimal
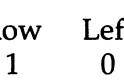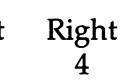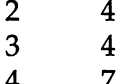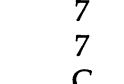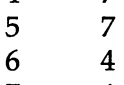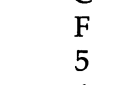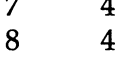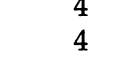
This time we are counting in Base 16. 16 units make one group (of 16). Because we only have 10 figures for numbers, we have to use letters for units over 10. See the right hand column in figure 2.

You have to use Hex for defining characters, but don't worry too much, you only have to use the first 16 numbers (0 to F). It is, when you are used to it, much easier, and quicker, to use Hex than it is to use binary.

Back to defining characters

Start by working out your character on squared paper. When you are happy with your design draw a line through the middle to divide it into left and right sides. Now take the rows one at a time from the top down, left side first, then right. Think of each coloured square as being the 1 of a binary number, and work out the value in Hex.

Here is how the rows worked out on the dog.

Hex	Decimal	Left	Right	Decimal	Hexa-decimal
0 =	0 =	0000 =	= 0100 =	4 =	4
4 =	4 =	0100 =	= 0111 =	7 =	7
4 =	4 =	0100 =	= 0111 =	7 =	7
7 =	7 =	0111 =	= 1100 =	12 =	C
7 =	7 =	0111 =	= 1111 =	15 =	F
4 =	4 =	0100 =	= 0101 =	5 =	5
4 =	4 =	0100 =	= 0100 =	4 =	4
4 =	4 =	0100 =	= 0100 =	4 =	4

Row	Left	Right	Together
1	0	4	04
2	4	7	47
3	4	7	47
4	7	C	7C
5	7	F	7F
6	4	5	45
7	4	4	44
8	4	4	44

Figure 3

That's the worst of it done. It's all downhill from here.

There is a built-in routine that does the actual definition. All you have to do is state which character number you are

going to redefine, and feed in those 8 pairs of hexadecimal numbers. All that is done in one line.

```
CALL CHAR(128,"0447477C7F454444")
```

When you are doing this check carefully that the character number you use is one that can be defined – i.e. between 128 and 159.

Check also that you have the right number of figures and letters in your hexadecimal number (there should be 16; Left-hand side 0's must be included); and that you have put quotes around the Hex string.

Figure 4 may help you sort out your Hex numbers.
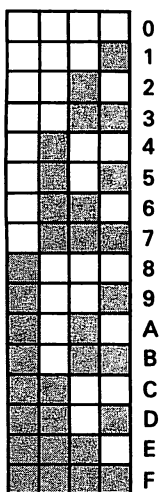
Figure 4

If, having read this far, you are not too happy with the thought of converting dots to binary to hexadecimal, then load the program CHARDEF. There you can work out your characters dot by dot and the 99 will tell you what strings of Hex numbers you need to define each character. Make a careful note of the numbers, and when you have done all the characters you will need, NEW that program and start on your own.

Note

You can, if you like, redefine every character from 32 to 127 as well, but there is a difference. These characters all go back to their normal state whenever the program is not running. This means that they are reset at every break point, and that you cannot see them in your program lines as you are typing in. The other characters (128 to 159) can all be called up from the keyboard using the CTRL key, as you may have seen in Starter Pack 1. (In case you haven't got Starter Pack 1, you will find a keyboard plan in Appendix C which shows you where these 32 characters are located.)

If you redefine the characters used in BASIC words, it makes not the slightest difference to the 99. Type this three-liner in and see. It redefines R, used in PRINT and in CALL CHAR.

```
10 CALL CHAR(82,"FFFFFFFFFFFFFFFF")
20 PRINT "CHARACTERS"
30 GOTO 30      (to hold the screen while the program's
                running).
```

All of which goes to show that the 99 can look after itself despite all that we humans might do to try and confuse it.

9
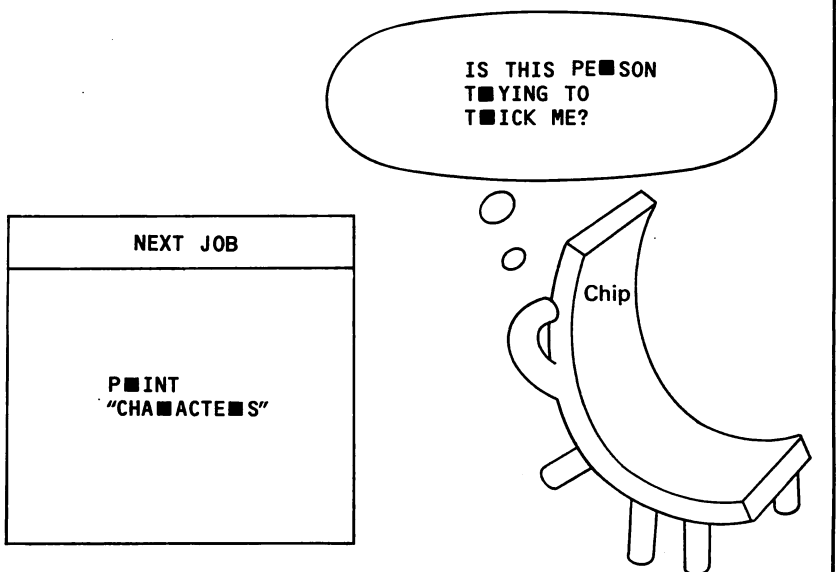
2
Program planning 1
-A bit at a time

As your programs become more complicated, so you will
need to spend more time working on and checking over the
different routines within the program. There are several
ways in which you can take your program a bit at a time, and
here they are.

The latest addition

You have a fairly lengthy program and have just added some
more onto the end. You want to look at how the latest
addition works, but don't want to wait ten minutes while the
program runs through to that point. You do not have to RUN
the whole program. You can tell the 99 which line to start to
RUN from. Type this in:

```
10  PRINT "HELLO"
20  PRINT "GOODBYE"
```

You know that if you now type RUN it will print "HELLO"
and then "GOODBYE". Type RUN 20 and see what
happens. If that had been line 20000, coming after 19999
other lines, the effect would have been more marked, but the
principle is the same. You can start to RUN at any line.
Watch out for these points though. If there is anything in the
latest routine that depends on earlier lines, it won't work.
You can't start in the middle of a FOR. . .NEXT. . . loop. If
graphics characters are defined in the sections that you have
jumped over, then they will be undefined in the part that
you run. You will also have problems if you are using
variables in the new section if these are supposed to have
numbers already stored in them from earlier. All your stores
are cleared on each RUN command, so that the numbers are

all 0 and string stores all "".

Watch your line numbers as well. Try RUN 15 on that little program and see what happens.
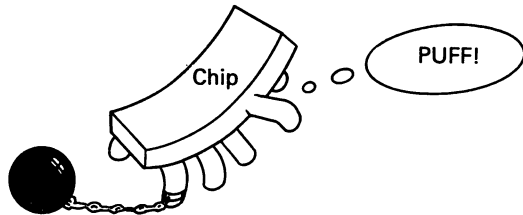
Slow down!

You are developing a fast-moving program and need to slow it down for checking. There are two simple ways to do this, as you have probably already discovered.

Put in a (temporary) delay loop.

```
1002 FOR D= 1 TO 500
1003 NEXT D
```

Or delay it with some sounds. Remember that one sound alone will slow nothing down. With a pair of sounds though, the program will wait for the second sound before it moves on.

```
CALL SOUND (1000,-1,1)
CALL SOUND (1,-1,1)
```

Here the second sound is the briefest possible blip, but it makes sure that the program waits while the loud beep sounds. That beep will also serve to draw your attention.

When the program is running as it should (apart from the delays) then you can knock out the sounds or the loop lines.

Hold it!

Hold the program at the end of a section by writing in a CALL KEY routine.

```
CALL KEY(3,K,S)   IF S=0 THEN....
```
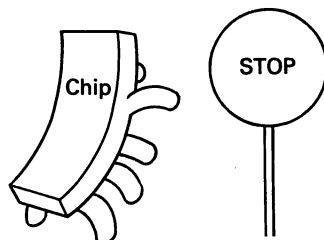
You can add a few extra lines to this to give yourself some options. You might want to run that section again, or go back to the start, or carry on, or simply stop so that you can rewrite lines. The following lines would do it:

```
IF K=82 THEN.....   (82 = "R" – Run this section
                     again)
IF K=66 THEN...     (66 = "B" – Back to the
                     Beginning)
IF K= 67 THEN....   (67 = "C" – Carry on)
STOP
```

You don't need to bother to write in any print lines as these lines are only for your own temporary use.

There is another way to hold the program, and it needs even less typing.

```
INPUT Z$
```

The computer now waits for an input. By pressing ENTER you can move it on. You could also include a set of check and redirect lines here, as with the CALL KEY routine. N.B. make sure that the store you use on this INPUT line is not already in use for something else. Notice also that a string store is used. This allows you to ENTER nothing. If you use a number store, you will have to ENTER a number if you want to avoid a * WARNING: INPUT ERROR report.

Take a break

You know already that you can break into the program whenever you want by using FCTN and ４. It is all too easy

though to miss the exact point that you wanted to break at. Here is the way to plan your breaks exactly. You can set your breakpoints before you run the program by typing in:

BREAK 150 (or whatever line number)

Run the program now and it will stop with a * BREAKPOINT AT 150 report. To carry on, type in:

CONTINUE (or CON which means the same to the 99)

and the program will continue from that point.

CONTINUE will restart the program after any BREAKPOINT, even one that has been produced by FCTN and ⁴, as long as you haven't changed the program in between. You can LIST (or perform any other command) at a breakpoint, and still Continue afterwards.

If there are a number of places that you would like to stop the program at, then type in the line numbers of all the BREAKPOINTS at the beginning.

BREAK 150,300,590

This will stop the program at each of those three places.

The BREAK command only lasts for the first run through the program. Run it a second time and it will go through without stopping.

If you have written a lot of breakpoints into the program at the beginning, and then decide half way through the run that you no longer need them, then rub them out with :

UNBREAK

This clears all the breakpoint markers. You can remove particular breakpoints by telling the 99 which ones you don't want.

UNBREAK 300
UNBREAK 300,590

Setting breakpoints in this way is much simpler than writing in Slow Down, or Hold it lines, but has the disadvantage that it prints on screen – and may therefore ruin a nice screen display.

3
Colour

Magic painting

Have you seen those children's painting books where you
simply paint with water and the colours appear, as if by
magic? You can produce the same kind of effect on the T.V.
screen by colouring characters that have already been
printed up in invisible ink. You will probably have seen this
effect on the Master Title screen in the KEYS program in Pack
1. Colour changing is much quicker than printing, and so is
very useful for giving the impression of movement, or for
producing (more or less) instant pictures.
   You can see this at work if you type this in:

```
10 CALL CLEAR
20 CALL COLOR(6,1,1)    (so set 6 letters are
                         transparent)
30 PRINT "HI"           (both "H" and "I" are in
                         set 6)
40 INPUT A$             (Hold it!)
50 CALL COLOR(6,2,1)    (to recolour them black)
60 INPUT A$             (Hold it!)
70 GO TO 20
```

Run this, and you won't know that the "HI" has been
printed until you press ENTER and move the program onto
line 50. ENTER again and send the program round the loop.
When the letters are reset to transparent, you can only tell
where they are by the odd gaps in the line of INPUT?'s that
go up the left hand side.
   LIST the program and remove line 10. Now run again and
you will see odd gaps scattered through the LIST as all the
set 6 letters become transparent. If we recolour all the letter

14

sets then we can make the whole lot disappear. Replace the single CALL COLOR lines with loops:

```
15 FOR S=5 TO 8
20 CALL COLOR(S,1,1)
25 NEXT S
```

Run it now and see how your program lines disappear, and reappear. The numbers and symbols are still around of course, but you can include these in the disappearing act by changing the Set range to: FOR S=2 TO 8.

Now knock out your INPUT A$ lines, and RESEQUENCE (just to tidy things up). You should be left with something like this.

```
10 FOR S= 2 TO 8
20 CALL COLOR(S,1,1)
30 NEXT S
40 PRINT "HI"
50 FOR S= 2 TO 8
60 CALL COLOR(S,2,1)
70 NEXT S
80 GO TO 10
```

LIST again (just to make sure there are plenty of characters on screen) and run. As the program runs, so the list will be gradually pushed up the screen by an increasing line of "HI"s. You will notice that where several different sets are involved, it takes a little time to get round and recolour them all – hence the flickering effect. However when the letters on screen are all of the same set (keep running until the list has worked its way right off the top) you get a nice steady flash.

If you particularly want a flashing message, it may be worth defining characters from one set into the letters you need, and then recolour that single set. In the example below characters 128, 129 and 130 are defined as L, O and K, so that a flashing LOOK can be printed on a steady screen.

```
10  CALL CHAR(128,"004040404040407E")    (L)
20  CALL CHAR(130,007C44444444447C)
30  CALL CHAR(130,"0044485060504844")    (K)
40  CALL CLEAR
50  PRINT "LOOK HERE"    (normal capitals)
60  PRINT "LOOK"    (L – CTRL and ⌐ O – CTRL and
                     Ⓐ K – CTRL and Ⓑ
70  CALL COLOR(13,1,1)
80  CALL COLOR(13,2,1)
90  GO TO 70
```

Notice how the CTRL key is used to get to the defined characters for that second "LOOK". You may well find it useful to run the program as soon as you have typed in the first three lines. That way the characters will be there, ready for use.

You will find a number of variations on this theme in the color section of the EFFECTS program in Pack 1. The RIPPLE effect (at 1000 –) works by defining two pairs of characters the same but in different sets. See figure 5. The are all printed on the screen at the start of the routine and remain there throughout, but by switching from red to invisible in turns, they give the illusion of movement.
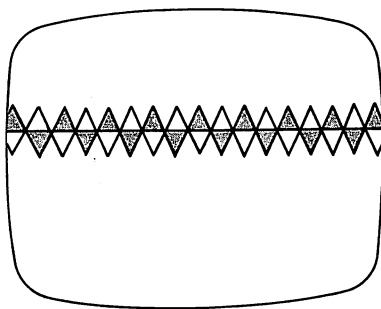
Figure 5

Here is a simple version of the routine:

```
 10  CALL CHAR(128,"FFFFFFFFFFFFFFFF")  (defines
 20  CALL CHAR(136,"FFFFFFFFFFFFFFFF")  two
                                         blocks)
 30  CALL CLEAR
 40  FOR C=1 TO 31 STEP 2
 50  CALL HCHAR(10,C,128)         (prints a line
 60  CALL HCHAR(10,C+1,136)       of blocks)
 70  NEXT C
 80  CALL COLOR(13,8,1)      (set 13 now red,
 90  CALL COLOR(14,1,1)      14 transparent)
100  CALL COLOR(13,1,1)      (and swaps them
110  CALL COLOR(14,8,1)      round)
120  GO TO 80
```

You will find that this all whizzes through rather quickly, which is why there are sounds included in the RIPPLE routine in the program.

The RUNNING LIGHTS routine on the EFFECTS program is produced by a similar routine, although there, three colours, and three sets of characters are used. This helps to give a greater sense of direction to the movement.

Figure 6

17
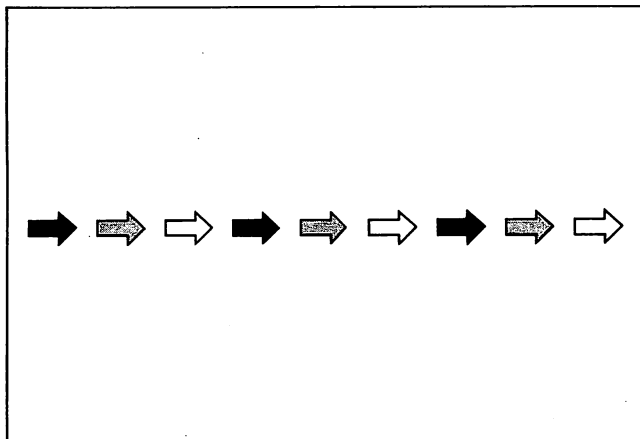
Where you have a lot of colour changes to make, and these changes follow a set pattern, it may be worth using a switching routine, rather than have dozens of CALL COLOR lines. In the example below, the program colours sets 5, 6 and 7, blue (5), red (9) and white (16), then changes blue to red, red to white and white to blue.

```
 10   PRINT "BITBITBITBITBIT"      (B, set 5, I set 6,
                                    T set 7)
 20   COL1=5     (blue)
 30   COL2=9     (red)      3 stores for color codes.
 40   COL3=16    (white)
 50   CALL COLOR(5,COL1,1)
 60   CALL COLOR(6,COL2,1)
 70   CALL COLOR(7,COL3,1)
 80   COLX=COL1
 90   COL1=COL2
100   COL2=COL3
110   COL3=COLX
120   GOTO 50
```

And here's what happens with that switching routine in lines 80 to 110.

A fourth store is brought into play (COLX). This takes the value of the first store (blue). The first store then changes the value of the second store (red), and this takes the number from the third store (white). The third store then collects blue (5) from the temporary store (COLX). Figure 6 shows this in operation

COL1
5

COLX
0

COL2
9
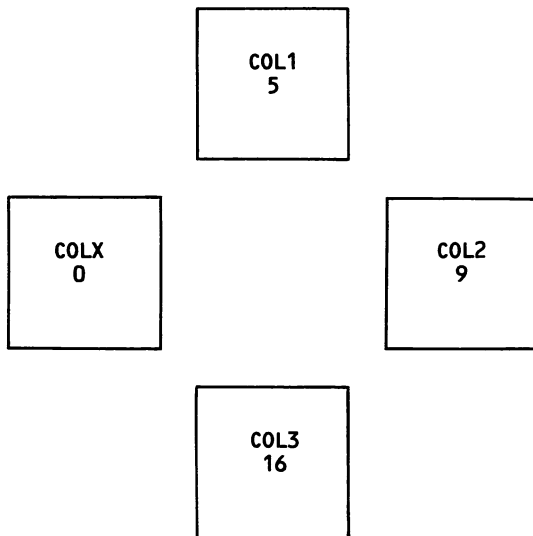
COL3
16

Figure 6A

COL1
5
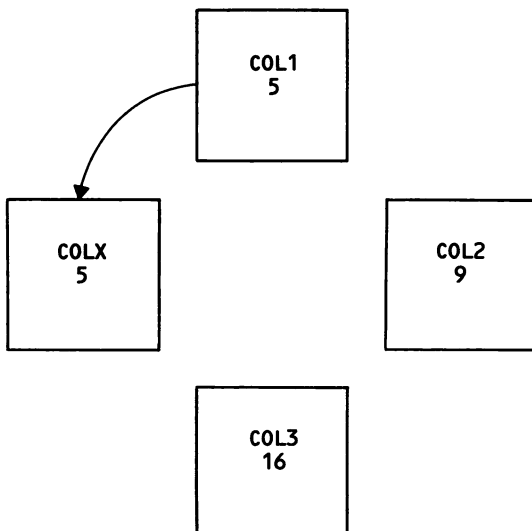
COLX
5

COL2
9
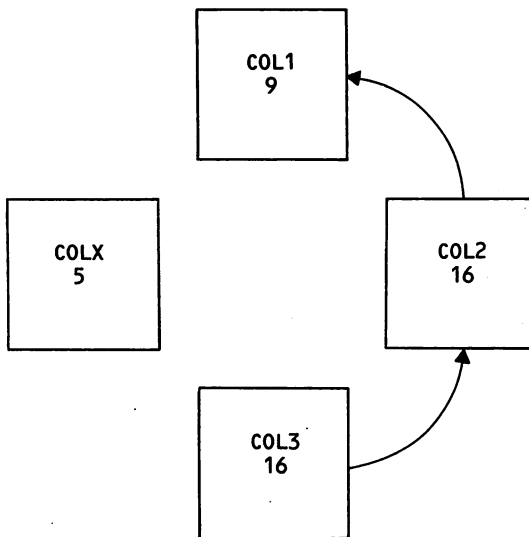
COL3
16

Figure 6B

Figure 6C

COL3 needs to collect 5 from COLX to complete the switch.

After a second run through the switches, COL1 is 16 (white), COL2 is 5 (blue) and COL3 is 9 (red).

Colour blocks

You can turn any character into a solid block of colour by making the foreground and background colour the same. You can then turn these into larger blocks, or solid lines using the HCHAR or VCHAR statements.

This program produces a large block of red in the middle of the screen.

```
10  CALL CLEAR
20  CALL COLOUR(13,9,9)
30  FOR R= 6 TO 15
40  CALL HCHAR(R,12,128,10)      (10 lines of 10
                                  characters)
50  NEXT R
60  INPUT A     (Hold it!)
```

The character used in line 40 (128) could have been any one
from set 13. They all give the same red on red block.

You can add to this program to give other blocks of
different colours. Here a block of green is added in the
bottom right of the screen.

```
60  CALL COLOR(14,3,3)
70  FOR R= 18 TO 21
80  CALL HCHAR(R,24,136,6)      (4 lines of 6
                                 characters)
90  NEXT R
100 INPUT A
```

In the last example HCHAR and VCHAR are combined to
produce a coloured border. Notice the STEPs in lines 30 and
60. These make sure that it is only the edges which are
coloured. Miss them out and see what happens.

```
10 CALL CLEAR
20 CALL COLOR(13,9,9)
30 FOR R= 1 TO 23 STEP 22      (leave line 24
                                clear for the
                                INPUT)
40 CALL HCHAR(R,1,128,32)
50 NEXT R
60 FOR C=1 TO 32 STEP 31
70 CALL VCHAR(1,C,128,23)
80 NEXT C
90 INPUT A      (hold it!)
```

Multi-coloured mosaics

In the PATTERN program you will see that the same shape
can be printed on screen in different colours at the same
time. There is no special trick to this. It is an extension of the
idea used in the RIPPLE effect covered earlier. There are
16 different shapes of tile, and each one has been defined
4 times, so that there are 4 full selections of the same tiles.
Each selection takes up a pair of colour sets. It is possible to
take this idea further and define characters over a wider

range of colour sets to give yourself even more colours on screen. That is, however, probably worth leaving until later, as there are a number of routines that can make complicated programming much simpler.

Meanwhile, why not load up some of your old programs and improve their presentation with colour blocks and flashing messages. When you have loaded in a program RESEQUENCE it to give yourself room to add lines. If you resequence in 20's, or even 100's, it will give you lots of space. The only thing you need to watch out for when resequencing is that the final number is no more than 32767. Even if you space out in 100's, you would need a program of more than 320 lines to reach this!

Patterned backgrounds

You can produce a simply-patterned screen with just one line:

```
100  CALL HCHAR (1,1,42,768)
```

This will print up 768 asterisks starting at the top left and working across and down. VCHAR would give the same result, but printing down and across. This may be useful as a background to a screen printed with HCHAR commands (see 'Strings'), or as a dividing screen between sections of a program.

You could start your pattern lower down the screen, beneath normally PRINTED material. Don't worry about getting the number of repititions exactly right. The 99 will not be fussed if you ask it to print too many.

4
Program planning 2
-Subroutines

A subroutine is a part of a program that you can re-use as often as you like. The beauty of it is that whenever you send the computer off to a subroutine it always comes back to the line it was on before, without being told which line number.

Send the 99 to a subroutine with the instruction:

GOSUB.....   (whatever line number)

and send it back to the program with:

RETURN   (no line number)

Type this is and listen:

```
10  PRINT "HELLO"
20  GOSUB 100
30  PRINT "HOW ARE YOU?"
40  GOSUB 100
50  PRINT "GOODBYE"
60  GOSUB 100
70  STOP
100 CALL SOUND(500,-1,1)
110 CALL SOUND(500,-8,1)
120 RETURN
```

SUBROUTINE

When Chip comes to a GOSUB he makes a note of where he is:

NEXT JOB

20 GOSUB

WHERE AM I?

Chip
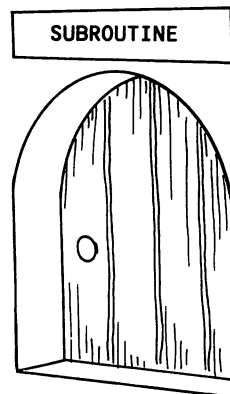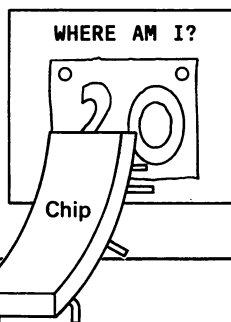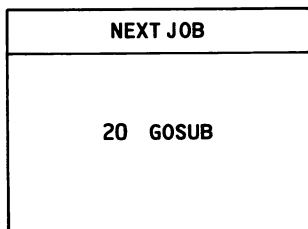
So that when he RETURNS, he knows where to go:
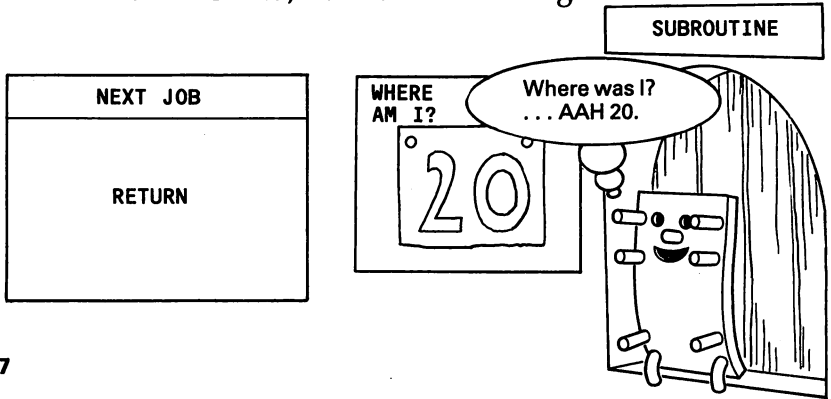
Figure 7

The RETURN line numbers are stored in a special place known as the GOSUB stack. There is room for more than one number here, which means that you can go to a subroutine from inside another one. Add these lines to the program given above:

```
115 GOSUB 200
200 CALL KEY(3,K,S)
210 IF S= 0 THEN 200
220 RETURN
```

Now, before the computer returns from its silly sound subroutine, it will go off to the second routine to wait for a key contact. After a touch, it returns to the end of the silly sounds, and then goes back to wherever it was in the print lines.

The line numbers are stored on the GOSUB stack on the last-on-first-off principle. You could think of it as a paper spike.
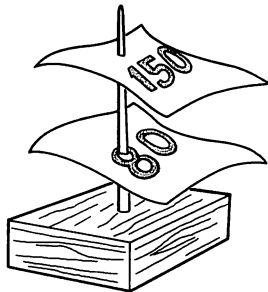
Figure 8

Here the program will return first to line 150, and when it reaches the next RETURN instruction, it will go back to 80.

NOTE: You can use an ON. . .GOSUB line in the same way as ON. . .GOTO.

You can follow the movements of the computer through a program by using the TRACE command. Type in TRACE now (no line number), and then run the program again. You should get something like this:

```
<10> HELLO
<20> <100> <110> <115> <200>   (silly sounds and
                                 GOSUB 200)
<210> <200> <210> <200> <210>  (whizzing round the
                                 CALL KEY loop)
<200> <210> <220> <120> <30>   (back to 120, then to 30)
HOW ARE YOU?
<40> <100> <110> <115> <200>
<210> <200> <210> <200> <210>
<220> <120> <50>               (back to 120 then 50)
GOODBYE
<60> <100> <110> <115> <200>
<210> <200> <210> <200> <210>
<200> <210> <220> <120> <70>   (back to 120 then 70)
** DONE **
```

Figure 9

You will see numbers printed in brackets. These are the numbers of the lines that the program is going through at that moment. If you ever have any doubts about whether or not a program is going through its routines in the proper order, you will find TRACE invaluable.

NOTE: Once you have entered a TRACE command, the computer will trace on every run until you cancel it with an UNTRACE command.

5
Teach your 99 to READ

There will be many times when you will want the computer
to perform a series of routines that are almost, but not quite,
the same. You might be playing a tune, so that the lines all
start "CALL SOUND. . ." but then have some different
numbers in the brackets. You might be printing out a
standard message to several people. You might be defining
characters. Let's start by reading some names.

Here you want the 99 to print a friendly "HELLO" to all
your friends. Your program might look like this:

```
10  PRINT "HELLO BILL"
20  PRINT "HELLO SUSAN"
30  PRINT "HELLO FRED"
40  PRINT "HELLO ALISON"
50  PRINT "HELLO SANDY"
 . . . . . .
 . . . . . .
```

You finish up with as many lines as you have friends. You
could also do it this way:

```
10  INPUT N$
20  PRINT "HELLO" ;N$
30  GO TO 10
```

but then you would have to sit there inputting all the names.

Here's a better way:

```
10  READ N$
20  PRINT "HELLO" ;N$
30  GO TO 10
40  DATA BILL,SUSAN,FRED,ALISON,
SANDY, . . . . .
```

When you tell the computer to READ it goes off in search of a line marked DATA and picks up the first thing it finds there. "BILL" goes into the N$ store, and is printed by line 20. When the computer goes back to READ for a second time it picks up the next name in the DATA list and puts that in the N$ store. It prints "HELLO" to SUSAN and goes round the loop, each time picking up the next name. When it has run out of new words it will stop with a * DATA ERROR IN 10 report.

In figure 10 you can see Chip READing names from the DATA list. Notice the little marker beside the names. This is to let him know where he should start to READ next time.
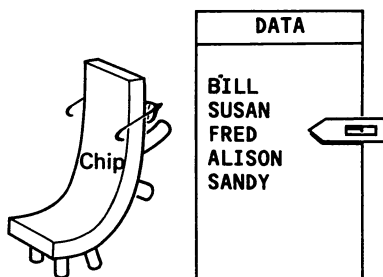
Figure 10

Here's a close-up of the list showing the marker. FRED gets READ next time.
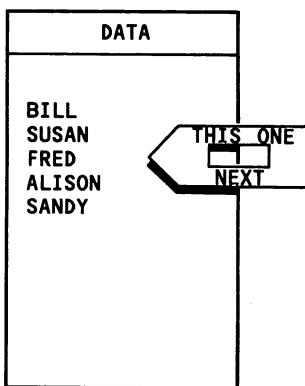
Figure 11

When you RUN the program the DATA marker is pushed back up to the top of the list. Write a program of your own to

say the same thing to a set of people. It might be "Welcome to the party", or "Hello"; (whatever your name is); "how are you." The DATA list can be as long as you like, and it can be split up over as many lines as you like:

```
40 DATA BILL,SUSAN,FRED,ALISON,
   SANDY,GEORGE,KATE
50 DATA MARGARET,TONY,MICHAEL,DAVID,
   ROY......
60 DATA WILLIAM SHAKESPEARE,R.REAGAN
```

Your DATA lines cannot be more than 4 screen lines long, of course, and you will find it much more convenient to keep them short. This makes it easier to correct typing errors, or to add, or take away from the list.

Make sure that the names are all separated by commas. You can, if you like, enclose the names in quotes, but this is not normally necessary. The only time you must put quotes round words in the DATA list is when you want to include a comma or quotes inside the group of words to be read.

```
70 DATA "MITTERAND,FRANCOIS",
   "P.K." "MAC" " MCBRIDE"
```
(note: double quotes needed to get one set printed)

When you run your program it will stop with a DATA ERROR report once it has run out of things to read. This could be very irritating if you wanted the program to carry on and do something else afterwards. It is, however, an easy error to avoid. Replace your simple GO TO loop with a FOR. . .NEXT. . . loop that will make sure that the 99 only reads as many data items as are there.

```
 5 FOR T= 1 TO 5     (or however many)
10 READ N$
.....
30 NEXT T
```

Now let's squeeze some more lines in between the "HELLO" loop and the DATA. (You can always tidy up with RESEQUENCE afterwards.

```
32  FOR T=1 TO 5     (or however many)
34  READ N$
36  PRINT "GOODBYE";N$     (or whatever message)
38  NEXT T
```

Run this, and you will get another DATA ERROR report. All the DATA has been used up by the first loop, and you need to reset the marker back at the top of the DATA list. No problem! Add another line.

```
31  RESTORE
```

RESTORE pushes the marker to the top of the list. Try it and see. You can see the effect of RESTORE even more clearly if you put a RESTORE line inside the loop. Then the marker will keep going to the top and the same name will be printed five times.

How to read characters

You will find it very useful to apply data-reading routines to your character definitions. The more characters you want, the more you save in typing time and memory space. Let's suppose that you wanted to define the following four characters – a solid block, a triangle, a space, and an empty 'box'. You are going to allocate them to character numbers 128, 129, 130, 131.
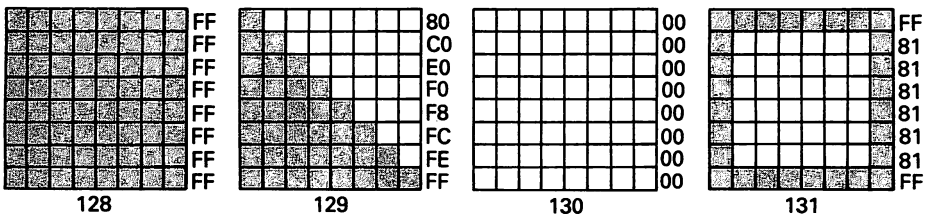
Figure 12

These are the hex strings you need to define them.
block (128) "FFFFFFFFFFFFFFFF"
triangle (129) "80C0E0F0F8FCFEFF"
space (130) "0000000000000000"
box (131) "FF818181818181FF"

Note that the hex numbers are actually strings of characters, so must be read into a string store.

This is the routine that defines them

```
10 FOR N=128 TO 131
20 READ G$      (G for Graphics)
30 CALL CHAR(N,G$)
40 NEXT N
50 DATA FFFFFFFFFFFFFFFF
60 DATA 80C0E0F0F8FCFEFF
70 DATA 0000000000000000
80 DATA FF818181818181FF
```

On the first run through the loop line 30 is, in effect, this:

```
CALL CHAR(128,"FFFFFFFFFFFFFFFF")
```

and similarly for later runs through.

Notice how each character has its own DATA line. This is a little wasteful of memory space, but it makes alterations so easy. Saving memory space does not really matter unless you intend to write extremely long programs, as 16k goes a long way. However, being able to alter your character definitions easily is very important, as it usually takes a lot of practice before you get your graphics looking right on the first attempt.

If you wanted to define a number of characters and their numbers did not run in sequence, then you could include the character number in the read and data lines.

```
10 FOR N= 1 TO 4
20 READ C,G$      (Character number, Graphic string)
30 CALL CHAR(C,G$)
40 NEXT N
50 DATA 128, FFFFFFFFFFFFFFFF
60 DATA 136, 80C0E0F0F8FCFEFF
```
. . . .

If you are mixing number and string stores in this way, then take even more care than usual when typing in your data lines.

6
Reading music

Did you know your 99 could read music? Not the little black dots on the lines type of music, but music as a computer understands it, and that means numbers.

Try this:

```
10 FOR T=1 TO 4
20 READ P     (Pitch)
30 CALL SOUND(500,P,1)
40 NEXT T
50 DATA 262,330,392,523
```

Isn't that easier than writing 4 separate CALL SOUND lines? When you realise that you can have as much data (and therefore as many pitches) as you like, you may begin to see some of the potential of READ instructions for music making.

'What about chords?' you say. No problem. The 99 can read several different things at once. The program below will read the three notes needed for each of four different chords.

```
10 FOR CHORD = 1 TO 4
20 READ N1,N2,N3     (3 notes)
30 CALL SOUND(1000,N1,1,N2,1,N3,1)
40 NEXT CHORD
```

Now let's work out the data that we need to make it work. The four chords that we are going to play are C, F, G and a second (different) C chord.

Chord	First note		Second note		Third note	
	letter	frequency	letter	frequency	letter	frequency
C	C	262	E	330	G	392
F	F	349	A	440	C	523
G	G	392	B	494	D	587
C	E	330	G	392	C	523

To get our data line out of this, we simply read across one
line at a time. It is probably worth typing each chord in as a
separate line. It will make editing easier in case of mistakes.

```
50 DATA 262,330,392    (C)
60 DATA 349,440,523    (F)
70 DATA 392,494,587    (G)
80 DATA 330,392,523    (2nd C)
```

So far, the time of our notes have lasted, and their volume
have remained the same for all the notes, but these could just
as easily be included in the data. The next example produces
a loud top C, and a longer, but quieter middle C.

```
10 FOR N= I TO 2    (2 Notes)
20 READ T,P,V    (Time, Pitch, Volume)
30 CALL SOUND(T,P,V)
40 NEXT N
50 DATA 500,523,1,    (short, top C, loud)
60 DATA 2000,262,10    (long, middle C, quiet)
```

If you have the time and the inclination, you can get the 99 to
play whole (single note) tunes, using that simple four line
routine, and lots of data. It will be easier on your first efforts,
to fix the volume in the CALL SOUND line, and simply read
Time and Pitch. The example below shows how the first line
of 'Girls and Boys come out to play' was converted into data
for a program. It was used for Violet's entrance in the
CHARLIES program in Pack 1, although there the rhythm of
the music was slightly affected by various character printing
routines.

Note length ♩ (crochet) = 600 } so each bar lasts
                ♪ (quaver) = 300 } about 2 seconds
                ♩. (dotted crochet) = 900

Figure 14

Music   99 coding

Number	Value	Note	Time	Pitch
1	♩	D	600	587
2	♪	B	300	494
3	♩	C	600	523
4	♪	A	300	440
5	♩	D	600	587
6	♪	B	300	494
7	♩	G	600	392
8	♪	G	300	392
9	♩	A	600	440
10	♪	B	300	494
11	♪	C	300	523
12	♪	B	300	494
13	♪	A	300	440
14	♩	D	600	587
15	♪	B	300	494
16	♩.	G	900	392

Figure 15

33

And so the DATA lines:

```
50 DATA  600,587,300,494,600,523,300,440
```
<div align="right">(first 4 notes)</div>

```
60 DATA  600,587,300,494,600,392,300,392
```
<div align="right">(next 4)</div>

```
70 DATA  600,440,300,494,300,523,300,494
```
<div align="right">(next 4)</div>

```
80 DATA  300,400,600,587,300,494,900,392
```
<div align="right">(next 4)</div>

For more complicated tunes, with three part harmony and rests (periods of silence), you would need to combine these routines, to give these lines:

```
20 READ TIME,P1,V1,P2,V2,P3,V3
30 CALL SOUND(TIME,P1,V1,P2,V2,P3,V3)
```

a DATA line for a 1 second chord would look like this:

```
..DATA  1000,262,1,330,1,392,1
```

which gives a C chord at volume 1.

If only 2 notes are needed, you would still neet to include a 'dummy note', so that you had a full set of numbers for the CALL SOUND line.

```
..DATA  500,294,1,440,1,-1,30
```

which gives half a second of D and A at volume 1, and a silent beep. It doesn't matter what that third sound it, and a beep only needs 2 characters to be typed in. You can't use 2 noises in a CALL SOUND line, so a single note for this routine would need this kind of DATA line:

```
DATA  1000,349,1,349,30,-1,30
```

so that here you have a silent note, and a silent beep.

Below you will find a table of note values, and the frequencies for the notes in the keys of C, G and F, which may be useful to you in your music making.

You will find a more complete list of frequencies in the User's Reference Guide, and if you want to explore further

into the joys of music on the 99, your Texas dealer should have the TI Music Maker in stock.

Note values

1 semi-breve	$\circ$	= 2 minims
1 minim		= 2 crochets
1 crochet		= 2 quavers
1 quaver		= 2 semi-quavers

A dotted note is half as long again

Timing	Slow	Fast
	2400	800
	1200	400
	600	200
	300	100

Figure 16

Note	Frequency
G	196
A	220
Bb	233
B	247
C	262
D	294
E	330
FF	349
F#	370
G	392
A	440
Bb	466
B	494
C	523
D	587
E	659
F	698
F#	740
G	784

Figure 17

```
KEY OF C   (all notes natural)   Main Chords
  C D E F G A B C                 C(CEG)  F(FAC)  G7(GBF)   Am(ACE)
KEY OF G   (one sharp F#)
  G A B C D E F# G                G(GBD)  C(CEG)  D7(DAC)   Em(EGB)
KEY OF F   (one flat Bb)
  F G A Bb C D E F                F(FAC)  Bb(BbDF)  C7(CEBb)  Dm(DFA)
```

If you load up the SOUNDS program, you will find that the
keyboard has been reprogrammed to make the number keys
play single notes, and some of the letter keys give chords.

    We will return to sounds later, and look at ways of
extending the organ keyboard on the 99, but first we need to
look at another programming technique.

7
ARRAYS

ARRAYS are special sorts of memory stores. Once you have got the hang of using them, you have at your fingertips a very powerful programming tool. Basically they are a means of collecting sets of information. Suppose you were doing a survey on the heights of your friends. You plan to measure them, type in their heights, and then have all the heights printed out at the end. You write this program:

```
10  INPUT "HEIGHT":H
20  IF H=0 THEN 40      (your escape from the loop)
30  GOTO 10
40  PRINT H
50  GOTO 40
```

You run the program and enter 150,134,168,145,0. The computer prints 0's all down the screen! Each new number has replaced the old one in the H store, so that 0 is there when it reaches line 40.

You need a separate store for every number, so you rewrite it:

```
10  INPUT "HEIGHT ":H1
20  INPUT "HEIGHT":H2
30  INPUT "HEIGHT ":H3
```
. . . .
. . . .   (how many people are you measuring?)
```
100  PRINT H1
110  PRINT H2
120  PRINT H3
```
. . . . . .
. . . . . .

Well, it works, but how much typing would you need if you

were surveying 30 people? You have no doubt noticed that you have a set of stores here that all start with H and end in a number – H1,H2,H3,. . . Why not use a loop?

```
10  FOR N=1 TO 5
20  INPUT "HEIGHT ":HN
30  NEXT N
40  FOR N = 1 TO 5
50  PRINT HN
60  NEXT N
```

You're getting warmer, but this still won't work. HN is the name of a single store. Put that N in brackets, and you are just about there. Now when the 99 sees H(N) it will check back to the loop to find the value of N, and store the heights in H(1), H(2) and H(3), etc. You have made an array.

Actually, you haven't made the array, the 99 did it for you. When it sees a variable name with a number in brackets after it, the 99 assumes you want an array, and sets up a little one for you. This array stops at H(10). If you want any more than that in your set of stores, you have to organise it yourself. Do this by telling the computer the DIMENSIONS of your stores before you try to use them.

```
5  DIM H(20)
```

You have now asked the computer to set up a ONE-DIMENSIONAL ARRAY. Its one dimension is its length, and here it has a length of twenty. It can be as long as you want, within limits. Eventually you run out of memory space. In figure 18 you can see Chip setting up that array for you.
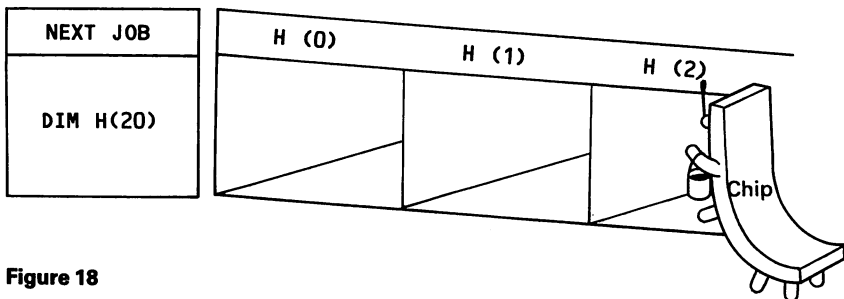
Figure 18

If you want to see just how rapidly you can use up 16k of memory, NEW and type this in:

```
10 DIM A(1813)
```

Run this, and you will see a * MEMORY FULL IN 10.

Change that to A(1812) and it will work, ending with a * DONE report. Mind you, there is no memory left at all. Add one short line:

```
20 PRINT "HI"
```

and you will get that MEMORY FULL report again.
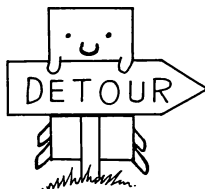
Numbers and the 99

Where has all the memory gone? The answer to that is also the reason why the 99 has a much higher level of mathematical accuracy than the great majority of home computers. Each number store actually takes up 8 bytes of memory. This allows it to handle very large numbers, very small numbers, and numbers with lots of decimal places (though it will only display 10 of them on the screen).

If you want to see the size of numbers that the 99 can handle, try this:

```
10 N=1
20 PRINT N
30 N=N*2
40 GOTO 20
```

Run this and it will start doubling and printing. When it reaches 8589934592 (that is a 10 digit number) it changes the

way it presents the numbers. The next one it prints looks like this:

`1.71799E+10`

This means 1.71799 times 10 to the power of 10, or 17179900000. 1.71799E+10 is the number shown in SCIENTIFIC NOTATION, and it is a more convenient way to handle very large (or very small) numbers.

When your number gets even larger you will notice another change. After 8.749E+99 the next number appears as 1.7498E+**. It is still keeping track of the powers of 10, but is no longer able to print them.

Eventually you reach this point:

`8.66459E+**`
`*WARNING`
`NUMBER TOO BIG IN 30`
`9.99999E+**`

That number is stored in the computer as 9.9999999999999 times 10 to the power of 127, or 9999999999999990000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000000000000 00000000000000000000000. If you could count a million numbers a second, it would take you 3170980000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000000000000 0000000000000 years to reach that number!

Change line 30 so that the numbers are halved rather than doubled, and you can see the process in reverse. Eventually, and it is after a very long while, the 99 decides the number is too small to bother about and treats it as 0.

All of which should help explain why the 99 needs 8 bytes to store a number. When the number is in an array, a further byte is needed for labelling. If you type in:

`PRINT 1812*9`

you should get 16308. You started off with 16384 bytes of memory, and have used a few for the DIM line itself.

Back to arrays

You will have noticed in figure 18 that the first of the array stores was labelled H(0). The 99 will start numbering arrays either from 0 or from 1, depending on your instructions. If you want the first one to be H(1) then you need a line at the beginning of the program:

```
1 OPTION BASE 1
```

If you want to start from H(0) you would write

```
1 OPTION BASE 0
```

If you don't tell the 99 what your OPTION BASE is, it will assume 0. Let's rewrite that height program properly and see what else we can do with arrays.

```
10 OPTION BASE 1
20 DIM H(20)
30 FOR N=1 TO 20
40 INPUT "HEIGHT ":H(N)
50 NEXT N
60 FOR N=1 TO 20
70 PRINT N,H(N)
80 NEXT N
```

Now, having collected all your information into an array, you process it in various ways very easily. You might start by working out an average. In Pack 1 you saw how you could total figures by adding a series of numbers into a store. We can do the same now by means of a loop:

```
 90 T=0
100 FOR N= 1 TO 20
110 T = T + H(N)
120 NEXT N
130 PRINT "TOTAL ";T
```

The average is the total divided by the number of entries, so add this:

```
140 PRINT "AVERAGE ";T/20
```

You can sort the array. Here the computer counts how many
of the heights were over 150 (cms) and how many 150 or less.

```
150 B=0      (Big people counter)
160 S=0      (Small people counter)
170 FOR N= 1 TO 20
180 IF H(N)> 150 THEN 210
190 S=S+1     (must be smaller, so add one on)
200 GOTO 220    (then jump over the next line)
210 B=B+1
220 NEXT N
230 PRINT "BIG PEOPLE ":B
240 PRINT "SMALL PEOPLE ":S
```

You can pick individual items out of your array:

```
250 PRINT H(3)
260 PRINT H(15)
```

You can see a one-dimensional number array at work in the
ARRAYS program. There it is used to store the pitches for a
tune. A similar array is used on the SOUNDS program to
store the pitches that are produced by pressing the number
keys. For details about that see 'More Sounds' below.

Into the second dimension and beyond

When you use a one-dimensional array, you have got, in
effect, a list of numbers (or whatever – see below). Suppose
you wanted to compare two or more lists? You could set up
two (or more) separate arrays, but there is an alternative,
which is often easier to handle. Your array can have two
dimensions – the first says how many lists, and the second
dimension is the length of the lists.

   Your family tells you that you are getting more and more
obsessed by the computer and that you are spending more
time every week playing with it. After you have told them at
great length, and in no uncertain terms, that you are not
'playing', you decide to keep records of the time you spend
at the typeface, and compare the results from a couple of

weeks. Here is the program you need to collect and analyse your data.

```
10  OPTION BASE 1
20  DIM H(2,7)     (Hours for 2 weeks, 7 days in
                    each)
30  FOR W=1 TO 2    (Weeks)
40  FOR D= 1 TO 7    (Days)
50    PRINT "WEEK";W, "DAY ";D (just so you can
                              see where you are)
60  INPUT "HOURS? ": H(W,D)
70  NEXT D
80  NEXT W
90  FOR D= 1 TO 7
100 PRINT H(1,D),H(2,D)     (same day, week 1
                            and week 2)
110 NEXT D
```

This is the basic program which collects, and prints out the numbers. Notice how the double loop between lines 30 and 80 manages the two dimensions of the array. If you wanted to compare the results of a month's programming hours, all you would need to change would be the number in line 30. If you used a separate (one-dimension) array for each week's entries, you would need four separate loops to collect the number from the four weeks.

Now, let's start to analyse the figure a little. Add this line:

```
105 PRINT "WEEK 2 - WEEK 1",H(2,D)-H(1,D)
```

and this will print up the number of extra hours you put in on the second week. The number may well be a negative one.

We can add a further set of lines to total up the hours in each week.

```
120 FOR W = L TO 2
130 FOR D = 1 TO 7
140 T(W) = T(W) + H(W,D)
150 NEXT D
160 PRINT "TOTAL WEEK ";W; " = ";T(W)
170 NEXT W
```

The key line in this is line 140 where an array is introduced –
T(2). This collects the totals for each week, and at the end of
the week is printed out by line 160. So now you can prove
that you aren't obsessed by the machine. Collect your daily
hours in a table like figure 19, and type this program back in
two weeks later. Your final display should look something
like figure 20.

	Week 1	Week 2
Monday	3 ·	4
Tuesday	4	2
Wednesday	3	3
Thursday	1	0
Friday	3	5
Saturday	4	4
Sunday	6	7

(busy day, Thursday)

Figure 19
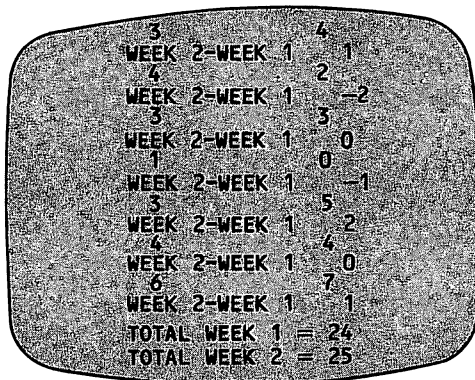
Figure 20

Well, it was only one hour extra!
   Two-dimensional arrays can be put to other uses besides
comparing lists. You can use them for storing the numbers

for sounds. The array would then be as many notes long as you wanted, and 3 numbers deep. S(10,3) would store full information on ten sounds. The sounds would be played by a routine like this:

```
100 FOR N= 1 TO 10
110 CALL SOUND(S(N,1),S(N,2),S(N,3))
120 NEXT N
```

When you are using arrays in bracketed instructions like this, make sure you put in the right number of closing brackets. It's very easy to miss one out. The 99 won't notice when you type the line in, but later, when it tries to work that line it will stop with an error report. By that time you will have carefully typed in all sorts of information, and you can't change a line and then CONTINUE a program. You have to run it again, and that means losing all the input data.

Two-dimensional arrays are also used in writing computer games, where you want to keep track of positions on a board. You will find more about this in the Games Packs, but here is a simple example of this kind of usage. The program takes in 9 numbers in a 3×3 array (perhaps for a noughts and crosses game) and prints them out.

```
10  OPTION BASE 1
20  DIM B(3,3)      (Board)
30  FOR R = 1 TO 3     (Row)
40  FOR C = 1 TO 3     (Column)
50  INPUT "NUMBER":B(R,C)
60  NEXT C
70  NEXT R
80  FOR R = 1 TO 3
90  FOR C = 1 TO 3
100 PRINT B(R,C);      (note the semi-colon at the
                        end)
110 NEXT C
120 PRINT      (to move the print position down)
130 NEXT R
```

So much for the second dimension, but what about beyond?

You can have arrays of up to 3 dimensions in TI BASIC. To go back to our "No-I'm-not-getting-obsessed" program, the third dimension could be for different months. Your new array is – H(13,4,7) – you work in lunar months, by the way! You are now able to compare the same week in different months, or even the same day of the week in different months. The program is basically the same as you had earlier, but with an extra loop to cope with months, and you have to remember to include all three subscripts (the numbers in the brackets) when you call up anything from the array. H(1,2,3) refers to the 3rd day of the 2nd week of the 1st month.

If you want to start comparing the results from different years (you are getting obsessed you know), then you will need a 4-dimensional array. You will also need a TI EXTENDED BASIC module. With this you can have up to 7 dimensions. Writing those sort of programs can be tremendously fiddly, but at the end of it, you have a very compact block of data that can be processed in all sorts of ways. Chip can cope with it!

Now where did I put Z(3,4,2,12,7,9,64)?

NEXT JOB

Z(3,4,2,12,7,9,64)

*Z(3,4,2,13,7,9,64)

+Z(1,2,4,6,8,12,64)

Chip
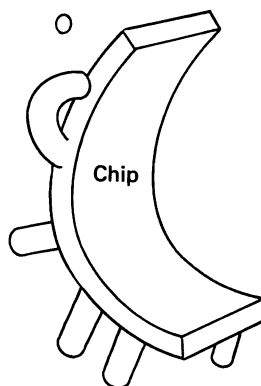
Figure 21

Strings and things

You may remember the "WHO'S THERE?" program in
Pack 1, where the 99 asked for peoples' names and printed a
friendly message. The names all went into the same store
though, so that each new name overwrote the old one. If we
set up an array of stores, the 99 could remember all the
names to say goodbye to later.

```
10  OPTION BASE 1
20  DIM NAME$(5)     (change the numbers later to suit
                      yourself)
30  FOR T=1 TO 5
40  INPUT "WHOS THERE?":NAME$(T)
50  PRINT "HELLO ";NAME$(T)
60  NEXT T
70  FOR T=1 TO 5
80  PRINT "GOODBYE";NAME$(T)
90  NEXT T
```

Run this and try typing in names of different lengths. You
will see that the string arrays automatically have two
dimensions – the first is the store number, and you fixed that
in the DIM line – the second is the length of the string, and
that depends upon what is entered at the INPUT line. This
could be a little inconvenient if you wanted to print out the
names as part of a survey in a nice tidy table. There is a
solution – of course – isn't there always? You can chop off as
much of the string as you want using the SEG$ instruction.
This tells the 99 you only want a SEGment of the string.
Change line 50 to this:

```
50  PRINT "HELLO ";SEG$(NAME$(T),1,5)
```

Now type in RUMPLESTILTSKIN and see what it prints.
    You should see "HELLO RUMPL". The SEGment you
asked for picked out five characters starting from the first.
There are three bits of information needed in the brackets.

SEG$ (which string, where to start, how many characters)

Here's a test to show it more clearly. Type in (no line number):

```
T$="TESTING"
PRINT SEG$(T$,3,3)
```
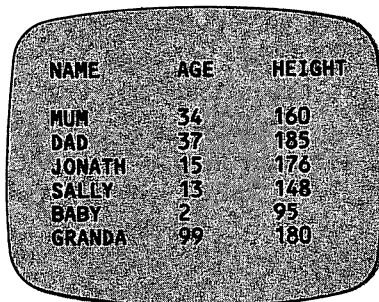
and you will get "TIN".

Now try: `PRINT SEG$(T$,1,1)`

You should see "T".

```
PRINT SEG$(T$,7,10)
```

Will print "G". You actually asked for a further 9 characters that weren't there. The 99 cannot be fooled that way. It went as far as it could and then stopped. This is very handy. It means that if you want to produce a nice neat table, like figure 22, you can set a limit to the length of names, by using the SEG$ instruction, and if the names are shorter than your limit, then the 99 will simply print all that is available. On some computers you would get into trouble if you asked them to print a long segment of a short word. It is another example of how the 99 takes some of the minor irritations out of programming.

Here is the program that produced it. Notice how in this the variable names have all been kept as short as possible. It really does save on typing errors.

```
 10 OPTION BASE 1    (the program will work without
 20 DIM N$(6)         any of these lines as the
 30 DIM A(6)          numbers are small, but write
 40 DIM H(6)          them anyway. You would need
                      them for a larger survey)
 50 FOR N=1 TO 6
 60 INPUT "NAME ":N$(N)
 70 INPUT "AGE ":A(N)
 80 INPUT "HEIGHT ":H(N)
 90 NEXT N
100 CALL CLEAR    (just to tidy up)
110 PRINT TAB(2);"NAME";TAB(12);
                 "AGE";TAB(22);"HEIGHT"   (titles)
120 FOR N = 1 TO 6
130 PRINT TAB(2);SEG$(N$(N),1,6);    (first 6
TAB(12);A(N);TAB(22);H(N)             letters
                                      only)
140 NEXT N
```

Do you see how the 99 chopped the names down to a
maximum of 6 letters in line 130? You can see what has
happened to Jonathan's and Grandad's names in figure 22.
Chopping up strings in this way is known as STRING
SLICING, and we will come back to it later. Meanwhile, let's
have a look at the way that arrays are used in the SOUNDS
programs, and see what else we can do with the 99's sounds
possibilities.

8
The musical keyboard

In the SOUNDS program you will find a routine that turns the 99 into a simple electric organ. You might like to build on this to give the '99-organ' a greater range of notes. There are two parts to that routine, one produces single notes, the other gives three-note chords. Let's look at the single notes first.

The frequencies for the notes in the scale of C are read into an array – P(8).

Array number	Frequency	Note
P(1)	262	C
P(2)	294	D
P(3)	330	E
P(4)	349	F
P(5)	392	G
P(6)	440	A
P(7)	494	B
P(8)	523	C

We now have a simple means of calling up those frequency numbers. A CALL KEY line finds the code number of the key that is touched. It is checked to make sure that it is in the range 49 to 56 (ASCII codes for 1 to 9), then 48 is taken off. Pressing ③ gives the key code 51, take away 48 leaves 3, and P(3) is 330, the frequency of E. The note playing line looks like this:

```
CALL SOUND(-2000,P(K),1)
```

A negative sign in front of the time number tells the 99 to cut any existing sound short, and play the new one immediately.

You can extend this idea in various ways. One way would

be to use the split keyboard scan so that notes could be played by the left hand, or the right hand or both. Figure 24 shows the flowchart for the program that would do that. You will notice that on this flowchart the boxes enclose routines that may well take up several programming lines, and which may have jumps and loops inside them. This is standard flowcharting practice. The first thing to work out is the overall shape of a program, and the details can be sorted out later. It may be that you find the need to draw up flowcharts for more complicated routines within the program.
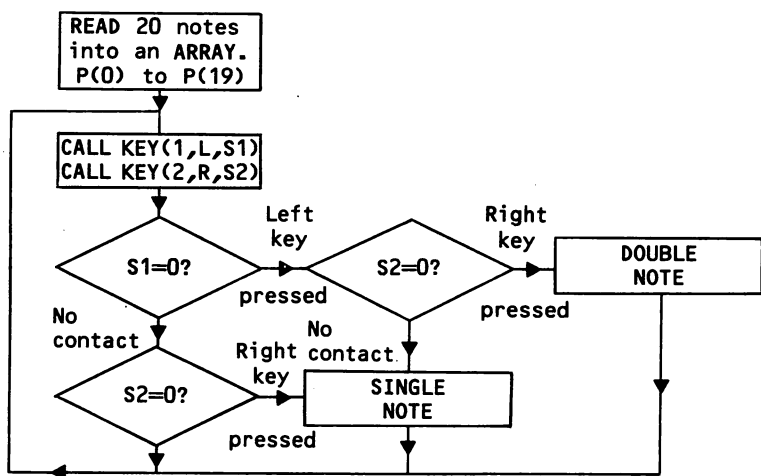
Figure 24

There are various little problems here, and many of the solutions can be applied to routines in other programs as well, so let's work our way through them.

If you look at figure 25 you will see the code numbers returned by the CALL KEY instruction when working on a split keyboard. As you can see, the numbers are not nicely arranged in order. This means that either you must fit the pitch frequencies into the array so that the keyboard becomes ordered, or you will have to hunt for your notes when you reach the playing stage. We will sort out the pitches in the array. It's fiddly, but worth it in the end. The

CALL KEY(1,....)   CALL KEY(2,....)

1	2	3	4	5	6	7	8	9	0	=
19	7	8	9	10	19	7	8	9	10	

Q	W	E	R	T	Y	U	I	O	P	/
18	4	5	6	11	18	4	5	6	11	16

A	S	D	F	G	H	J	K	L	;	ENTER
1	2	3	12	17	1	2	3	12	17	

SHIFT	Z	X	C	V	B	N	M	,	.	SHIFT
	15	0	14	13	16	15	0	14	13	

ALPHA	CTRL	SPACE	FCTN

Figure 25

pitches can cover any range of notes you want, but here we will use those given earlier in figure 17, which go from a low G through to G 2 octaves higher. There are only 19 pitches there, so we will add a twentieth – 1397, this is the frequency of the high beep that the 99 makes when the INPUT ? appears. The keyboard is arranged as in figure 26. This shows only the left hand side. The right hand side follows the same key codes, and produces the same range of notes. The high beep is the only note whose key is not in the equivalent place on the right hand side. (Right code 16 is above ENTER)

High
D	E	F	F#	G
19	7	8	9	10

G	A	B♭	B	C
18	4	5	6	11

C	D	E	F	F#
1	2	3	12	17

Low
G	A	B♭	B	High beep
15	0	14	13	16

Figure 26

We can arrange these into a table:

Code	0	1	2	3	4	5	6	7	8	9
Note	A	C	D	E	A	B*b*	B	E	F	F#
Frequency	220	262	294	330	440	446	494	659	698	740

Code	10	11	12	13	14	15	16	17	18	19
Note	G	C	F	B	B*b*	G	beep	F#	G	D
Frequency	784	523	349	247	233	196	1397	370	392	587

Figure 27

And then read them into an array.

```
10 OPTION BASE 0    (this time we need to start at 0)
20 DIM P(19)
30 FOR N = 0 TO 19
40 READ P(N)
50 NEXT N
60 DATA 220,262,294,330,440,
   466,494,659,698,740
80 DATA 784,523,349,247,233,
   196,1397,370,392,587
```

Don't type this in yet. Wait until you can assemble the whole program.

The central part of the program is the keyboard check, and there are four possibilities to be accounted for – left hand key only, right hand key only, one from each side, and none. Either single key contact will lead on to the single-note playing routine. You can see the type of routine that will check in this way in the example below. This prints the code number(s) of the key(s) pressed.

```
100  CALL KEY(1,K1,S1)      (left)
110  CALL KEY(2,K2,S2)      (right)
120  IF S1<>0 THEN 200    (left contact)
130  IF S2<>0 THEN 300    (right contact only)
140  GOTO 100    (no contact)
200  IF S2=0 THEN 300    (no right side key)
210  PRINT K1,K2    (both sides touched)
220  GOTO 100
300  K=K1+K2+1    (single key routine)
310  PRINT K
320  GOTO 100
```

Compare this with the flowchart, and take a few 'dry runs' through it. What would you expect to happen under different types of contact?

The line which needs a little further explanation is line 300.

```
K = K1 + K2 + 1
```

The Key Contact variables (K1 and K2) are reset to a value of −1 whenever the CALL KEY is performed. As only one of these has been altered by the time the program reaches line 300, the other must still be −1 we need to neutralise it by adding 1. The end result is that the variable K takes the value of the key which was pressed, whichever side of the board it came from. Type that routine in, and run it and see.

Those CALL KEY lines can be simply altered to produce sounds, rather than print the code numbers. The single note line becomes:

```
310  CALL SOUND(−2000,P(K),1)
```

and the double note:

```
210  CALL SOUND(−2000,P(K1),1,P(K2),1)
```

Put it all together and you will have a 2-octave, 2-handed organ. It is not as quick to respond as a modern electric organ, but plays more like the old-fashioned wind-powered organs. How about a bit of Bach, with counterpoint and two-part harmonies?

P.S. There is no reason of course why you should have the same sets of notes on both sides of the keyboard – except that it's easier to write. You would need two arrays, or a two-dimensional array for the two sets of frequencies, and you would need to split the single note line into two routines, one for each half. You would also need a minor adjustment to the double note line.
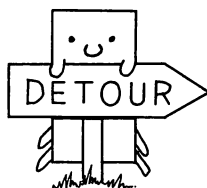
Musical intervals

If you look at a table of note frequencies, you might wonder how all those numbers fit together. Here's a program which shows the pattern.

```
10 OPTION BASE 1
20 DIM P(12)
30 FOR N = 1 TO 12
40 READ P(N)
50 NEXT N
60 DATA 262,277,294,311,330,349,
   370,392,415,466,494
```

This puts into the array the frequencies of the 12 semitones from middle C upwards – C, C#, D, D#, E, F, F#, G, G#, A, B♭, B.

```
70 FOR N = 1 TO 11
80 PRINT P(N);P(N+1);(P(N+1)-P(N))/P(N)
90 NEXT N
```

This prints out each pair of adjacent notes, and also the difference between them (the interval) – $(P(N+1)-P(N))$, divided by the first of the pair.

Run this and you will see that the third number is always between .057 and .061. The number varies because the

frequencies used are all rounded up or down to the nearest whole number, while they should really be more complicated decimals.

The difference averages .05935. You can use this to create any scale. Add this:

```
100  S=262    (middle C)
110  FOR N= 1 TO 12
120  PRINT INT(S +.5)    (to give the nearest
                          whole number)
130  CALL SOUND(1000,S,1)
140  S=S +S*.05935    (add on the difference)
150  NEXT N
```

Run this and you will see that the numbers printed agree, almost exactly, with the frequencies used in the data line. There is a slight difference on a couple of notes, but not the sort of differences that most people could hear. Change the frequency given in line 100 and try this for different scales. This might prove useful at some point if you ever want to generate the values for notes, rather than reading them in from data. The program might also suggest to you other ways in which you might use the 99 to work out mathematical problems.

NOTE: The Nearest Whole Number

The INT instruction always rounds down, by chopping off the decimals. You can get it to round to the nearest whole number by adding .5 before you INTegerise the number. If you start with 1.9 (nearest whole number 2), add .5 (to get 2.4) and then INT, you are left with 2. Start with a smaller decimal bit, e.g. 1.4, add .5 (= 1.9) the INTeger is still 1.

9
Program planning 3
Controlled inputs

Large and small letters

The fact that the 99 has large and small capitals means that you can vary your presentation of text on the screen, it also means that you have to be extra careful when checking inputs. You may remember how, in Pack 1, the answers in the branching programs (PETS and TRANSPORT) were all numbers, or "Y/N".

The string answers are checked by lines like these:

```
IF  A$="Y" THEN....
IF  A$= "N" THEN.....ELSE.....    (back to
                                  INPUT line)
```

The program checks for large capitals. What happens if you type in small capitals? Type in the routine below, and find out:

```
10  INPUT A$
20  IF A$="Y" THEN 50
30  PRINT A$
40  GOTO 10
50  PRINT "BIG Y"
60  GOTO 10
```

If you don't use a large Y, the 99 won't jump to line 50. This could be inconvenient, expecially if you were writing your program for young children or other people who would be unlikely to spot the difference between "Y" and "Y". You need to add some means of making both Y's acceptable to the 99. There are several ways of doing this – aren't there always several ways of doing everything?

The simplest method is to add this line at the beginning of the program:

```
5 CALL KEY(3,K,S)
```

No more is needed. You are not interested in which key is pressed – that will come through the INPUT line. What this line does for you is to reset the way in which the 99 looks at the keyboard. Run the program again, and you will see that large capitals appear on screen at the INPUT, whether or not you are pressing SHIFT, and whether the ALPHA LOCK is on or off. This effect only works in keyboard mode 3. In the other CALL KEY variations the keys give both big and small capitals (modes 4 and 5) or special code number (modes 1 and 2). On A CALL KEY (3. . .) line will reset the keyboard for the rest of the program, or until a different CALL KEY line is used.

You may wish to allow small letters to be input, in which case the CALL KEY routine is no use. Here's the second way, and it's the more obvious one:

```
IF  A$ = "Y" THEN....
IF  A$ = "Y" THEN.....
etc.
```
(both go to same line)

These two lines can actually be compressed into one using logical operators, and you will find out about them in chapter 11.

The third way of checking your inputs is to use the ASCII codes.

```
10  INPUT A$
20  IF ASC(A$)<91 THEN 40
30  A$ = CHR$(ASC(A$)-32)
40  IF ASC(A$) = 89 THEN 60
50  GOTO 10
60  PRINT "THIS IS Y"
```

The ASCII code is checked twice. At line 20 the check is to find out if a small letter has been input. Small letters have codes from 97 to 122. Large letter codes finish at 90. If a large "Y" has been entered, the program jumps from 20 to 40 and

there it leaps on to 60, as ASCII "Y" is 89. If a small "Y" was entered the program would go through line 30. This finds the ASCII code of the letter that was input, (ASCII"Y" is 121) takes off 32 (121 − 32 = 89) and gives A$ a new code number.

NOTE: ASC(A$) actually checks only the first letter of A$, no matter how long the string may be, and the new A$ at line 30 is only one letter long. Add a line:

35 PRINT A$

and run the program, typing in "Y" and "YES" in large and small letters. You will see that no matter what the answer, as long as the first letter is a "Y" of some sort, the check line at 40 will accept it. This routine is particularly useful when you can't be sure just how people will try to answer.

INPUT anywhere

You may be writing a program where you would like your user to write his answer (or other message) at a particular point on the screen, rather than on the bottom line. The routine below uses CALL KEY instead of INPUT to get the information from the keyboard, and CALL HCHAR instead of PRINT to put it on the screen.

```
10 CALL CLEAR
20 C = 10     (Column starting point)
30 CALL KEY(3,K,S)
40 IF S<>1 THEN 30     (so that only new contacts
                        are picked up)
50 IF K = 13 THEN 90   (13 is the code for
                        ENTER)
60 CALL HCHAR(10,C,K)  (starts to "PRINT" at
                        10, 10)
70 C = C + 1     (to move the "PRINT" position
                  along)
80 GOTO 30
90 PRINT "END OF INPUT"
```

Type it in and try it. You can fix the starting position by changing the value of C in line 20, and the Row number in the CALL HCHAR line.

If you are only interested in what the first letter of the INPUT is ("Y" ?) then you would include a line at 45 or 55 to check this. If you want to store the whole of the input for later use, you need to collect it into a string store. This introduces a new technique for handling strings known as STRING CONCATENATION. String slicing is about chopping strings up, string concatenation is about sticking them together (see below). Add this line to the program:

75 A$ = A$ & CHR$(K)

The "&" (ampersand) joins strings together. This line adds the character with the code K onto the end of A$. Add another line:

100 PRINT A$

You should have the same message at the bottom of the screen as you have in the middle. A$ can now be checked and compared in the usual ways.

You can dress this kind of input routine up, to produce a better effect. Add a flashing cursor and a beep or two. You may have come across the imitation inputs in the KEYS program in Pack 1.

10
Strings

Which bit would you like?
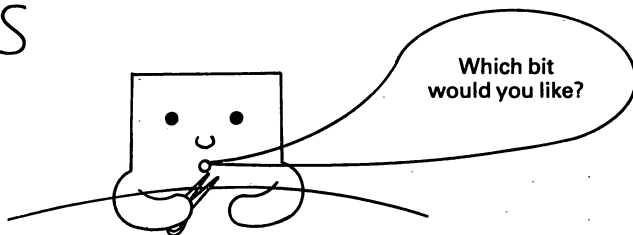
String slicing

You can 'slice' any string. It doesn't have to be in an array, it doesn't even have to be in a variable. Prove it with this:

```
PRINT SEG$("TESTING",4,3)
```

Type this in, and it will print three letters starting from the fourth one:

```
TIN
```

You can slice your way through a string by using a loop. Try this:

```
10 INPUT "NAME ":N$
20 FOR T= 1 TO 5
30 PRINT SEG$(N$,T,1)
40 NEXT T
```

Run this an type in "JAMES". You will see this

Figure 28

61
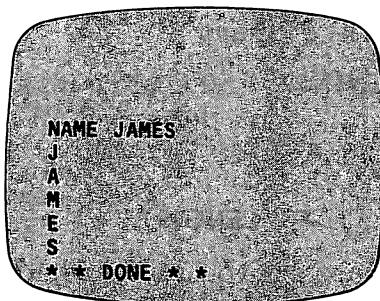
You don't have to restrict yourself to single-letter slices in line 30. Change it to `PRINT SEG$(N$,T,2)` and you will see this:
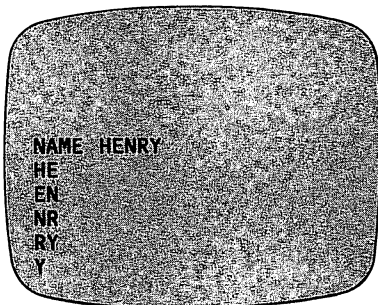
Figure 29

You will notice that when the loop reaches 5 there is nothing after the 5th letter, so the single "Y" is printed.

Now this is all very well for names of 5 letters, but what about longer ones? You need to find how long the name is before you fix the number in the loop. You can do this using the LEN command. This tells you the LENgth of a string. You can see this by typing in : (no line number);

```
PRINT LEN("WORD")
```

The 99 will print 4. Notice that you must put quotes around the word in the brackets so that the 99 knows it is a string. Failure to do so will give the error report * STRING – NUMBER MISMATCH.
Add a line to that last program:

```
15  PRINT LEN(N$)
```
(the $ says it's a string, you don't need "")

and change line 20

```
20  FOR T = 1 TO LEN(N$)
```

Line 15 is just there so that you can see what's happening, it is line 20 which has really altered the program. Now the loop will always be as long as the word. Try it with names like "JIM", "JB" and "RUMPLESTILTSKIN".

Printing anywhere

You saw in Controlled Inputs how to input anywhere on the screen. You are now almost ready to print anywhere on the screen. To do this you will use an HCHAR (or VCHAR) instruction, and to use that instruction you need to know the character codes of the letters you want printed. We can find this out by getting the ASCII codes of our single letter slices.
Add a line:

25 A = ASC(SEG$(N$,T,1))   (watch those brackets!)

and change line 30

30 PRINT A

Now type in some names and you will see the codes of the letters printed on the screen. Add one more line and you have got controlled printing:

35 CALL HCHAR(10,T,A)

It's also worth adding a CALL CLEAR line at the start of the program, just to tidy things up.
When you run this now you will get something like this:
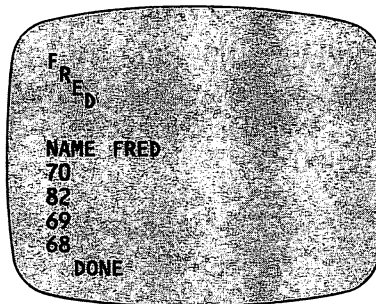
Figure 30

Aargh! What went wrong? The problem is, you are PRINTING (which makes the screen scroll) while you are HCHARing (which really needs a steady screen). Knock out line 30, you don't actually need it. Now try again. Better, but

not too good. The left hand side of the picture tends to disappear off the screen. Move the column position across by adding to it in the HCHAR statement:

```
CALL HCHAR(10,T+10,A)
```

You should finish up with something like this:
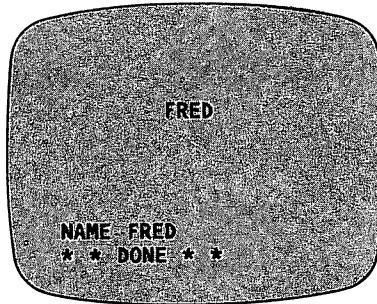
Figure 31

You will find this sort of "print-anywhere" routine in most of the programs on the cassette. It is usually worked in as a sub-routine from line 6000 on. The program then specifies the string that needs printing, and the row and column start-points, before the GOSUB command.

```
350  W$ = "HELLO AND WELCOME"
360  R = 3      (will print near the top
370  C = 7      middle of the screen)
380  GOSUB 6000
390  ....
....
6000 FOR Q= 1 TO LEN(W$)
6010 X = ASC(SEG$(W$,Q,1))    (watch the
                              brackets!)
6020 CALL HCHAR(R,C+Q,X)      (C + Q moves
                              "PRINT" position
                              along)
6030 NEXT Q
6040 RETURN
```

You may notice here that Q is used to label the variable in the loop, rather than N or T, which are the loop variables normally used in these packs. It is, of course, entirely up to the individual what variable names he uses. What is important is that you do not use the same name for two different stores, especially if there is the slightest possiblity that these might cross paths during the course of a program. It helps a great deal if you stick to the same names for the same types of stores in different programs. In most of the cassette programs, Q is used as the loop variable in the print routine. W$ is used for the string of words that are to be printed.

What's in a string?

Suppose you are looking for a particular word or letter that may appear in an answer. You might have written a quiz program and have as a question "WHAT IS THE CAPITAL OF FRANCE". If your check line looks like this:

```
IF A$ ="PARIS" THEN.....    (well done routine)
PRINT "WRONG"
```

It could be awkward if the player had entered "THE CAPITAL OF FRANCE IS PARIS". What you need is some means of checking whether or not "PARIS" appears anywhere in the string. There is a simple way of finding that out. It uses the POS function. This looks for the POSition of a string inside another string. If the computer reaches the end of the answer string without finding the word it was looking for, then it gives a value of 0, otherwise it gives the number of the letter where the word starts. The line looks like this:

```
IF POS(A$,"PARIS",1)>0 THEN....    (well done)
```

It looks through A$ for "PARIS" starting from the first letter (1). Here's what happens with some different answers to the question.

ANSWER	POS number
"THE CAPITAL OF FRANCE IS PARIS"	26
"PARIS"	1
"I THINK IT'S PARIS"	14
"LONDON"	0

Take care as the results are not always what you would expect. Try this and see;

```
10  INPUT "MALE OR FEMALE ":S$
20  IF POS(S$,"MALE",1)>0 THEN 50
30  PRINT "YOU'RE A GIRL"     (must be if not male)
40  GOTO 10
50  PRINT "YOU'RE A BOY"
60  GOTO 10
```

Now enter first "MALE" and then "FEMALE". What happened?
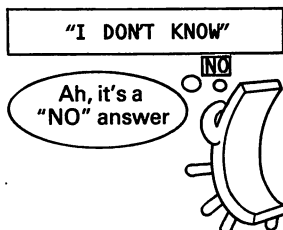
Split and shuffle

You can use the POS function to split a string up into its separate words. The words are marked off by the spaces between them, so all you need to look for are those spaces. We can build up a program to do this. Start with this:

```
10  INPUT S$     (sentence)
20  X = POS(S$," ",1)
30  Z$=SEG$(S$,1,X)
40  PRINT Z$
```

Line 20 finds the position of the first space, and line 30 puts into the Z$ store the characters from the beginning of the string up to that space. Run it and enter a short sentence.

We now need to chop that first word off the string, and leave the rest. It would help if we knew how long the string was to start with:

```
15 L = LEN(S$)
```

Rather than put the rest of the string into a different store, we will change S$ so that we can go back through the same lines to chop off the next word:

```
50 S$ = SEG$(S$,X+1,L)
60 PRINT S$    (just so you can see)
70 GOTO 15     (back for more)
```

Add the new lines and run it again. You should see something like this:
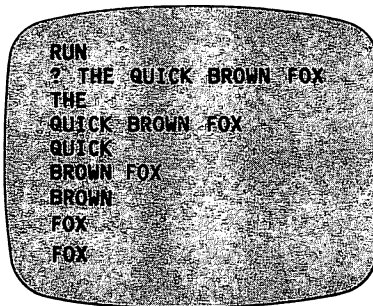
Figure 32

The problem is, you haven't told it when to stop. The obvious time to stop is when you are down to one word, and there are no spaces left. Add another line:

```
35 IF X = 0 THEN 80
80 STOP
```

Variations on this technique are much used in word-processing programs, where the text is entered in a continual string, and then chopped into line lengths by the machine, using this type of check to make sure that each line ends with a whole word. Words may also be separated by different types of punctuation (,.;:-) so these must be looked for as well.
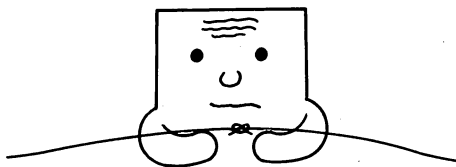
Knots

You would think that the term used for joining strings together would be knotting – it isn't. As mentioned earlier, joining strings is referred to as
STRING CONCATENATION.

If you simply want to print strings together on screen, you don't bother with this – you just arrange them in your print line.

```
10 A$ = "HELLO"
20 INPUT "NAME":N$
30 B$ = "HOW ARE YOU ?"
40 PRINT A$;N$;B$
```

However, if you were going to use a pretty printing technique – with an HCHAR subroutine, you would need to join all the strings together before you sent them off for processing and printing. Replace line 40 with this:

```
    40 W$ = A$ & N$ & B$
```
Add   `50 PRINT W$`   (just so you can see)

You cannot include a simple number in a string. It has to be strung up first. You might have a program that uses a CALL KEY line, and you want to print the letter of the key that was touched. Use the CHR$ function:

```
...W$ = "YOU HAVE TOUCHED THE " & CHR$(K)
& " KEY."
```

Notice the space left in the strings immediately before and after the CHR$(K) part. Without these it will be squashed together in the middle.

You might want to include a number (perhaps in a sums program) in a line of comment. The function STR$ puts

quotes round a number, or a number variable, thus making it into a string:

```
... INPUT ANSWER
```

if it's wrong. . .

```
... W$ = " NO, "& STR$(ANSWER) & " IS WRONG"
```

By the way, if you have got a number in a string store, and want to know its value, you can reverse the process. VAL(ANSWER$) will give you the VALue of the numbers in the string. You will find VAL used in the Games Packs for getting the value of numbers that have been stored in string arrays. Number arrays you will remember are very expensive on memory space. String arrays are much cheaper to use.

You will see a number of these string handling techniques used in the HANGMAN program. This, and other string-based games are covered in Game Writer's Pack 1.

11
Random logic

Random Logic !?! There's no such thing, you say. Quite right, there isn't. That misleading title is just to get you interested. What we are going to do is to combine random numbers, which you will have come across in Pack 1, with LOGICAL OPERATORS which you probably haven't met.

Random

Just to refresh your memory – the RND function produces a (more or less) random number between 0 and 1. To turn this into a decent sized number you multiply it. RND*10 gives you a random number between 0 and 10. This can be turned into a whole number by the INT function, which chops off the decimal part.

```
INT(RND*10)
```

will produce a random number between 0 and 9.

If you wanted the range of numbers to be 0 to 10 add .5 before you INT.

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

or push your range of numbers up by adding another number.

```
INT(RND*10) + 1
```

gives numbers between 1 and 10.

Don't forget to include RANDOMIZE at the start, so the numbers really are random.

Logical

The 99 will check any equation to see if it is TRUE or FALSE. A TRUE equation will give a value of −1, a FALSE equation is worth 0. You can see this in the program below:

```
10 A = 99
20 PRINT (A =99)
30 PRINT (A = 1)
```

Run this and the 99 will print −1, because it is true that A = 99, and then 0, because the equation in line 30 is false. Add another line:

```
40 PRINT (A>90)
```

This is true, and you will see −1 printed. In just the same way (A<100) would give a −1 value.

True and false works for strings as well. NEW that program and type this in:

```
10 A$ = "TEST"
20 PRINT (A$ = "TEST")
30 PRINT (A$ = "TRY")
```

Run this and you will see −1 and then 0 printed.
You can work these into IF. . .THEN. . . lines:

```
10 INPUT A$
20 IF (A$ = "TEST") = −1 THEN 30
ELSE 10
30 PRINT "YOU TYPED IN TEST."
```

Try it. You will see that it works just as well as the normal sort of check line – IF A$ = "TEST" THEN 30. . If you want to check a single variable you might just as well stick to the standard lines. However, using the logical operators you can include more than one thing in your check line. Change lines 20 and 30 to these:

```
20 IF (A$ = "TEST") +(A$ = "TRY") = −1
THEN 30 ELSE 10
30 PRINT "YOU TYPED IN TEST OR TRY"
```

Now run it a few times entering "TEST","TRY" and some other words. The two equations in line 20 are valued, and their results added together. If both are false the total is 0. If either of them is true, the result is the same. $-1 + 0 = 0 + -1 = -1$. You can include as many checks as you want in that line. As long as any one of the equations is true the result is still -1.

This routine works in the same way as what is known in mathematics as the Boolean OR operator. IF either this OR that OR something else is TRUE. . .

You can produce a routine to work the same as an AND operator. IF this AND this are TRUE. . . Try this:

```
10  INPUT A$
20  INPUT B$
30  IF (A$ ="TEST") + (B$ = "RUN") = -2
THEN 40 ELSE 10
40  PRINT "TEST RUN"
```

Type in "TEST" and then "RUN" and see what happens. Try different combinations of inputs. Both equations must be true to give a total value of -2. Change line 30:

```
30  IF (A$ = "TEST") + (B$ = "RUN")<>0
THEN 40 ELSE 10
```

You now have combined AND/OR. The program will jump to line 40 if "TEST" AND "RUN" are entered. It will still jump if either "TEST" OR "RUN" are entered. Try it and see.

NOTE: Actually, you can save a bit of typing on that line. Miss out the "$<>0$" and it still works. On any IF. . . THEN. . . line, IF the condition gives a non-zero result THEN the program jumps. This is what happens with a normal check line.

```
IF A = 99 THEN...
```

When the 99 hits this line it works out the truth value of $(A = 99)$. If it is true (value -1, which is not zero) then it jumps.

You will find logical operators in the check lines in some of the programs on the cassette. They are frequently used to check the ASCII code of inputs. Look at the listing of the SOUNDS program and you will find this:

```
1230  CALL KEY(3,K,S)
...
1230  IF (K>64) + (K<72) =-2 THEN 1400
```

This line checks to see if one of the letters A to G has been pressed. The range of acceptable character codes is therefore 65 to 71 (>64 AND <72).

Hunt the thimble

Do you know this game? One person hides a thimble (or something) and the other player(s) try to find it. To help them in their search the thimble-hider tells them if they are getting closer to it, or further away. You are 'cold' if you are a long way from it, and 'getting warmer' as you draw near. We can get the 99 to hide thimbles for us, using random numbers, and compare our guesses with the true position using logical operators. We can play the game on the screen using co-ordinates. The flowchart for the game is shown in figure 33.
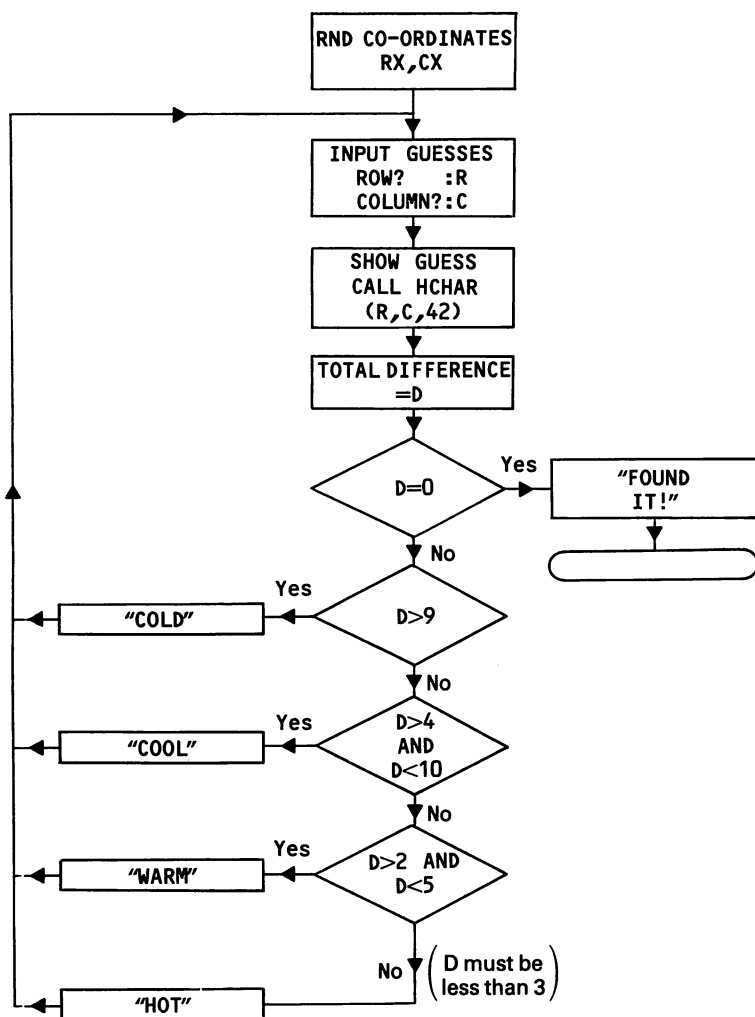
Figure 33

You can work most of the program out already, but there may be a few points worth looking at more closely.

Random co-ordinates

There is no reason why you shouldn't use the whole screen, leaving just the bottom line free for printing messages on. INT(RND * 23)+1 will give you a value for RX between 1 and 23. Use a similar line for the column.

Inputs

If you use a normal input your screen will scroll, which will ruin the display, so use an 'Input anywhere' routines to get the player's guesses. The numbers could well be two-figure ones, so collect the key contacts into a string, until ENTER is pressed, and then evaluate the string. R = VAL(R$).

Show the guess

A simple CALL HCHAR(R,C,42) will print an asterisk at the guessed spot, but you might like to define a different character, and give it a splash of colour.

Total difference

You need to find the absolute difference between the thimble's hiding place and the player's guess. This line gives it:

```
D = ABS(R-RX)+ABS(C-CX)
```

The ABS function tells you the ABSolute difference between two numbers. It is essential here. Look what might happen without it. Suppose the thimble was at 10,15 and the player guessed 12,13.

```
R-RX = 10 - 12 = -2
C-CX = 15 - 13 = 2
```

Add these together and the total is 0. He's found it!!!
ABS knocks any negative sign off the front of a number.

```
ABS(-2) + ABS(2) = 2+2 =4
```

Check lines

Simple check lines only are needed for the "found it" – IF
D = 0 THEN. . . and for the "COLD" – IF D>9 THEN. . .
Use AND type checks for the others:

```
IF(D>4)+(D<10) = -2 THEN...
```

Messages

Have these printed on the bottom line using the HCHAR
routine. If you make sure that each message is the same
length (by padding up with spaces where needed), then they
will overprint each other, and you won't have to worry about
rubbing them out.

Limits and scores

You could limit the number of guesses by using a
FOR. . .NEXT. . . loop, rather than a simple GOTO, and
print the thimble's position if it still hasn't been found after
(say) 10 goes. You could allow as many guesses as need be,
but keep a count, printing it out at the end. This would give
players a best score to aim at.

Have fun, and if you want to know more about the uses of
random numbers in games, then have a look at the Games
Packs.

12
Morse and more sounds

If you LIST the SOUNDS program you will see these lines:

```
2100 CALL KEY(3,K,S)
2110 IF S=0 THEN 2100
...
2130 IF (K>64)+(K<90) =-2 THEN 2495
...
2495 PRINT CHR$(K);
2500 K = K - 64
2505 IF K>13 THEN 2800
2510 ON K GOTO.....
```

This is the part of the program which prints the letters and
sends the computer off to make the morse beeps and buzzes.

Notice the semi-colon at the end of line 2495. This is so that
the letters are printed across the screen like a typewriter.
Line 2500 then brings the value of K down into the range 1 to
26. This is so that an ON. . .GOTO. . . line can work. That
line cannot, however, cope with the full 26 GOTO line
numbers, so line 2505 redirects the letters of the second half
of the alphabet (K over 13). At line 2800 K is again revalued
to bring it back into the range from 1 up, and a further
ON. . .GOTO. . . line sends the computer off for the
appropriate beeps. You might have expected this part of the
program to be packed full of CALL SOUND lines. It isn't.
The dot and dash sounds are produced by two separate
subroutines (at 2400 and 2450), so the program is packed
with GOSUBS instead.

```
GOSUB 2400
GOSUB 2450
```

produces the dot and dash for the letter A.

Figure 34 shows the Morse code for use with this program.

0 -----	A .-	K -.-	U ..-
1 .----	B -...	L .-..	V ...-
2 ..---	C -.-.	M --	W .--
3 ...--	D -..	N -.	X -..-
4 ....-	E .	O ---	Y -.--
5	F ..-.	P .--.	Z --..
6 -....	G --.	Q --.-	
7 --...	H	R .-.	
8 ---..	I ..	S ...	
9 ----.	J .---	T -	

Figure 34

You could adapt the program so that the dots and dashes were printed out. The sections that deal with the different letters and numbers are all clearly REMed. Include a print line in each of the sections like this:

```
2520  REM A
2521  PRINT ".-"     (your new line)
2522  GOSUB 2400     (dot)
2525  GOSUB 2450     (dash)
2530  GOTO 2100      (back for the next)
```

Notice that there is no semi-colon this time. The screen you get will look like this;
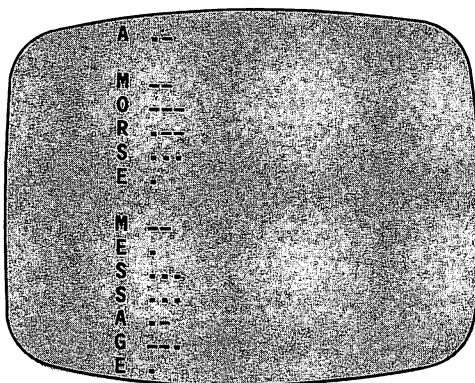
Figure 35

After a while you will get to know the morse code, and no longer need to cheat by making the 99 work it out for you. That's the time to build your own morse buzzer. Here is the flowchart for a simple buzzer.
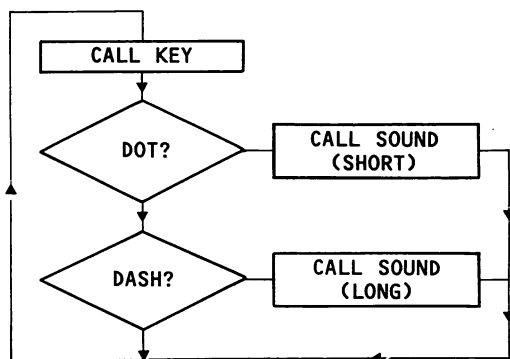
Figure 36

Sound effects

The best way to find out the sort of sound effects you can get out of the 99 is to sit and play with it for a few hours, but here are a few ideas that might lead you towards some good sounds. Some of these you may have come across in the EFFECTS program in Pack 1, others are included in the SOUNDS program.

Falling bomb

Because each sound on the 99 is distinct, it is not possible to get a completely smooth sliding or dropping effect, however, this doesn't sound too bad:

```
10 FOR P = 1500 TO 200 STEP -10   (down the
                                   scale)
20 CALL SOUND(10,P,1)
30 NEXT P
```

and when it lands you need a good blast:

```
40 CALL SOUND(1000,110,0,215,0,
   315,0,-8,0)
```

"0" is as loud as you can get. The combination of three notes out of harmony and the crackly white noise is fairly striking.

The explosion must then die away, so this time the volume goes into a loop:

```
50 FOR V = 0 TO 30
60 CALL SOUND(100,150,V,200,V,
   300,V,-5,v)
70 NEXT V
```

You don't have to change the note pitches in line 60. They could be the same as in line 40, but it definitely helps to use −5 here rather than −8. There is a distinct difference in the quality of the noise.

Sirens

These do present a real problem, because the best sirens have a fast smooth up and down sound. A straight two-tone is easy enough:

```
10 CALL SOUND(300,523,1)
20 CALL SOUND(300,392,1)
30 GOTO 10
```

Sliding ones are a bit fiddlier. How does this sound to you?

```
10 FOR P = 400 TO 600 STEP 10
20 CALL SOUND(-50,P,1,-1,1)
30 NEXT P
40 FOR P = 600 TO 400 STEP -1
50 CALL SOUND(-50,P,1,-1,1)
60 NEXT P
70 GOTO 10
```

Notice the negative time (−50,. . .). This gives a smoother change from one note to the next. Take the minus sign out and listen to the difference.

The inclusion of the noise in the sound gives a more interesting edge to the effect. Alter the ranges in the P loops, and also the size of the STEPs and try some variations. You will tend to find that if you make your noises any shorter the end-of-sound clicks become more of a nuisance.

Beeps

The 99 produces two types of beeps as part of its user guidance system. There is a high-pitched one, when you have an INPUT and when you first turn the machine on, and a lower one that you will hear on an INPUT WARNING, or a BREAKPOINT. If you are developing your own "Home Confuser" you will need to include these.

```
High beep  CALL SOUND(150,1397,1)
Low beep   CALL SOUND(200,220,1)
```

Engines

The 'white noises' (−5 to −8) can be used on their own for space-age engine sounds, or they can be combined with notes to produce different effects. This is the train on SOUNDS.

```
4610  FOR V = 1 TO 10 STEP 9
                    (first time loud (1), next quiet (10))
4620  FOR N = 1 TO 3
4630  CALL SOUND(40,220,V,−7,V+6)
                    (to give an undertone of noise)
4640  NEXT N
4650  CALL SOUND(100,165,V,−7,V+6)
4660  NEXT V
```

The whole routine is then enclosed in another loop to keep it going. The train's 'whistle' also combines noise and notes.

```
4680  CALL SOUND(1000,440,1,622,1,880,1,−1,1)
```

Two different sounds are used to make the 'old car' noise – in fact it uses three sounds.

```
4510  CALL SOUND(10,-3,V)     (Volume also
                               looped)
4520  CALL SOUND(10,-5,V)
```

The third noise is the end of sound 'click', which is very noticable with short sounds.

13
Graphs and maths

While the 99 has an excellent 'number crunching' capability, it isn't so hot on graphic displays, as only low-resolution graphics are available in TI BASIC. This means that the 'graph paper' of the screen is only 24 squares by 32. With high-resolution graphics you might have 'graph paper' as fine as 200 by 250 squares.

This doesn't matter with fixed displays, as you can produce finely detailed pictures using the character definition routines, but it is a little limiting for mathematical displays. However, that said, here is some idea of the sort of things you can do.

Equations of lines

You can get a display for any equation of the type $X = Y * Z$ using this routine.

```
10   INPUT Z
20   FOR Y = 1 TO 24      (to go on every line)
30   C = Y*Z
40   IF C<=32 THEN 70      (it's going off the screen)
50   Y = 24
60   GOTO 90
70   R=25 — Y     (so it works from the bottom)
80   CALL HCHAR(R,C,42)      (prints asterisks)
90   NEXT Y
100  INPUT A$     (wait for it!)
110  CALL CLEAR
120  GOTO 10     (try another number)
```

In normal maths X and Y co-ordinates start at the bottom left and go up and right. The 99's co-ordinates, of course, start at

the top left. You have got two choices – either stand the T.V. set on its side and look at it in the mirror, or swop them around. If you look closely at this program you will realise that much of it is devoted to producing the right type of presentation. Type it in and run it and try entering different numbers, whole numbers and decimals. $X=Y/2$ is the same as $X = Y * .5$. Figure 37 shows (in line form) some of the displays you should see.
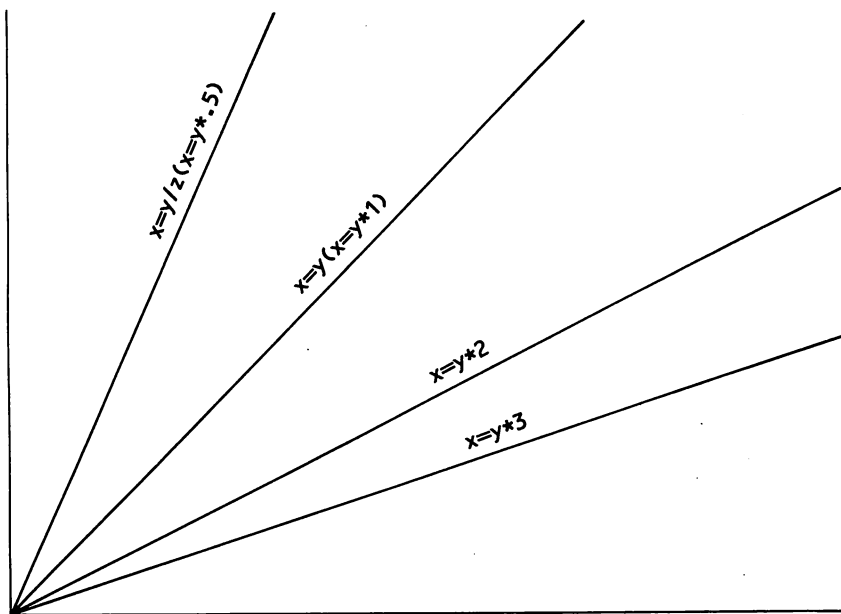
Figure 37

(Note – this program will not work where Z is less than .5, as this gives a value for C, in line 80, of 0.)

More complicated equations of the type $X = Y * Z + A$ can be catered for by a couple of minor additions.

```
15  INPUT A
30  C = Y*Z + A
```

You should now get displays something like this:
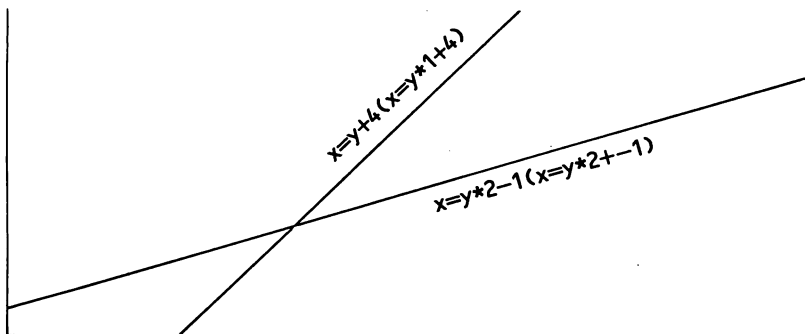
Figure 38

The 99 can also be used to plot SINE waves using a program like this:

```
10 CALL CLEAR
20 FOR C = 1 TO 32 STEP .5    (note the step)
30 R = INT(SIN(C)*5 +15)
40 CALL HCHAR(R,C,42)
50 NEXT C
```

The numbers in line 30 can be varied to produce different effects. The wave will remain lumpy whatever you do.

Angles

For some reason best known to themselves, computers do not measure angles in degrees like most folk do. They measure in RADIANS. An angle of 1 RADIAN is what you have got when the arc is one radius long. (see figure)
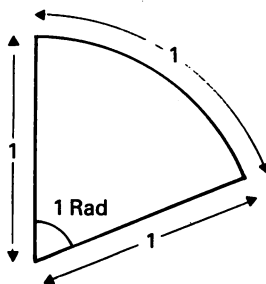
Figure 39

A whole circle therefore has 2 PI radians (approximately 6.3 rads). This is all very inconvenient to those of us used to working in degrees, but you can get the 99 to convert degrees to radians for you. The neatest way to do this is by using the ARCTANGENT function. As any mathematician will tell you, 4 * ATN(1) is equal to PI. (If you want to know why, ask that mathematician.) PI/180 is what you need to turn degrees into radians. Try this:

```
10 DR = (4*ATN(1))/180     (Degrees to Radians)
20 INPUT ANGLE
30 PRINT SIN(ANGLE*DR)
40 GO TO 20
```

Run it and try a few different angles. Sine 30° is .5, sine 90° is 1, sine 60° is .866. Change line 30 to read:

```
30 PRINT COS(ANGLE*DR)
```

and you can get the cosines of angles in the same way.

Sines and cosines can be used to plot 'circles'. (They are not very round)

```
10 FOR A = L TO 6.3 STEP .2     (we will use
                                 radians this
                                 time)
20 R=SIN(A)*5 + 10     (5 to make it bigger, +10 to
                        move it to the centre)
30 C=COS(A)*5 + 15
40 CALL HCHAR(R,C,42)
50 NEXT A
```

Change the numbers in 20 and 30 to alter the size and position.
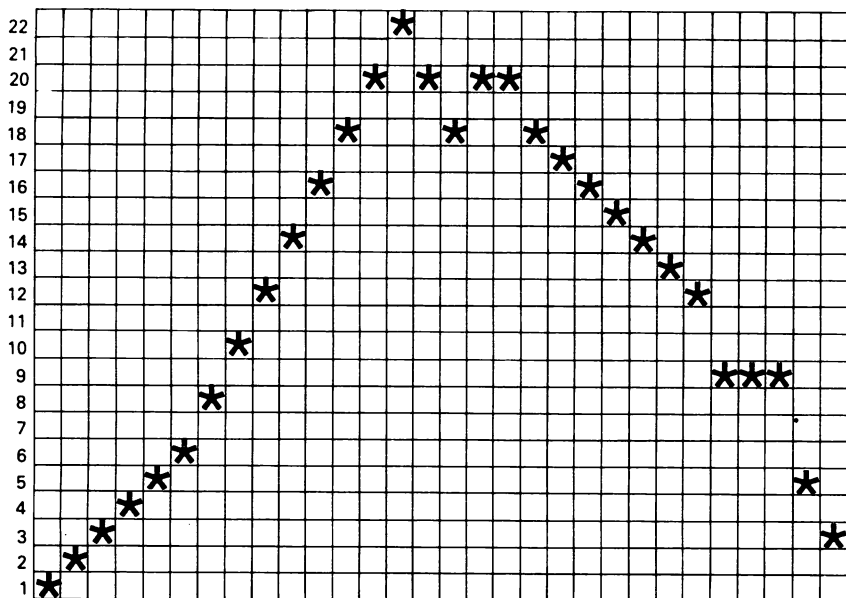
Figure 40
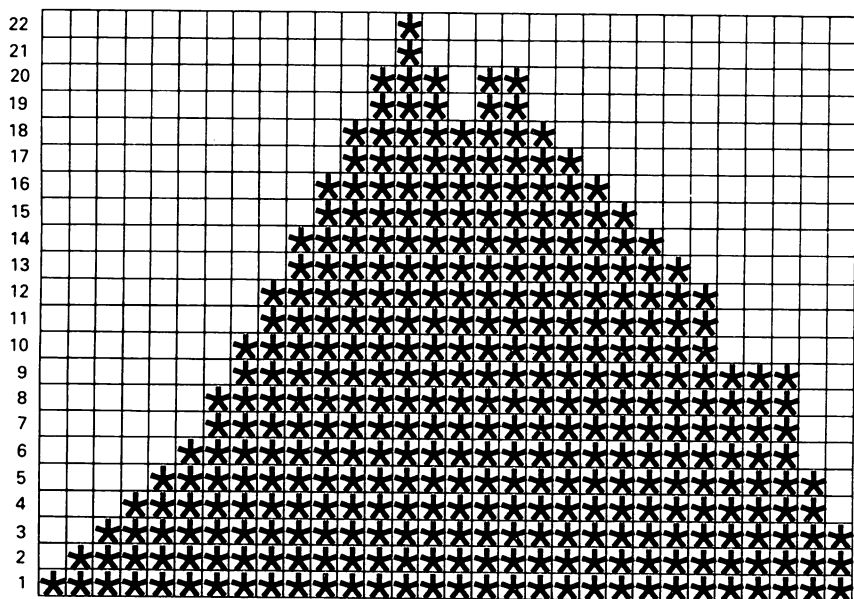
Figure 41

Displaying statistics

Here's a way of displaying figures as graphs. It is suitable for things like monthly sales or production figures. The first half of the program collects in a set of thirty numbers into an array. You will notice that the numbers are all stood on their heads so the graph comes out the right way round. These numbers must all be in the range of 1 to 24 – this is only a simple demonstration after all.

```
10  OPTION BASE 1
20  DIM R(30)
30  FOR N = 1 TO 30
40  INPUT "NUMBER ":R(N)
50  R(N) = 24 - R(N)      (upside down)
60  NEXT N
```

The next part plots the figure across the screen.

```
70   CALL CLEAR
80   FOR C = 1 TO 30
90   CALL HCHAR(R(C),C,42)
100  NEXT C
110  INPUT A$      (hold it!)
```

The graph in figure 40 was produced by this set of numbers:

1,2,3,4,5,6,8,10,12,14,16,18,20,22,20,18,20,20,18,17,16,15,14, 13,12,9,9,9,5,3

A simple conversion will make this program produce barcharts instead of line graphs. We can use the VCHAR routine to drop columns down from the line. All you need to calculate is the length of the VCHAR column. Add this:

```
85  L = 24-R(C)
```
and alter `90  CALL VCHAR(R(C),C,42,L)`

The same set of figures now look like figure 41.

The Record Keeper's Pack goes into far more detail on the display of statistics, and of their analysis, and provides some programs to do it for you.

Maths on the 99

What follows is simply an outline of the different mathematical functions that are available on the 99, and not an attempt to teach you how to use them. Some of them will be dealt with in more detail in later books in this series, as and when they are needed for particular routines.

ABS – The ABSOLUTE value of a number is the value of the number itself, never mind whether it is positive or negative.
    ABS (99) – the ABSOLUTE value of 99 is 99
    ABS (−99) – the ABSOLUTE value of −99 is still 99

ATN – gives you the ARCTANGENT, that is, the angle for a given tangent. The angle will be given in radians. To convert these to degrees multiply by 180/π which is 57.3 (approx.).

COS – gives the COSINE of an angle, where the angle is in radians. If the angle (A) is in degrees you must multiply by π/180. You can use this formula for accurate work:

    COS(A * 4 * ATN(1)/180)

Otherwise, use

    COS(A)*0.02

which is roughly the same.

EXP – the EXPONENTIAL function is the inverse of the LOGARITHM function. In other words, it will convert a natural logarithm back to a decimal number.

INT – gives the INTEGER of a number. The INTEGER is the whole number part, with the decimal lopped off the end. This means that INT always rounds down. To make it round to the nearest, you must use this formula:
    INT(X + .5)

If X was 1.2 to start with, it becomes 1.7, with the extra .5. and is INTEGERED to 1. When X is 1.8 it becomes 2.3 (with the extra) and is rounded to 2.

LOG gives you the natural logarithm of a number.

RANDOMIZE makes the random numbers produced by RND start at an unpredictable part of their sequence, and thus appear truly random. If you do not include a RANDOMIZE line early in a program that uses RND, then the numbers will come out the same every time you run the program.

RND – produces pseudo-random numbers. The range is normally between 0 and 1 but can be pushed into whatever range you need by multiplying, and adding. (See chapter 11 and Pack 1).

SGN – the SIGNUM function looks at the sign in front of a number and tells you if it is positive or negative.

```
PRINT SGN(99)
```

will give you 1. (Positive number)

```
PRINT SGN(-99)
```

will give you −1. (Negative number)

```
PRINT SGN(0)
```

will give you 0

SIN – the SINE of an angle. This assumes the angle is in radians. If it is in degrees, use the same formula as for COS.

SQR – tells you the SQUARE ROOT of a number

TAN – the TANGENT of an angle. Again, normally in radians, use the COS formula for angles in degrees.

14
Why won't it work?

The thing to do when you have a program that doesn't work when you run it for the first time, is keep calm. Don't worry, it happens all the time. It often happens for the silliest reasons. A major cause of program failure is typing error. You spelt a variable name wrongly, or typed the wrong letter for it. You missed the last closing bracket off a line that had lots of brackets. You missed out the space between two words, where your program line spills off the edge of the screen and onto a second line. Whatever the error, you will almost certainly be told what line it's on. Check that you have typed what you meant to type.

Variables can also cause trouble if they are not kept under control. If you suspect that one may be the cause of your bug, then add lines to the program to print it out, whenever it is used or altered in any way. You can then check that it is what it should be. Make the lines something like this:

101 PRINT "N=";N

and you will know what it is there for. It is definitely worth including the variable's name in your print line if you are following two or three different ones at the same time.

TRACE your programs and check that the computer actually goes through the routines you want it to.

Check your error reports using Appendix B or your User's Reference Guide.

If you still can't work it out, SAVE the program on tape, turn the 99 off and go away. When you come back to it again the next day you will look at it with fresh eyes and will be much more likely to spot the bug.

Appendices
A
Glossary

Array   a set of variables having the same name, but with different numbers, e.g. A(1),A(2),A(3), etc. The 99 will set up small arrays automatically, but if you want an array of more than one-dimension, or with more than 10 items, you must organise it yourself with a DIM statement. (See 'Arrays')

ASC   gives you the ASCII code number of a character. (See Pack 1 and 'Strings')

Binary   a system of counting used by computers and people with only two fingers. (See 'Creating your own Characters').

BREAK   used to halt the program at particular places during its run. Restart with **CONTINUE**. (See 'Program Planning 1')

Bug   a fault in the program. Complicated programs very rarely work well first time (or second time, or third time). You have to go through them sorting out minor errors. This is known as 'debugging'.

Byte   a unit of memory. The smallest unit of memory is a BIT – Binary digIT. This is either 1 or 0 (on or off). 8 BITS make one BYTE (a number between 0 and 255). 1024 BYTES make 1 KILOBYTE (1K). The 99 is a 16k computer, so has 16384 bytes of memory. Roughly speaking, 1 byte will hold one character, or BASIC instruction. The average program line will use around 25 bytes.

CALL CHAR   built-in sub-program used for defining characters. (See 'Creating your own Characters')

CALL HCHAR   Horizontal CHAracter Repitition sub-program. For examples of use, see 'Color', 'Strings' and Pack 1.

CALL VCHAR   Vertical CHAracter Repitition. As above.

CALL GCHAR   used for finding out what's where on the screen. (See Game Writer's Pack 1')

CALL KEY used to get information direct from the keyboard, without the user having to press ENTER. See Pack 1 and also 'The Musical Keyboard' and 'Strings'.

CALL COLOR fixes the colours of sets of characters. (See Pack 1 and 'Colour')

CALL SCREEN fixes the screen colour. (See Pack 1)

CALL SOUND the sub-program that produces music and noises. (See Pack 1 and chapters 6, 8 and 12 in this Pack.)

CALL JOYST the sub-program which takes in information from joysticks. (See Game Writer's Pack 1)

Character A letter, number, symbol, or user-defined graphic. Each character is known by its own special ASCII code. ASCII code lists are given in Pack 1 and the User's Reference Guide.

**CHR$** followed by a number (in brackets) will give the character that has that code number. CHR$(65) is "A". (See Pack 1)

CONTINUE will restart the program after a BREAKPOINT report. (See 'Program Planning 1')

Data means information given to the computer to work on. Inputs are Data. **DATA** also refers to the information written into the program for the computer to READ. (See 'Teach your 99 to read')

DELete the function key is used to rub out unwanted characters when writing or editing lines.

DIM short for DIMension. Used to set the size of arrays. (See 'Arrays')

DISPLAY means the same as PRINT. In TI EXTENDED BASIC, DISPLAY allows you to include extra information in your print line.

EDIT the command used to pull a program line back to the workspace for rewriting. Can be more simply replaced by the line number followed by FCTN and E or FCTN and X.

ELSE included in an IF. . .THEN. . . line when you want to include an alternative jump in the line. (See Pack 1)

END Traditionally, the last line in a program is . . .END. Can be used instead of STOP, and works exactly the same.

FOR. . .NEXT. . . loops ways of running a series of numbers through the same part of a program. (See Pack 1)

GOTO or **GO TO** sends the computer to a particular line. This is an UNCONDITIONAL JUMP. The computer must GOTO the line, what ever has happened. (See JUMPS below)

GOSUB sends the computer off to a subroutine. (See 'Program Planning 2')

Hexadecimal number system used in defining characters. Easy to operate if you have 2 tame spiders, or a 16-bead abacus. (See 'Creating your own characters')

IF...THEN... makes the computer jump to a line IF a certain condition , or set of conditions exist. If things aren't right, then the computer will jump to the line number given after ELSE (if used), or simply go on to the next line. IF. . .THEN. . . produces a CONDITIONAL JUMP. (See Pack 1 and 'Random Logic')

INPUT waits for some data to be ENTERED from the keyboard. (See Pack 1)

Jumps where the computer goes off to another line further on, or back up the program, rather than simply to the next line.

an **Unconditional Jump** occurs with GOTO or GOSUB, or ON. . .GOTO (GOSUB)

a **Conditional Jump** is what you get with IF. . .THEN. . .

LEN measures the LENgth of a word or a string variable, by counting its letters; e.g. LEN("METRE") = 5. (See 'Strings')

LET puts a number or a string into a store. It can be omitted. LET A = 5 is the same as A = 5 to the 99. Use it if it makes the program easier to read. (See Pack 1)

LIST makes the program appear on screen. Stop the listing by pressing FCTN and $\boxed{4}$.

LIST 1000 would make the single line appear.

LIST 1000- will LIST the program starting from line 1000. (See Pack 1)

LIST 1000-2000 will list the lines from 1000 down to 2000 and stop there.

Load to transfer a program from tape (or disk) into the computer's memory, using the OLD command. (See below)

Loop where the program runs through the same routine several times. (See Pack 1)

NEW   wipes a program and all its data from memory, ready for a new program. (See Pack 1)

NEXT   see FOR. . .NEXT. . . loops.

OLD   command used to load programs from tape (or disk) into memory. The 99 must know where the program is coming from, as it has different routines to deal with tape and with disk loading. Load from tape using "OLD CS1".

ON. . .GOTO (GOSUB)   a way of controlling jumps. The number in the variable after ON must be part of a simple series, 1,2,3. . . etc, and must not be higher than the number of possible lines to jump to.

   ON K GOTO 100,200,300

will only work as long as either 1,2 or 3 is the value of K. (See Pack 1)

OPTION BASE   fixes the lowest number to be used in an array. Can be either 0 or 1. (See 'Arrays')

POS   finds the POSition of a string inside another string. (See 'Strings')

PRINT   puts characters on the bottom line of the screen.

PRINT " "   prints whatever is in the quotes.

PRINT A+B   works out the sum of the two numbers and prints that.

Print Separators   different sorts of punctuation, to give different screen spacings.

Semi-colons (;)   print things close together.

Commas (,)   leave half-screen spaces.

Colons (:)   push the print position to the next line. (See Pack 1)

READ   tells the computer to collect some information from the DATA list and put it into a store. (See 'Teach your 99 to Read')

REM   short for REMark. Used in programs to write notes to yourself so you know what you are doing. (See Pack 1)

RESEQUENCE   tells the 99 to renumber the program so that the lines are nicely spaced out again. (See Pack 1)

RESTORE   puts the DATA marker back to the beginning of the DATA list. Can also be used to set the DATA marker to particular points in the program, and is very useful if you are

using several different blocks of DATA and cannot tell which will be used when. RESTORE 4000 will make the 99 READ the next DATA item it finds after line 4000. Well worth including a RESTORE . . . (whatever line number) at the start of any section where you are going to do some reading. (See 'Teach your 99 to Read')

RETURN sends the computer back from a subroutine to the main program. (See 'Program Planning 2')

Routine a section of a program that does a particular job. It may be one line long, or hundreds. See also Subroutine.

RUN clears all the variables and starts the program from the beginning. RUN followed by a line number will start the program from that point instead. (See 'Program Planning 1')

SAVE copies a program from the 99's memory onto tape (or disk.) The program in memory is unaffected by SAVE. (See Pack 1)

SEG$ gives a SEGment of a string. (See 'Strings')

STEP sets the difference between the numbers in a FOR. . .TO. . .NEXT. . . series. Miss it out if the difference is 1. (See Pack 1, and 'Morse and More Sounds' for examples of its use.)

STOP stops the program, with a * DONE * report.

STR$ converts a number (or number variable) into a string (variable). (See 'Strings')

Sub-program one written into the 99's operating system, and used with a CALL instruction.

Subroutine a routine which can be re-used as often as need be. The computer goes to it on a GOSUB instruction, and goes back to the main program with RETURN. (See 'Program Planning 2')

Subscript the number in brackets after an array name. It must be a whole number, and not higher than the number set by the DIM instruction. (See 'Arrays')

TAB fixes the column number at which a print item will be placed. (See Pack 1)

THEN see IF. . .THEN. . . above.

TO sets the range of numbers in a FOR. . . TO. . . line. 1 TO 5 means 1,2,3,4,5.

TRACE allows you to see where the program is going while

it runs. (See 'Program Planning 2')

UNBREAK   cancels a BREAK command. (See 'Program Planning 1')

UNTRACE   cancels TRACE. (See above)

VAL   works out the value of any numbers in strings, or in string variables. (See 'Strings')

Variables   a memory store. Number variables store numbers, and string variables store words, letters, symbols or numbers in quotes.

B
Error messages

A list of common Error Messages is given in Pack 1, and a full list can be found in the User's Reference Guide. The following may be useful to you when you are first exploring different areas of programming, as this is the time when you make most mistakes, and when the messages mean least to you.

CALL sub-programs

■ BAD VALUE – you are using a number which is outside the possible range; e.g. a color-set number, or color code above 16, or a row number over 24. Easily happens with the hex-strings (the pattern-identifiers) in CALL CHAR lines. If you type CALL HCHAR, instead of CALL CHAR, you will also see this.

■ INCORRECT STATEMENT – you have most likely mistyped the word after CALL, or your are trying to produce more than three notes, or more than one noise with a CALL SOUND line.

■ MEMORY FULL – and there is no room left for you define a new character.

Sub-routines

■ BAD LINE NUMBER – the line number you have told it to GOSUB to, doesn't exist.

■ MEMORY FULL – you have mistyped and finished up with a line like this:

    100 GOSUB 100

The 99 whizzes round this loop unitl the GOSUB STACK (where it keeps a note of the line numbers it must RETURN

to) is full of 100's. Hence MEMORY FULL.

■ CAN'T DO THAT – the program has arrived at a RETURN without going past a GOSUB. Most likely it ran off the end of the main program and into the first sub-routine.

■ INCORRECT STATEMENT – there is something typed after RETURN, or you have mistyped an ON. . .GOSUB. . . line.

■ BAD VALUE – you have an ON. . .GOSUB. . . line and the number that it is supposed to jump ON is either 0, or too big. If you have 4 possible sub-routines, the only acceptable numbers are 1,2,3 and 4.

Data

■ DATA ERROR The four most likely causes of this are:
1 trying to READ too many items.
2 commas missing between items in the DATA list.
3 mixing strings and numbers so that an item is READ into the wrong type of store.
4 using RESTORE with a line number, and the line number is higher than the last line of the program.

Arrays

Most Array Errors are picked up by the computer while it is checking the program before it actually starts to run. This is the point at which arrays are dimensioned.

■ BAD VALUE – the array has a dimension greater than 32767, or of 0, when the OPTION BASE is 1.

■ CAN'T DO THAT – the OPTION BASE line comes after the DIM line, or you are trying to use more than one OPTION BASE line in the program.

■ INCORRECT STATEMENT – check the typing of your DIM line, you have got an invalid name, a bracket missing, or too many dimensions. (Max. 3 in TI BASIC)

   Check also the OPTION BASE line. It might be mistyped, or lack a number (0 or 1) at the end.

■ NAME CONFLICT – You have got two arrays with the

same name, or, more likely, an array and a simple variable with the same name. You cannot have A(10) and A in the same program.
■ MEMORY FULL – you are too ambitious. Try again with a smaller array.
You may also get this when the program is running:
■ BAD SUBSCRIPT – subscripts must be whole numbers, and within the dimensions of the array.

String handling

■ BAD ARGUMENT – most likely with ASC and VAL where you are asking the 99 to do something to a string that doesn't exist at all, or is empty. ASC(A$) won't work if A$ is "" at the time.
■ BAD VALUE – this might crop up with CHR$,POS and SEG$ if the number in brackets is 0 or more than 32767.
■ STRING – NUMBER MISMATCH. You have got your strings crossed and are asking the 99 to perform a string function on a number, or a number function on a string. ASC(N) won't work. Neither will CHR$(N$). Remember you can always convert a number to a string by using STR$, and that VAL will give you the number value of a string (as long as it is a number – e.g. "99").

C
Keyboard plan (CH$128 to159)

1	2	3	4	5	6	7	8	9	0	=
							158	159		157

Q	W	E	R	T	Y	U	I	O	P	/
145	151	133	146	148	153	149	137	143	144	

A	S	D	F	G	H	J	K	L	;	ENTER
129	147	132	134	135	136	138	139	140	156	

SHIFT	Z	X	C	V	B	N	M	,	.	SHIFT
	154	152	131	150	130	142	141	128	155	

ALPHA LOCK	CTRL	SPACE	FCTN

CHARACTERS 128 TO 159 FROM THE KEYBOARD

Use this when you have defined your own characters to numbers in this range. Hold down CTRL and press the key to get the character number shown. CTRL and □ A is the same as CHR$(129)

   Characters called up this way can be included in the LISTS and used in PRINT lines. (See Pack 1).

English
Limited warranty — U.K.

This Texas Instruments (TI) warranty extends only to the original consumer purchaser.

1. The electronic and mechanical components of the product are warranted for a period of twelve (12) months from the date of original purchase under normal use and service against defective materials or workmanship. This warranty is void if the product has been damaged by accident, unreasonable use, neglect, improper service or other causes arising out of defects in materials or workmanship.

2. Any implied warranties arising out of the sale are limited in duration to the above twelve (12) month period.

3. ** During the above twelve (12) month period, the product will be repaired or replaced with a new or reconditioned one of equivalent quality at Texas Instruments option, without charge to the purchaser when the product is returned, with proof of purchase date to a Texas Instruments retailer. The repaired or replacement product will continue to be warranted until the end of the original twelve month period or ninety (90) days from the date of repair or replacement, whichever comes later.

4. Important notice of disclaimer regarding the software programs and book materials — read this carefully before purchasing the console and/or programs.

T.I. does not warrant that the software programs and book materials will be free from error or will meet your specific requirements. Each user is notified that the programs may contain errors and assumes sole responsibility for any decision made or actions taken based on information obtained from using the programs. No information given concerning the utility of the programs is to be construed as an express or implied warranty.

5. ** In no event shall T.I. be liable to anyone for special, incidental, or consequential damages in connection with or arising out of the purchase or use of the console. Hardware and/or programs and the sole and exclusive liability of T.I. shall not exceed the purchase price of the console, hardware and/or programs. T.I. shall not be liable for any claim of any kind whatever against the user of the programs by any other party.

** **Paragraphs 3 and 5 shall not affect the statutory rights of the consumer as defined in the consumer transactions (restrictions of statements) order 1976, as amended.**

- -

TEXAS INSTRUMENTS

Model	Serial N°		1 □ MR.	2 □ Miss, Mrs	3 □ Company
Modell	Serien Nr.		Herr	Frau, Fräulein	Firma
Modèle	N° de série		Monsieur	Mme, Melle	Societé
Modello	Numero di serie		Sig. re	Sig. ra, Sig. na	Ditta
Malli	Serie nr.		Dhr.	Mevr., Mej.	Bolag.
Modelo	Sarja No		Herr	Fru., frk.	Yhtiö
	No de serie		Hr.	Nti, Rva	Empresa
Last Name	N° de serie		Hra.	Senhora, Menina	
Familienname			Senhor	Sra. o Srta.	
Nom			Sr.		
Cognome					
Achternaam			First Name		
Efternamn			Vorname		
Efternavn			Prénom		
Sukunimi			Nome		
Ultimo nome			Voornaam		
Appellidos			Förnaam		
			Fornavn		
			Etunimi		
Address			Primeiro nome		
Adresse			Nombre		
Indrizzo					
Adres					
Gatuadress					
Osoite					
Endereco					
Dirección					

Date, Datum, Data, Päivämäärä, Dato, Fecha	Town	P.O. Code	Country
	Ort	Postleitzahl	Land
	Ville	Code Postal	Pays
	Città	Codice Postale	Paese
	Stad	Postcode	Maa
	By	Postnr.	Pais
	Kaupunki	Postinumero	
	Citade/Vila	Zona postal	
	Ciudad	D. Postal	

TEXAS INSTRUMENTS
HOME COMPUTER

COLLINS
MICROSOFTWARE