MASTERING THE TI-99

Peter Brooks

Mastering the TI-99

Mastering the TI-99

Peter Brooks

MICRO PRESS

First published in 1984 in the United Kingdom by
Micro Press
Castle House, 27 London Road
Tunbridge Wells, Kent

British Library Cataloguing in Publication Data

Brooks, Peter
Mastering the TI-99.
1. Texas TI-99 (Computer)
I. Title
001.64'04 QA76.8.T/

ISBN 0-7447-0008-6

Typeset by Keyset Composition, Colchester
Printed and bound by Mackays of Chatham Ltd

Contents

Acknowledgements

I owe thanks to more people than I could mention, but I would like gratefully to acknowledge the assistance of the following people during the period leading up to the preparation of this book:

Elizabeth, for putting up with the incessant chatter about computers; my parents, who aided me in the purchase of my equipment; Paul Dicks, for having patiently endured endless telephone calls over the last three years, for having allowed me to shoot my mouth off in public through the now-defunct Tidings, and for having placed me in a position to write this book; Trevor Hood, of Castle House, for comments and invaluable help in the production of the manuscript; Matthew Childs, for having checked some of the routines for me; George Moore, for constructive criticism during editing and proofing; and anyone who ever had a software problem and brought it to me — I always found the experience beneficial.

Peter Brooks
Oxford
October 1983

Introduction

For the newcomer, home computing can be a veritable quicksand of jargon and strange concepts. Confusion and frustration are the two most common conditions suffered by those who have 'taken the plunge', bought a home computer, and are engaged in a desperate struggle to understand their machine and its manual, and perhaps even to find a practical use for it. (These conditions seem to be experienced more severely in the U.K. by TI-99/4 and 99/4A owners, because the machine appears to be marketed in Europe with the American consumer in mind.)

Their efforts are hampered by comparatively minor problems — printing errors or omissions in manuals, books, or magazines, for example — and often the solutions, when uncovered, are so simple as to cause much smacking of foreheads and utterances of self-deprecation.

Even after struggling through the early stages, however, the difficulties are not reduced, as attempts are made to translate a program written for one machine, in a supposedly 'common' language, into TI BASIC.

You have become the owner, possibly proud, of a Texas Instruments Home Computer. If you bought your machine some time ago, you will already be aware that it has a number of annoying deficiencies when compared with its competitors. In the majority of cases, these can be overcome through careful programming, some examples of which are presented in this book. In others, sadly, the only alternative is to keep throwing money at the problem(s) by buying more and more equipment, which can often result in your being able to do less than before.

There are two things to bear in mind when close to capitulation: firstly, that there is NOTHING which cannot be understood by you eventually; if an explanation doesn't 'click', that is not your fault. Either ask, if you can, for it to be re-expressed, or look elsewhere for enlightenment. Secondly, perseverance is the name of the game. No matter how intractable the problem may appear to be, someone, somewhere either has the solution, or can point you in the right direction toward one.

Become a member of a local computer club if there is one, and join one (or more) of the specialist groups which cater specifically for the TI-99/4 and 99/4A. Read computer magazine articles, even if you don't entirely follow them, and don't restrict your subject matter to the TI computer alone. Tricks on other machines may not be directly applicable to the Texas computers, but they can trigger ideas which end up as useful routines or techniques.

Keep on coming back to programs or ideas which you have worked on in the past: you'll find that the experience gained in just a few weeks can help you to improve on what went before, even highlighting errors which you may have missed.

Above all, if you think that you have found something useful, interesting, or even staggeringly amazing, don't keep it to yourself. Discuss it with fellow owners, write to magazines about it, but never think that it is unimportant. Even if it does turn out to be so, you will still have learned something, and that is most important.

This book lies somewhere between a vast tome of totally understandable explanations, and a packed, incomprehensible reference work, and as such is of necessity something of a compromise. The one thing it cannot do is to answer every question that TI owners might pose, so if you have anything which you would like to discuss, any questions that you would like help in answering, I am happy to accept any correspondence. You can contact me via the publisher, CASTLE HOUSE, but please include a stamped, self-addressed envelope.

And finally, if Darth Vader on the front cover resembles me, it is purely coincidental!

Translation

The task which many new owners most often undertake is to try to translate into TI BASIC a program written for another machine and published in a book or magazine.

Although BASIC is a common language, you may already have realised that there is BASIC, and then there is BASIC. There are almost as many dialects of BASIC as there are machines, and although each dialect has its origins in the early American 'Dartmouth' BASIC, each has been added to and 'improved' in order to enhance the facilities available. Translation from one dialect into another can be fraught with difficulties, because of subtle language differences and printing errors or omissions in the published material. Sometimes the only alternative to giving up is to buy the owners' manual for the machine concerned, and attempt to use that as a source of reference. The problem is that if you use that approach you could end up with hundreds of manuals and still no translated program.

This chapter looks at some of the more common BASIC reserved words which are used on other machines, and, where possible, gives the alternatives which TI BASIC can employ. It also gives some suggestions on how to cope with some of the 'constructions' used in other BASICs: for example, 'IF A = B THEN C = D' or 'IF A = B THEN PRINT "HIT"'.

In addition there are differences between the 99/4 and 99/4A which need highlighting, especially in respect of the keyboard and keycodes, and, to a lesser extent, the variable names.

To begin with, we will look at some of the commands which are likely to be found in published listings of programs written for other machines. There are certain commands which cannot be implemented in TI BASIC without additional equipment — some uses of PEEK and POKE, for example — and unless you are particularly adventurous or masochistic it is best to avoid those programs which make use of many PEEKs or POKEs, especially if those commands appear to form a central part of the structure of the program. You might well be better off in such cases if you wrote a TI BASIC version from scratch. Some magazines, notably *Personal Computing Today*, actually provide general translation information with every program that they publish; would that all magazines did so.

The following discussion is by no means exhaustive, and with the rapidly changing state of the microcomputer market the subject of translation is one which will gain prominence as time goes by. Once a year a magazine called *Personal Computer World* publishes a large reference sheet of BASIC reserved words for a number of the most popular machines; despite the recent high sales of the TI-99/4A it has never figured in this chart, but it can be a source of valuable (if occasionally misleading) information.

The BOOLEAN Operators

These are: AND, OR, XOR, and NOT. Their function is discussed in the chapter on **Jargon**, while their TI BASIC equivalents are explained in more detail here.

Briefly, the operators AND, OR, and XOR may be seen in any of the following contexts, using AND for the examples:

```
1) IF A = B AND C = D THEN 1000
2) IF A = B AND C = D THEN E = F
3) A = B AND C
4) PRINT A AND "ON TARGET"
```

The equivalents to AND, OR, and XOR are implemented in TI BASIC using brackets (parentheses) to form *'Relational Expressions'*, with the operators '*', '+', and '−' respectively. Example (1) becomes:

```
IF (A = B)* (C = D) THEN 1000
```

The expressions (A = B) and (C = D) are evaluated by the computer, which returns a value of −1 if the expression is TRUE; that is, if A does equal B, or if C does equal D; and 0 (zero) if FALSE. In the above example, then, we have four possible combinations:

```
TRUE* TRUE     or   −1* −1   which is +1
```

```
TRUE* FALSE   or  -1*  0  which is  0
FALSE* TRUE   or   0* -1  which is  0
FALSE* FALSE  or   0*  0  which is  0
```

Of these combinations, only the first evaluates to a 'non-zero' result. (Note that any number multiplied by zero is still zero.)

The 'IF . . . THEN . . .' statement in TI BASIC is really 'IF . . <> 0 THEN . . .', where the '<> 0' is said to be 'implied'. Of the four results possible from the pair of expressions above, only TRUE * TRUE gives a result which '<> 0', or is non-zero.

If example (1) had been:

```
IF A = B OR C = D THEN 1000
```

it could have been implemented in TI BASIC thus:

```
IF (A = B) + (C = D) THEN 1000
```

In this instance, the jump to line 1000 will take place if either A = B or C = D; we have four possible combinations again:

```
TRUE + TRUE    or  -1 + -1  which is -2
TRUE + FALSE   or  -1 +  0  which is -1
FALSE + TRUE   or   0 + -1  which is -1
FALSE + FALSE  or   0 +  0  which is  0
```

Remember that we are looking for a non-zero result, so the first three combinations will cause a jump to line 1000. This imitates the function of OR, otherwise known as INCLUSIVE OR, because it means 'if one or the other or both are TRUE'.

XOR, or EXCLUSIVE OR, which has '−' as its TI BASIC equivalent, operates slightly differently:

```
IF A = B XOR C = D THEN 1000
```

It means 'if either one or the other BUT NOT BOTH are TRUE', so in terms of the example which we have been discussing, it becomes:

```
IF (A = B) - (C = D) THEN 1000
```

Again we have four possible combinations of TRUE and FALSE:

```
TRUE - TRUE    or  -1 - -1  which is  0
TRUE - FALSE   or  -1 -  0  which is -1
FALSE - TRUE   or   0 - -1  which is +1
FALSE - FALSE  or   0 -  0  which is  0
```

This time note that the only two results which are non-zero are those given by TRUE − FALSE and FALSE − TRUE, which is exactly as we stipulated above.

These TI BASIC versions of AND, OR, and XOR are generally best used with only two expressions at a time, although it is entirely feasible, for AND and OR at least, to use them with more than two:

```
IF A = B AND C = D AND E$ = "FIRE" THEN 1000
```

which becomes

```
IF (A = B)*(C = D)*(E$ = "FIRE") THEN 1000
```

With care they can be mixed:

```
IF A = B AND C = D OR E$ = "FIRE" THEN 1000
```

which becomes:

```
IF (A = B)*(C = D) + (E$ = "FIRE") THEN 1000
```

BUT . . . you must be very careful to make sure that you have understood exactly what conditions must be fulfilled before the jump to line 1000 can be made.

XOR cannot really be used with more than two expressions, partly because of the fact that its purpose is to operate on only two values, and partly because its TI BASIC equivalent ceases to be accurate.

The mixing of expressions can become quite complicated, and when it reaches this sort of level:

```
IF (A = B OR C = D) AND (E = F OR G = H) OR (I = J AND
K = L) THEN 1000
```

it is time to stop and decide whether to continue using the TI BASIC equivalents or to translate in terms of multiple 'IF . . . THEN . . .' statements.

Continuing with the discussion, this time of example (2), it is not possible in TI BASIC for anything other than a line number to follow 'THEN' (or 'ELSE'), so the statement has to be split up:

First line:	IF A = B AND C = D THEN Second line ELSE Third line
Second line:	E = F
Third line:	continuation of program

We can now apply our TI BASIC equivalents (for AND, OR, and XOR) as previously discussed:

First line:	IF (A = B) * (C = D) THEN Second line ELSE Third line
Second line:	E = F
Third line:	continuation of program

and so on. As you become more adept both at translating and at finding methods of shortening programs, you will be able to reduce the statement to a simple 'IF . . . THEN' without the need for an 'ELSE'.

It is feasible to translate the above example as one line even in TI BASIC:

First line: E = E * (1 − (A = B) * (C = D)) + F * ((A = B) * (C = D))

but as you can see, it can be horrendously difficult. Just to go over the operation of that line: if both expressions are TRUE, both will be evaluated as −1 by the computer. We can rewrite the line thus:

First line: E = E * (1 − (−1) * (−1)) + F * ((−1) * (−1))

This simplifies to:

First line: E = E * (1 − (−1 * −1)) + F * (−1 * −1)

Simplifying again:

First line: E = E * (1 − (+1)) + F * (+1)

And again:

First line: E = E * 0 + F

which means that the variable E is assigned the number represented by the variable F. If you have the stamina: if either or both of the expressions is FALSE, the equation will evaluate eventually to this:

First line: E = E * (1 − 0) + F * 0

which is:

First line: E = E + 0

which results in the variable E retaining its original value.

It is complex, and it works, but I would never recommend it for practical use.

Example (3) is difficult to follow, and it does depend upon the dialect of BASIC which uses it. Generally, the computer will work on the BINARY equivalent of the numbers represented by variables B and C. The two values will be ANDed (or ORed, or XORed) bit by bit (see the **Jargon** chapter), and the resulting binary number is translated back into decimal and assigned to variable A. The reasons for wanting to do this can be quite complex and are beyond the scope of this book to discuss, as they usually apply to *Machine Language* programming.

Having said that, example (3) can actually be translated into TI BASIC, but the procedure is long and involved as well as being slow, and would require a chapter all to itself. It is best to avoid translating a program which has statements of the type shown in example (3). If you have the patience (and the funds) a second, more comprehensive book is planned which will indeed spend a chapter on explaining how to translate this statement.

Example (4) is a rare form, found almost exclusively in Sinclair BASIC, and can be broken down firstly into:

First line:	IF A = a value THEN Second line ELSE Third line
Second line:	PRINT "ON TARGET"
Third line:	continuation of program

and then subsequently as discussed previously. The value which the variable A must represent is dependent upon the type of BASIC used by the respective computer: some might use 0, others 1 or −1. The translation is similar to that for example (2).

The Boolean operator NOT is seen in Sinclair BASIC programs and occasionally in others (e.g. TI Extended BASIC). Generally it is equivalent to '<>' or 'is not equal to'. For example:

```
IF NOT A = 10 THEN 1000
```

which is equivalent to:

```
IF A <> 10 THEN 1000
```

NOT also has another use:

```
A = NOT B
```

NOT is called an 'inverter', and operates on the binary equivalent of the decimal number represented by the variable B. Its use makes all the binary ones into zeros and vice versa, and the resulting number is translated back into decimal and assigned to the variable 'A'. The effect is known as ONES COMPLEMENT in machine language programming, and although a detailed discussion is beyond the scope of this book, it can be implemented in TI BASIC. ONES COMPLEMENT is a step towards producing a negative number in binary notation. The actual procedure is called TWOS COMPLEMENT, and consists of ONES COMPLEMENT followed by the addition of binary 1. ONES COMPLEMENT is therefore the negative number minus 1. In TI BASIC this is simply:

```
A = -B - 1
```

You are unlikely to encounter this very often if at all, mainly because it serves little purpose. What you are likely to encounter, in Sinclair BASIC, primarily for the ZX81, is:

```
LET A = NOT PI
```

This peculiar statement is in fact one of a number of memory-saving tricks on the ZX81. In Sinclair BASIC, PI is the constant 3.14159 . . . and an evaluation of NOT PI gives zero as its result. It is, in terms of memory saved, far more economical on the ZX81 than:

```
LET A = 0
```

You will also see its counterpart:

```
LET A = SGN PI
```

which is the same as:

```
LET A = 1
```

A discussion of the reasons why the seemingly longer NOT PI and SGN PI are actually shorter than 0 or 1 in Sinclair BASIC is, as usual, beyond the scope of this book.

Other BASIC Functions

The command ACCEPT AT is similar in function to INPUT AT; for further details on INPUT AT see INPUT and PRINT, and for a translation of ACCEPT AT using a TI BASIC subroutine see the **Hints and Tips** chapter.

The ASC() function in Sinclair BASIC is CODE(), where the ZX81 uses a non-ASCII code system.

The command CALL, although used to similar purpose on the 99/4 and 99/4A, is in fact invoking an *Assembly Language* subroutine; it has other titles: SYS, EXEC, USR, and so on, and unless you are given specific information with the listing as to what the subroutine does, it is not worth while trying to implement in TI BASIC.

Similarly, CHAIN is another command which TI BASIC cannot imitate — it permits one program to load and run another, wiping out the 'calling' program as it does so. This is a fairly powerful facility, often used to make up for small memory capacity, as it allows you to run a program which would not normally be able to sit in its entirety in the available memory.

Although CHR$( ) is available in TI BASIC, in some BASICs it is used with PRINT to perform screen or printer functions. A range of different numbers can be seen with CHR$( ), some clearing the screen, others causing a scroll, while yet others place the cursor at the top left-hand corner of the screen — called 'HOME'. The numbers and their functions seem to vary and it is best to try to guess from the context in which they are used as to their function.

CLEAR is another function which cannot be implemented easily in TI BASIC: it sets all numeric variables to zero, and all string variables to 'null'. The only way to imitate this action is to have a section of the program set aside to perform the function, perhaps as a subroutine, for all the variables which you have used.

Even the command CLOSE, which is present in TI BASIC, can be used differently on other machines. For the 99/4 and 99/4A, file #0 accesses the screen for OUTPUT and the keyboard for INPUT, and it cannot be opened, closed, or altered in any way. (The **File Handling** chapter gives a few more details.) However, CLOSE#0 may be used on some machines to close ALL open files, a facility which can only be imitated in TI BASIC by systematically closing all files by # number. CLOSE alone may also have a similar function and be imitated similarly.

For information on DEEK and DOKE see PEEK and POKE later.

DEF exists in many different forms with different capabilities:

some BASICs use DEF FN, permitting the use of FNA to FNZ, while others may allow the use of several 'parameters' — e.g. DEF A$(B, N, M) — against TI BASIC's one. You may end up using the function as a subroutine instead as the only practical way of implementing it. For example, you could implement a function like:

```
DEF A$(B, N, M) = SEG$(STR$(B), N, M)
```

which might perform the 'slicing up' of a number, simply thus:

```
A$ = SEG$(STR$(B), N, M)
```

We will examine the string functions more fully later.

DIM is not always as straightforward as it looks, either. In some BASICs you need to DIMension string space just for ordinary strings. Sinclair BASIC's DIM A$(50) can be either a single string with just 50 characters (compared with our 255), or it can also be manipulated as if it was a string array of 50 elements, each having a maximum of one character. Some BASICs, like that used by Tandy's TRS 80, may not require you to DIMension arrays of over 11 elements (0 to 10, remember), as TI BASIC does, and to complicate matters, it may not be possible in the other dialects of BASIC to have array subscripts of zero (we can choose with OPTION BASE). What is more, it may also be possible to DIMension the arrays using variables:

```
DIM ST4$(A, B)
```

where A and B have been assigned values either by the program or by INPUTting from the keyboard or files. This is not possible in TI BASIC without additional equipment (e.g. MiniMemory module). A TI BASIC DIM needs actual numbers within the brackets.

EVAL( ) is a function which is very powerful, existing in the form VAL( ) on the Sinclair computers, and lacking on the 99/4 and 99/4A. Although TI BASIC has VAL( ), it is a very limited form, and does not compare with EVAL( ), which functions with strings and string variables, allowing you to enter an expression which will be EVALuated, using current values for any variables:

```
A = EVAL("SIN(C / 9)* 40")
```

or:

```
A = EVAL(G$)
```

where G$ holds an equation or expression. It can only be implemented on our computer by directly entering the equation or expression as a statement with a line number — not nearly as flexible, as any editing causes not only all program variables to be reset to zero or null, but also prevents a program from being CONTinued.

The function FRE is not one usually seen in programs — it can also appear as SIZE, MEM, etc., and refers usually to the amount of memory available. More often than not it is a way of checking to see whether the data being generated during the running of a program is coming close to filling the available memory. It doesn't have a programmable equivalent in TI BASIC, although it is possible, after a fashion, to discover the memory usage of a program: see the chapter on **Hints and Tips**.

GET is a function which is similar to INPUT when used with either disk or tape files, but on occasion can seem more like CALL KEY( ). It is best to try to discover from the context in which it is used what the most suitable way to translate it might be.

Even GOSUB is not immune to variation. On some machines you can use numeric variables and even expressions:

```
GOSUB A + 7* C
```

In one form of BASIC, line numbers can also have letters associated with them, and it is possible to GOSUB to these letters:

```
300  GOSUB t
—
—
—
400t PRINT "READY TO FIRE!"
410  RETURN
```

GOTO is similarly treated. In fact, the use of equations with GOTO and GOSUB is very similar to our ON . . . GOTO and ON . . . GOSUB, and you will need to calculate carefully, using the possible values for any variables involved, all the line numbers which result. These can then be put in the list which goes with the ON . . . command. One warning, though. The use of equations could conceivably produce such a long list of line numbers that you would not be able to fit them all in the single ON . . . statement, and you would then need to look at reducing the equation into two parts (or more), a technique which requires the use of IF . . . THEN . . ., and is beyond the scope of this book.

IF . . THEN . . . ELSE is usually more complex and flexible in other BASICs, allowing the 'nesting' of tests:

```
IF A = B THEN IF C = D THEN IF A$ = "E" THEN PRINT
"HIT!" ELSE 400 ELSE 800 ELSE PRINT "MISSED!"
```

This needs some explaining as far as translation is concerned, so let us begin by isolating the different levels of IF . . . THEN:

```
IF A = B THEN
  IF C = D THEN
    IF A$ = "E" THEN PRINT "HIT!" ELSE 400
  ELSE 800
ELSE PRINT "MISSED!"
```

The technique is to place every separate 'IF' on a new line, 'indenting' each one a little further in, until you begin to encounter 'ELSE's, whereupon you begin working back out again.

Let us take a step away from BASIC and towards English:

```
1) IF A = B THEN (4) ELSE (2)
2) PRINT "MISSED!"
3) GOTO (7)
4) IF C = D THEN (5) ELSE 800
5) IF A$ = "E" THEN (6) ELSE 400
6) PRINT "HIT!"
7) continuation of program
```

If you examine the flow of the routine above you will find that it is the same as for the earlier indented version, except that actual line numbers are not in brackets. The numbers in brackets will be turned into line numbers by you to fit your translated program.

Furthermore, it can be possible, in other BASICs, to use GOSUB after THEN or ELSE, and the translation is similar to that for IF . . . THEN E = F discussed earlier:

```
1000 IF A = B THEN GOSUB 750
```

becomes:

```
1000 IF A = B THEN 1001 ELSE continuation of program
1001 GOSUB 750
```

or perhaps better:

```
1000 IF A < > B THEN continuation of program
1001 GOSUB 750
```

It is not unusual for 'multiple statements' to be involved with IF . . . THEN and this can often require very careful translation. You may only be aware of multiple statements by their separation from each other by single or double colons for example:

```
1000 A = 17::T = 21::GOSUB 750::GOTO 250
```

In these cases you must simply separate out each individual statement and assign a line number to it, taking care that you don't overwrite any existing lines:

```
1000 A = 17
1001 T = 21
1002 GOSUB 750
1003 GOTO 250
```

The command IN is a difficult one to attempt to translate, as by and large it is used with 'ports', or communication with equipment outside the computer (peripherals). It could be used to scan joysticks or paddles, in which case translation can be effected with CALL JOYST( ), although you might have some difficulty when it comes to deciding what values any subsequent equations might be working on.

INKEY and INKEY$ are often seen, and they are counterparts to CALL KEY( ). INKEY$ simply returns the actual character of the key pressed rather than its ASCII code, and is usually simple to translate. For example:

```
100 IF INKEY = 0 THEN 100
```

or:

```
100 IF INKEY$ = "" THEN 100
```

are the same as:

```
100 CALL KEY(O, K, S)
101 IF S = 0 THEN 100
```

Some BASICs have a time limit for INKEY and INKEY$, using instead GET as the equivalent to CALL KEY.

INPUT can appear in a slightly different form. In one BASIC it is possible to use:

```
900 INPUT "NAME :": N$ ; "ADDRESS :": A$ ; "AGE (yr) :":
    A
```

which has to have either separate INPUTS, or a single INPUT of the type:

```
900 INPUT "NAME, ADDRESS, and AGE (yr) :": N$, A$, A
```

Take note of the separators used too; in some forms of INPUT a semi-colon or a comma is used where we would use a colon, and you could find yourself typing the wrong separator. INPUT can sometimes be seen with AT or USING; for further details see PRINT.

A string function you will see on many other computers is LEFT$( ), which is one of three concerned with string 'slicing', but which are not all exactly the same as SEG$( ), and which are presented here in alphabetical order. LEFT$(string, length) extracts the string segment which starts at character position 1, and of a specified length. It is directly translated as:

```
SEG$(string, 1, length)
```

In Sinclair BASIC this appears as:

```
A$(1 TO length)
```

or:

```
A$( TO length)
```

where you would substitute the required variable for A$.

LIST with a specified line number, or range, might possibly be found within a program, as in other BASICs it is permitted. LLIST is likely to be used to produce a listing on a printer.

The logarithmic function LOG( ) can be a little confusing. There are two kinds of logarithm, one called *Briggsian*, the other *Napierian*. The first is based on 10, while the second is based on 2.7182818285, and the confusion can arise because the standard notation for Briggsian logarithms is 'log', while for Napierian it is 'ln' or 'loge'; on most computers although the notation is LOG, the function is in fact Napierian. To the best of my knowledge only the BBC computer uses the correct notation.

Should you wish to obtain the Briggsian logarithm of a number, the following expression will convert from Napierian:

```
DEF BLOG(X) = LOG(X) / LOG(10)
```

The antilogarithm for this defined function is simply:

```
10^ X
```

MEM is an available-memory function; see also FRE and the **Hints and Tips** chapter.

The second of the three string functions mentioned earlier is MID$( ), and on most machines it appears to equate directly with SEG$( ); the format is the same:

```
MID$(string, start, length)
```

although the form:

```
MID$(string, start, end)
```

has been known.

NEXT (of FOR . . . TO . . . STEP . . . NEXT fame) in some cases uses a form which is less demanding than that in TI BASIC; you may find this:

```
FOR I = 1 TO 10
—
—
NEXT
```

which doesn't require the variable to be specified. Translation is straightforward. Some argue that this leads to sloppy programming and difficult debugging. Another form, less criticised apparently, is:

```
FOR I = 1 TO 10
FOR J = 1 TO 20
—
—
NEXT J, I
```

ON ERROR, or ON ERR, are not present in TI BASIC, and are a way of preventing the computer from calling a halt every time an error occurs. It can also appear as TRAP, apparently, although I have never seen that form. It is not possible to emulate, and only by careful programming (i.e., attempting to cater for every eventuality) can you get round its use.

OUT, the counterpart of IN (see earlier) is another which, under exceptional circumstances, can be implemented in TI BASIC, but as usual it depends very much on the circumstances in which it is used. One function might be to transfer a character to a peripheral, so if you know that the peripheral is a printer, then you may be able to make a reasonable translation through the use of OPEN, PRINT, and CLOSE. If you have a printer, that is. It might also be used to operate a sound generator, in which case if you can obtain details about the type of sound produced then you might be able to use CALL SOUND( ). In some instances, OUT might also clear the screen.

PAUSE is similar to WAIT (see explanation later), and under certain circumstances it can be used to halt execution until a key is pressed. That function can be imitated by:

```
300 CALL KEY(O, K, S)
310 IF S < 1 THEN 300
320 continuation of program
```

PEEK and POKE, and their companions DEEK and DOKE, are not available in TI BASIC, and generally cannot be imitated. However, under certain circumstances they do have counterparts. The distinction between PEEK and POKE and DEEK and DOKE is minimal — on one or two machines PEEK and POKE are used with hexadecimal values instead of the usual decimal, and DEEK and DOKE were created to cope with decimal values, and occasionally DEEK and DOKE represent *double* PEEKs and POKEs — but for the most part you will see only the standard use of PEEK and POKE. PEEK allows you to copy the decimal equivalent of the contents (usually 1 byte) of a single address (location) in memory ('peeking' at it), while POKE is used to place a number in a particular location, replacing the number which was previously stored there. Their forms are:

```
1) A = PEEK(B)
2) POKE(C, A)
```

which are: (1) copy into A the contents of an address in memory specified by B, and (2) put the value given by A into the address given by C.

Now, 99% of the time PEEK and POKE cannot be imitated in TI BASIC. They can be used on other machines to clear the screen, find the current position of the last PRINT, discover whether a key is being pressed, operate a sound generator, specify a colour, place a machine code program into a specified area of memory, or re-write part of the BASIC program while it is running.

But 1% of the time PEEK and POKE are used to 'manipulate the display file', which to you and I means using CALL GCHAR( ), or CALL HCHAR( ) and CALL VCHAR( ) with single characters. For example:

```
POKE(17716, 65)
```

which might put the letter 'A' (ASCII code 65) into the top left-hand corner of the screen (perhaps memory location 17716, which is used purely as an example), would be translated into TI BASIC like this:

```
CALL HCHAR(1, 1, 65)
```

Similarly,

```
H = PEEK(17716)
```

which would copy the value it found in location 17716 into the variable H, would be:

```
CALL GCHAR(1, 1, H)
```

To confuse matters further, on at least one machine the symbols '!' and '?' may be used instead, the former being equivalent to POKE. You must be careful, though, since there are older machines which permit the use of '!' for PRINT, and '?' for INPUT.

PRINT can sometimes appear with other 'reserved words':

```
PRINT AT( )
PRINT USING
```

PRINT AT can be emulated using a short routine which can be found in the **Hints and Tips** chapter; PRINT USING is a way of 'formatting' the items being printed — for example, lining up all the decimal points of a series of numbers being printed in a list. TI Extended BASIC also uses IMAGE, which allows further formatting. The exact format should be specified with the program using it; if it isn't, trying to answer the question 'how should I attempt to translate it?' is a little like trying to answer 'how long is a piece of string?'

RANDOMIZE (see also RND) may appear as RAND in other BASICs, but may not always be flexible enough to allow you to specify a 'seed' (e.g. RANDOMIZE 2) in order to have separate, but reproducible, series of 'pseudo-random' numbers, as in TI BASIC. (In Sinclair BASIC, you may see RANDOMIZE with USR — see USR.)

REM can appear as '!' (a much over-worked symbol, I think you'll agree), for example in TI Extended BASIC, and in other BASICs REM doesn't require a space between it and the remark. Thus you may find:

```
1200 REMARKABLE SUBROUTINE
```

which, should you be desperate to translate it, is:

```
1200 REM ARKABLE SUBROUTINE
```

It seems to lose something in translation, though, and anyway, as a general rule, I would not recommend the inclusion of REMs in a program unless they were absolutely necessary. REMarks are a hang-over from the days when ASSEMBLERs were the only 'translation' programs available, and they were designed to ignore REMs (often implemented as colons ':') — thus you could REM your original listing to your heart's content as none of them was incorporated in the final, assembled, program. Today's INTERPRETERs however do incorporate REMs, and they can occupy valuable memory in TI BASIC listings.

REPEAT . . . UNTIL is similar to WHILE . . . WEND (see later).

RESUME, which is not one I have seen myself, apparently continues execution of a program when completing the ON ERR (see earlier) instruction. Again, there is no equivalent in TI BASIC.

RIGHT$( ), the third of the three string functions mentioned earlier, is usually of the form:

```
A$ = RIGHT$(B$, length)
```

and instead of 'slicing' the string beginning at the first (or leftmost) character, it begins from the end (or rightmost). Therefore:

```
RIGHT$("HELLO", 2)
```

is "LO". In TI BASIC, SEG$( ) provides the equivalent:

```
A$ = SEG$(B$, LEN(B$) − length + 1, length)
```

Using our "HELLO" string again, which has a length of 5 characters, the TI BASIC equivalent results in:

```
SEG$("HELLO", 5 - 2 + 1, 2)
```

which is:

```
SEG$("HELLO", 4, 2)
```

or "LO". Notice that '5 − 2 + 1' is evaluated as '(5 − 2) + 1', and not '5 − (2 + 1)'.

RND, which may also appear as RND( ), does vary a little. Some forms, like RND(n), give a whole number (integer) between 1 and 'n', so in TI BASIC this is:

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

Others use RND( ) in certain ways to give either negative numbers, or to repeat a series of fractional numbers. A discussion of the generation of such 'pseudo-random' numbers would require a book all to itself.

RPT$( ), which appears in TI Extended BASIC, is similar to STRING$( ) — see later.

RUN, or RUN with a line number, cannot appear within a program in TI BASIC, but can be used by some forms of BASIC. Its use usually resets all the variables (see CLEAR) before running the program again, so it can only be implemented with (i) a subroutine to emulate CLEAR, and (ii) a simple GOTO to the start of the program, or wherever specified. For example:

```
3000 RUN
```

becomes:

```
3000 GOSUB 400
3010 GOTO start of program (which need not be the very first
     line)
```

where the subroutine referred to might be like this:

```
400 A$ = ""
410 A = 0
420 B = 0
430 ST4 = 0
440 FOR I = 1 TO 20
450 C$(I) = ""
460 NEXT I
470 RETURN
```

With a large number of variables you might have a problem, especially if the program is a large one.

SAVE is another command which cannot be used in a TI BASIC program but does occur in programs for other machines. In at least one case, where it is possible to save a program with a 'name', the use of SAVE in a program causes that program to be saved in a special way. When you subsequently re-load that program it RUNs automatically once it has loaded successfully. In Sinclair BASIC, this causes the first character of the 'name' to be inverted — making it white on black.

SIZE, like FRE and MEM, amongst others, gives the amount of memory available. To imitate it in TI BASIC requires a short routine which is explained in the **Hints and Tips** chapter.

STRING$( ), like RPT$( ), is a useful command which does not occur in TI BASIC, and can really only be emulated through the use of a loop and string concatenation (tacking one string onto the end of another). The format is STRING$(string, number), which produces a string containing repetitions of the specified string a specified number of times. Thus STRING$("HELLO", 2) produces "HELLO HELLO". To make a string of 16 zeros, for example, it would require STRING$("0", 16).

TRON and TROFF equate to TRACE and UNTRACE in TI BASIC. It is highly unlikely that you will see them in programs, but just in case you do, you now know how to cope with them.

USR, with the format USR number, or sometimes USR "letter", has no direct equivalent in TI BASIC. It generally transfers control from a BASIC program to a machine code routine, but can also be associated with graphics commands in, for example, Sinclair BASIC. Additionally, you may see various words associated with USR, for example RAND USR number, PRINT USR number, LET A = USR number, and so on. These are most likely again in Sinclair BASIC, and are simply mechanisms whereby the computer can accept the command USR. The reasons for this are fairly complex and involve a discussion of machine code programming, which is beyond the scope of this book. As a rule, programs which contain this command cannot be translated successfully into TI BASIC, primarily because we cannot provide equivalents to USR, which is usually a critical factor in the program.

VAL( ), also mentioned earlier under EVAL( ), does operate in many cases like EVAL( ), and the explanation of that function's operation applies to VAL( ) also; TI's VAL( ) is not as powerful as that found in other BASICs.

WAIT is a pause command, and can be imitated through the use of a simple loop:

```
1000 FOR I = 1 TO value

1010 NEXT I
```

where the value dictates the length of the pause.

WHILE . . . WEND, similar to REPEAT . . . UNTIL, has the format:

```
250 WHILE X = 10
260 PRINT X
270 X = X + 1
280 WEND
```

Producing an equivalent involves IF . . . THEN:

```
250 IF X < > 10 THEN 290
260 PRINT X
270 X = X + 1
280 GOTO 250
290 continuation of program
```

WIDTH is another function which might be difficult to reproduce in TI BASIC. It sets the width in characters for the screen, or perhaps a printer. As far as the printer is concerned, because the OPEN command in TI BASIC can specify a 'record length' it might be feasible to mimic WIDTH by setting the record length to whatever is specified by WIDTH, bearing in mind that a limit will be imposed by the physical characteristics of the printer you are using. For the Thermal Printer, for example, the maximum width would be 32 characters, although it must be said that with some very complex programming it is possible to produce a virtually infinite-length line on the thermal printer, but the task is so complicated it requires a book all to itself.

Closing Notes

In Sinclair BASIC, if the variable A$ contains "HIS LITTLE FRIEND", then:

```
A$(1 TO 3) = "HER"
```

turns the string into "HER LITTLE FRIEND" by replacing HIS with HER. In TI BASIC this has to be produced using SEG$( ):

```
A$ = "HER" & SEG$(A$, 4, LEN(A$) - 3)
```

Similarly, if B$ contains "GIVE ME A KICK", then:

```
B$(6 TO 7) = "IT"
```

turns the string into "GIVE IT A KICK", and is imitated thus:

```
B$ = SEG$(B$, 1, 5) & "IT" & SEG$(B$, 8, 255)
```

Take careful note of the 255 in the second use of SEG$( ); it is usually a mark of sloppy programming, but in this case it is the only practical way to perform the implementation. If you use 255, the computer will include the remaining segment of the string, however long it may be, without causing any errors.

The IF . . . THEN form:

```
IF A = B THEN PRINT "YES" ELSE PRINT "NO"
```

may be translated in one line (although, as noted earlier, it is generally impractical):

```
PRINT SEG$(" YES NO  ", 5 + 4 * (A = B), 5)
```

(Note that there is a space before the 'YES', one between the 'YES' and 'NO', and two after the 'NO'.) If A does equal B, then (A = B) evaluates to −1, which is multiplied by 4 to give −4, then added to 5 to give 1, thus beginning the string segment at position 1, with length 5 characters. This extracts " YES ". If, however, A does not equal B, then (A = B) will evaluate to 0, which, when multiplied by 4 and added to 5, will give 5, thus beginning the segment at position 5, with length 5. This extracts " NO ".

All The Fours

There are at least four identifiably different TI-99s in existence, and they are sufficiently different for there to be translation problems if an attempt is made to copy or run a program intended for one specific machine.

The four known machines are: NTSC 99/4, NTSC 99/4A, PAL 99/4 and PAL 99/4A.

The NTSC 99/4A has not been sold in this country, but the other three versions have, beginning with the NTSC 99/4, followed by the PAL 99/4. These two have identical keyboards, using keys which are similar to giant calculator buttons. The functions which are obtained on the later 99/4A via the FCTN key are obtained on the 99/4s through the Shift key. The 99/4A manuals give details concerning the 99/4 keyboard, but 99/4 owners might have a little trouble deciding exactly what the function keys are.

Function	*99/4*	*99/4A*
AID	SHIFT A	FCTN 7
CLEAR	SHIFT C	FCTN 4
DELete	SHIFT F	FCTN 1
INSert	SHIFT G	FCTN 2
QUIT	SHIFT Q	FCTN =
REDO	SHIFT R	FCTN 8
ERASE	SHIFT T	FCTN 3
LEFT ARROW	SHIFT S	FCTN S
RIGHT ARROW	SHIFT D	FCTN D
DOWN ARROW	SHIFT X	FCTN X
UP ARROW	SHIFT E	FCTN E
PROC'D	SHIFT V	FCTN 6
BEGIN	SHIFT W	FCTN 5
BACK	SHIFT Z	FCTN 9
ENTER	ENTER	ENTER

In addition, the 99/4A has a CONTROL (CTRL) key, which is intended primarily for use either with communications devices like MODEMs or ACOUSTIC COUPLERs, or for transmitting special instructions to peripherals like the impact dot-matrix printer. It is also possible to use both FCTN and CTRL keys to obtain some of the User-definable characters directly, instead of having to use CHR$( ) as on the 99/4s. These differences, although important as far as programming goes, are over-shadowed by the different character set possessed by the 99/4A.

On the 99/4, the standard character set is only part of the full ASCII; the lower case (small letters) were omitted, but on the 99/4A they have been included, albeit as small upper case (capitals). The problem is that a program written on the 99/4A, which uses informative sections of text written in this small upper case, will produce totally incomprehensible garbage on the 99/4s. It is important for 99/4 owners to be aware that this might happen, and to be on the look-out for such programs. It is possible, through the inclusion of large amounts of extra data, to redefine the 99/4 User-definable set to give the small upper case of the 99/4A, but few if any 99/4A programs recognise this problem so no attempt is made to compensate. The small upper case is in fact the standard set for the 99/4, while the standard set on the 99/4A is a larger version — if you have seen the two character sets available on the Thermal printer, then the resident TP set is equivalent to the standard 99/4A set.

On top of this the 99/4A has three additional characters which can be used as variable names. These are the 'back slash' (\), the left square bracket ([) and the right square bracket (]). None of these is directly accessible from the 99/4 keyboard, but they are nevertheless active when a 99/4 is running a 99/4A program which utilises them.

There are slight differences in the COS or Cassette Operating System. The early NTSC 99/4s can use CS2 for both SAVE and OLD, although the cassette cable may not have the necessary connections fitted. There are programs around which process data files using CS2, and of course later models will be unable to run such programs unless all CS2 file commands are altered to address CS1.

There are also, alas, bugs in all models, some of which are specific to a particular model. Some of these are detailed in the **Hints and Tips** chapter, with suggestions as to how they may be overcome.

This chapter has been able to do little more than scratch at the surface of the subject of translation — for example, no specific details have been given about the makes of computer whose dialects of BASIC sport the non-TI BASIC commands and functions examined, and certain areas are virtually impossible to explain in sufficient detail so as to allow trouble-free translation. The most efficient tutor is perhaps experience, and if you spend some time attempting to translate programs into TI BASIC that is one attribute you will acquire rapidly.

Jargon

What is Jargon?

It is hard when discussing any subject, be it preparing Sunday dinner or piloting the Shuttle, to avoid the use of jargon. Its purpose is usually poorly understood, and its use is sometimes criticised unreasonably. There are, naturally, cases where jargon is used — or abused — unnecessarily, and yet others where a word may be specially created to describe something which is in fact already adequately described by an existing word.

Many examples of jargon do not appear on the surface to be so, but this is because the specialised words are in general use and are recognised by almost everybody. For example, the words 'grill', 'roast', and 'fry' are all specialised terms yet almost all readers will recognise not only the field to which they belong, but also the action which each describes. Some readers may also be able to apply them to other fields, and will even be able to distinguish their different meanings according to the context in which they appear; that is, they can tell, by the way in which the jargon word is used, exactly which meaning is to be taken.

The field of Computer Science is immense and covers an enormous number of topics, many of them appearing only in recent years. When discussing any aspect of any one of these topics, the wealth of detail involved is so great that the dreaded jargon has to be used in order to reduce complicated ideas to single representative words or phrases. Often these words or phrases are chosen from those in everyday use, not deliberately to confuse or mislead innocent bystanders, but precisely because those words or phrases are in common use, as well as being apt descriptions. Initially these jargon words form an apparently impenetrable barrier to understanding as far as the newcomer is concerned, but once they are explained in an understandable manner, and have been used often enough to become familiar, then the novice can find him/herself peppering everyday conversation with them, and shortly their friends and colleagues will be shaking their heads in bewilderment as they in turn struggle to understand what is being said to them.

It is a trap which is all too easy to fall into, as many will find; once over the initial 'hurdle' it becomes difficult to remember just what it was like to be one of the 'uninitiated', and I certainly cannot claim to be able to avoid being incomprehensible even part of the time.

No matter how hard you try, the use of a small amount of jargon is inevitable, primarily because any attempt at discussion would become bogged down in explanations in fine detail of complex concepts, so it is as well to read up on some of the more common words and expressions. This chapter will provide explanations of some items in a way which is as free of jargon as is possible; you may begin to appreciate why jargon is used so frequently when you see the lengths to which you have to go in order to avoid its use.

We'll begin by looking at some of the aspects of BASIC which are discussed in the chapters in this book. Because of space limitations only a few of the enormous number of concepts can be looked at here, but it should give you an insight into them.

BASIC

The word BASIC is a 'mnemonic' (pronounced 'knee mon ik') — that is, it is a 'word' made up of the initial letters of a group of other words, in such a way that it is easier to remember or discuss them. We are familiar with this idea through, for example, our Trades Unions: NUPE (usually pronounced 'new pea'), which is the National Union of Public Employees; CoHSE ('cozy'), the Confederation of Health Service Employees; and so on.

BASIC stands for Beginners' All-purpose Symbolic Instruction Code, and was devised in about 1964 in America's Dartmouth College by Thomas Kurtz and John Kemeny. (The Americans have a penchant for producing mnemonics which themselves are recognisable words.) It was originally intended as an introductory language to help new students understand the principles of programming before ploughing on into the more complex, and less easily understood, languages then available. Unlike most other languages, BASIC is simple and easy to learn — within a very short time even young children can learn and understand enough to be able to write programs. It is, in its many and slightly varied forms, perhaps the most popular and successful language yet produced, although in many academic eyes it is a 'dangerous' language because it encourages 'sloppy thinking'.

It is perfectly true that it is possible to produce a BASIC program which works, but which is virtually impossible to understand or alter in the event of error or changed requirements.

Perhaps the best way to view BASIC is as a stepping-stone to more advanced things. After all, we don't expect a newborn infant to speak or understand grammatically correct English; in fact we go out of our way to use very simple words and expressions (not to mention the 'goo-goo-ga-ga' syndrome), so why should we treat the learning of a language on a computer any differently?

Words, Bytes, Nybbles, and Bits

These are among the most commonly used jargon words, and annoyingly they are the ones rarely used in TI BASIC, simply because we don't have ready access to the kind of facilities which would enable us to discuss them in detail, with useful examples of their function.

All four are measures of the data-handling function of the computer. *Bit* is a contraction of *Binary Digit*, and is the smallest piece of information which the computer can hold. Inside the computer, a 'bit' corresponds to the state of an electrical pulse: it has one of two (*binary*) states — relatively low voltage, corresponding to a 0, and relatively high voltage, corresponding to a 1. A group of four of these bits is called a *Nybble*, while 8 are a *Byte*.

The combinations of 1s and 0s in a nybble total 16 (0000 to 1111), and in a byte total 256 (00000000 to 11111111). When you read the section later on numbers this should have a little more meaning.

All the letters, the numbers 0 to 9, punctuation marks, etc. (called 'characters'), which appear on the screen display have to have details about their shape stored somewhere inside the computer. For the 'User-definable characters', in TI BASIC, the graphics command CALL CHAR( ) uses 16 nybbles to specify those details, representing each nybble with the 'digits' 0 to 9 and A to F. These digits form the basis of a numbering system called *Hexadecimal*, which we will cover in more detail later. All the shape details are stored internally in 8 bytes per character.

Using a special system of codes, one byte can be used to refer to the code of one 'character' (a letter, number, or punctuation mark), so the 256 combinations of 1s and 0s in a byte can be used to refer to the codes of 256 characters — again, this is covered in a little more detail later in the chapter. Using ASCII (see later), the standard coding system, 7 out of the 8 bits in a byte can be used to represent the 128 character codes (coded 0 to 127).

A *Word* is slightly more difficult to define. Its size in bits is determined by the number of bits which can be held at one instant in an area of the computer called the *Accumulator*, which usually resides in a 'micro-chip' called the *Central Processing Unit*, or cpu. The accumulator holds the results of processing by the cpu, and on most of the computers used by enthusiasts has a capacity of 8 bits. On the 99/4 and 99/4A though, the accumulator can hold 16 bits, so a word on other machines is equivalent to a byte, while our word is two bytes long. This has advantages as far as *Machine Language* programming is concerned, for it permits a wider variety of instructions, greater mathematical precision, and the processing of larger quantities of information each time. Machine language is discussed below. All of these aspects are, alas, closed to the TI BASIC programmer, for to gain access to them we must invest in further equipment.

Machine Language

The language which we understand, known as *Natural Language*, is not one which the computer is capable of 'understanding'. Our language is a *High Level* language, immensely complex, full of contradictions and 'exceptions to the rule'. A computer is nothing more than a sophisticated machine, and its 'language' bears little comparison to ours. So how then, you might ask, does it 'understand' BASIC, which has words like PRINT, IF, and NEXT which are all English words? The short answer is that it doesn't. It 'understands' the words in the same way that a car 'understands' pressure on a brake pedal means 'apply the brakes'; in other words, there is no 'thought', as we would know it, behind the computer's actions — the machine responds to a command in a particular way because it is designed to. The difference, and a very important one it is too, is that on the car *You* cannot redefine the purpose of the brake pedal. You cannot change the function of the brake pedal into, say, one of steering, or of windscreen-wiping. On the computer, however, because of its design, *You* can make changes to the machine which enable it to perform an enormous variety of tasks, simply by re-programming it.

As all matter is composed of atoms, so all computer programs are composed of *Machine Code Instructions*. They are the basic building blocks, so to speak; the 'language' which the computer can 'understand'. In BASIC, the English-like words are held as a kind of 'foreign language dictionary' in the computer's memory. A special program called an *Interpreter* scans your BASIC program, looking in the dictionary for the words which go to make up your 'listing', and as each is found, the interpreter invokes sequences of machine code instructions which are used to make the computer perform the tasks specified by the English words.

The 'vocabulary' of machine code instructions is called the *Instruction Set*, and is composed of really quite simple functions. Using them, the computer's cpu can add two bytes (because our cpu has a 16 bit capacity, it can work with larger numbers of bits), subtract them (which in fact involves making one byte 'negative' and then adding the two), compare them to see if they are equal (the cpu in the 99s can also multiply and divide them); it can work on the individual bits themselves, testing them to see if they are 1s or 0s, shunting them around within the byte, making them 1 if they were 0 and vice versa; it can avoid executing (jump over, like GOTO, or IF . . . THEN . . .) sequences of instructions; all in all, it can perform only surprisingly simple tasks, yet these can be used in combination to produce some startling results: with the right programming the computer can generate poetry, sing (I have heard the 99/4A, Terminal Emulator II, and Speech Synthesiser being used to produce a creditable rendering of 'Daisy, Daisy'), and commands spoken by an individual can be translated by the computer into physical action — turning lights on, dialling a telephone number, and so on.

Each machine code instruction is a group of bits, usually a byte or so, stored in an address in memory. *Machine Code* is usually the decimal or hexadecimal equivalent of the combination of 1s and 0s in the instruction, while *Assembly Language* is the set of almost-English mnemonics (see BASIC), a sort of 'shorthand' for the instructions, which makes them easier to remember. A special program called an *Assembler* allows you to write your machine language program in near-English (but not as near as BASIC) and the computer translates it into machine code. An example of machine code might be 'CD FE 11 09', while the assembly language version might be 'MPY 11, 09'. In English this would be 'Multiply 11h by 9h'. (Please note that this example is totally artificial.)

There are therefore several 'levels' of language which can be used with a computer, ranging from BASIC ('HIGH') to machine code ('LOW'). There are many different languages which can be used on the same computer; some of their names you will doubtless have come across: Cobol, Forth, Lisp, APL, Algol, Fortran, LOGO, Pascal, etc. They each have different facilities for handling data: one will concentrate on high-precision arithmetic, another on processing text. Depending upon your requirements, so you would choose a language which is the most suitable tool for handling your information.

Boolean Algebra

The title suggests some weird form of arithmetic, but in fact it describes a way of making decisions which we all use, probably every day. George Boole, a 19th century mathematician, after whom it is named, developed this notation which uses LOGIC, rather than arithmetic. It is concerned with two attributes: TRUE and FALSE. Instead of the arithmetic 'operators' (add, subtract, multiply, divide) it uses 'logical' operators: AND, OR, and NOT (and there are combinations of these). Examine this statement:

If it is raining AND you have a raincoat then wear it.

There are two statements within it:

a) It is raining
b) You have a raincoat

and, when tested (for 'veracity', if you like) they BOTH must be TRUE before you can comply with the instruction 'wear it'. If it isn't raining, or if you don't have a raincoat, then you won't or can't wear it. The example is contrived, but it should take you a little way towards understanding the next section.

The operators named above are used in two languages on computers: BASIC, and *Machine Language*. Their exact operation differs in each language, but the principle is the same. In machine language, AND, OR, and NOT (as well as another called XOR or EXCLUSIVE OR) perform functions on bytes of data. AND, OR, and XOR operate on two bytes, while NOT operates on one. In effect, AND, OR, and XOR place one byte 'above' the other, thus:

```
11010110   (top line)
01111111   (bottom line)
--------
           (result line)
--------
```

The operator AND compares the first bit in the top line with the first bit in the bottom line, and only when both are '1' will it place a '1' in the 'result' line. It then looks at the second bits in top and bottom lines, and so on until it has compared all the bits. In the above example the result line will look like this:

```
01010110
```

which, if you look very carefully at the top line, is the first byte with the first '1' turned to '0'. It is a useful way of 'isolating' bits or groups of bits (see later in ASCII the discussion of checking for errors, when the PARITY BIT is 'stripped off'). When however OR (also known as INCLUSIVE OR) is used, the effect is one of 'mixing' or superimposing one byte on another, for with OR the operation will place a 1 in the result line when a 1 appears in either or both of the top and bottom lines. This kind of function can be useful when making, say, one shape on the screen pass 'through' another. For example, ORing these two bytes gives the result shown:

```
10011001   (top line)
00111100   (bottom line)
--------
10111101   (result line)
--------
```

XOR is slightly different in that a 1 will appear in the result line if either of the compared bits is 1, but *not* if BOTH are 1. Thus:

```
10011001  (top line)
00111100  (bottom line)
--------
10100101  (result line)
--------
```

NOT works on a single byte, and is also called an *Inverter.* Its function is also referred to as *Ones Complement.* It simply works from one end of the byte to the other, making all the 1s into 0s, and vice versa. Thus the effect of NOT on:

```
10100101
```

is:

```
01011010
```

Although this could be used to invert a shape (make it green on red instead of red on green for example), it is used in machine language programming as a step towards producing a *Negative* number in binary notation. The actual process is called *Twos Complement,* and the second step involves adding a binary 00000001 to the value obtained after *Ones Complement.* One use for negative numbers in machine language is as part of the process of subtraction (see above).

String and Numeric Variables

In BASIC you can store either text (i.e. any quantity of letters, digits, punctuation marks, known as a *String*) or numbers (i.e. combinations of the digits 0 to 9, plus a few others: the decimal point '.', '−', '+', and 'E'. 'E' is the symbol for *Exponentiation,* and stands for 'multiplied by 10 to the power of', so that 1E4 is '1 times 10 to the power of 4' or 1 × 10000. These are called *Numeric*). The distinction on the computer between a *String* and *Numeric* item is that the arithmetic operators ('+', '−', '*', '/', and '^') cannot be used with *Strings.*

A *Variable* is a method of storing data in such a way that, by referring to the name of the variable, you refer to the data itself. The numeric and string data stored in variables are both referred to and stored differently on the computer, and the distinction between the names of string variables and numeric variables is made by adding a '$' to the name. Thus 'A' or 'ZZ' are names of numeric variables, while 'A$' or 'TIME$' are names of string variables.

Arrays, Elements, Subscripts

These are descriptions of a method in BASIC for referring to a stored list of information. The *Array* is a sort of list; the *Elements* are the number of separate 'items' stored in the array; and each element has a number which refers to it — the *Subscript*. Assuming that you already know a little of STRING and *Numeric Variables*, here is an array called SHOP$ which has 10 words stored in 10 elements:

Word	*Stored in:*
Tea	SHOP$(1)
Butter	SHOP$(2)
Cheese	SHOP$(3)
Bread	SHOP$(4)
Eggs	SHOP$(5)
Milk	SHOP$(6)
Cake	SHOP$(7)
Sugar	SHOP$(8)
Salt	SHOP$(9)
Mustard	SHOP$(10)

Element 1 of SHOP$ — SHOP$(1) — contains the word 'Tea', SHOP$(2) contains 'Butter', and so on. This type of array is known as *One Dimensional*. The number of dimensions relates to the number of subscripts necessary to refer to the elements of the array. A *Two Dimensional* array might be:

	Columns:	1	2	3
Rows:	1	Sandra	Blue	Sky
	2	Jane	Green	Wham
	3	Susan	Brown	Duran Duran

Here the data, which are spread in two directions, are a descriptive table of the girl friends of one G. Whizz, playboy and deckchair attendant on Cannes' nudist beach, listing them by name and giving their eye colour and music preference. If the array is called WOW$, then WOW$(3,3) contains Susan's music preference, while WOW$(1,2) contains Sandra's eye colour (referring to the elements by row and column). Here there are 9 elements in all, arranged as 3 by 3 'pigeon holes'.

A *Three Dimensional* array is perhaps a little too complex to do anything more than describe; a book could be regarded as a three-dimensional array, with each letter and punctuation mark being referred to by page number, and row and column on the page. The format would be BOOK$(page, row, column). An example of a *Four Dimensional* array might be a series of volumes, this time with the letters and punctuation marks on a given page also being referred to by volume number as well. Thus the format might be: BOOK$(volume, page, row, column). A *Five Dimensional* array could be represented by a library filled with volumes of books, in which case a single punctuation mark on one page of one volume of a series on a shelf could be referred to by BOOK$(shelf number, volume, page, row, column).

Arrays are very useful when it comes to storing data which is related in some way: for example, surname, hours worked, and wage due. On the 99s in TI BASIC we are limited to a maximum of three dimensions (and seven in Extended BASIC), but some other machines allow you to specify as many dimensions as your available memory will allow.

Numbers: Binary, Decimal, Hexadecimal

The counting system that we are familiar with is called *Decimal*, and is based on the number 10. Combinations of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 are used to represent quantities, and for most of us it has been so long since we learned about it that it has become second nature to us.

However, there are numbering systems in use which are based on numbers other than 10, and two of these surface regularly in computing. They are called *Binary*, which is based on 2 and uses combinations of the digits 0 and 1, and *Hexadecimal*, which is based on 16 and uses the digits 0 to 9 and the letters A, B, C, D, E, and F, where A represents decimal 10, B represents decimal 11, and so on up to F, which represents decimal 15. Hexadecimal, or hex, uses the letters because it enables single 'digits' to represent all of the numbers in the range 0 to 15.

If you were never exposed to the idea of counting in something other than tens, or it has been so long that you have forgotten all you ever knew, then you might well experience some difficulty initially. It is unfortunate that you need to understand the functioning of a number of facilities which are absent from TI BASIC, like the operators AND, OR, XOR, and NOT, and machine language, all of which are fundamental parts of this area of computing, in order to build up a reasonably complete picture of the operation of any computer.

To begin with, let us study the decimal system of notation, in order to understand the principles involved and thus hopefully follow their application to binary and hex numbers.

When we write the decimal number 123 we are in fact using shorthand for:

$$(1 * 100) + (2 * 10) + (3 * 1)$$

You may be familiar with the childhood practice of performing calculations by writing the numbers under column headings:

Hundreds	*Tens*	*Units*
1	2	3

which can be rewritten thus:

100	10	1
1	2	3

In each case the column headings begin at 1 on the right and increase by factors of 10 (the 'base') as you travel to the left.

In binary, the headings increase by a factor of 2:

4	2	1
1	0	1

(so this binary number, 101, is (1* 4) + (0* 2) + (1* 1), or decimal 5) and in hex, by 16:

```
256  16  1
———  ——  –
  F   C  7
```

(so this hex number, FC7, is (15 * 256) + (12 * 16) + (7 * 1), or decimal 4039). Remember, zero multiplied by any number is still zero.

Using the information presented so far, how would you convert the binary number 1100 to decimal, and to hex? One method might be to revert to the childhood practice of putting the number under columns:

```
8 4 2 1
– – – –
1 1 0 0
```

This is then 8 + 4 + 0 + 0, or 12 decimal. 12 decimal is C in hex, for C is the symbol which represents 12.

Note that in each case I have tried to make sure that you knew which numbering system I was referring to; suppose your task had simply been to convert 1100 to hex? Without knowing exactly what 1100 represented, you wouldn't be able to do it. Is it 1100 binary, or 1100 decimal? Or even 1100 in some less well-known base — say 8, which is called OCTAL, incidentally. It is important to stipulate which numbering system you are using. There are different ways to notify an observer of the base of the number being used, but by and large you can use 'd' for decimal (e.g. 23d), 'b' for binary (e.g. 11011b), and 'h' for hex (e.g. A403h). Some people prefer to write the number with the base as a trailing subscript, but that can lead to confusion if your penmanship is none too good, or if your typewriter can't produce subscripts.

ASCII

ASCII — pronounced 'askey' — stands for *American Standard Code* for *Information Interchange*. It is analogous to Morse code in many respects. It is a commonly agreed, standard way of representing symbols in BASIC, where 'symbol' means letter, number, punctuation mark, or *Control Character*. Control characters are obtained using the *Control Key* (only available on the 99/4A) and are used to give special instructions to attached equipment: for example, to mark the beginning and end of a piece of text which is being transmitted, perhaps to a printer. There is not space here to go into ASCII in great detail, for it is far more complex than it looks. There are 128 ASCII characters, which are listed in your manual, numbered 0 to 127, and each is stored in memory as a *byte*. These 128 codes are represented by only 7 out of the 8 *bits* in a byte, and the 8th bit is used in a technique for checking for errors in transmitting bytes from one location to another. The full 8 bits allow codes 0 to 255, and in TI BASIC you can print all 256 characters to the screen, but of the characters with codes greater than 127, some are used for graphics purposes, while the others have no practical use, generally speaking.

The error checking is simple. The 7 bits which are used to represent ASCII character codes are examined, and if there is an odd number of 1s, the 8th bit is also set to 1 to make it an *Even* number (called *Even Parity*). Alternatively, if using *Odd Parity*, the number of 1s is counted, and if there is an even number of 1s, the 8th bit is set to 1 in order to make the number *Odd*. Otherwise, the 8th bit is always 0.

Then, depending on the type of *Parity Checking* (*odd* or *even*) chosen in the computer, any received character has its bits counted up once it has reached its destination, and, if there is an odd or even number of 1s the computer 'knows' that some data have been lost, and can either try asking for the data to be transmitted again or just stop and display an error message.

The effect of the 8th bit can be undone by 'stripping' it off using the Boolean operator AND (see *Machine Language*). If any byte is ANDed with the binary number 01111111, only the 7 rightmost bits will remain, leaving the 8th (leftmost) bit set to 0. This is a common use of AND in machine language programming.

ASCII is not the only coding system around, but it is the one most commonly used and after a time you will, through repeated use, become more familiar with some of the codes — 65 for the letter 'A', or 48 for '0' for example.

Conclusion

It has not been possible to explain in anything like sufficient detail the vast majority of jargon words and expressions which you are likely to encounter. Hopefully, though, you will have experienced a little of the 'flavour' and may not be completely overawed by some of the concepts with which you may have to deal as you delve deeper into any aspect of computing. If you finish this chapter just as confused (or more so) as when you started, then don't forget: that's not *Your* fault, it's *Mine*.

File Handling

The subject of File Handling is broad and quite complex. It covers not only the handling of data files, but also of program files. We will limit ourselves to a discussion of the handling of cassette data files in TI BASIC, as the basic system does not permit manipulation of other kinds of file (and few owners are likely to be able to afford the disk system, or the printer).

Cassette file handling on the 99/4 and /4A is made more difficult by the fact that no two individuals' filing requirements are exactly the same. Also, like learning to program, once one has overcome the initial hurdles one quickly forgets what it is like not to understand, and it can be something of an effort to second-guess the kind of problem which is likely to face the newcomer.

Generally, the most common problem appears to be an inability to create a mental picture both of what is required, and of what is going on inside the machine. Jargon abounds in this area — I/O BUFFER, DEVICE.FILENAME, FILE TYPE, etc., all serving to trip the unwary.

TI's own manual does give condensed information on File Processing, but the average owner needs to be led slowly and gently through each section. It can require quite some will-power to ignore the gut feeling of 'I'll never understand this lot!', and to force yourself to go through, sentence by sentence — even word by word if necessary — and even then total success is never guaranteed.

Having apparently overcome the difficulties involved in understanding the manual, the next problem lies in understanding why the computer keeps rejecting your carefully formulated and surely-correct filing instructions. You may be certain that you have followed everything in the manual to the letter; you can see absolutely nothing wrong with the statements in your program; and yet . . . and yet . . . it insists on crashing.

Now, file handling shouldn't really present any problems at all

(I can say this now that I've overcome my own difficulties) and yet it does so, and frequently. I have spent more time than I care to admit to, just trying to store and retrieve the simplest items of data. I couldn't even get the examples in my 99/4 manual to work, and eventually I had to admit defeat.

It wasn't until around 18 months after buying my machine that I finally discovered why I had not been successful, when a fellow enthusiast sent me some text filed on cassette, with a program to retrieve it.

Now that I had something on tape which I knew must be retrievable, I was determined once and for all to master this one grey area.

It transpired that the reason for my former lack of success stemmed primarily from two things:– firstly, my habit of disconnecting the 'remote' lead from my cassette recorder in order to speed up tape handling for programs, and secondly the omission from my TI manual of a vital piece of information.

When you open a data file to cassette — a process which we will examine in more detail later — the computer suspends operations for quite a while before continuing. This is *before* it begins storing/retrieving data. The intention (presumably) is to ensure that the leader tape is not under the tape recorder's record/playback head, and to pass over the first few seconds of tape, which is often damaged (due, for example, to 'shedding oxide'). A similar delay occurs when a program is being SAVEd to tape.

However, the manual didn't warn of this delay, and because I didn't have the 'remote' lead connected I was unaware that the computer was not yet ready to begin transferring data. Consequently, when I tried loading in some previously recorded data, I was playing the tape to a deaf machine, and then when I had reached the end of the data file and no DATA OK message had appeared — or whatever was due to happen — I was rewinding the tape while the computer listened out for incoming data. By the time that my data again came under the playback head, the computer had had enough of waiting, and told me NO DATA FOUND.

You can avoid this experience by becoming familiar with the 'pause' through this short program:

```
100  PRINT "BEGIN OPENING THE FILE"
110  OPEN #1: "CS1", INPUT, FIXED
```

```
120 PRINT :::"FILE OPENED"
130 CLOSE #1
140 PRINT :::"FILE CLOSED"
```

The cassette leads do not need to be connected; you won't need any data on your tape because the routine doesn't attempt to read any; the specifications in line 110 are not really important; and the computer thinks that the system is working normally. (You may find that your tape recorder will not function correctly with your computer for technical reasons — perhaps the best policy is to contact one of the TI User groups and ask them if they can recommend a tape recorder.)

You may, of course, have had no problems whatsoever in mastering filing — in which case award yourself a gold star!

From this point the chapter will depart from the standpoint adopted until now, and instead of carrying on from where the manual leaves off, it will attempt a more comprehensive explanation of the facilities offered by TI BASIC for cassette file handling.

There are two aspects to filing which we shall examine here: firstly, file handling or data processing and what it involves; and secondly, what it entails as far as TI BASIC is concerned.

To most people who don't spend their working days handling files the subject of File Handling can be confusing if not daunting. Commonly we expect a file to be a folder or something similar, used to store sheets of paper containing details of business transactions, or medical records, or school project notes, and so forth.

In computer terms, though, a file is simply something outside, and linked to, the computer, to and from which data are transferred. The keyboard and the screen are files, as is the printer, the cassette tape recorder, the disk drive, the speech synthesiser, and so forth. Depending on the capability of the file, data can be transferred in either or both directions: from computer to file, and/or from file to computer.

When the computer needs to transfer information between itself and a file, it first has to set aside a kind of 'waiting area' in its memory, called an INPUT/OUTPUT, or I/O, BUFFER. This waiting area is something like a holding bay, where information which is 'in transit' sits and waits to be moved to its destination. Generally the movement operates on the principle of 'we go when the wagon is full' — that is, the data are not moved until a certain quantity is ready to be transferred. This quantity is dictated either by the capability of the file which is its destination/origin, or by the requirements of the person doing the filing.

The specification of the size of the waiting area (buffer) is contained within the BASIC command 'OPEN', giving the name of the file and some details as to whether the 'traffic' is to be incoming only, outgoing only, or both. (Some TI peripherals have a small amount of memory on board which 'identifies' them by name to the computer and also indicates what minimum specifications they require; the system is designed so that if you attempt to use a peripheral which doesn't possess this identification, the computer will 'refuse' to work with it.)

Each waiting area (buffer) — there can be several on the go at any one time — is identified by a 'file number', rather like a 'platform number' at a rail station. Platform #1 might have 'outgoing traffic' to the printer, while platform #2 might have 'incoming traffic' from a cassette tape recorder.

On the TI-99s, platform #0 is a special case, having its function preset by the manufacturers. Outgoing traffic using platform #0 is sent to the TV screen, while incoming traffic arrives, in varying amounts, from the keyboard. Whereas the programmer can determine the characteristics and destinations/origins of platforms #1 upwards (to #127 or #255, depending on the capabilities of the peripheral), the characteristics of platform #0 cannot be altered, nor can the waiting area be shut down.

The specification for information being transferred also involves a description of its file's capabilities: some files simply allow you either to add information onto the tail-end of the last 'train-load' sent, or to take the information out in exactly the same sequence in which it was originally stored (*Sequential Access*), while others allow you to be more selective and to store or withdraw any item from any position (*Random Access*) — rather like taking library books from, or replacing them on, shelves: you don't have to start by the library entrance and work through every book, cover to cover, until you reach the one you were interested in. You can walk directly (after consulting the library index) to the book you want, take it out, read it, and put it back again (not necessarily in the same place).

Unfortunately, due to the physical characteristics of the cassette tape filing system you have to do the equivalent of working through every book, cover to cover, until you reach the one you want; the other, more direct, and therefore faster, system requires a disk controller and drive(s).

To complicate matters further, there are certain specifications which do not have to be 'declared' to the computer; these are said to be 'defaults', which means that unless you tell the computer otherwise, it will assume certain things about your filing requirements.

If the explanation so far conjures up a mental image of a railway goods yard bustling with activity then that is not too far removed from the actual state of affairs; the only difference is that here the trains tend to leave only when they are full or when the station is closing its platforms down.

To recap: a file as far as we are concerned is really a peripheral — the screen, the keyboard, a cassette tape recorder, the printer, a disk system (disk controller plus disk drive or drives), the speech synthesiser, etc. In some cases the name of the peripheral is also the name of the file (e.g. CS1, CS2), while in others the peripheral name and the file name combine to give 'device-name.file-name' (e.g. DSK1.INVADERS). All data which are being moved, either from a peripheral into the computer, or from the computer out to a peripheral, have to stop temporarily in a buffer (waiting area in memory for data in transit, specifically set aside for certain peripherals), before proceeding to their destination.

We have covered in broad outline what a 'file' is as far as TI BASIC is concerned. TI BASIC is based on what is known as a COS, or Cassette Operating System, which means that, as standard, it is designed to operate with a cassette-based or tape-based filing system. A 'file' created using TI BASIC is rather clumsy to manipulate, and is thus not a suitable medium for storing large amounts of data. There is the added difficulty that there is no inbuilt checking mechanism as there is for programs, so that you either have to have a very trusting nature or time-consuming and repetitive checks of your own — and these could all come nastily unstuck because of a piece of iron oxide which chooses to go missing from the section of tape which the computer is trying to read.

Having opened your file and perhaps generated some data and put it into (onto?) the file, you can then proceed to 'process' it. This processing can take many forms: reading in old information, updating it, or performing calculations upon it (providing it is in a form suitable for this) and writing it back out again; searching for a particular item or items in a file; sorting (arranging) the data in a file in a particular way (alphabetically, numerically, ascending or descending order, etc.); printing it out, either on the screen or to the printer; and so on.

Let us now examine cassette filing using the commands available in TI BASIC. The commands are OPEN #, INPUT #, PRINT #, and CLOSE #, and in fact the TI manual makes something of a meal of the whole thing by not restricting its discussion to those details which are specific to cassette files alone. In addition there are certain items of information which have no real function whatsoever, whether in TI BASIC or Extended BASIC.

The OPEN command is the logical command with which to begin. It is this which specifies the format which the transferred data will take, and it has several components:

File-number: all files are addressed through a unique number between 0 and 255 (or less, depending upon the capabilities of the file). File #0 is the screen and keyboard and cannot be OPENed or CLOSEd. In fact PRINT #0 and INPUT #0 are exactly the same as plain PRINT and INPUT, but when using file numbers it can be convenient to refer to them through variables; thus PRINT #F could be used to print data onto the screen when F = 0, and to another peripheral (a printer perhaps) when F =1 (or whatever is chosen). Likewise INPUT #F can be used to obtain data from the keyboard when F = 0, and from a peripheral when F = 1 (or whatever). However, you cannot include an 'input prompt' (i.e. the material in quotes in 'INPUT "ENTER YOUR INSIDE LEG LENGTH:" : I$') with INPUT #F, and anyway, none would usually be necessary — what purpose would it serve to 'talk' to the tape recorder? A file number is always entered as the hash sign (#) followed by the number (e.g. #1, #22; you can even have an expression: #12 * 12 or #7 + A * B). A file number *must* be used with OPEN.

File-name: the file name in this case refers to either CS1 or CS2. You can use either of these in quotes (OPEN #1:"CS1"), or as string variables (OPEN #1: Z$ — where Z$ contains "CS2"), or even a mixture: OPEN #1: "CS" & STR$(N) — where N is either 1 or 2. A file name *must* be used with OPEN.

File-organisation: the only organisation permitted with cassette files is *Sequential*, and it is a default option, which means that you don't have to use it with OPEN. Sequential files have to be read/written one after the other. The tape/LP contrast is very apt: think of being able to put the stylus straight onto the fifth track of an LP, while with a tape the previous four tracks have to pass the playback head first — much slower.

File-type: again this is rather a daft inclusion, as for all practical purposes the only realistic specification is *Internal* (which means that the data are stored in a manner which the computer can process quickly). However, the default option is DISPLAY (a much more involved, inefficient, time-consuming and limited format) which seems illogical. You *must* therefore specify INTERNAL as the file type.

Open-mode: this tells the computer whether the data are to be transferred from peripheral to computer, or vice versa. The two specifications are INPUT and OUTPUT. INPUT refers to data coming into the computer, OUTPUT to data going out to a peripheral. The default option is one called UPDATE which apparently cannot be used by cassette files. You *must* therefore specify either INPUT or OUTPUT, not forgetting that only CS1 can be used for OUTPUT; CS2 can't (except on some machines: see **Hints and Tips**).

Record-type: a record is an item of data (a name, an address, a number, etc.), a group of which go to make up a file. The two record-types which can be specified are *Variable* or *Fixed*, followed by a number (of which more later). Because of the fact that cassette files can only be *Sequential*, the default type is *Variable*, which is confusing because although that is efficient in terms of storage, the manual gives FIXED as the appropriate record type with cassette files. The reason for this is not totally clear. However, the confusion is further increased because although you can specify the number of characters in each record, if you do specify FIXED with a number of characters (e.g. FIXED 100, which makes the buffer size 100 characters in length) the computer will to an extent ignore what you've told it, because with cassette tape records there are only three lengths allowed: 64, 128, and 192. A specification like FIXED 100 is actually set up by the computer to be FIXED 128, as 128 is the nearest higher value to 100. The difference in length between 100 characters and 128 is made up by the computer through 'padding' — i.e., wasting space. On top of all that, certain peripherals have their own restrictions as to the number of characters which they can handle in any one record (e.g., the Thermal Printer — one from the distant past and not now available — which can only take 32 characters per line), which the computer will compare with what you have told it, and woe betide you if you got it wrong. You must therefore specify FIXED, and if you don't specify a length in characters, the default is 64.

File-life: finally, in an act of sheer lunacy, a specification is included which refers to the 'life' of the files you create. It is lunatic because there are two possibilities: *Permanent* and something which we must assume is called *'Temporary'*; the default is *Permanent*; and you can only have *Permanent* files on the 99s so it doesn't matter anyway!

There is an awful lot of information here; even when the trees have been pruned it is still difficult to see the wood. As a general recap and guide to cassette file information, note the following:

File number:	1 to 255
File name:	CS1 or CS2 subject to restrictions
File organisation:	don't bother
File type:	INTERNAL
Open mode:	INPUT or OUTPUT accordingly
Record type:	FIXED according to the manual, and you may want to add a number: 64, 128, or 192, whatever is equal to or greater than your requirements
File life:	don't bother

So far we have only examined the format for the OPEN statement; once we've opened a file and done something with it we will then want to close it. Don't argue; because of the way that the buffers work some of your data could still be languishing somewhere, waiting to be added to, so that the computer will send them on their way. The only way to get round that once you've finished is to use the CLOSE command, which empties the relevant buffer by sending any remaining data packing, and then closing down the buffer. That's why the manual recommends that you should always use BYE to exit from BASIC, because BYE empties all buffers and then closes down. QUIT (FCTN =, or Shift Q, depending upon your machine) doesn't empty the buffers first, it just shuts up shop, thus losing any data which might still have been around. Under what circumstances would you want to use BYE? Well, if the program has just crashed with an error you might not be pleased to lose any information currently still on board.

The CLOSE format is just CLOSE with '#' and the relevant file number (e.g. CLOSE #1, CLOSE #F, etc.).

At last we come to the more informative section: just how DO you decide on your file structure?

A file is very much like a form (and if you hate filling in other people's forms you're in for a treat, because here you get to design your own form, for you to fill in) where all the little boxes in which information is normally put have been placed end to end — a weird sort of form, but that's the way things work with cassette files (and with most others).

Taking an example and working it through, let us cater for Mr X's need to store details on his friends (1984 where are you?). Suppose he wants to store these details:

1) Forename
2) Other names
3) Surname
4) Address
5) Sex
6) Age
7) Birthday
8) Hobbies, Likes, Dislikes (so that he can buy them the most pleasing presents at various times of the year)

To begin with, they will have to be entered into the computer so that it can then store them on tape. All the items listed above can be entered as strings (and assigned to string variables): for example, the age given can be processed to yield an integer number of years, and then CHR$( ) used to give a single character which will cover all ages from 0 to 255 (which, of course, is more than adequate); the sex can be simply M or F (unless some clever-clogs wants to put something else); and the birthday can be encoded as two characters whose ASCII codes refer to the month and the day. The aim is to compress the data as much as possible. The following variables might be used to hold the data:

1) F$ (for Forename)
2) M$ (for Middle names)
3) S$ (for Surname)
4) L$ (for Location)
5) G$ (for Gender)

6) A$ (for Age)
7) B$ (for Birthday)
8) T$ (for Tastes)

We won't go into exactly how the data are validated after input (to make sure that an impossible birthday date has not been entered, for example), but here is a sample input:

1) Ian
2) Petrovicz
3) Knightly
4) 2, The Cottage, Milton, Beds
5) M
6) CHR$(27)
7) CHR$(25) & CHR$(12) :i.e., day & month
8) Hobbies: Mugging little old ladies; Likes: Football; Dislikes: Televised Sport

So far, so good. We have our information format sorted out; now we need to work out how to open the file to CS1. Certain items of information could prove to be rather lengthy — the address, for example, or the list of hobbies etc. — so being wary we might settle on the maximum 192 characters per record (what you see above constitutes one record), so FIXED 192 is the appropriate specification. We're transferring the data from computer to tape, so the specification will be OUTPUT. The remaining specifications will be INTERNAL (see the list of cassette file specifications above), and a file number — say 1. The final OPEN statement will be:

```
OPEN #1: "CS1", OUTPUT, INTERNAL, FIXED 192
```

We now have the vital data to begin writing the program to place the data on tape — the instructions for setting up the file. The only problem remaining is how actually to get them onto tape. That is solved by the use of PRINT #. We simply PRINT out all our strings (see above) thus:

```
PRINT #1: F$:M$:S$:L$:G$:A$:B$:T$
```

What happens when the statement above is executed by the computer is this:

Pause
Steady tone (called the Header tone)
Data are transferred

This happens every time the computer encounters a PRINT #, so that doing separate PRINT # for each one of the string variables individually would have taken ages. It is always best to put out as much data as possible each time when using PRINT #.

That's all there is to it (!). When you wish to read the data from tape, you simply specify almost the same OPEN statement — this time specifying INPUT and not OUTPUT — and you use INPUT # with a list of string variables (or whatever you used) which match up exactly in number to those in the corresponding PRINT # statement earlier (note that they don't have to have the same variable names — as long as you know which item is assigned to which string variable).

As to exact space considerations, the manual does give some help. A number always occupies 9 'positions' (bytes, characters, spaces in the record, etc.), while a string always occupies its length plus 1 ("HELLO" occupies 6 positions, for example). In our odd example file above, there are 8 strings with a maximum total record size of 192 positions. This means that the actual data must be contained within 192 − 8 = 184 characters. If it is longer than this then problems will arise as the first 192 characters are transferred to tape, and the remainder will wait in the buffer until either more data are added to it to take it to the 192 character threshold, or until a CLOSE # command (or other closing of the file) causes it to be transferred. Subsequent reading of the file will result in virtually useless data being transferred, so it is a point to watch very carefully.

The other items in the manual (EOF, DELETE, REC, RESTORE, RELATIVE, APPEND, UPDATE) do not apply to cassette files and will only concern you if you purchase either a disk system or certain other items of equipment.

Graphics and Plotting

There is something inherently satisfying in being able to point proudly to a sketch or drawing and claim responsibility for it. Those of us not blessed with skills in reproducing life with pencil and paper must often have wished that we could draw. Well, now perhaps we can — with a little help from a friend.

Most of the computers now available allow you to define shapes and put them together on the screen to make reasonably satisfying images. Some offer facilities which enable you to plot a series of dots on the screen, and to produce straight lines, curves, even circles, with simple commands. Unfortunately the PLOT and DRAW commands have not been included in TI BASIC, but with a little skill and a lot of patience the diligent programmer can mimic those two commands by making use of the facility for defining 'characters'.

Indeed, some highly pleasing displays can be produced using both the redefinable and user-definable characters, as I hope you will see.

This chapter deals in outline with the general subject of Graphics and in particular with Plotting. Many computers which plot graphs or draw pictures do so at a very rapid rate, but because PLOT and DRAW have to be imitated in TI BASIC through the use of subroutines and character redefinition, the speed of plotting does not bear comparison with other machines.

Likewise, without access to the Sprites that Extended BASIC and certain other modules offer, the movement of shapes over the screen is also slow, but there are programming techniques which can help to improve matters.

We will look at two main areas:

1) PLOT and DRAW
2) Moving shapes about

In some cases there will be short routines as examples.

Plot and Draw

Because of the facility for defining characters, we can produce subroutines which give several degrees of resolution. The degree of resolution is a measure of the size of the smallest point which can be plotted on the screen. The degrees are:

a) High resolution:	192 rows by 256 columns
b) Medium resolution:	96 rows by 128 columns
c) Low resolution:	48 rows by 64 columns

Although it is well beyond the scope of this book, it is worth noting that it is possible — with the aid of further equipment — to plot to ALL 192 by 256 dots on screen on the 99/4A, but not on the 99/4. The two models differ in the chip used to produce the display.

The principles of operation of the subroutines which give different degrees of resolution are different from each other, but the general principle of redefining characters applies throughout. To save space only the high resolution plot will be discussed as an example (see Listing GP.1.), although listings (each with its own initialisation) are given for all degrees, as well as for DRAW. In each case the listings begin at line 100 for simplicity's sake; they may need to be renumbered to fit in with your own routines. Note the location of the subroutines — they are all placed BEFORE the main routine for reasons which are given in the **Hints and Tips** chapter.

The explanation of the High Resolution subroutine is presented here as a 'scenario'. There is a tendency when dealing with complex mathematics to explain a particularly difficult procedure through the use of anecdotes — parables, if you like. This method might help the reader to gain an insight into the workings of the subroutine in the least painful manner possible. Either way, it should give you something to think about.

Imagine a room in which the floor is covered with square tiles. There are (surprise, surprise) 24 rows of 32 tiles, each tile being subdivided into 8 by 8 smaller squares. There are 256 types of tile, some with differing patterns or shapes drawn on the upper surface, and there are 768 of each type of tile.

Each tile has its type marked as code number on the back, ranging from 0 to 255. On some of the tiles the pattern has been

```
        G R A P H I C S   &   P L O T T I N G
        = = = = = = = =   =   = = = = = = = =
```

LISTING GP.1. : HIGH RESOLUTION SUBROUTINE AND INITIALISATION

```
100  CALL CLEAR
110  DIM C$(128)
120  GOTO 340
130  Y = INT(R / 8 + .875)
140  X = INT(C / 8 + .875)
150  CALL GCHAR(Y, X, H)
160  IF H > 31 THEN 230
170  IF S = 159 THEN 330
180  S = S + 1
190  C$(S - 31) = Z$
200  CALL CHAR(S, "")
210  CALL HCHAR(Y, X, S)
220  H = S
230  H = H - 31
240  B = C - X * 8 + 8
250  P = 2 * R - 16 * Y + 16 + (B < 5)
260  IF B < 5 THEN 280
270  B = B - 4
280  I$ = SEG$(B$, POS(H$, SEG$(C$(H), P, 1), 1), 4)
290  I$ = SEG$(I$, 1, B - 1) & "1" & SEG$(I$, B + 1, 4 - B)
300  I$ = SEG$(H$, POS(B$, I$, 1), 1)
310  C$(H) = SEG$(C$(H), 1, P - 1) & I$ &
     SEG$(C$(H), P + 1, 16 - P)
320  CALL CHAR(H + 31, C$(H))
330  RETURN
340  S = 31
350  CALL HCHAR(1, 1, S, 768)
```

```
360  B$ = "0000.0001.0010.0011.0100.0101.0110.0111.1000.1001
     .1010.1011.1100.1101.1110.1111"
370  H$ = "0....1....2....3....4....5....6....7....8....9...
     .A....B....C....D....E....F"
380  Z$ = "0000000000000000"
```

RESERVED VARIABLES

The following variables should NOT be altered by any routine other than the subroutine:-

S, C$(), B$, H$, Z$

The following variables may be altered, but will be overwritten by the subroutine:-

B, H, I$, P, X, Y

CO - ORDINATES

ROWS : 1 TO 192

COLUMNS : 1 TO 256

LISTING GP.2. : MEDIUM RESOLUTION SUBROUTINE AND INITIALISATION

```
100  CALL CLEAR
110  DIM C$(128), H$(2)
120  GOTO 330
130  Y = INT(R / 4 + .75)
140  X = INT(C / 4 + .75)
```

```
150  CALL GCHAR(Y, X, H)

160  IF H > 31 THEN 230

170  IF S = 159 THEN 320

180  S = S + 1

190  C$(S - 31) = Z$

200  CALL CHAR(S, "")

210  CALL HCHAR(Y, X, S)

220  H = S

230  H = H - 31

240  W$ = C$(H)

250  B = C - X * 4 + 4

260  P = 4 * R - 16 * Y + 14 + (B < 3)

270  IF B < 3 THEN 290

280  B = B - 2

290  I$ = SEG$(H$(B), POS(B$, SEG$(W$, P, 1), 1), 1)

300  C$(H) = SEG$(W$, 1, P - 1) & I$ & SEG$(W$, P + 1, 1)
     & I$ & SEG$(W$, P + 3, 14 - P)

310  CALL CHAR(H + 31, C$(H))

320  RETURN

330  S = 31

340  CALL HCHAR(1, 1, S, 768)

350  B$ = "03CF"

360  H$(1) = "CFCF"

370  H$(2) = "33FF"

380  Z$ = "0000000000000000"
```

RESERVED VARIABLES

The following variables should NOT be altered by any routine other than the subroutine:-

B$, C$(), H$(), S, Z$

The following variables may be altered, but will be overwritten by the subroutine:-

B, H, I$, P, W$, X, Y

CO - ORDINATES

ROWS : 1 TO 96

COLUMNS : 1 TO 128

LISTING GP.3. : LOW RESOLUTION SUBROUTINE AND INITIALISATION

```
100  CALL CLEAR
110  GOTO 210
120  Y = INT(R / 2 + .5)
130  X = INT(C / 2 + .5)
140  CALL GCHAR(Y, X, H)
150  H$ = SEG$(B$, POS(C$, STR$(H), 1), 4)
160  P = C + 2 * R - 2 * X - 4 * Y + 4
170  H$ = SEG$(H$, 1, P - 1) & "1" & SEG$(H$, P + 1, 4 - P
180  H = VAL(SEG$(C$, POS(B$, H$, 1), 2))
190  CALL HCHAR(Y, X, H)
200  RETURN
210  DATA 000000000F0F0F0F, 00000000F0F0F0F,
     00000000FFFFFFFF, 0F0F0F0F, 0F0F0F0F0F0F0F0F,
     0F0F0F0FF0F0F0F0
```

```
 220  DATA OFOFOFOFFFFFFFFF,FOFOFOF,FOFOFOFOOFOFOFOF,
      FOFOFOFOFOFOFOF,FOFOFOFOFFFFFFFF, FFFFFFFF,
      FFFFFFFFOFOFOFOF
230  DATA FFFFFFFFOFOFOFOF,FFFFFFFFFFFFFFFF
240  FOR I = 33 TO 47
250  READ D$
260  CALL CHAR(I, D$)
270  NEXT I
280  B$ = "0000.0001.0010.0011.0100.0101.0110.0111.1000.
     1001.1010.1011.1100.1101.1110.1111"
290  C$ = "32...33...34...35...36...37...38...39...40...
     41...42...43...44...45...46...47"
```

RESERVED VARIABLES

The following variables should NOT be altered by any routine other than the subroutine:-

B$, C$

The following variables may be altered but will be overwritten by the subroutine:-

H, H$, P, X, Y

CO - ORDINATES

ROWS : 1 TO 48

COLUMNS : 1 TO 64

```
LISTING GP.4. : DRAW SUBROUTINE
-------------------------------

100  DX = X2 - X1
110  DY = Y2 - Y1
120  IF (DX = 0) * (DY = 0) THEN 200
130  IF (DX = 0) + (DY = 0) THEN 270
140  IF ABS(DX) > ABS(DY) THEN 210
150  FOR L = Y1 TO Y2 STEP SGN(DY)
160  R = INT(.5 + L)
170  C = INT(.5 + X1 + DX / DY * (L - Y1))
180  GOSUB plotting subroutine entry point
190  NEXT L
200  RETURN
210  FOR L = X1 TO X2 STEP SGN(DX)
220  C = INT(.5 + L)
230  R = INT(.5 + Y1 + DY / DX * (L - X1))
240  GOSUB plotting subroutine entry point
250  NEXT L
260  RETURN
270  IF DY = 0 THEN 340
280  C = INT(.5 + X1)
290  FOR L = Y1 TO Y2 STEP SGN(DY)
300  R = INT(.5 + L)
310  GOSUB plotting subroutine entry point
320  NEXT L
330  RETURN
340  R = INT(.5 + Y1)
350  FOR L = X1 TO X2 STEP SGN(DX)
360  C = INT(.5 + L)
370  GOSUB plotting subroutine entry point
380  NEXT L
390  RETURN
```

```
    RESERVED VARIABLES
    ------------------


    The  variables  which should NOT be altered by  any  calling
routine are as directed by the relevant plot subroutine.


    The  following  variables  may  be  altered,  but  will  be
overwritten by the DRAW subroutine:-

                          L, DX, DY, R, C
```

glazed over, and you cannot change it, while others have not been glazed, and you can change the pattern at will by filling in or rubbing out the smaller squares. When you do this to any tile, however, the new pattern is automatically copied onto the 767 other tiles of that type.

The rules are simple. The floor is always covered in tiles. Tiles numbered 32 to 159 inclusive can have their patterns changed by you, while 0 to 31, and 160 to 255, have fixed patterns. Tiles 32 to 127 (or 32 to 95 if you have a 99/4) are supplied with a pattern already printed on them, but you can rub this out and replace it with your own shape if you want to. The restriction is that you cannot add to those patterns; you must always wipe them clean and start afresh with your own, which of course you can add to.

Tile 32 is blank to begin with, as are a number of others (e.g. tile 31), which is very important.

The story begins with tile 31 (which is blank, remember) being placed in all 768 locations on the floor, leaving tiles 32 to 159 inclusive to be used to produce the image.

Your part is fairly straightforward. An instructor will give you a tile location on the floor (rows 1 to 24, columns 1 to 32) and you have to find out what the code number of that tile is, which you can accomplish by simply looking at the back.

Depending upon the code, you will do one of two things.

Either: you will have found that the tile code is 31, in which case you will select a tile code which has not so far been drawn upon, wipe one clean (thus automatically wiping all 767 others of that type), and replace the code 31 tile on the floor with it. This you can do until the tile code reaches 159, after which no further tiles are free to be redrawn.

Or: you will find that the tile code is greater than 31, in which case the tile will already have a pattern on it (you will have drawn it previously), and having found the code number you will then obtain from the instructor information as to which of the smaller squares have already been filled in.

You will then be given the location of a smaller square on that tile's surface which is to be filled in, and having done so — even if the square was already filled in — you will replace the tile on the floor, noting which of the smaller squares are now filled in, and informing the instructor of this.

You do this until either an image has been produced, or perhaps until you have run out of fresh tiles on which to draw.

Not much of a scenario, admittedly, but it should serve to give an outline description of the way in which the high resolution subroutine works.

The 'floor' is the television screen, while the 'tiles' are the characters which appear on that screen. The 'code' of each tile type is in fact the ASCII code — an explanation of which can be found in the **Jargon** chapter. Finding out the code of a tile at a given floor/screen location is performed by CALL GCHAR( ), while replacing a tile at a location involves either CALL HCHAR( ) or CALL VCHAR( ). The information 'as to which of the smaller squares have been filled in' is the hexadecimal definition string used by CALL CHAR( ), and which, for the purposes of the subroutine, has to be stored in a string array, because there is no counterpart to CALL CHAR( ) in TI BASIC as there is in Extended BASIC and with other modules (i.e. CALL CHARPAT( )).

There is a link between the element number of the array and the ASCII code of the character whose definition string is held in that element. That is, element 1 holds the definition string for character 32 (the 'space'), element 2 holds the string for character 33 (which is supplied pre-drawn with the shape of "!" — the shriek, shout, or exclamation mark), and so on. The link is that the element number is always the ASCII code of the relevant character minus 31. (32 − 31 = 1; 33 − 31 = 2; etc.) If you have difficulty understanding what an array is, or what an element is, then you will find an explanation in the **Jargon** chapter, and similarly for the hexadecimal definition strings.

You may have been surprised to learn that the screen is always filled with characters ('the floor is always covered in tiles'), for it may not at first be apparent when the screen is cleared with CALL CLEAR.

However, CALL CLEAR functions by swiftly placing the 'space' character (code 32) all over the screen. Try this on your machine:

> Switch on and select TI BASIC. We will use what is known as the *Immediate Mode*, in which the computer will immediately execute whatever instructions are entered instead of storing them away for later execution — i.e., accepting them as a *Program Listing*. When READY and the cursor re-appear, type:
>
> ```
> CALL HCHAR(1, 1, 31, 768)
> ```
>
> and press ENTER when you are certain that you have typed a correct instruction. You have now filled the screen with character 31, although the scrolling of the screen display once the instruction has been carried out will have placed different characters on the bottom (24th) line.
>
> When the cursor re-appears, type:
>
> ```
> CALL GCHAR(12, 16, H)
> ```
>
> and press ENTER, checking as before. What you are doing now is the equivalent of looking at the back of the tile at location row 12, column 16, for its code, and storing that code in a variable called H. Now type:
>
> ```
> PRINT H
> ```
>
> and the magic number 31 should appear (unless anything has gone drastically wrong), showing that although the screen appears empty, there is in fact something there. You could try using CALL CLEAR instead of CALL HCHAR(1, 1, 31, 768), and checking further for yourself.

Now we come to a direct discussion of the subroutine. It actually comes in two sections: a few lines which set up the basic items which will be used by the subroutine (the string array, for example, which will be used to hold the definition strings of the ASCII characters used in the plotted image), and then the subroutine itself. For reasons which are given in more detail in the **Hints and Tips** chapter, the subroutine is placed close to the beginning of the program listing; this enables it to be found much more quickly when required to be executed.

Listing GP.1. gives the subroutine and its 'initialisation' section. The initialisation is itself divided into two parts: the major part is placed after the subroutine so that the start of the subroutine is only three statements away from the first program line.

The routines which will make use of the subroutine begin at line 390, but before we delve that deeply into things, some of the statements will probably look rather daunting to the newcomer to computing, so a line-by-line explanation is in order. There are also concepts involved (for example INCLUSIVE OR) which are explained in more detail in the **Jargon** chapter.

There are a few points to note before we commence the explanation: firstly, there are some variables used by the subroutine which must not be used by the main routine (the 'calling' routine), as their alteration will almost certainly result in either unpredictable plotting, or even a crash.

Secondly, the usual format for the PLOT command is PLOT x,y where 'x' and 'y' refer to screen column- and row-coordinates. Because we are using a subroutine, and because those values cannot be included in the GOSUB statement, two variables, R for rows, and C for columns, will have the coordinates assigned to them BEFORE the GOSUB statement is executed.

Thirdly, in the interests of speed there are no 'validation' checks; that is, there are no checks within the subroutine to see if the calling routine has specified a coordinate which is outside the 1–192, 1–256 range allowed, or if the coordinate is anything other than a positive whole number (positive integer).

Let us now look in detail at the subroutine:

100: Although the intention is to cover the screen with character 31s, the fastest way initially to clear the screen (and therefore tidy things up) is by using CALL CLEAR, which puts 768 spaces on the screen.

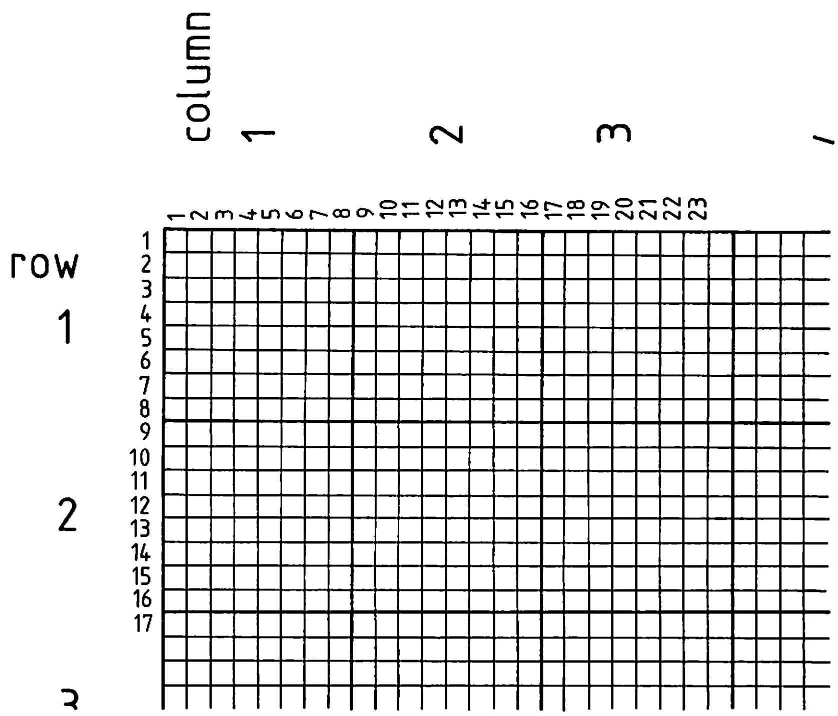
110: This statement has to appear in the program listing BEFORE any references to the array; the reason for this is given in some detail in the **Hints and Tips** chapter. An array, C$( ), with 129 elements (0 to 128, remember), is set up. I have chosen to use elements 1 to 128 to hold the hexadecimal definition strings which will be created by the subroutine, leaving element 0 free for future use (for example, to hold extra information). Each definition string will be exactly 16 hex digits long. It would be possible to add an extra two characters whose ASCII codes would give the screen row and column location of the redefined character concerned. These could be tacked on to the end of the definition string, and as CALL CHAR( ) uses only the first 16 digits in the string and ignores anything extra, there would be no interference with the program's execution. (This facility can be of use when needing to hold several definition strings for the same character, as you will see later.)

120: This causes the execution of the program to be transferred to line 340, where the second part of the initialisation resides. It ensures that the subroutine is not executed until required.

130: This is the 'entry point' of the subroutine; it is the 'destination line number' that goes with the GOSUB, so that once R and C have been assigned their respective values, all that is needed to perform the equivalent of PLOT x,y is GOSUB 130.

The equation in this statement has been reduced from ((R − 1)/8 + 1) in order to keep the number of brackets to a minimum (another trick to help improve execution speed, which is given in the **Hints and Tips** chapter), and also to keep the numbers to a minimum. This equation calculates the row location on screen of the character whose 8 by 8 dots are covering the area in which the point to be plotted is located. If the value for R is between 1 and 8, the plotted point lies under the character which is located on screen row 1; between 9 and 16, and it lies on screen row 2; and so on. The resulting screen location (minimum 1, maximum 24) is assigned to the variable Y.

140: A similar function is performed by this statement for the column coordinate. The value represented by C is processed to obtain the location of the screen column (minimum 1, maximum 32) under which the point to be plotted lies. This value is then assigned to the variable X. Diagram GP.5. illustrates how the 'dot' and screen co-



GP.5. 'Dot' and screen co-ordinate relationship

ordinates are related to each other. The large numbers are the screen locations, while the smaller ones refer to the actual plot locations.

150: This is the statement which performs the equivalent to looking at the back of the tile to find out what its code is. The command uses the values represented by Y and X to locate the character position on the screen, and then assigns the ASCII code of that character it finds there to the variable H. Remember that if there has been no previous plot to this general (8 by 8 dot) area the code of the character will be 31. Although the first part of the initialisation cleared the screen with CALL CLEAR, the second part of the initialisation replacing this by filling the screen with character 31s, thus releasing character 32 for redefinition.

160: This is the point at which the computer must make what we would call a 'decision'. If the code of the character found at the chosen location is not 31, then a different course of action needs to be followed compared with that to be followed if the code is 31. If a point has already been plotted in the general area (8 by 8 dots) then the code will not be 31, and the program will continue execution from line 230. Otherwise execution continues from the next line.

170: Another decision must now be taken. Because the computer has reached this line, it means that a point has been specified which requires a fresh 'tile' to be selected, and wiped clean. If, however, there are no more fresh tiles available, the computer must return from the subroutine without making a plot. The variable S is used to keep track of the availability of fresh tiles by storing the code of the last tile which was wiped clean. If that code reaches 159, then there are no further tiles available. Line 330 is the end of the subroutine, and it is to this line that execution is transferred if S is 159. If S is less than 159, then there are still tiles free for redrawing, and execution will continue from the next line.

180: We are now selecting the code of the next tile to be redrawn, so S needs to be incremented by 1.

190: The array which will hold the information on the pattern of each of the tiles needs to have an initial 'blank' hexadecimal string stored each time a fresh tile is selected. This so-called blank string is in fact 16 zeros, and all of the zeros are required so that subsequent processing can produce exactly the right definition string to describe the current pattern on the tile. Remember that the link between the code of the tile (its ASCII code) and the relevent element of the array is 'code minus 31', hence the S − 31.

200: Here we need to wipe the tile clean, and the simplest method is to use a null string ("") with CALL CHAR( ). The variable S holds the code of the tile which is to be wiped.

210: The newly-wiped tile is placed on the screen using CALL HCHAR( ) at row Y, column X.

220: The statements from line 230 onwards are those actually concerned with redefining the character (redrawing the tile); either the one just placed on the screen, or the character placed previously. The processing uses the variable H to refer to the ASCII code of the character (see line 150), but because the current screen location was previously not plotted to, and therefore will have caused

31 to be assigned to H, we need to copy the code of the new character into H. This code is held in S, so we simply use H = S.

230: To try to reduce the wastage of space and time during the processing, which will involve addressing the array C$( ) using (H − 31) — remember, the relationship between the ASCII code and the definition string element number — this line subtracts 31 from H so that it now corresponds directly to the relevant element of the array.

240: This line calculates the column location (1 − 8, within the 8 × 8 dot character matrix) of the point to be plotted.

250: This rather imposing expression calculates the position, in the 16 digit hexadecimal definition string, of the relevant single hex digit which defines the four-bit line on which the point to be plotted lies. Trying to explain how the equation was arrived at is a major undertaking, and outside the scope of this book, alas.

260:
270: A detailed explanation of the purpose of these lines is too complex to go into here; they simply reduce the value by 4 if the current value is greater than 4. The effect could have been achieved with just one line:

```
B = B + 4 * (B > 4)
```

but it executes more slowly in this form.

280: A series of complicated string manipulations begins at this point. This complex BASIC statement is intended to translate the relevant hex digit 'pointed to' by the variable P into its binary equivalent, ready to perform an INCLUSIVE OR. It functions thus:

1) The relevant element of the C$( ) array is addressed using H.
2) The relevant hex digit is extracted using SEG$( ) and P.
3) The position of this digit within the reference string H$ is obtained.
4) That position is then used to obtain the corresponding 4 digit binary equivalent from the reference string B$, and which is then assigned to I$.

In I$ we now have a four digit binary number between 0000 and 1111 which is the exact binary equivalent of the hex digit which defines the point we wish to plot.

290: The OR is performed by simply replacing the binary digit (which corresponds to the point that we wish to plot) with a "1", even if it is already a "1" (because it would be slower to check).

300: We now reverse the procedure which was followed in line 280. The statement in 300 translates the modified 4 digit binary number back into hex, assigning the resulting hex digit back into I$.

310: The new hex digit now replaces the old digit in the correct position in the definition string.

320: The new definition string (which may in fact have not been altered at all if the plot was to a point previously plotted anyway) is used to redefine the relevant character. Remember that in line 230 we subtracted 31 from H to make subsequent processing faster. Now we have to compensate for that difference of 31 when using H to address the ASCII code of the character we are redefining.

330: End of the subroutine, with an instruction to return to the calling routine.

Lines 340 to 380 contain the second part of the initialisation section, in which the variable S is set to a starting value of 31 and the screen is re-cleared, this time using CALL HCHAR( ) and character 31s. S is used in line 350 to speed up execution (see the chapter on **Hints and Tips** on why this is so). Line 360 contains the reference string of binary equivalents to the hex digits 0 to F; each equivalent is separated by a full stop to distinguish it from its neighbours. Line 370 contains the counterpart reference string which holds those 0 to F hex digits, which are spaced out with full stops so that the position of each hex digit corresponds exactly with the position of the first binary digit in the corresponding equivalents in the B$ reference string. Finally, line 380 holds the 'blank' definition string which is used to initialise each element of the C$( ) array as it is called up by the subroutine. Again, these 16 zeros are used within a string variable to speed up execution of the subroutine.

Well, there you have the plotting subroutine. Note that the explanation has had to be greatly shortened, or this chapter would never end. Now for some examples, which will show you not only how to use the subroutine, but also what this computer can do if you give it enough time.

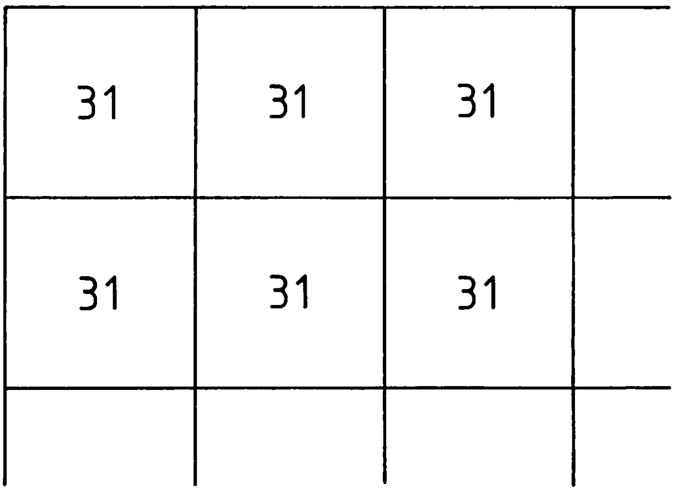
Diagrams GP.6. to GP.9. show, in a very brief and simple sequence, what happens when the first point (perhaps of a series) is plotted on screen. GP.6. reminds us that after the second part of the initialisation the screen is covered with characters 31s. If our plotted point lies on R = 14, C = 11 (see GP.7.), it will occur within the 8 by 8 dot matrix of the character at screen location Y = 2, X = 2 (cf. GP.5.). Selecting the next available character for redefinition gives us code 32, and after due calculation a definition string is produced (see GP.8.). This character now replaces the character 31 which previously occupied screen location Y = 2, X = 2; see GP.9., where the plotted point has been shown together with the codes of the characters now on the screen. (Note that if the point had been plotted using the *medium* Resolution subroutine, because of the larger point size — and hence reduced resolution — the actual plotted 'point' would have been a group of four of the high resolution points; if by the *low* Resolution subroutine, using a slightly different approach, it would have consisted of a 'point' produced by no less than 16 (4 by 4) of the high resolution points.)

Let us continue with a simple sine wave. It is, unfortunately, outside the scope of this book to discuss in detail the operation of some of the equations which will be presented here. The sine wave plot can be produced by adding these lines to the listing GP.1. and running the program.

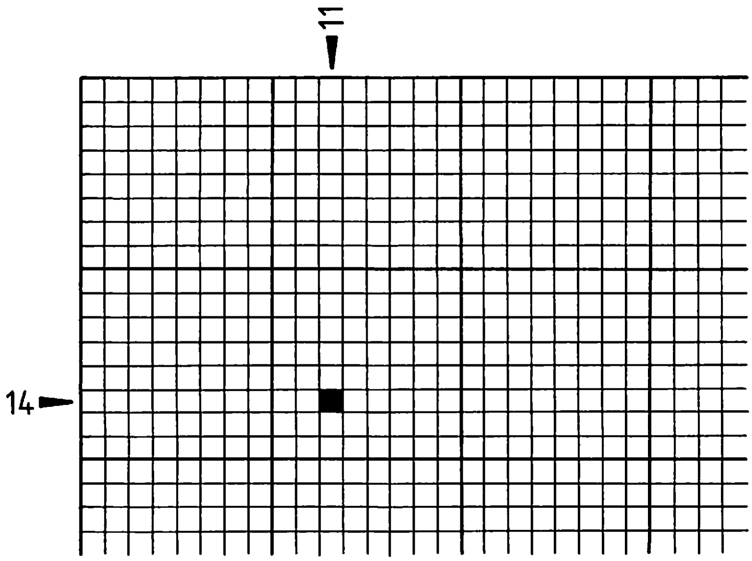
```
390 FOR C = 1 TO 256
400 R = 96 + INT(.5 + SIN(C / 9) * 40)
410 GOSUB 130
420 NEXT C
430 GOTO 430
```

(Note that the same block of lines — with different line numbers, due to the different size of the respective subroutines — can be used with all three of the resolutions given; however, the maximum value for C in line 390 will have to be reduced to fit within the restrictions for each degree of resolution: viz., 128, and 64. Likewise, the value 96 in line 400 here will also need to be reduced: 48, and 24.)

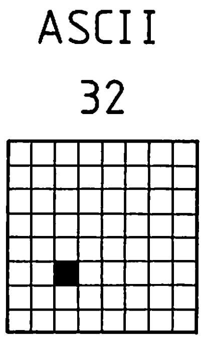
The screen colour will be green — you may change this to a more agreeable hue by inserting the necessary line somewhere in the second initialisation section — and the plotted points will be black. The routine will take a few minutes to draw the sine wave



GP.6. State of the screen after the second part of the initialisation

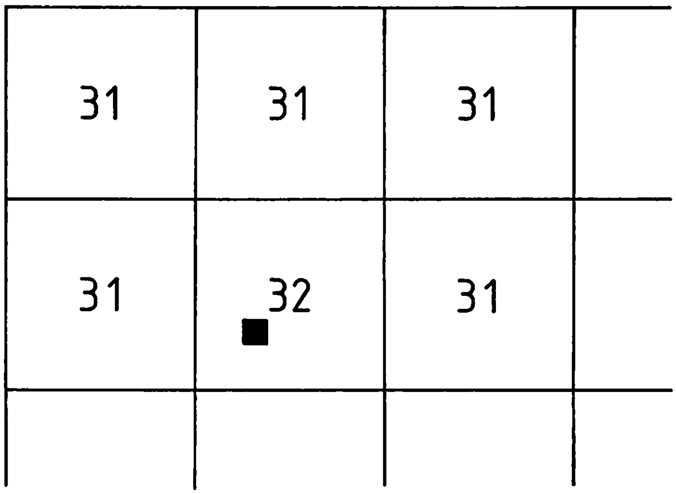


GP.7. 'Dot' location of example point



0000000000200000h

GP.8. Re-defined 'space' character containing a single point



GP.9. State of the screen after plotting the example point

across your screen; you can speed up the execution, and degrade the quality of the wave, by making the loop in line 390 step in 2s or 4s, or a value of your choice. By varying the factor of 40 you can affect the height (amplitude) of the wave, and by altering the division factor of 9 you can affect the frequency of waves on the screen. Experiment with the function in line 400, trying COSINE or even SINE + SINE. Some will crash, some will produce quite pleasant effects. The .5 in the equation is a simple trick to 'round' the value obtained — see the **Hints and Tips** chapter for further details.

By way of an example of just what the system is capable, here is a routine which will produce a three-dimensional image on the screen. A discussion of the principles involved would take the rest of this book, so you may have to content yourselves with simply looking at the end result. It is likely that printing errors will have crept into the listings, but do not despair. First read the chapter on **Printing Errors**, which won't solve your problems outright but should point you in a more successful direction. If you still find no enlightenment, write to me at the publishers' address, which is given elsewhere. Now, here is the routine:

```
390 CALL SCREEN(8)
400 V = 104
410 X1 = 96
420 X2 = X1* X1
430 Y1 = V / 2
440 Y2 = V / 4
450 FOR X5 = 0 TO X1
460 X4 = X5* X5
470 M = -Y1
480 A = SQR(X2 - X4)
490 FOR I1 = -A TO A STEP V / 10
500 R1 = SQR(X4 + I1* I1) / X1
510 F = (R1 - 1)* SIN(R1* 12)
520 R = INT(I1 / 5 + F * Y2)
530 IF R <= M THEN 600
540 M = R
550 R = Y1 - R
560 C = X1 - X5 + 32
570 GOSUB 130
580 C = X1 + X5 + 32
590 GOSUB 130
600 NEXT I1
610 NEXT X5
620 GOTO 620
```

Be warned that this routine will take about half an hour to produce its image. You could spend the time biting your nails in front of the TV, or you could go shopping. The function which is plotted occurs in line 510, and its degree of tilt is given by line 520, should you wish to tinker around with either of them. If you have difficulty visualising exactly what the final image is, imagine part of a series of ripples in a pond viewed in close-up.

So far we have looked at the High Resolution plotting subroutine; there are two others, both used in a similar manner to the 'hi-res' subroutine, but giving a lower resolution. The difference between them is a comparison between writing with a fine ball-point and a thick felt-tip. This time, because the explanations would be so space-consuming, the routines are simply presented as listings, with details regarding the reserved variables, etc. You can use them in exactly the same way that the 'hi-res' routine is used, except of course that you would use smaller values to correspond to their smaller ranges.

Drawing Continuous Lines

One routine which can be of use when creating simple technical drawings is an equivalent to DRAW. This function on other machines usually allows you to draw a continuous straight line between two specified points. Because of the limitations of TI BASIC this effect has to be obtained through a further subroutine which itself makes use (slowly) of the plotting subroutine. The variables X1, Y1 mark the starting column and row of the line, while X2, Y2 mark the end column and row. To draw a square you would need to specify the start and finish locations of four lines, which could be contained within a DATA statement. The values for X1, Y1, X2 and Y2 would need to be assigned before using GOSUB to transfer control to the DRAW subroutine. For example:

Line 1: X1 = 1
Line 2: Y1 = 1
Line 3: X2 = 64
Line 4: Y2 = 48
Line 5: GOSUB draw subroutine entry point

The values have been chosen so that they can be used with any of the degrees of resolution without crashing and without needing any re-scaling of coordinates.

Because of a lack of space, the DRAW subroutine cannot be dealt with in detail as was the High Resolution subroutine, but it should be possible to discern its operation from the listing. It is given as a block of lines beginning at line 100; this is for convenience only, and for practical purposes the routine would be placed in a listing AFTER the plotting subroutine which it would use, but BEFORE the second initialisation section for that subroutine.

Aspects of Plotting and Drawing which have had to be omitted because of space considerations include: Mixing text and plots — how to 'protect' specific characters from being (a) selected for redefinition, and (b) displaced by plotted points; a technique for storing the screen locations with the respective definition strings, restoring the standard character shapes for printing text on screen (menus, etc.) and then re-creating the screen display quickly; the possibility of producing a cassette or disk file containing all the necessary data; the possibility of producing a series of images and swapping between them at will; the list is endless (almost).

Moving Graphics

The TI manual deals adequately with the definition of shapes using strings of hexadecimal digits, but gives little or no information on the two most important aspects of graphics: animation of a shape where it stands on the screen, and movement of a shape across the screen. Few owners appear to experience difficulty with the first aspect, although they may not always have found the easiest approach. The greatest problems seem to arise when attempts are made to move a character (or group of characters) over the screen, coping with interaction with other characters and also keeping within boundaries — either those imposed by the screen, or those imposed by design.

Changing Shape

There are two routes to changing or animating a shape on the screen: either by using CALL CHAR( ) with different definition strings on the same ASCII character, or by using different characters which have been already defined with those strings — the brute force approach. The slow speed of TI BASIC means that the processing of definition strings while a program is running will not improve the quality of animation, so any strings which are used may well have to be processed outside the program — meaning that all eventualities have to be catered for and definition strings produced, which can severely limit the variety of response possible.

It is very useful to prepare a series of 'utility' routines for graphics work, freeing you from hours spent over graph paper laboriously creating consecutive image sequences. The kind of utilities from which you are likely to experience greatest benefit are these:

1) Movement of a shape upward
2) Movement of a shape downward
3) Movement of a shape left
4) Movement of a shape right
5) 'Mirror' imaging of a shape
6) Inversion of a shape

The first two are simplicity itself: If you take an example definition string thus: "1122448888442211" (it produces a 'chevron' shape), then to make it appear to move upward within the 8 by 8 dot matrix of a character all you need to do is successively to remove the first two hex digits. This will give you a series of definition strings (note that CALL CHAR( ) assumes trailing zeros if the length of the string is less than 16 hexadecimal digits):

a) 1122448888442211
b) 22448888442211
c) 448888442211
d) 8888442211
e) 88442211
f) 442211
g) 2211
h) 11
i) null string

which can be used with either 9 CALL CHAR( ) statements, 9 DATA statements and a loop to read them plus one CALL CHAR( ), a string array with the definitions stored in 9 elements plus a loop to assign them via the CALL CHAR( ), or even a single string containing all 9 strings (plus trailing zeros, which you must include — very important, as you will find) which is accessed 16 digits at a time using a loop, SEG$( ), and CALL CHAR( ).

Similarly, movement downward is achieved by adding leading zeros (you may not have to remove the last pair of digits each time, as CALL CHAR( ) only examines the first 16 digits in any string), giving the following strings:

a) 1122448888442211
b) 0011224488884422
c) 0000112244888844
d) 0000001122448888
e) 0000000011224488
f) 0000000000112244
g) 0000000000001122
h) 0000000000000011
i) 0000000000000000

You may now have begun to see possibilities for graphics manipulation: what about animating only a part of the 8 by 8 dot matrix — either vertically, horizontally, or diagonally — leaving the remainder intact; this might give the impression of one image disappearing behind another? Or perhaps moving the shape at a faster rate — the examples show movement one dot at a time, but you could slice the string up in fours instead of pairs of digits.

Moving a shape left and right is not so straightforward. Take the top row of dots in our chevron shape and move them left one dot at a time until the shape has moved out of the matrix completely:

a) 11
b) 22
c) 44
d) 88
e) 10
f) 20
g) 40
h) 80
i) 00

Do it to the entire string, and you obtain:

a) 1122448888442211
b) 2244881010884422

c) 4488102020108844
d) 8810204040201088
e) 1020408080402010
f) 2040800000804020
g) 4080000000008040
h) 8000000000000080
i) 0000000000000000

The example is a symmetrical shape, so it is possible to make the false assumption that there are simple, repeating patterns which can be made use of to shorten the calculation.

There are really only two practical routes to obtaining the 'transformed' definition strings: the first is perhaps clumsy and consumes proportionally more space than the second. The straightforward (or so it seems on the surface) method is to translate the entire definition string into its binary equivalent (as a two dimensional array of 8 by 8 elements), perform a shift right or left by as many elements as are required, and then translate back into hexadecimal. The alternative is to make use of the fact that movement left is equivalent to 'doubling' the value represented by the hex digit pair defining each line, and movement right is equivalent to 'halving' that value. To give you some idea of how this works, let us examine 'doubling' on the first row of dots (first hex pair), using decimal as an intermediate to show the working of the calculation:

> The first hex pair is 11h. This is 1 * 16 + 1 in decimal, which is 17d. Doubling gives 34d, which in hexadecimal is 22h.

There is one problem with this approach, apart from the slow execution: what happens if doubling produces a value in excess of 254d (11111110b or FEh), or if halving produces a value which is a fraction (i.e., ends in .5d)? For these two cases you would need checks and extra processing, reducing the speed of execution still further, and for those reasons I tend to prefer either the manipulation of an array of binary equivalents, or 'pair by pair' processing of a hexstring. There is also the additional factor that if you need to rotate your character in steps through 360 degrees then the method based on the array is suited to that processing.

The last two utilities, mirror-imaging and inversion, may be of only limited use to you, but they are well worth looking at if only for the experience of string manipulation.

So far I have not given any specific example routines, primarily because there are aspects of input validation involved which themselves would take a chapter to explain. However, for these last utilities I will give example routines, and leave the readers either to provide their own input validation or to be very careful when entering data.

Mirror-imaging simply involves 'flipping' a shape over, either vertically (left to right) or horizontally (top to bottom). The horizontal mirror image is very easily produced: all you have to do is to reverse the definition string, pair by pair. Thus the string "0123456789ABCDEFh" becomes "EFCDAB8967452301h", achieved by this short routine:

```
100 CALL CLEAR
110 CALL SCREEN(8)
120 P$ = ""
130 INPUT "STRING" : S$
140 FOR L = 1 TO 15 STEP 2
150 P$ = SEG$(S$, L, 2) & P$
160 NEXT L
170 PRINT "HORIZONTAL MIRROR IMAGE IS:" : P$
180 GOTO 120
```

Note that the string entered in line 130 ought to be the full 16 hexadecimal digits in length (I leave you to add your own validation routines).

The vertical mirror image is a little more complex. The 'left to right' mirror image of "01h" is "80h"; examine the binary equivalents to see that this is so:

00000001b becomes 10000000b

Each of the hexadecimal digits (0–9, A–F) has its mirror image counterpart:

HEX	BINARY	becomes:	HEX	BINARY
0	0000		0	0000
1	0001		8	1000
2	0010		4	0100
3	0011		C	1100
4	0100		2	0010
5	0101		A	1010

HEX	BINARY	becomes:	HEX	BINARY
6	0110		6	0110
7	0111		E	1110
8	1000		1	0001
9	1001		9	1001
A	1010		5	0101
B	1011		D	1101
C	1100		3	0011
D	1101		B	1011
E	1110		7	0111
F	1111		F	1111

Again the hexadecimal definition string is processed by pairs of digits, only this time each pair is 'swapped' — thus the mirror image of "32h" is not merely "C4h" but is "4Ch".

Everyone has their own particular way of doing things, and my favourite method for translating digits in a case like this is to use 'reference' strings coupled with POS( ) and SEG$( ). This routine is an example of that approach:

```
100 CALL CLEAR
110 CALL SCREEN(8)
120 H$ = "0123456789ABCDEF"
130 R$ = "084C2A6E195D3B7F"
140 P$ = ""
150 INPUT "STRING" : S$
160 FOR L = 2 TO 16 STEP 2
170 I$ = SEG$(S$, L, 1)
180 J$ = SEG$(S$, L - 1, 1)
190 I = POS(H$, I$, 1)
200 J = POS(H$, J$, 1)
210 I$ = SEG$(R$, I, 1)
220 J$ = SEG$(R$, J, 1)
230 P$ = P$ & I$ & J$
240 NEXT L
250 PRINT "VERTICAL MIRROR IMAGE IS:" : P$
260 GOTO 140
```

Again, there is no input validation, but the full complement of 16 digits is expected. This particular routine has been simplified in order to make its working more obvious; it is usually more complex than this because I tend to make each statement do as much work as possible. Thus lines 170 to 230 would normally appear as the single statement:

```
P$ = P$ & SEG$(R$, POS(H$, SEG$(S$, L, 1), 1), 1) & SEG$(R$,
POS(H$, SEG$(S$, L - 1, 1), 1), 1)
```

which taxes the eyes a little.

Finally, image inversion. This can be achieved in two ways: either you can alter the contrast by swapping fore- and background colours using CALL COLOR( ), but which will affect sets of characters rather than just one individual character, or you can again manipulate the definition string. This time the manipulation is digit by digit (whereas previously it has been by pairs of digits), and again I have used a reference string. The table of inversions is as follows:

HEX	BINARY	becomes:	HEX	BINARY
0	0000		F	1111
1	0001		E	1110
2	0010		D	1101
3	0011		C	1100
4	0100		B	1011
5	0101		A	1010
6	0110		9	1001
7	0111		8	1000
8	1000		7	0111
9	1001		6	0110
A	1010		5	0101
B	1011		4	0100
C	1100		3	0011
D	1101		2	0010
E	1110		1	0001
F	1111		0	0000

The pattern here is simple, but actually making use of it is not, and there may be some argument as to which is the best approach. Here is one suggestion:

```
100 CALL CLEAR
110 CALL SCREEN(8)
120 H$ = "0123456789ABCDEF"
```

```
130 P$ = "  "
140 INPUT "STRING" : S$
150 FOR L = 1 TO 16
160 P = POS(H$, SEG$, L, 1), 1)
170 P$ = P$ & SEG$(H$, 17 - P, 1)
180 NEXT L
190 PRINT "INVERTED IMAGE IS:" : P$
200 GOTO 130
```

As before, there is no input validation.

In each of the routines above you can 'visualise' the effect of the new definition string by inserting a couple of CALL CHAR( ) statements and redefining the characters on-screen.

Returning to the subject of animation, the second method involves movement of an entire character over the screen, keeping within certain boundaries and coping with interaction with other characters on-screen. What helps most here is not a huge series of standard routines which must be either adhered to (thus forcing your program to follow the dictates of someone else's programming) or modified to suit your purpose (and therefore may as well be written from scratch), but a careful, detailed examination of the processes involved. In other words, long before you reach the keyboard you would do well to have evolved an *algorithm* to which you can refer. Let us examine a possible algorithm with such specifications.

Firstly, we must define exactly the composition of the screen, the location of static objects, the existence and location of any boundaries, independent of the boundaries set by the computer's own limitations (here I am thinking of the limitations of CALL HCHAR( ) and its two companions). We must also decide which effects are to be used when exceeding those boundaries: should any mobile shape be prevented from crossing the boundaries; should it be allowed to 'wrap-around' from one boundary to another; should it be permitted to cross the boundary, but not be shown on-screen (thus avoiding complications with CALL HCHAR( ) etc.); or should it disappear in a glorious technicolor explosion? Likewise if its location after a move coincides with that of another static or mobile object, should it bounce off at the appropriate angle, should it be prevented from making the move, or should it explode as above?

In addition, we must also decide on both the degree of movement allowed, and on the total complexity of the movement on-screen; that is, we must decide whether the shape is to move one square at a time, and whether the program can support the movement of several shapes simultaneously (or nearly simultaneously).

Here is a suggested algorithm, not finely detailed, for that will depend solely upon your own requirements:

1) Locate all objects on screen
2) Accept instructions for movement
3) Calculate new co-ordinates
4) Check new location for:
   being within boundaries
   coincidence with other objects
5) Depending upon the results of the checks, respond appropriately
6) Calculate and validate the movements of other objects on screen
7) Perform other processes (e.g., time elapsed, fuel consumed, etc.)
8) Check other processes
9) Depending upon the results of those checks, respond accordingly
10) Go to (2)

Note that the use of the 'look before you leap' principle is perhaps the best approach to movement.

This chapter, despite its length, has only been able to scratch at the surface of the subject, and an enormous amount of material has been left undiscussed; hopefully later publications will redress the balance.

Printing Errors

One of the most exasperating problems which can face the newcomer to computing is that of printing errors and omissions in programs published in popular magazines and books. On the surface it seems straightforward enough: all the novice has to do is to type published listings into his machine (assuming that the programs are intended for his model) and soon he will have a thriving library of challenging and absorbing material with which he can delight and impress his friends and colleagues, all for the cost of a few pence each week/month.

Unfortunately, it rarely works that way. Apart from the fact that quality programs are not cheap, and TI-99 commercial programs still less so, the library tends to be filled with material which either would take the average school child an afternoon to write, or which has never run properly because there are errors in the algorithm (the 'rules of thumb' by which the program operates) or the listing is 'bugged', or the typist didn't quite copy the listing exactly — all the statements appear to be correctly entered, but somewhere something is not right. The latter are called *transcription* errors.

This chapter examines both those and *typographical* errors — misprints — often the reason why such programs won't run properly (or sometimes won't run at all). It is not really intended to be a professional aid to debugging, although it does go some way towards that. What it is intended to do is to give the new owner as much help as possible in uncovering and resolving errors caused by omissions and mis-spellings in published listings. It cannot possibly be exhaustive — the permutations are too numerous for that — and, as has been said elsewhere in this book, if you have anything to contribute or if you have insurmountable problems, then write to me, care of the publishers, enclosing a stamped, self-addressed envelope. I can't promise to hold the answer to Life, The Universe, and Everything, but if I can help I will.

Usually the listings are taken direct from the computer — many magazines insist on this precisely to avoid as many errors as possible — but even then careless programmers who contributed the programs often added program sections (patches) without first checking that they worked properly.

Common errors to be found involve one or more of the following:

1) Mis-spelling Reserved Words (these are the TI BASIC words which you cannot use as variable names — they are reserved for the computer: LET, IF, PRINT, etc.).
2) Mis-spelling of variable names. The biggest culprit here is the letter 'O' — often confused with the digit '0'.
3) Other mistaken entries — I for 1, etc.
4) Omission of spaces. Often the computer needs spaces between certain Reserved Words and other data.
5) Omission of punctuation marks. Many of these become evident only after someone else has pointed them out to you.
6) Omission of an entire line or lines. These can be among the most difficult to spot.
7) Omission of symbols: #, $, ∧, &, *, −, +, =, <, >, /, which are easy to spot, but unless the context gives some clue it can be difficult to decide exactly which symbol is missing.
8) Omission of sections of parameters for CALLs (the Subprograms).

To enlarge on those categories:

1) It is not usual to find reserved words being mis-spelled in computer-generated listings, but it can happen if the listing is typed. By and large these errors are very easy to spot, and even if you are an 'abject beginner' you can quickly work out what the word should be, if only by examining the list of reserved words given in the TI manual. Rarely you might find that the reserved word has been omitted (the exception is LET in TI BASIC, the use of which is optional), although the context will usually tell you which word is missing:

```
IF A = 7 1200
```

is straightforward — the missing word is THEN, which needs to go between the 7 and the 1200. You might think that this is a laughable example, but somebody (who shall remain nameless) got stuck on it. Occasionally you may find what appears to be a mis-spelling, but which is not. TI BASIC allows both GOTO and GO TO (and GOSUB and GO SUB); check the manual. It appears however that GOTO is stored differently in memory from GO TO, although the function it performs is identical. A discussion of the internal storage of a listing is beyond the scope of this book.

2) Another opportunity for mis-spelling — in some instances it can come down to a transcription error — arises with the names of Variables. As has been said, the confusion which can occur over the letter 'O' and the digit '0' results in that being the most common reason for a program either stopping with a BAD NAME or BAD ARGUMENT error, or failing to run as designed. Whoever decided not to follow the convention that either zero or the letter 'O' have an 'oblique' — a diagonal stroke — through them, has a lot to answer for. It can be all too easy when reading from a poorly reproduced listing to mistake an 'O' for a '0', and one program which I examined for an owner was filled with this type of transcription error. For example:

```
a) O = 0 + 1
b) A = B* 0* C
c) FO = F0 - 2
d) 0 = O - Z
```

Here the errors are quite numerous; the correct lines, as far as can be discerned, are:

```
a') O = O + 1
b') A = B* O* C
c') FO = FO + 2
d') O = O - Z
```

In (a), the value assigned to variable O will never exceed 1, and it can be obvious from the context alone that O is being used as a 'counter', incrementing by 1.

In (b), multiplying any number by zero produces zero, which makes nonsense of B * 0 * C — there is no purpose to having a zero in the equation unless it is one of the values represented by a variable — so again it is obvious from the context that the '0' should in fact be the variable 'O'.

In (c), there are two variables: FO and F0, both of which are valid variable names, but in the listing FO cropped up in several places, while F0 appeared only in the faulty line. In this case it is best to tread warily, as the statement could in fact have been entirely correct. Additional information was necessary (obtained by examining the whole listing) before F0 could be isolated as an error.

In (d), the faulty statement caused the program to stop with an INCORRECT STATEMENT error before it would run, because the computer could not accept the number zero as a valid variable name — the manual tells you exactly what constitutes a valid variable name.

It is well worth while, before beginning to type in a listing, to compile a list of all the variable names in the listing, to which you can refer. It is likely that, as your concentration can be elsewhere while typing, you will not notice any errors in respect of variable names; the act of compiling the list of variables will go some way to preventing this.

3) Here again the compilation of a variable list can help to reduce the incidence of transcription errors; the most likely are 'I' for '1', 'S' for '5', 'Z' for '2', 'B' for '8', and 'G' for '6'; occasionally you may come across '?' for '7'. The general rule is to examine the program before you attempt to key it in, *and* to check it as you type, being as aware as possible of the sense of what you are typing. The kind of thing to watch for is:

```
FOR L = I TO B
```

and if you check the listing and find that the variables I and B do not occur anywhere else (and are not assigned values anywhere), then it is a safe bet that '1' and '8' have been misprinted or wrongly copied.

4) Problems due to omission of spaces can sometimes cause extreme frustration. Many of the reserved words will, when entered, automatically add spaces where they are necessary when listed, but, for some reason, one or two words need the spaces to be entered for them. The reserved word DATA is one of that type, and if no space is entered between DATA and the first item in the data-list the computer may not reject the statement even when it attempts to run the program. For example:

```
DATA0, NAME, 1, NUMBER, 22
```

may eventually be rejected by the computer with either

INCORRECT STATEMENT, or DATA ERROR, but under differing circumstances. If the computer encounters the erring DATA statement while running a program (i.e., if it has to 'pass over' as it would do with a REM statement) then the error is INCORRECT STATEMENT IN . . .; if however the computer has not encountered it, but instead elsewhere executes a READ instruction which attempts to read data from that data list, the error will be DATA ERROR IN. . . .

Similarly, FOR . . . TO . . . STEP . . . NEXT needs a space to be inserted between the FOR and the 'loop control variable':

```
FORL = 1 TO 15
```

will not run until you put a space between the FOR and the L.

One other instance involving 'spaces' is actually an error of 'inclusion'. It is possible (on the 99/4A alone) accidentally to press either the Control key (CTRL) or the Function key (FCTN) coupled with another key and insert what looks like a space on the screen. If your haste is such that you are rarely watching the screen to check on what you are typing, it is likely that if you notice the extra 'space' (and the omission of whatever character you intended to type) you will assume that it IS a space. Sometimes these 'pseudo-spaces' will be accepted by the computer (and you could be in for some surprises when you list the program — see the chapter on **Hints and Tips**) but will not allow the program to run.

5) One thing that the 99s are quite strict about is punctuation, and on occasion it can be difficult trying to decide what is missing from a statement which TI BASIC won't accept. The most common omissions are: one or both quotation marks which enclose a string constant ("HELLO MOTHER" is a string constant); print separators in a PRINT statement (i.e., ",", ";", and ":", missing from a 'print list' — e.g. PRINT A B Z$); and too many or too few brackets. Generally the omitted quotation marks are easily noticed in a listing (and there can even be too many, although that rarely happens), but the intended effect of the missing print separators can often only be guessed, sometimes by trial and error.

The omitted brackets can usually be checked on by adding up the number of left-hand brackets and comparing the total with the right-hand number. That isn't the end of the story, however; you need to break the statement down into its components as far as bracketed expressions are concerned, and examine each in turn, to ensure that the correct number of brackets have been used for each expression/function.

In one case I know, a DEF statement contained one too few left-hand brackets. On running the program, no error message was generated concerning the DEF statement itself, but when another statement which appeared later in the program attempted to make use of the DEFined function, an INCORRECT STATEMENT error was produced referring to the statement being executed. It took half an hour before the true origin of the error was uncovered.

6) It is not unusual for published listings to lack a line or even a block of lines, and the first indication may be either an error elsewhere as variables which have been undefined (due to the omission) are involved in 'division by zero' errors or similar situations, or a failure of the program to perform stated functions. A little detective work may be necessary to determine whether lines are missing and, if so, exactly what form they should take. For example, if these statements appeared in a program which controlled a shape on screen:

```
R = R + 1
C = C + 1
C = C - 1
```

you would be justified in thinking that there ought to be one more, to complete the pattern:

```
R = R - 1
```

This kind of omission underlines the necessity of providing full documentation with all programs, for it can often be the only practical source of information to which you can refer when in difficulty.

7) Missing symbols are among the more difficult to cope with, especially if there are no clues as to the likely function of the statements concerned. For example:

```
C = R  A * B
```

seems to have an omission between the variable R and A. It might, of course, be the case that there is a variable named, say, RTA in the program, which tends to point strongly to the omission being a letter 'T' (once again highlighting the usefulness of having produced a list of variables). Often, by scanning the rest of the listing, you may be able to see other, similar, omissions and thus be able to build up a picture about the nature of the omission. Some magazines have to insert certain symbols after typesetting the rest of the listing, and that step has, on occasion, been forgotten in the past. You may notice, therefore, that all the symbols are present bar one or two, thus reducing the number of possibilities which have to be considered. In the above example, the asterisk (*) has been printed, so the missing symbol is unlikely to be that. You might have found from examining other statements that the addition and subtraction symbols were present elsewhere, as was the division (/) sign. Out of the remaining symbols, the 'Relational Operators' usually are to be found within brackets, which leaves the ampersand (&) which is the string concatenation operator (and as there are no strings involved here, it is not a likely candidate) and the symbol for exponentiation, the caret (∧). For the most part, you will be very fortunate indeed if you can similarly isolate the most likely symbol so easily.

Sometimes the string symbol ($) which identifies a string variable can be missing (there is a strange form of BASIC which allows you to DEFine the name of a numeric variable to be that of a string variable: DEF R = R$, and, if you are translating a non-99 program, that might explain the apparent omission) but the context usually clearly indicates exactly what the omission is:

```
T$ = STR(J)
```

In this case, there are two possibilities: either T$ is in error and the '$' is an unwanted inclusion, or the apparent numeric array STR( ) is in reality the string operator STR$( ). It is worth bearing in mind the possibility that there could be another operator missing:

```
T$ = CHR$(STR(J))
```

and the context and the variable list should give valuable pointers to the solution of the problem.

8) Rarely you may find that parameters are missing from the Subprograms (CALLs), (a 'parameter' is one of the items of information within the brackets, separated by commas) and either the lack of certain effects on screen will indicate the problem statement (for example if only one graphics character

instead of several is placed on the screen, indicating that the fourth parameter in HCHAR/VCHAR has been omitted), or an examination of the erring CALL after an error message will reveal the answer. For example:

```
CALL GCHAR(Y, X, H, Q)
```

Knowing that GCHAR requires a screen row, screen column, and 'return variable' to be specified and no more, the Q indicates one of two things. Either there has been an error in the name of the subprogram (perhaps it should have been HCHAR or VCHAR), or part of another statement elsewhere has replaced what should have been within the brackets of GCHAR. You may find another statement a few lines earlier which uses:

```
CALL HCHAR(Y, X, H, Q)
```

which, while it doesn't tell you what is missing from the GCHAR subprogram parameter list, does at least tell you that there is a very strong possibility of a typographical error.

As has been the case with most of the chapters in this book, lack of space has prevented a detailed discussion with analysis of actual programs — we just couldn't have fitted all the examples in. Further publications planned will deal with all of the subjects covered in outline in this book, and will go on to uncover further areas of interest.

Hints and Tips

This chapter is intended to give you as much help as possible in getting the most from your computer. Because of internal differences, there is a small chance that your machine may not be able to make use of all of the hints and tips given here, in which case I would be very glad to hear from you, both to make a note of the problem(s) and to help you towards a solution. Please write to me care of the publishers, enclosing a stamped, self-addressed envelope.

Let's start with a couple of very useful items.

The 99s don't have a command in TI BASIC which gives you the amount of memory 'used' by a program. (I say 'used', because a program uses not only the amount of memory occupied by its listing — and the listing is not all it appears to be — but also uses memory when you RUN it: an amount is used to hold a table of the names of the variables involved together with their contents, and SUBPROGRAMs or CALLs also require memory).

However, there is a very simple two-line program which can give a reasonable approximation.

Why would you want to find the amount of memory used? Well for one thing, if you decide to publish your latest version of *Alien Mothers-in-Law*, the editor will want to know how much memory is needed to run the program so that he can publish the details for the benefit of those wishing to translate your masterpiece into their own dialect of BASIC. The owner of an 8K machine is not going to be happy if he keys in your program (taking four hours to do so) only to find that it needs 9K in order to run successfully.

On the other hand, you might need to know just how much room for expansion the program has, especially if it is the type which generates and/or stores large quantities of information. You will curse pretty loudly if, after two hours' running, your program stops with a MEMORY FULL error.

To begin with, let's see what the two-line program mentioned earlier looks like, and how it works when there is no other program in memory.

Switch your machine on and select TI BASIC. Enter the following two lines:

```
100 A = A + 8
110 GOSUB 100
```

Now RUN the program. After a few moments, the program stops with a MEMORY FULL error. Now comes the clever bit. Type:

```
PRINT A
```

and press the ENTER key. Depending upon the variant of TI-99 that you have, a number around the 14600 mark should appear (14792 on the NTSC 99/4, 14536 on the PAL 99/4A; others not currently known). This is the number of bytes, roughly, that are available for you to play around with.

By now you may be thoroughly confused, because you may not be able to see how on earth the routine manages to do what it does. On top of that, you may wonder why the number of bytes available is not 16384, or 16K.

Things are reasonably straightforward, however. Whenever the Texas computer performs a GOSUB, it uses up about 8 bytes of memory to store details about the point from where it is to continue once it RETURNs from the subroutine. So, every time the computer encounters GOSUB 100 it uses up 8 bytes. It is possible to 'nest' the GOSUBs so that one subroutine causes a GOSUB to another. In this case the computer is continually told by the subroutine to GOSUB to the same subroutine, without ever encountering a RETURN. It therefore uses up blocks of 8 bytes until there is no more memory available, whereupon it halts with the MEMORY FULL error. Each time that it encounters the beginning of the subroutine (line 100), 8 is added to a 'counter' — the variable A — so that when the program halts, that variable will hold the number which corresponds to the number of bytes of available memory which the computer was able to use.

So why does the computer only find around 14K instead of 16K of available memory, when the machine is described as 16K? Well, the computer uses some of the 16K to produce the screen display — 768 bytes, which refer to the ASCII codes of the characters on the screen. The definitions for the shapes of all the predefined characters (codes 30 to 95 on the 99/4, 30 to 127 on the

99/4A) also occupy some memory (8 bytes per character), and the remainder is taken up with storing what are known as *System Variables*, or variables which are generated and used by the computer during the execution of your program.

How can you use the routine to discover the size of your own programs? All you need to do is to add the two-line routine to your program, subject to a few restrictions, and then RUN. Place the two lines at the very beginning of your own program, and it will measure the initial use of memory. Place it somewhere in the middle, and it will measure the amount in use by that point in the program.

One restriction is that in the first line of the two-line routine you should use a variable which is *not* used by your own program.

To see exactly how you should go about using this tool, here is an example routine:

```
100  CALL CLEAR
110  PRINT "EXAMPLE PROGRAM"
120  END
```

Add the routine as:

```
1  A = A + 8
2  GOSUB 1
```

and RUN. When the program halts (note that it never gets any further than line 2) type in the following, and then press ENTER:

```
PRINT A, 14792 − A
```

where the number (here 14792 as an example) is the one you got from running the routine with no other program in memory.

The values which are printed out are the number of bytes remaining, and the number of bytes used. Note that you should never edit any lines before printing out those values, because this will reset all numeric variables to zero, and therefore prevent you from proceeding further.

If you have the disk system switched in (i.e., so that the computer 'knows' that it is there) you will find that the numbers you obtain are smaller, because the presence of the system causes an amount of memory to be set aside for transferral of data between computer and disk. This amount will depend upon the number of files permitted by CALL FILES( ), so to free as much memory as possible for your program you should specify the smallest number of files necessary before loading any programs.

The second useful item is a trick which enables you to monitor the initial stages of the loading of a program from tape. Get your cassette system ready to load a short program of your own which you are sure of being able to OLD successfully. Select the 'title page' (either switch on or, if already in BASIC, use the QUIT function — shift Q or FCTN =, depending upon machine). Select TI BASIC. This is necessary to redefine the User-definable characters as 'blanks' — you'll see why later.

If you have a 99/4 (PAL or NTSC), enter this:

```
100 FOR I = 96 TO 159
```

If you have a 99/4A, enter this:

```
100 FOR I = 128 TO 159
```

Then add:

```
110 PRINT CHR$(I);
120 NEXT I
```

The semi-colon after the CHR$(I) in line 110 is important.

Now RUN the program. When it has finished, type OLD CS1 and load your program in your usual manner, but watch the screen carefully.

If everything goes smoothly, as the 'header tone' is succeeded by the 'angry bee' sound, you should see a small army of 'insects' begin marching across your screen at the bottom, in short bursts. It will last for only a couple of lines (less if you have a 4A) but it is a visible sign that the program is being loaded successfully. The chances are that if the program initially loads without difficulty, the rest of the program will also load successfully. With some programs, getting the volume just right can be difficult, and with this trick if the volume is way off the mark, the insects will not march; however, you do have some time to rewind the tape to the point just before the angry bee starts buzzing, enabling you to alter the volume, and press PLAY to try loading again. You can do this several times until either you take a little too long and the computer says 'NO DATA FOUND', which tends to suggest that the volume was far too low, or 'ERROR DETECTED IN DATA' which can mean either that the volume was far too high (causing distortion of the signal), or that it was fractionally below optimum (when slight changes in sound level could mean that data were lost during transfer). In either event, the resulting error messages and options cause your insect army to be shunted off the top of the screen, so ideally you really need to start from scratch (i.e., the title page), having used 'E' to exit from the OLDing instructions.

How does the three-line program help? It makes use of the fact that the area of memory which stores the definitions for the User-definable (NOT Redefinable) characters is also the area of memory into which your program is placed when OLDed from tape. What the short routine does is to place the User-definable characters on screen so that when the incoming program is transferred into that area of memory, you can 'see' it arriving via the 'window' on that area afforded by the characters. Incidentally, when you define one of the User-definable characters, your BASIC program is not interfered with. The computer simply moves everything 8 bytes further along inside to make room for the definition. You may notice that redefining characters with CALL CHAR( ) is slightly faster when using the predefined characters; this is because memory has already been set aside for their definitions, and so the computer doesn't need to shunt data around in order to make space for them.

The reason why you need to go back to the title page to wipe the User-definable characters clean instead of simply typing NEW is that on most computers the use of NEW does not in fact remove the BASIC program from memory. Generally a computer will use some *system variables* (see the beginning of the chapter) which will record where in memory the BASIC program listing begins and ends. NEW simply resets those variables so that the computer 'thinks' that there is no program resident. No one knows for certain with the 99s, because Texas Instruments play their cards pretty close to their corporate chest, but the fact that, after NEW, the undefined User-definable characters maintain the shapes produced by the incoming program seems to suggest that the 99s are no different from most of the others in this respect.

While on the subject of OLDing programs, it is worth noting that certain keys are 'active' — that is, if you press them the computer will respond — during the OLDing and SAVEing instructions. If you have a program in memory, for example, and a program on tape which you suspect may be identical, you can invoke the CHECK facility by moving the tape to the start of the header tone, and typing SAVE CS1, as if you were going to make a copy of the program. Instead, however, once the REWIND instructions appear, you can press C to check, even though the computer doesn't tell you that you can, and proceed as if you had already SAVEd the program. By playing the program on tape to the computer, you will cause it to be compared with the one currently resident in memory. The two must be identical (which may involve far more than simply having identical listings on screen, but which is beyond the scope of this book to discuss) or you will receive the ERROR DETECTED IN DATA message, whereupon you can exit with E. The program in memory will not be affected. Almost all the options are active during SAVEing and OLDing, so that you can exit from a command immediately by typing E without having to wait until the computer tells you that you can. Note that early NTSC 99/4s can receive instructions concerning the OLDing of programs from CS2, even though the cassette cable may not support the necessary hardware connections.

Still on the subject of SAVEing programs, it can be helpful to you, more so if you are using C60 tapes or something similar, if you preface each SAVEd program with a short spoken résumé of what the program's title is, the version, etc. It will save you time when you are looking for a specific program later.

Many other BASIC dialects (and indeed TI's own Extended BASIC) permit the use of the Boolean Operators AND, OR, XOR, and NOT. The subject of Boolean Algebra is covered broadly in the **Jargon** chapter, and specific details on implementing the TI BASIC equivalents of the operators are given in the **Translation** chapter.

There are a few bugs which are known to exist in TI BASIC. Some which were evident on the NTSC and PAL 99/4 have been removed, only to be replaced by others on the 99/4A. Two which could cause problems for the 99/4 occur in the use of POS( ) and in the use of *relational expressions* with the ASCII codes of certain characters.

The POS( ) bug is fairly straightforward. If you instructed the computer to search a string for the occurrence of another string, beginning at the first character, you had no problems unless the string to be searched was greater than 127 characters in length AND the string to be looked for occurred for the first time in a position which was greater than 127th. The value returned by

POS( ) would be zero (i.e., string not found) unless you altered the instruction so that the computer began the search from the 128th character of the major string. This has been resolved on the 99/4A, but 99/4 owners ought to be aware that a 99/4A program might fall foul of it if run on a 99/4.

The second bug occurred when comparing the ASCII codes of two characters. If one of these codes was less than 127, and the other greater, then the evaluation of:

```
CHR$(code less than 127) < CHR$(code greater than 127)
```

would give a FALSE result (i.e., that the lower-coded character was not 'less than' the higher-coded one). This is not likely to make many programs collapse in ruins, but it exists and you should be aware of it.

On the 99/4A the bugged command is CALL KEY( ) when using key units 1 and 2. When pressing X or M, which return values of zero while being scanned with CALL KEY(1, K, S) or (2, K, S) respectively, a small problem arises in that although if X (or M) is pressed and the contents of variable K is printed it will produce a zero on screen, that value is not really zero, and it fails the test for being zero:

```
CALL KEY(1, K, S)
IF K = 0 THEN . . .
```

The only way really to decide what value you have there is to execute this:

```
PRINT EXP(LOG(K))
```

whereupon if X (or M) is pressed the magic number 400 appears. Something is interfering with the correct function of CALL KEY( ); I understand that the use of key unit 3 as a 'dummy' (i.e., not used for any purpose) before using CALL KEY( ) with key units 1 or 2 will avert disaster, but I have been unable to confirm this. The known solution is to test for (K + 1) = 1, rather than K = 0, which looks daft but it works.

```
CALL KEY(1, K, S)
IF (K + 1) = 1 THEN . . .
```

Moving on to memory-saving techniques, and general hints, there are a number of simple points which can be easily overlooked:

1) If DIMensioning arrays, do them all in the same program statement if at all possible.
2) Use PRINT :::::::::: rather than simply lines of PRINT, or loops of PRINTs.
3) If you want to redefine a character as a 'blank', then use a null string with CALL CHAR( ); i.e., CALL CHAR(code, ""), rather than "0" or "00".
4) If an IF . . . THEN . . . is testing for '<> 0', then save space by replacing:

   IF variable <> 0 THEN . . .

   with:

   IF variable THEN . . .

   for the '<> 0' is 'implied'.
5) If the same number or string is going to be used in several places throughout the program, assign it to a variable, which will usually save time and space.
6) Use string variables and string arrays instead of their numeric counterparts wherever possible in a program; they may slow things down, but they can reduce the memory requirements markedly.
7) Reduce unnecessary calculation whenever possible, and reduce the numbers of brackets involved as well — these can slow a program down considerably.
8) Use DIM on any arrays whose maximum subscript is going to be less than 10, rather than waste valuable space.
9) If there are sections of the program which are repeated in several places, turn each section into a subroutine, and place it as close to the beginning of the listing as is possible. All GOSUBs appear to cause a search for the line number to begin at the first line in the listing, so it makes sense to place subroutines as close to the first line as is possible.
10) Don't use REMs unless you really can't avoid it. There are really only two practical uses of REM: to 'inactivate' a program line so as to provide optional additional lines in a program, or to give copyright and other information. The only other use for REMs is in your early versions of a program — the ones which are not for public consumption. Too often memory is wasted with pointless REMs, and far too often they are made the destination of a jump (i.e.,

GOTO/GOSUB/ON GOTO/ON GOSUB/IF . . . THEN . . ., where the line number, or one of the line numbers, which is given as the destination, contains a REM statement), which means that if you decide to make room by deleting the REMs, you'll also end up with a crop of BAD LINE NUMBER error messages.

11) Always *resequence* your finished program, especially if it is for publication, and any explanatory REMs should ideally be added afterwards on line numbers ending in the digits 1 to 9. Then LIST the program, and any destination line numbers which are 32767 will stick out like sore thumbs — which means that somewhere you have made a bit of a boo-boo.
12) Use CALL KEY( ) when only a single key selection is to be made, rather than using INPUT, which requires the user to press ENTER in order to get the program to continue.
13) Don't touch the ADVANCE button on the Thermal Printer while it is listing a program; the system will 'lock', forcing you to switch the console off and then on in order to regain control — by which time you'll have lost the program you were listing.
14) Beware of typing errors in the DEF statement. These may cause problems only when the defined function is used and the error message may not give any indication that it is the DEF statement itself which is at fault.
15) If you exit a FOR . . . NEXT loop before it has 'finished' you may run into problems when the program attempts to begin a fresh loop. The message 'FOR — NEXT ERROR' will indicate the problem.
16) Make sure that if a subroutine has been placed early in the listing, and it has any references to arrays, that those arrays are DIMensioned in a statement whose line number lies before that of the subroutine.
17) You can round a number up or down to the nearest integer according to the standard convention by using INT(.5 + number), where 'number' is a numeric variable or expression.
18) You can round a number up or down to a specified number of decimal places thus:
    a) Call the number of places required: 'D'.
    b) Let D become 10 ^ D.
    c) Let the value to be rounded be: 'V'.

d) The new, rounded value for V is given by:

```
V = INT(.5 + (V * D)) / D
```

19) When typing in a program, make it a habit to SAVE a copy every twenty lines or so; if you have a power failure, or the machine 'locks' for some reason, you will lose only those lines which you have entered since the last SAVE. Don't forget to make a final copy of the finished program before RUNning in case the system crashes, or in case you lose the program for some other reason.
20) Use NUM for entering programs whenever possible; it will encourage you to keep your program listings neat and tidy, a condition which is helpful when searching for bugs. It can also be useful if entering a program from a published listing, especially if the other author has placed REMs on line numbers ending in 1–9, as NUM will automatically skip over them.
21) If you want to know how many statements your program has, simply 'RES 1, 1' and then LIST 32767. The last program line will be listed, and its line number will be the number of lines in the listing.
22) If you want a function which isolates the fractional part of a number (i.e., the opposite of INT( )) then DEF can provide it. Simply use the line:

```
DEF F(X) = X - INT(X)
```

and then use it thus:

```
A = F(B)
```

where the fractional part of the number represented by B will be assigned to A. Incidentally, once a program stops, the functions defined by DEF remain operational in the *Immediate Mode* until editing takes place. This means that you can create special functions and test them out manually without needing to write a program to do so. You can also use the function defined above to test for an integer:

```
IF F(C) THEN . . . (routine for non-integer)
```

which translates as 'if the fractional part of the number represented by the variable C is not equal to zero then . . .'.

23) If you want to place a border around your screen display, you can do so very quickly with just two program lines. Choose the character which will form the border, for example CHR$(30) — the cursor — and use these statements:

```
CALL VCHAR(1, 32, 30, 48)
CALL HCHAR(24, 1, 30, 64)
```

(with line numbers, of course). This makes use of the 'wrap-around' effect of H- and V- CHAR. To thicken the border, simply double up:

```
CALL VCHAR(1, 31, 30, 96)
CALL HCHAR(23, 1, 30, 128)
```

You could use a redefinable character and produce all sorts of variations on the border theme.

24) The command INPUT has some useful attributes. If you don't specify an 'input prompt', the computer will print a helpful '?' for you. However, if you want to take advantage of the full 112-character input line you must trick the computer into thinking that there is an input prompt when in fact there isn't. To do this all you need is a 'null string' input prompt:

```
INPUT "" : S$
```

In addition, if you create a 'pending print' condition by following the last PRINTed item with a semi-colon or comma, a subsequent INPUT will cause the prompt/query to be printed immediately following the item.

25) The ENTER key is not the only one which can be used to put data into the computer. Both X and E keys, when used with Shift (99/4) or FCTN (99/4A), are also capable of acting like ENTER, not only when editing (which will move you up and down through the listing) but also when using INPUT. This means that INPUT can take on a degree of flexibility if CALL KEY( ) is used in the statement immediately following the INPUT. If the key used for entry is held down for a short period and CALL KEY( ) tests for the codes for the three valid keys, then a branch to alternative sections of the program can take place.

26) If you want to ask a user to select from a range of choices represented by single characters, i.e.:

PRESS:	FOR:
1	First choice
2	Second choice
3	Third choice

then you can use CALL KEY( ) and test if the code of the pressed key lies outside the permitted range. In the above example this could be programmed thus:

```
First line:    CALL KEY(0, K, S)
Second line:   IF (K < 49) + (K > 51) THEN
               First line
Third line:    ON K − 48 GOTO . . ., . . ., . . .
```

This is also applicable to ranges which use letters of the alphabet. The process is known as *input validation*, or *mug trapping*. A more sophisticated solution which permits selection from a list of keys which do not form a simple range involves the use of POS( ). If, for example, your valid keys were W, E, R, S, D, Z, X, C, 0, 1, and 2, then the following routine would be of use:

```
First line:    CALL KEY(0, K, S)
Second line:   IF S < 1 THEN First line
Third line:    ON 1 + POS("WERSDZXC012",
               CHR$(K), 1) GOTO First line,
               . . ., . . ., . . .
```

If an invalid key is pressed then POS( ) will return a value of 0. Adding 1 means that if the first line number in the list following GOTO (or GOSUB) is 'First line' then a branch will be made back to the 'First line', and only the valid keys will cause the relevant section of program to be branched to. By making use of careful arrangement of the valid characters in the string in POS( ) and the 'start position' facility for a search for CHR$(K), you could further impose restrictions — for example, if using the 'arrow' keys in an Alien Invader-type game, a hit on your defending ship could disable the 'up' keys simply by making the 'start position' 4 instead of 1 in our example above. In that case the search for CHR$(K) will begin with the 4th character in the string, ignoring "WER".

27) When the CALL KEY( ) command is used, the usual test for keys being pressed is to examine the status variable, S, to see if it is −1, 0, or 1. However, the key-return variable, K, can also hold some useful information. Besides containing the code of any key being pressed, when no key is pressed K contains −1, so it too performs a kind of status function. And instead of:

First line:	CALL KEY(0, K, S)
Second line:	IF S < 1 THEN First line
Third line:	IF K <> valid code THEN First line

you could use:

First line:	CALL KEY(0, K, S)
Second line:	IF K <> valid code THEN First line

28) The use of CALL KEY( ) with key unit 0 on a 99/4A has a slightly different effect from that on a 99/4: on a 99/4A, key unit 0 will scan the keyboard according to the key unit specified in the last scan. Thus if key unit 1 is followed by key unit 0, on a 99/4A this will function like key unit 1. The key unit on a 99/4A which corresponds to unit 0 on a 99/4 is 3: in all the examples here it has been assumed that CALL KEY(0, –, –) will be replaced by CALL KEY(3, –, –) on a 99/4A, but not all of the software on the market may take this into account, so there may be problems when using 99/4A software on a 99/4.

Further Useful Routines

To scan the keyboard for the 'arrow' keys — W, E, R, S, D, Z, X, and C, the following routine will automatically alter row (R) and column (C) variables, increasing/decreasing by 1 each time the relevant valid keys are pressed:

```
100  R = 12
110  C = 16
120  CALL KEY(1, K, S)
130  IF S < 1 THEN 120
```

```
140  R = R + (K = 4) + (K = 5) + (K = 6) − ((K + 1) = 1) − (K =
     14) − (K = 15)
150  C = C + (K = 2) + (K = 4) + (K = 15) − (K = 3) − (K = 6) −
     (K = 14)
160  continuation of program
```

Note that the CALL KEY( ) bug mentioned earlier is catered for in line 140; this will not affect its correct use on an unbugged machine. The scan is for key unit 1, which is the left side of the keyboard. Change the key unit to 2, and the right side of the keyboard will be scanned.

To scan the joysticks, you could use this simple routine:

```
100  R = 12
110  C = 16
120  CALL JOYST(1, X, Y)
130  IF (X =  0) * (Y = 0) THEN 120
140  R = R − Y / 4
150  C = C + X / 4
160  continuation of program
```

Line 120 scans joystick #1, line 130 doesn't continue the program if the joystick is not being used, and lines 140 and 150 automatically increment/decrement the row and column variables R and C, depending upon the direction in which the joystick is being pushed.

In both instances, there is no check for the values of R and C being off-screen; you can prevent any crashes due to attempts perhaps to use HCHAR with invalid row or column co-ordinates, simply by checking the row and column values thus:

```
nnn  IF (R > 0) * (R < 25) * (C > 0) * (C < 33) THEN . . . both
     co-ordinates are on-screen, so skip over the section
     which copes with them being off-screen
```

The section of program which copes with the co-ordinates being off-screen is simply the reverse of lines 140 and 150 in both of the above examples:

```
nnn + 1  R = R − (K = 4) − (K = 5) − (K = 6) +
         ((K + 1) = 1) + (K = 14) + (K = 15)
nnn + 2  C = C − (K = 2) − (K = 4) − (K = 15) +
         (K = 3) + (K = 6) + (K = 14)
```

or for joysticks:

```
nnn + 1  R = R + Y / 4
nnn + 2  C = C - X / 4
```

PRINT AT

Mimicking PRINT AT in TI BASIC is reasonably straightforward, and requires a simple subroutine. All you need to do is to assign your item to be printed to a string variable and specify a row and column for printout, and then branch to this subroutine:

```
First line:     FOR L = 1 TO LEN(M$)
Second line:    CALL HCHAR(R, C + L - 1, ASC(SEG$(M$,
                L, 1)))
Third line:     NEXT L
Fourth line:    RETURN
```

In the above example subroutine, the text to be printed is in M$, the screen row is given by R, and the screen column by C. To use such a subroutine, simply follow this format:

```
500  R = 4
510  C = 10
520  M$ = "HELLO FOLKS"
530  GOSUB First line
```

You can put numbers on screen using such a subroutine, by turning them into strings using STR$( ):

```
520  M$ = STR$(V)
```

It is advisable to clear the line on which printing is to occur, as a long item followed by a short item on the same line will produce garbage. You can get round the problem by inserting a line before 'First line' in the example above:

```
Zero line:   CALL HCHAR(R, 1, 32, 32)
```

and causing the routine to GOSUB First line if no clearing is required, and GOSUB Zero line if it is. Partial clearing can be managed by using the length of the text item to calculate the exact part of the line to be cleared:

```
Zero line:   CALL HCHAR(R, C, 32, LEN(M$))
```

This technique can be used to leave untouched sections of text or data already on the screen. One final warning: if the text to be printed is longer than the amount of space left on the row between the end of the screen and the start of the text, then an error will occur, so beware of making that mistake. If you want to be really sophisticated, you could amend the subroutine so that it copes with several lines of text at one go, perhaps printing in blocks of specified dimensions to produce a screen display like this:

```
                    DINSDALE'S DOINGS

        An Adventure Game For Up To Four Players,
                                   Requiring Skills
                                     Of Both Mind
                                        And Body.
```

ACCEPT AT

Implementing ACCEPT AT is far less easy. To begin with, there are not only the valid key ranges to be considered, but also the provision (or not, as the case may be) of cursor control to edit the line before inputting it, and the concomitant changes to the contents of the variable being used to hold the input.

The aspect of *input validation* has already been discussed earlier, and the finer details need not concern us here. Let us examine a very simple routine which will accept any keys pressed, compiling them into a string variable — A$ — until ENTER is pressed. First, we need a keyboard scan, effected through CALL KEY( ).

```
1000  CALL KEY(0, K, S)
1010  IF S < 1 THEN 1000
```

This will only continue processing if a key is pressed, and because 'S < 1' is used, will 'debounce' the keyboard — i.e., you will have to release a key and press it again before the system will recognise that you are pressing keys. Using 'S = 0', if you kept your finger on a key for a fraction of a second too long, more than one character would be accepted.

Next, we need to test for the ENTER key, in which case the entry is complete:

```
1020  IF K = 13 THEN 1050
```

Then we simply add to A$ the CHR$( ) of the key pressed thus:

```
1030  A$ = A$ & CHR$(K)
1040  GOTO 1000
1050  RETURN
```

Apart from the obvious things like the lack of control over the input — you can't retrace an entry and re-enter a corrected version — and the fact that A$ is going to fill up pretty quickly with successive jumps to the subroutine because it is not reset to a null string ("") before being used, there is one glaring omission:

How can you tell what you are typing?

What is needed is an 'echo' to the screen; that is to say, every key that you press should result in the character represented by that key appearing on the screen. There are several routes to this, but one of the simplest, if your subroutine is to be a true mimic of ACCEPT AT, is to use CALL HCHAR( ), specifying a screen row and column location at which the input is to occur:

```
1000  CALL KEY(0, K, S)
1010  IF S < 1 THEN 1000
1020  IF K = 13 THEN 1070
1030  A$ = A$ & CHR$(K)
1040  CALL HCHAR(R, C, K)
1050  C = C + 1
1060  GOTO 1000
1070  RETURN
```

This subroutine makes no checks to see if the 'echo' is about to go off-screen, although these could be added with a little thought. To call it, the main routine might look like this:

```
3000  R = 14
3010  C = 4
3020  CALL HCHAR(R, 1, 32, 32)
3030  A$ = " "
3040  GOSUB 1000
3050  IF A$ = " " THEN 3040
3060  IF A$ = "FIRE" THEN 6000
        .
```

and so on. Please note that this is by way of example only, and is not intended as a piece of model programming.

There are still a couple of small points which may not be immediately obvious: there is no cursor, so you won't know where you are on the screen until you begin typing something. This can be avoided by adding a line to place the cursor (character 30) on the screen in such a way that it doesn't interfere with what you type, yet shows you exactly where you are. In addition, the data which are echoed to the screen are not removed by the example routine, so you will need to add such a line, between lines 1060 and 1070, modifying line 1020 accordingly so that the jump is made to that 'clearing' line. The line providing the cursor might best be placed between lines 1020 and 1030, thus:

```
1025  CALL HCHAR(R, C, 30)
```

You may have your own ideas or preferences.

Quirks

There are a number of quirks on different models which may be specific to a particular machine. For example, when using the Thermal printer with an NTSC 99/4 you may refer to the printer with only "." instead of the specified "TP". Thus "TP.U.S.E" becomes ".U.S.E".

Again, a specific quirk which can be examined on those models which have the *Equation Calculator* involves deliberately inducing an error condition. This particular condition shows the sprites in action in TI BASIC, although alas no useful functions can be performed with them (unless someone finds otherwise). They are invoked with this line:

```
CALL KEY(0, K, "@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
```

You need to fill the four screen lines with characters (here represented with '@') and then press ENTER. Some quite extraordinary things can happen if you place the flawed CALL KEY( ) statement in a short program and then RUN it. Recent experiments have produced some interesting effects, but they will be of little interest to 99/4A owners.

Another quirk which seems to work on all but the most recent machines involves the addition of comments to certain types of program line. There are several different ways of adding these, some of them more useful than others, but the pick of the bunch involves taking advantage of a weakness in the computer's error-checking of program statements. I have called it a *quoted comment* because it uses the quotation mark ("). To check whether your machine can make use of this facility (although I would not recommend its use for anything other than development of a program — any routine using the quoted comment could cause a crash on a machine which cannot make use of the effect), type this into your machine and RUN it:

```
100  GOTO 110"JUMP TO THE NEXT LINE"
110  PRINT "IT WORKED"
```

If the computer crashes with an error message which refers to line 100, then your machine cannot support quoted comments. If the text in line 110 is printed, then your machine can support them. They can be used, as shown in line 100, with GOTO, GOSUB, ON . . . GOTO, ON . . . GOSUB, and IF . . . THEN . . . ELSE. With the IF . . . THEN . . . ELSE type, the quote can appear only after the ELSE and line number, and with the ON . . . type the quote must appear after the last line number in the list. When you use them, you don't need to begin the routine(s) to which they refer with a REM, thus avoiding the pitfall of accidentally using the REM line as the destination for the jump. In addition, if you are writing a program and there are forward jumps involved (where you may not always have a clear picture of what the exact line number will be) you can substitute 'labels':

```
1000  GOTO START
```

or:

```
1010  GOSUB PRINTOUT
1020  GOSUB KEYSCAN
1030  IF R = 20 THEN ENDGAME
```

and so on. Note that these do not need quotes, and, more importantly, will not permit you to run the program — they are an aid to entry only, not to actual execution. Once you have the program written you can replace PRINTOUT, KEYSCAN, and ENDGAME with their relevant line numbers.

There is another quirk which is specific to the 99/4A and involves the CTRL (control) and FCTN (function) keys. A full discussion is beyond the scope of this book, but if you type a line number followed by REM and then hold the CTRL key down and press a few others, followed by ENTER, and then attempt to list the line, you will find a whole string of characters and TI BASIC words in the REM line. In TI BASIC this has very little practical use as yet, but it is worth experimenting with if you like an intellectual challenge.

Finally, an oddity which appears on some, if not all, 99/4s, concerning the Equation Calculator. If you enter:

```
B = 10
```

then the variable B appears in the box with its associated value. However, if you enter:

```
LET C = 100
```

nothing appears in the box, although the computer has actually executed the instruction.

Index

MASTERING THE TI-99

Peter Brooks covers the essential aspects needed to master the '99's (PAL 4a, PAL 4 and NTSC). Building from the User's Manual, the author explains the mystifying jargon which frustrates so many '99' owners. He fills the gap left by the popular magazines by showing you how to translate programs written on other machines for your '99'. High resolution plotting and the graphics capability are fully explained. File handling for all uses is made simple to operate, and the author's Hints and Tips are the best around: finding the size of a program simply and quickly, coping with printing errors in published material, and explaining equivalents to the Boolean operators AND, XOR and OR.

Become a Master of your '99' and use your machine to the full.

The Author
Peter Brooks has impeccable credentials for introducing '99' owners to the intricacies of their machines. As area contact for TI HOME members—the TI User's Group—he has built up a picture of recurrent difficulties and presents his solutions in this book. He contributes regularly to journals such as Computer Talk, as well as reviewing software for Home Computing Weekly.

GB £ NET +005·95

ISBN 0-7447-0008-6

00595

9 780744 700084