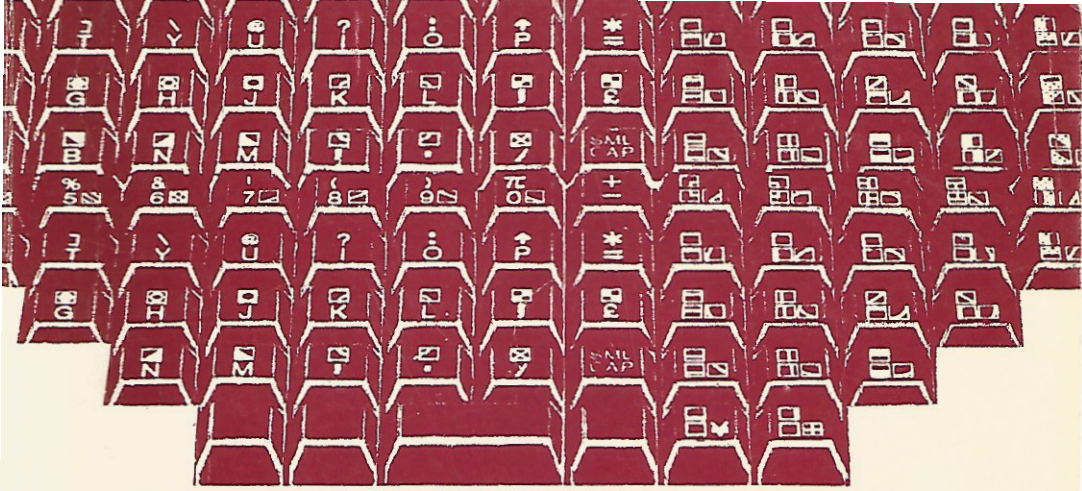


SOFTWARE SECRETS

Input, Output and Data Storage Techniques

GRAHAM BEECH

Approved by SHARP Electronics (UK)

Sigma Technical Press

5.95

SOFTWARE SECRETS:

Input, Output and Data Storage Techniques

GRAHAM BEECH

Sigma Technical Press

ISBN: 0 905104 15 3

Published by:

Sigma Technical Press
5 Alton Road, Wilmslow, Cheshire, UK.

Distributors:

Europe, Africa:
John Wiley & Sons Ltd., Baffins Lane, Chichester, Sussex, England.
Australia, New Zealand, South East Asia:
Jacaranda-Wiley Ltd., Jacaranda Press, John Wiley & Sons Inc., GPO Box 859, Brisbane, Queensland 40001 Australia.

Typeset by:

Bizmark Business Services
365 London Road, Hazel Grove, Stockport, Cheshire, UK.

Acknowledgment

The Display Code and ASCII tables in section 4.2 are reproduced by kind permission of Sharp Corporation (UK) Ltd.

INTRODUCTION

This book is a collection of ideas, techniques and about 30 programs and sub-routines for a personal computer. The actual computer used was a Sharp MZ-80K on which each of the programs have been tested.

The computer was equipped with dual disk drives and 48K of random access memory. But, most of the programs will run with smaller memories on machines without disk drives. Also, users of other personal computers will find that the techniques are easily applied to their machines.

The MZ-80K is not as well known as some of its rivals, but it does compete very favourably. It is extremely rugged and is particularly well suited to domestic and educational applications. But I felt that the factors holding back this computer were the standard of the manuals and the lack of available software. Hopefully, this book will clarify the contents of the manuals, particularly in regard to:

screen output
keyboard input
file handling

These operations are of crucial importance to getting data into and out of the MZ-80K or any other computer. If you can program your way around these areas, the rest of a program is easy!

I am most grateful to the Sharp Corporation for the extended loan of an MZ-80K, and for their help and advice during this project.

December 1981

CONTENTS

1. GETTING STARTED

This is not a book about BASIC programming - it is concerned with techniques that will be useful to programmers or teachers. But, it is necessary to have a quick review of the MZ-80K implementation of BASIC. This should be equally useful to users of this and other machines. I shall assume that you already have some knowledge of BASIC.

1.1 BASIC Commands and Statements

On the MZ-80K and most other personal computers, there are two ways of using BASIC: in direct mode or in statements.

1.1.1 Statements

As you know, BASIC statements are numbered when used in a program. The MZ-80K supports most of the familiar statements (such as LET, IF.....THEN, FOR.....NEXT) although some less familiar ones need a little explanation:

Format	Example	Explanation
ON x GOTO (list)	100 ON B GOTO 200, 300, 400	Control is passed to line 200 if B is 1, to 300 if B is 2 and to 400 if B is 3
ON x GOSUB (list)	100 ON GOSUB 200, 300, 400	Similar to ON..GOTO, but control passes to a subroutine
MUSIC string	10 MUSIC "CDEFG"	Sound is output in the sequence defined by the characters in the string
TEMPO value	20 TEMPO 5	This determines how quickly MUSIC is played
DEF FNx(y)	30 DEF FNP(Y)=Y↑2+1 40 B=FNP(10)	The value of 10^2+1 is assigned to B using the function definition of line 30
GET value	50 GET X	The keyboard is scanned for a single key-press. Line 50 looks for a numeric key

GET string	60 GET A$	Line 60 accepts any key
PRINT	70 PRINT "ABCD" 20 PRINT"■■▼■AB"	Both text and/or special graphics characters can occur in PRINT statements. ■ clears the screen and ▼ moves the cursor down
SIZE	10 PRINT SIZE	Displays the amount of unused memory in bytes
TI$	10 TI$="000000" 20 PRINT TI$	Set the internal clock (line 10) and display the time (line 20) in hours, minutes and seconds

There are, also, the expected functions and string processing statements, as well as PEEK, POKE and other statements which access the RAM. These will be introduced later.

If, after trying a BASIC program, you need to make changes to it, the MZ-80K has a versatile screen editor. This enables you to list part of the program and to modify a line by using:

1) The insert key, to include extra characters.

2) The delete key, to remove extraneous characters.

3) The cursor keys to select where (1) and (2) will be performed. The cursor keys can also be enclosed within PRINT strings.

There are some very minor differences between the BASIC statements on the cassette and disk versions of the MZ-80K. Unlike some machines, the MZ-80K loads the BASIC interpreter into RAM from disk or tape. This enables new dialects to be developed, and we will probably see further enhancements in the future.

1.1.2 Direct Commands

Certain commands are only executed in "direct mode" - this simply means that no line number is used. Examples are:

LOAD "program name"	(load a program from disk or tape)
SAVE "program name"	(copy it to disk or tape)
LIST 100-500	List the program to the

LIST/P	screen or printer
CONT	Continue execution after pressing BREAK.

Some other commands can appear in direct form or as statements:

NEW	(erase the previous program)
or 100 NEW	
CLR	(set all variables to zero,
or 20 CLR	and all strings to null)

You can also execute any BASIC statements in the direct mode:

```
A = 5+3: PRINT A
```

This gives you an immediate calculator mode, even in the middle of a program. Notice that multiple statements are separated by a colon, up to a maximum length of 80 characters. I often use this mode to check the current values of variables in a program that I am developing.

1.2 Arithmetic on the MZ-80K

Variable names are one letter or one letter followed by a letter or a digit. Variable names longer than this are only significant in the first two characters, so "ABC" is equivalent to "AB" as a variable.

Simple BASIC arithmetic expressions are written in normal algebraic form, as with most dialects of BASIC. But, all results are rounded to 8 digits, which can cause problems where precision is of importance - for example in stock control or accounting programs.

As a first example of the pitfalls of trusting too much in a computer, consider the simple program:

```
10 K=0
20 INPUT I
30 K=K+I
40 PRINT K
50 GOTO 30
```

With small values of I, everything proceeds normally. But, in time, for any value of I, K eventually exceeds the normal 8 digit representation (such as 12345678) and is expressed in exponential notation:

```
0.1567E+16
```

where "E" means "10 raised to the power". Whilst this

does permit very large or small numbers to be expressed approximately, it must be realised that the last digits are lost. For example,

```
    0.2345 E+10
```
is a representation of
```
    2345000000
```

Although this may be the true result of a calculation, it is more likely that the trailing digits have been lost. On a point of BASIC format, if you wish to INPUT numbers in E format, you must type them complete with signs, such as 0.2345E+10. Do not omit the + sign. E format numbers input with digits before the decimal point are automatically adjusted; so, an input of 24.26E-09 is stored as .2426E-07.

If rounding errors are a problem to you, there are several ways round the problem, mostly using changes of base. For example, a many-digit number can be re-expressed by grouping the digits in sets of 3. This expresses the number to base 1000. Full details are given in the MZ-80K manual.

Division can also be a problem. All results are rounded to 8 significant figures. Thus,

```
    PRINT 5/3
```
gives 1.6666667.

Both numeric and string arrays are available on the MZ-80K. In each case, the maximum length of an array is 256 elements, and the maximum width is 2 columns. So, these are valid DIM statements:

```
    10 DIM V(3,4)
    20 DIM A$(100)
    30 DIM A$(20), B$(10,12), X(150)
```
But these are incorrect:
```
    10 DIM V(3,10,2)        - too many columns
    20 DIM A$(256)          - too many elements, since
                              A$(0) is the first one.
```

Arrays can be dimensioned dynamically so that space is not wasted:

```
    40 INPUT"Array size?";A1
    50 DIM B$(A1)
```

1.3 Program Design

Most of the programs in this book are described in words. I do not use many remark (REM) statements because they are tedious to write and it is all too easy to leave incorrect REMs in programs. I also dislike flow charts,

as readers of my previous book "Successful Software for Small Computers" will know.

My preference is for a simple program description language (PDL) to describe the operation of a program. I have used PDL in this book only to describe the more complex programs, although I always used PDL to design the programs. Briefly, here is a summary of the PDL method:

1. There are 5 types of program sub-units:
   (a) the simple sequence - just a series of actions.
   (b) the alternative clause:

```
if (this is true) then do something
  else do some other thing
end if
```

   (c) the choice unit:

```
case of
  (1)  action 1
  (2)  action 2
  - - - - - -
  (n)  action n
end case
```

   This selects from several alternatives

   (d) the iteration unit:

```
while something is true
   do
      a sequence of actions
   end do
```

   (e) The repetition unit:

```
repeat
  a sequence
until a condition is true
```

   Unlike (d), this will execute at least once

   Alternatively, using a **for** loop as a special case of repetition:

```
For var:= start until end step increment
{carry out      }
{    a          }
{simple sequence}
next var
```

   Note that a BASIC FOR...NEXT loop always executes at least once.

2. Standard actions such as print or input can be used in PDL.

3. Any program, of any complexity, can be expressed as an appropriate linear sequence of PDL units.

4. PDL discourages backward GOTO's, as these lead to program complexity.

5. There is a direct and simple method of translating a PDL design into a BASIC program.

6. It is virtually certain that a properly designed PDL sequence will, when coded into BASIC, work first time.

Although space does not allow a full exposition of PDL in this book, I do recommend its use. Some examples are found in the following chapters.

2. CHARACTERS and STRINGS

2.1 Characters

2.1.1 Keyboard Characters

The characters available on the MZ-80K include all of the letters of the alphabetic, digits, and graphical symbols (many of them useful, some a little unusual). You can see what they look like from the ASCII character table in the manual, though you should note that some of these are incorrectly depicted as black on white, and should be reversed.

To investigate them for yourself, you can not simply use an INPUT/PRINT pair because this will not permit you to display characters such as the one represented by the delete key. A better method is to use the single character GET statement, as in this short program:

```
100 PRINT"PRESS A KEY"
110 GOSUB 1000
120 PRINT AK$,NK
130 GOTO100
1000 GET AK$:IF AK$="" GOTO 1000
1010 NK=ASC(AK$)
1020 RETURN
```

Line 1000 uses the GET statement, and waits until it notices something other than a null character from the keyboard. It then converts AK$ into its numeric ASCII code and prints this together with the character itself. You can discover some quite interesting characters - carriage return "looks like" ↵ and the delete key resembles a small car or flying saucer! Equally interesting, you can discover that the cursor keys do not have a printable symbol, but they do have ASCII codes:

Cursor key	ASCII Code number
Right	19
Left	20
Up	18
Down	17

ASCII codes below 32 are all reserved for special functions (such as the cursor) and for printer control.

As you will see throughout this book, the GET statement is used extensively. There is a similar statement for obtaining numerical values:

```
100 GET X : IF X=0 THEN 100
110 PRINT X
120 GO TO 100
```

This ignores any keys other than 1 to 9.

You may also find GET to be a useful way of "freezing" a program temporarily:

```
100 REM - start of program
-----     ---------
-----     ---------
400 GET A$ : IF A$="" THEN 100
410 GET A$ : IF A$="" THEN 410
420 GOTO 100
```

This will cause the program to pause at line 410 whenever a key is pressed. Press it again, and the program continues!

2.1.2 Using ASC (string) and CHR$ (expression)

From the previous section, we have found that ASC(AK$) returned the numeric ASCII code of whatever AK$ happened to be.

Similarly, ASC("F") returns the ASCII code 70. If the brackets enclose a quoted **string** of characters such as

```
ASC("FRED")
```

only the first character is examined, so the result is still 70. To generalize, ASC can examine anything other than a null string "". The string can also be specified in terms of the string operators such as MID$, introduced later.

CHR$ is the reverse of ASC. It converts a valid ASCII code into its ASCII character. Therefore

```
PRINT ASC(70)
```

will print F. This enables you to print unusual characters (such as flying saucers) and such unprintable items as quotation marks. The " has a code of 34, so this will work:

```
10 PRINT "THIS IS THE";CHRS(34);MZ-80K;CHR$(34)
```

CHR$ ( ) can be used to display the entire range of allowed symbols with this simple program:

```
10 FOR I=1 TO 255
20 PRINT CHR$(I)
30 NEXT I
```

Note that, although line 20 prints all of the ASCII characters, the special function keys are **not** simulated through CHR$. Thus "clear screen" has ASCII code 22, but PRINT CHR$(22) has no effect.

2.2 Character Strings

As you saw in the ASC(string) examples, a string is a sequence of ASCII characters. By analogy with numeric variables, string variables are used to store internal representations of strings. Any numeric variable followed by $ is a valid string variable; for example:

```
A$
B$
Q1$
```

String arrays are defined similarly. They must be declared in a DIM statement, for example

```
      DIM A$(20)
or    DIM Q2$(20,30)
or    DIM Q2$(20,30),A$(20)
or    DIM A$(X,2)
```

The last example illustrates dynamic dimensioning.

Storage is allocated initially on the basis of 1 byte for each string variable or element of a string array. Thus DIM A$(200,2) has an initial overhead of approximately 400 bytes. As characters are placed into strings, their memory requirement increases at a rate of 1 byte per character, so the string "12345" consumes just 2 more bytes than "123". But, space is quickly consumed by string arrays, which are subject to a maximum of 256 elements and to two columns. Thus, DIM A$(10,4) is acceptable, but DIM A$(200,200) will cause an immediate "out of memory" error.

Large string arrays also slow programs down. For example, the simple program, below, uses a two-dimensional string to "white-out" the screen:

```
1 PRINT"■"
10 DIM A$(25,40): REM...25 ROWS, 40 COLUMNS
20 FOR J=1 TO 25
30 FOR K=1 TO 40: A$(J,K)="▒": NEXT K
40 NEXT J
50 FOR J=1 TO 25
60 FOR K=1 TO 40: PRINT A$(J,K); : NEXT K
80 NEXT J
90 GOTO 90
```

This is not intended as a good "whiteing" method but, interestingly, if you deliberately increase the DIM to

```
10 DIM A$(100,100)
```

the program slows down appreciably. So, over-dimensioned

strings reduce both the speed of your program and waste space.

2.2.1 STR$ (expression) and VAL (string)

In some applications, a numeric variable must be stored as a character string. For example, you may wish to find the position of the decimal point in 45.893; this is easier if the equivalent character string is examined. This is where STR$( ) comes in. Any numeric constant or expression can be converted into its string equivalent:

```
Q$= STR$(48.2) fills Q$ with "48.2"
```

and

```
A1=48.2 : Q$=STR$(A1) has the same effect.
```

The VAL function has the reverse effect. It converts a string into a number:

```
A$="48.2" : Q=VAL(A$) assigns 48.2 to Q
```

But, if the string contains any non-numeric characters, a zero is returned, as in:

```
PRINT VAL("5 POUNDS")
```

2.2.2 Musical Strings

For those with a musical ear, the MZ-80K has a built-in sound output module. Sounds are output by a statement such as

```
100 MUSIC M$
```

where M$ is a character string composed of any mixture of the letters C,D,E,F,G,A or B, representing the notes of a complete scale. Also, the ▁ and ▔ select the lower or upper octaves in which the string will be played. A semitone is represented by # and a rest by R. Except for their obvious features, musical strings can be processed in a program exactly as any other character string. For example, these 3 lines convert your computer into a rudimentary piano:

```
5 TEMPO 7
10 GET A$: IF A$="" THEN 10
20 MUSIC A$
30 GOTO 10
```

Since any string can be stored, you might like to devise a means of saving your best tunes!

2.3 Manipulating Strings

Apart from the CHR$, ASC, VAL and STR$ functions, there are several ways to manipulate strings:

String functions

There are three functions for the examination of parts of strings:

LEFT$(string, n) selects the first n characters. So,

```
10 A$= "SHARP MZ-80K"
20 PRINT LEFT$(A$,5)
```

prints the word SHARP. In an expression, LEFT$ must always appear on the right hand side, as in:

```
30 B$=LEFT$(A$,6)
```

RIGHT$(string,n) selects the last n characters. Change line 20 to

```
20 PRINT RIGHT$(A$,6)
```

and the program will print MZ-80K.

MID$(string,m,n) selects the characters in the string beginning at position m and continuing for n-1 characters. Therefore,

```
20 PRINT MID$(A$,2,4)
```

will select the word HARP from our test string.

LEN(string) measures the length of a string. If you add this statement:

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

you obtain 12 (not 11, since a blank space is a valid character).

Strings are concatenated (joined together) with the + operator, this being the inverse of the LEFT$, RIGHT$, or MID$ functions. For example,

```
10 A$="S"
20 B$="HARP"
30 C$"MZ-80K"
40 D$=A$+B$+C$
50 PRINT D$
```

Concatenation can include string variables, string constants, string expressions and string functions. Here are some examples:

```
10 A$="G"
20 B$=A$+"BEECH"
```

which produces the string "G BEECH"

This one uses the CHR$ function:

```
10 B$="THIS IS A QUOTATION MARK" + CHR$(34)
```

and

```
10 C$="MERRY CHRISTMAS"
20 D$="HAPPY + RIGHT$(C$,9)
```

would give a seasonal greeting, HAPPY CHRISTMAS

A particular use of concatenation is in conjunction with the GET statement. This short program uses the little subroutine from the beginning of this chapter to construct a character string of a fixed length:

```
10 U=10
20 C=10
30 B$=""
40 GOSUB 1000
50 PRINT AK$
60 B$=B$+AK$: C=C+1
70 IF C = U THEN 90
80 GO TO 40
90 STOP
```

This forces the user to type a 10 character string, which is stored in B$. This technique is useful in applications requiring specific user inputs. You can use it with the cursor control to get the user to fill in a reply to prompts such as:

```
ENTER day, month, year: --/--/--
```

The completed string can then be checked for validity. This is a necessary alternative to a simple INPUT statement in many commercial programs.

2.3.1 User-Defined Functions

Quite a few more functions are needed in most string processing applications, but these have to be constructed by the programmer (i.e. you!) as needed. Here are some examples:

User Defined Keys

Using the same 3-line subroutine as before, both the character and ASCII code of any key can be obtained, without displaying the character.This enables you to re-define any key. For example, you may wish to use the left-arrow (not the left-cursor) to delete a whole line. The subroutine in this program will do that:

```
1 PRINT"E"
5 CURSOR 0,5
6 B$="TYPE A LINE:     "
10 PRINT B$; : INPUT A$
30 CURSOR 15,20
40 PRINT"PRESS ← TO DELETE LINE"
50 GOSUB 1000
60 IF NK=95 THEN 70
65 GOTO 6
70 FOR I=1 TO LEN(A$)
80 CURSOR LEN(B$)+I+1,5 : PRINT" ";
90 NEXT I
100 GOTO5
1000 GET AK$: IF AK$="" GOTO 1000
1010 NK=ASC(AK$)
1020 RETURN
```

Note that CURSOR is available on Sharp's Disk BASIC or it can be simulated (see page 22).

Other definable keys might perform the following functions:

Move the cursor at 45°angles
Delete a word, rather than a single character
Force the user to choose or avoid certain keys

This last application is often useful. You can prevent the delete key, for example, from being used by something like:

```
10 GOSUB 1000
20 IF NK<>96 THEN 40
30 CURSOR 0,24 : PRINT "DELETE NOT ALLOWED"
40 PRINT AK$
50 GO TO 10
```

2.3.2 Additional String Functions

INSTRING

This searches a string, A$, for another string, B$, and records its starting position in A$ if it is present. Otherwise it finds a value of zero.

```
3000 REM...INSTRING
3010 A=LEN(A$): B=LEN(B$)
3020 AB=0: REM- AB IS START OF B$ IN A$
3030 FOR I=1 TO (A-B+1)
3040 IF MID$(A$,I,B)=B$ THEN 3050
3045 GOTO 3060
3050 AB=I: GOTO 3070
3060 NEXT I
3070 RETURN
```

Thus, if A$ is "QWERTY" and B$ is "ER", AB will be 3.

STRING$ (n, "character")

This produces a string of n characters such as ++++++++++, that may be used in tables or graphs:

```
3500 REM... STRING$
3505 A$="": IF N=0 THEN 3530
3510 FOR I=1 TO N
3515 A$=A$+K$
3520 NEXT I
3530 RETURN
```

The character is stored in K$.

REPLACE (string1,a,string2,b)

A string can be wholly or partly replaced by another. Replacement begins at the a'th character of string 1, using b characters from string 2.

```
4000 REM
4005 B=LEN(B$)
4007 AL=A-1:  AR=LEN(A$)-A-B+1
4008 IF AL<0 THEN AL=0: AR=LEN(A$)
4009 IF AR<0 THEN AR=0
4010 A1$=LEFT$(A$,AL)
4020 A2$=RIGHT$(A$,AR)
4030 A$=A1$+B$+A2$
4040 RETURN
```

For example, if:

```
string 1(A$)= "ABCDEF"
A=3
string 2( B$)= "RST"
B=2
```

then, string 1 becomes:

ABRSEF

This function enables you to search for particular character sequences in lines of characters, and to replace them if you wish. For example, you can use it to change the spelling of particular words.

STRING COMPARISON

Sharp BASIC only supports string equality:

```
IF A$  =B$ THEN PRINT "strings are equal"
```

Single characters can be compared by their ASCII codes but, often you need to know which string precedes another, alphabetically. One rather complex method partially achieving this result is shown in the Disk BASIC manual. A simpler method is to pad out each string with blanks and then to test the ASCII codes of each character. This ensures that a string "ABC" is greater than "AB" and vice-versa. This routine returns a value of 3 for equality, 2 if A$ precedes B$ and 1 if A$ follows B$ alphabetically:

```
4500 REM-STRING COMPARISON
4515 XX$="                                        ": REM- 40 BLANKS
4520 XX=40
4522 A1$=A$+RIGHT$(XX$,XX-LEN(A$))
4524 B1$=B$+RIGHT$(XX$,XX-LEN(B$))
4525 IF A1$=B1$ THEN SP=3: GOTO 4550
4528 FOR ZZ=1 TO XX
4530 IF ASC(MID$(A1$,ZZ,1))<ASC(MID$(B1$,ZZ,1)) THEN SP=2: GOTO 4550
4535 IF ASC(MID$(A1$,ZZ,1))>ASC(MID$(B1$,ZZ,1)) THEN SP=1: GOTO 4550
4540 NEXT ZZ
4550 RETURN
```

Although written for 40 character strings, it is easily extended to the maximum length of 256 characters. Note that only the upper case letters are in ascending ASCII code sequence, so comparisons of strings with lower case letters in them is unpredictable.

2.3.3 Building Subroutine Libraries

You may find it useful to collect a library of routines such as those above. All you need to do is is store them as a program on tape or disk and:

(1) Use high line numbers, so that your actual program precedes the library.
(2) Make a note of the variable names used in the subroutines, so that you can avoid them in your program.

3. SCREEN INPUT/OUTPUT

3.1 Extended PRINT Features

We have already seen most of the PRINT features, but some others are useful.

TAB (expression)

When used in a print statement, this moves the cursor to the value of the expression, which must lie between 0 and 255 inclusive. If the expression does not give an integer result, the cursor is moved to the next lowest integer. So

```
PRINT TAB(10);"#"       prints in column 10
PRINT TAB (0);"#"       prints in the left-most
                        column
PRINT TAB(79);"#"       prints in the right-most
                        column
PRINT TAB(150);"#"      prints in column 70
```

but both

```
PRINT TAB(-1);"#"
```

and

```
PRINT TAB(256);"#"
```

produce errors.

Note that a semicolon is essential after the TAB for its correct operation.

SPC (expression)

This is identical to TAB except that any characters between the present cursor position and the value of the expression are replaced by spaces. The TAB simply moves the cursor.

CURSOR (X,Y)

This function, available on Sharp's Disk BASIC, moves the cursor to the horizontal position given by X and the vertical position Y. X must be between 0 and 39, Y must be between 0 and 24. Although CURSOR is not available on the standard cassette BASIC, it can be simulated by a subroutine such as:

```
2500 PRINT"⊟";
2510 FOR Y1=1 to Y
2520 PRINT"⊡";
2530 NEXT Y1
2540 FOR X1=1 to X
2550 PRINT"⊡";
2560 NEXT X1
2570 RETURN
```

[H] is the screen "home" character (upper left).

With such a routine, you replace CURSOR (X,Y) with

```
X = value: Y = value : GOSUB 2500
```

Note that this is only applicable for X and Y not equal to zero, since FOR loops are executed at least once. You can overcome this with the SET instruction described later in this chapter.

3.2 Formatted Output

The standard methods of printing are

* Along the line
* Down a column

Whilst this is convenient for small amounts of data, some extra control is usually necessary since, after 25 lines, the screen scrolls upwards. This causes the top line of data to disappear from view. One way to prevent this is to count the output lines to a "clean" screen and to freeze the output when the screen is full. For example, this program prints the values of angles and their sines:

```
 20 A=0
 30 L=1 : PRINT"[CLS]"
 40 PRINT "ANGLE";A;"SINE";SIN(A*π/180)
 50 L=L+1 : A=A+5
 60 IF L=20 THEN 80
 70 GO TO 40
 80 PRINT : PRINT "PRESS ANY KEY TO CONTINUE";
 90 GET A$ : IF A$="" THE 90
100 GO TO 30
```

This displays the results fairly neatly, page by page. It is, of course, a rather specific application but the technique can easily be extended. The only snag is that when the value of "A" increases from, say, a 2 to 3 digit number, it shifts the second column over by an extra space. This can happen part way down the screen, slightly upsetting legibility.

A more flexible approach is shown in the subroutine COLUMNS, below. This prints any data as a column, or set of columns, down the screen. When the bottom of the screen is reached, it will attempt to print (if there is room) a fresh set of columns to the right of the present column. It carries on in this fashion until the screen is full, at which point the user can clear the screen for more data. This makes for the most efficient use of the screen.

The secret behind this subroutine is that the data

is generated and then stored as a string before printing. Strings are easier to control than numeric fields. The logic of the subroutine is as follows:

```
initialise x-min, y-min, x-max, y-max
if y=top then output-string:=header
         else output-string:=data-string
end if
length:=(length of output-string)
if y ≠ y-max then if (x + length)>x-max
                  then wait for user to clear screen
                  else print output-string;y:=y+1
              end if
     else y:=y-min;x:= x + length (data-string)
        print output-string;y:=y+1
end if
```

A user of this subroutine needs to put some work into arranging the data string, but it is fairly simple. In the example, there are two numbers - the angle (I) and its sine. These are stored in S1$ and S2$ in line 15 of the example. In line 20, the purpose of LEFT$(B$,4-LEN(S1$)) is to add leading blanks to force S1$ to have a total length of 4 characters. The blanks are then concatenated with S1$, 3 further blanks and then S2$. No extra blanks were used for S2$, although you may prefer them. As an exercise, use LEFT$ on the string S2$ to print the first 5 characters, and then print sets of 3 columns of data.

Demonstration Program

```
1 C=π/180
2 H$=" ANGLE     SINE"
5 PRINT"■"
10 FOR I=0 TO 360 STEP 5
15 S1$=STR$(I): S2$=STR$(SIN(I*C)): B$="     "
20 S$=LEFT$(B$,4-LEN(S1$))+S1$+"   "+S2$
25 GOSUB 4000
30 NEXT I
40 CURSOR 0,24:PRINT"PRESS 'S' TO STOP";
50 GET A$:IF A$="" THEN 50
60 IF A$="S" THEN 80
70 GOTO 50
80 END
```

The subroutine

```
4000 REM-COLUMNS SUBROUTINE. NEEDS H$,S$
4001 REM-AND INITIAL XC,YC VALUES
4005 X0=0:Y0=0:XM=39:YM=20
4020 IF (YC=Y0)+(YC=YM) THEN 4050
4030 P$=S$
4040 GOTO 4052
4050 P$=H$
4052 PL=LEN(P$)
4058 IF YC<>YM THEN 4070
4060 YC=Y0
4066 XC=XT+1
4070 IF(XC+PL)> XM THEN 4100
4080 CURSOR XC,YC: PRINT P$
4090 YC=YC+1
4095 GOTO 4220
4100 CURSOR 0,24:PRINT"PRESS 'C' TO CONTINUE";
4110 GET P1$
4120 IF P1$="C" THEN 4130
4125 GOTO 4110
4130 PRINT"■": XC=X0: YC=Y0: XT=0
4220 IF (XC+LEN(P$))>XT THEN XT=XC+LEN(P$)
4230 RETURN
```

An example display

```
 ANGLE     SINE        ANGLE     SINE
   5    0.087155742     105    0.96592583
  10    0.17364818      110    0.93969263
  15    0.25881905      115    0.90630779
  20    0.34202014      120    0.8660254
  25    0.42261826      125    0.81915204
  30    0.5             130    0.76604444
  35    0.57357644      135    0.70710678
  40    0.64278761      140    0.64278761
  45    0.70710678      145    0.57357644
  50    0.76604444      150    0.5
  55    0.81915204      155    0.42261827
  60    0.8660254       160    0.34202015
  65    0.90630779      165    0.25881905
  70    0.93969263      170    0.17364818
  75    0.96592583      175    0.087155745
  80    0.98480775      180    0
  85    0.99619469      185   -0.087155742
  90    1               190   -0.17364818
  95    0.99619469      195   -0.25881904


 PRESS 'C' TO CONTINUE
```

3.3 PRINT USING

Some computers provide a PRINT USING option to format data prior to printing. Although not designed for paged output, it does permit you to print data exactly as you want it.

In this section, we present a simplified PRINT USING subroutine which allows you to format data within a "mask" of this type:

```
##.###↑↑↑↑
```

The # specifies the position of each digit in the value to be displayed. The ↑↑↑↑ indicates that exponential format is required.

Here are some examples:

```
Datum     Mask      As printed


 48.92    ##.##      48.92
 48.926   ##.##      48.93 (rounded up)
418.93    ##.##     *418.93 (over flow)
  4.3     ##.##       4.30 (leading zeroes suppressed)
  0.00001 #.#↑↑↑↑  .1E-04
```

In the following subroutine you need to supply

(1) XV, the value to be printed
(2) MT$, the format mask (e.g. #.###)

X$ is then printed out in the precise representation of XV required.

The program makes use of most of the string handling techniques seen so far. In lines 8900-8935, the input value is examined for a decimal point or for the existence of "E" notation (line 8925). Lines 8940-9015 examine the mask, MT$, and the rest of the program matches the mask to the value of XV in string form. Lines 9115-9300 are used for rounding up a value to the required number of decimal places. If a value is already in E format, the E notation is automatically used.

Some further techniques of input/output are covered in the next two chapters.

Subroutine Name: PRINT-USING

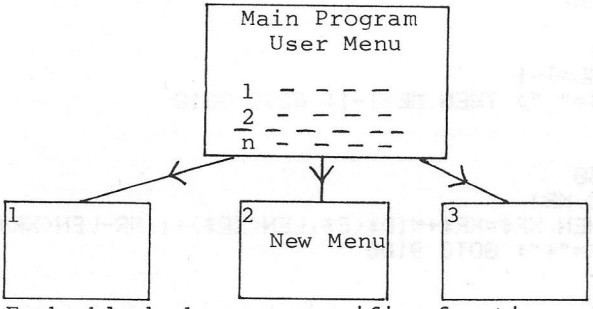
```
8000 REM
8900 XT$=STR$(XU): XN=LEN(XT$): XL$="":B$="00000000"
8901 XE$="": IF RIGHT$(MT$,1)="↑" THEN XE$="E 00"
8905 XL=LEN(XT$)
8910 FOR I=1 TO XN
8915 MD$=MID$(XT$,I,1)
8920 IF MD$="." THEN XL=I-1
8925 IF MD$="E" THEN XE$=RIGHT$(XT$,4): XT$=LEFT$(XT$,XN-4)
8930 NEXT I
8935 XR=VAL(RIGHT$(XT$,XN-XL+2))
8940 MN=LEN(MT$)
8950 ML=MN: MR=0: ME=MN
8960 FOR I=1 TO MN
8970 MD$=MID$(MT$,I,1)
8980 IF MD$="." THEN ML=I-1
8990 IF (MD$="↑")+(MD$=" ") THEN ME=I-1: GOTO 9010
9000 NEXT I
9010 MR=ME-ML-1
9015 IF ML=MN THEN 9030
9020 XR$=MID$(XT$,XL+2,MR)
9025 IF LEN(XR$)<MR THEN XR$=XR$+MID$(B$,LEN(XR$)+1,MR-LEN(XR$))
9030 IF XL>ML THEN XL$="*": GOTO 9100
9040 I=XL
9050 IF I=ML THEN 9100
9060 XL$=XL$+" "
9070 I=I+1: GOTO 9050
9100 XL$=XL$+LEFT$(XT$,XL)
9110 IF ML=MN THEN X$=XL$: GOTO 9500
9115 IF VAL(MID$(XT$,ME+1,1))<5 THEN X$=XL$+"."+XR$+XE$: GOTO 9500
9120 X1=VAL(XL$): X2=VAL(XR$)+1
9140 X$=XL$+"."+STR$(X2)+XE$
9150 GOTO 9500
9300 X2$="1."+RIGHT$(STR$(X2),2)
9310 XU=X1+VAL(X2$)
9320 GOTO 8900
9500 PRINT X$
9600 RETURN
```

4. INTERACTING with PROGRAMS

4.1 Menu Selections

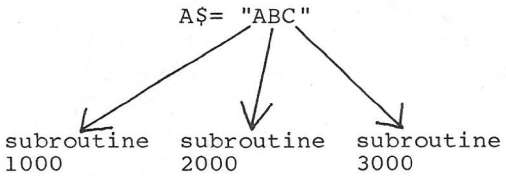
Many programmers fall into the trap of starting with a simple idea, then adding "bells and whistles" until the logic behind the program is incomprehensible. A better approach is to use "structured programming" methods to break a complex program down into simple sub-units. These sub-units are often implemented as BASIC subroutines, so it is necessary to have an efficient way of communicating with them.

The way to do it is to design the program like this:



Each block has a specific function. Some blocks may themselves contain menus to provide a finer selection of user actions. One advantage of this approach is that each block can be developed and tested separately, without interference from extraneous program lines. Data required by more than one subroutine can be defined in the main program, which also contains the menu. Remember, of course, that variable names must be carefully chosen since if, for example, the variable X is set to 9 in any subroutine it will reset X in any other subroutine.

Menus can be designed in several ways. One is the "Chinese Take Away" method where a user selects a number or letter from a list. An example is shown below, complete with a sample program to produce the menu. In this case, the possible user choices are stored in a string, A$. The user presses a key and, using the MID$ function, the string A$ is examined for his selection. If the key is not contained in A$, the menu is re-displayed, otherwise the correct subroutine is selected:



Example Menu

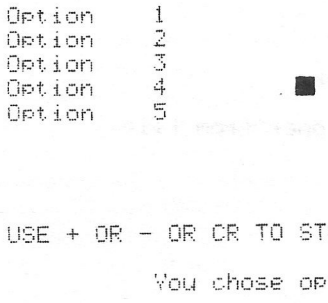
```
                ** MAIN MENU **


        (A) New Customer

        (B) Delete Customer from file

        (C) Stop




        Type A,B or C:B
```

Example Menu Module

```
10 REM- MAIN MENU
15 PRINT"■"
20 CURSOR 10,4:PRINT"** MAIN MENU **"
30 A$="ABC"
35 LA=LEN(A$)
40 CURSOR 0,8: PRINT"(A) New Customer"
50 CURSOR 0,10:PRINT"(B) Delete Customer from file"
60 CURSOR 0,12:PRINT"(C) Stop"
70 CURSOR 0,24:PRINT"Type A,B or C:";
80 GET K$: IF K$=""THEN 80
85 PRINT K$;
90 FOR I=1 TO LA
100 IF K$=MID$(A$,I,1)THEN 130
110 NEXT I
120 GOTO 15
125 REM- now go to the correct subroutine
130 ON I GOSUB 1000,3000,5000
135 REM- ask the user for his next choice
140 GOTO 15
```

On returning from a subroutine (in line 130), the menu is re-presented. It may be necessary to pause before clearing the screen, if the user has to read the screen output from a subroutine.

A variation of this menu scheme is to select from the menu with a cursor. This is similar to ticking the item you require, as in this example:



The program to produce this menu is shown below:

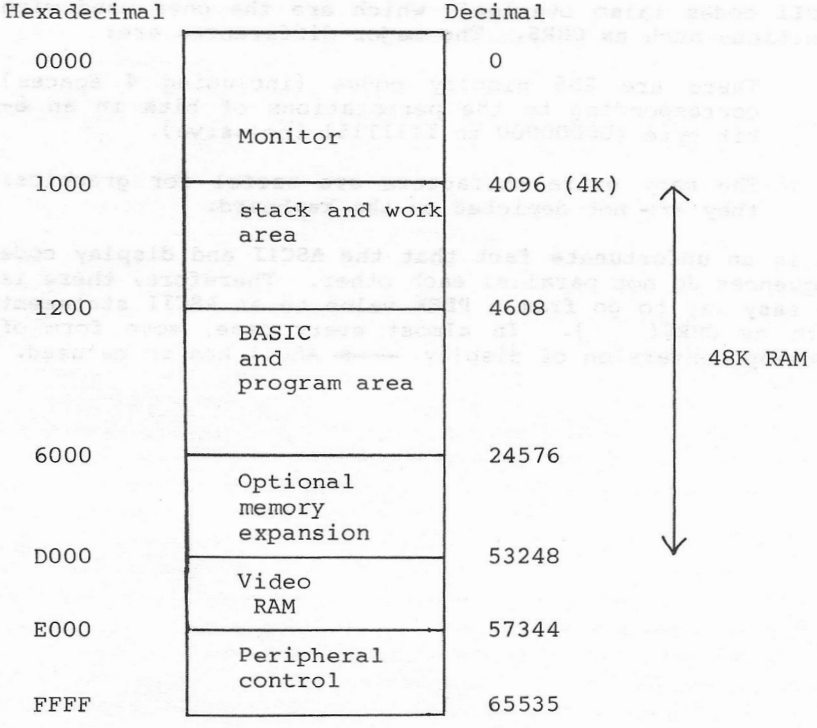
```
1 PRINT"{CLR}"
5 M=5: REM.... no. of options
6 P=10: REM....to position the cursor
10 FOR CP=1 TO M
20 CURSOR 0,CP+P: PRINT"Option   ";CP
30 NEXT CP
50 CP=1
60 CURSOR 0,20: PRINT"USE + OR - OR CR TO STOP"
65 CURSOR 20,CP+P : PRINT"{REV}";
70 GOSUB 1000
80 IF AK$="-" THEN CP=CP-1
85 IF CP=0 THEN CP=1
90 IF AK$="+" THEN CP=CP+1
95 IF CP=M+1 THEN CP=M
100 IF NK=102 THEN 120
105 PRINT"{OFF} "
110 GOTO 65
120 CURSOR 10,22: PRINT"You chose option ";CP
125 REM- THE NEXT STATEMENT IS TO FREEZE THIS DEMONSTRATION!
130 GOTO 130
1000 GET AK$: IF AK$="" GOTO 1000
1010 NK=ASC(AK$)
1020 RETURN
```

The program requires little explanation. CP is used to record the current cursor position, from 1 to 5 in the example. The + and - keys are chosen to move the cursor, subject to CP being maintained betwen 1 and 5. Line 105

erases the old cursor before displaying the new one. This technique is substantially extended in a "screen-handler" described later in this chapter. Before describing more advanced techniques we need, however, to understand the PEEK and POKE commmands.

4.2 Simplified PEEK and POKE

The PEEK (address) statement is used to examine a memory location and to return the value of the data item it finds. This has no effect on any location in memory, but only certain areas of memory are worth PEEKing. The MZ-80K RAM is arranged like this:



An example of a PEEK is

```
B=PEEK(4509)
```

If you print out the value of B, a value of 32 is obtained. Interestingly (?) this is NOT because 32 is stored at location 4509, but because the designer of the system would prefer that you did not find out what is

really there! All PEEKs into addresses in the Monitor stack, BASIC and program areas yield a value of 32.

So, looking for more interesting areas, PEEK into the peripheral control area (61440-65535) and you will find a wide variety of contents. But, the most useful area, for our purposes, is the video RAM. The screen on the MZ-80K contains 1000 locations which are treated as an addressable area of memory - hence the term "memory mapped display". There are 25 lines of 40 characters that make up the total of 1000. Therefore, the "visible" part of the video RAM only extends from 53248 to 54247. Presumably, the rest of the RAM is for larger screens!

The values obtained from PEEKing into the video RAM (screen) are the **display codes** of the MZ-80K. These are shown on the next page. They are not the same as the ASCII codes (also overleaf) which are the ones used with functions such as CHR$. The major differences are:

> There are 255 display codes (including 4 spaces) corresponding to the permutations of bits in an 8-bit byte (00000000 to 11111111 inclusive).
>
> The many extra characters are useful for graphics; they are not depicted on the keyboard.

It is an unfortunate fact that the ASCII and display code sequences do not parallel each other. Therefore, there is no **easy** way to go from a PEEK value to an ASCII statement such as CHR$( ). In almost every case, some form of look-up conversion of display ⟶ ASCII has to be used.

ASCII Code Table

The following are the ASCII codes for characters:

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
32	SP	64	@	96		128	SP	160	q	192	SP	224	
33	!	65	A	97		129		161	a	193		225	
34	"	66	B	98		130		162	z	194		226	
35	#	67	C	99		131		163	w	195		227	
36	$	68	D	100		132		164	s	196		228	
37	%	69	E	101		133		165	u	197		229	
38	&	70	F	102		134		166	i	198		230	
39	'	71	G	103		135		167		199		231	
40	(	72	H	104		136		168	Ö	200		232	
41	)	73	I	105		137		169	k	201		233	
42	*	74	J	106		138		170	f	202		234	
43	+	75	K	107		139		171	v	203		235	
44	,	76	L	108		140		172		204		236	
45	-	77	M	109		141		173	ü	205		237	
46	.	78	N	110		142		174	ß	206		238	
47	/	79	O	111		143		175	j	207		239	
48	0	80	P	112		144		176	n	208		240	
49	1	81	Q	113		145		177		209		241	
50	2	82	R	114		146	e	178	Ü	210		242	
51	3	83	S	115		147		179	m	211		243	
52	4	84	T	116		148		180		212		244	
53	5	85	U	117		149		181		213		245	
54	6	86	V	118		150	t	182		214		246	
55	7	87	W	119		151	g	183	o	215		247	
56	8	88	X	120		152	h	184	l	216		248	
57	9	89	Y	121		153		185	Ä	217		249	
58	:	90	Z	122		154	b	186	ö	218		250	
59	;	91	[	123	o	155	x	187	ä	219		251	
60	<	92	\	124		156	d	188		220		252	
61	=	93	]	125		157	r	189	y	221		253	
62	>	94	↑	126		158	p	190	¥	222		254	
63	?	95	←	127		159	c	191		223		255	

Note: The code is based on the decimal system. SP represents a space.

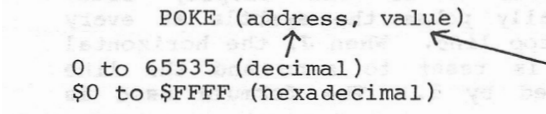
Display Code Table

The following is the display code of the MZ-80K. The code is based on the decimal system.

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
0	SP	32	0	64	SP	96	ア	128	SP	160	｠	192	ｰ	224	ｴ
1	A	33	1	65	♠	97	イ	129	a	161	｡	193	ｱ	225	ｵ
2	B	34	2	66	◢	98	ウ	130	b	162	｢	194	ｲ	226	ｶ
3	C	35	3	67	□	99	エ	131	c	163	｣	195	ｳ	227	ｷ
4	D	36	4	68	♦	100	オ	132	d	164	、	196	ｴ	228	ｸ
5	E	37	5	69	←	101	%	133	e	165	・	197	ｵ	229	ｹ
6	F	38	6	70	♣	102	&	134	f	166	ヲ	198	ｶ	230	ｺ
7	G	39	7	71	●	103	'	135	g	167	ァ	199	ｷ	231	ｻ
8	H	40	8	72	◎	104	(	136	h	168	ィ	200	ｸ	232	ｼ
9	I	41	9	73	?	105	)	137	i	169	ゥ	201	ｹ	233	ｽ
10	J	42	=	74	○	106	+	138	j	170	ェ	202	ｺ	234	ｾ
11	K	43	≡	75	◤	107	*	139	k	171	ォ	203	ｻ	235	ｿ
12	L	44	;	76	◣	108	◥	140	l	172	ャ	204	ｼ	236	ﾀ
13	M	45	/	77	◥	109	✕	141	m	173	ュ	205	ｽ	237	ﾁ
14	N	46	■	78	◢	110	◜	142	n	174	ョ	206	ｾ	238	ﾂ
15	O	47	'	79	▮	111	◟	143	o	175	ッ	207	ｿ	239	ﾃ
16	P	48	¯	80	↑	112	□	144	p	176	ー	208	ﾀ	240	ﾄ
17	Q	49	▏	81	＜	113	□	145	q	177	ア	209	ﾁ	241	ﾅ
18	R	50	▁	82	[	114	□	146	r	178	イ	210	ﾂ	242	ﾆ
19	S	51	▁	83	♥	115	□	147	s	179	ウ	211	ﾃ	243	ﾇ
20	T	52	▔	84	]	116	▔	148	t	180	エ	212	ﾄ	244	ﾈ
21	U	53	▕	85	@	117	▕	149	u	181	オ	213	ﾅ	245	ﾉ
22	V	54	▔	86	◢	118	◹	150	v	182	カ	214	ﾆ	246	ﾊ
23	W	55	▏	87	＞	119	◹	151	w	183	キ	215	ﾇ	247	ﾋ
24	X	56	▔	88	↓	120	▔	152	x	184	ク	216	ﾈ	248	ﾌ
25	Y	57	▕	89	◥	121	▕	153	y	185	ケ	217	ﾉ	249	ﾍ
26	Z	58	▬	90	→	122	▬	154	z	186	コ	218	ﾊ	250	ﾎ
27	€	59	◨	91	▚	123	◧	155	ä	187	サ	219	ｏ	251	ﾏ
28	▗	60	□	92	▗	124	▁	156	◹	188	▙	220	ﾋ	252	ﾐ
29	▖	61	□	93	▗	125	▁	157	◸	189	▟	221	ﾌ	253	ﾑ
30	▘	62	▁	94	▘	126	▬	158	◸	190	ゃ	222	ﾍ	254	ﾒ
31	▘	63	◨	95	▘	127	◨	159	◹	191	ゅ	223	ﾎ	255	ﾓ

Note: SP represents a space or blank.

The complementary statement to PEEK is POKE , having the form



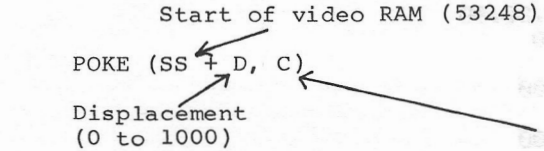
Note that hexadecimal notation can be used for POKE (in the form $nnnn) but not for PEEK. Valid POKEs would be:

```
10 POKE (53400,129)
20 POKE ($62B6,115)
```

Line 10 places the display code for "a" into location 53400. Since this lies in the video RAM area, the letter "a" will actually be displayed on the screen.

The video RAM is the only safe place to POKE into; if you POKE into the monitor RAM area, you will modify its contents. Only do this when you have a thorough grasp of, for example, the monitor and of machine code programming.

Although the various string functions are commonly used for screen output, PEEK and POKE are often more useful - especially in graphical applications. These only involve the video RAM, and to avoid POKEing outside this safe area, we can use the general formula:



The POKE technique has a particular advantage over PRINT: the video RAM (together with the rest of RAM) is continuously scanned by the processor. Therefore any change in the contents of the video RAM is almost immediately displayed.

For example, we can see all of the display code symbols with a simple POKE program. There are 256 display codes and we can print them with, say, 3 spaces between each one as follows:

```
5 PRINT"■"
10 SS=53248: H=40
15 I=0: J=0
20 FOR D=0 TO 255
30 POKE (SS+I*H+J),D
40 J=J+3
50 IF J>40 THEN I=I+1: J=0
60 NEXT D
70 GOTO 70
```

Notice how quickly the program displays its output. In this program, SS is the start of video RAM, H is the line width (40 characters) and D is the display code. Therefore, Line 30 initially pokes the symbols at every third location along the top line. When J, the horizontal counter, exceeds 40, it is reset to zero and the line counter (I) is incremented by 1. The formula used is simple:

Start of video + (40 x line position) + number of spaces across

Getting slightly more ambitious, we can use a particular display code and POKE it at different positions. For example, the next program simulates a random meander of character 202 (a little man) around the screen: Lines 40 to 90 generate a random choice of North, South, East or West and adjust the line position (LP) or row position (RP) accordingly. Of course, we must ensure that we do not wander off the screen, and that is the function of lines 100-130. Each run of the program will give a different result. (But is it Art?!)

```
10 PRINT"■"
20 SS=53248: H=40
30 LP=20: RP=12
35 RM=24: LM=39
40 R=INT(4*RND(1))+1
50 ON R GOTO 60,70,80,90
60 LP=LP+1: GOTO100
65 GOTO 100
70 LP=LP-1: GOTO 100
75 GOTO 100
80 RP=RP+1: GOTO 100
85 GOTO 100
90 RP=RP-1
100 IF LP>LM THEN LP=0: RP=RP+1
110 IF LP<0 THEN LP=LM: RP=RP-1
120 IF RP>RM THEN RP=RM
130 IF RP<0 THEN RP=0
140 POKE (SS+RP*H+LP),202
150 GOTO 40
```

In this program, you can see the advantage of POKEing compared to PRINTing - the output speed for our randomly directed characters is very fast. Random character display is particularly useful for games.

Certain other POKEs are of occasional use. POKE 4509,1 causes a bleep to be sounded each time a key is pressed. Silence is restored by POKE 4509,0.

4.3 Using PEEK and POKE for Output Processing

These statements are commonly used in graphics applications. Some examples are shown in the next chapter. Another aspect is in the processing of data on the screen. This is illustrated with two examples: dumping the screen contents to a printer and the use of protected fields.

4.3.1 Screen/Printer Dumping

Although the MZ-80K has LIST/P and PRINT/P statements, there is no standard way to output a screen full of information to the printer as and when you want it. One way around this is to PEEK at each location on the screen to obtain the display codes, convert them to ASCII codes and print the answer - tedious, but it works!

In this next program, the ASCII equivalents of the display codes are stored in DATA statements. The program reads the ASCII codes into an array S( ) running from S(0) to S(255). This array is used as a "look-up" table since the display code is used to directly index S( ). That is, given a display code D, S(D) contains the equivalent ASCII code.

Lines of ASCII text are built up as follows:

1) The loop starting at 9030 sets the row number from 0 to 24. A character string, S$, is initially set to a blank.

2) The loop starting at 9050 PEEKs at each position on the present row and stores the display code in SV.

3) The function CHR$(S(SV)) converts the display code into the equivalent ASCII code and produces the ASCII character.

4) The character is concatenated into the string S$ until the end of the current row and, in line 9090, it is printed out.

Since the display table is larger than the ASCII table, some simple expedients have been adopted: if there is no exact correspondence, the most similar ASCII character is substituted; where no substitution is possible, such as the symbol with display code 227 ( ), a blank is substituted.

This program was used to prepare all of the computer output for this book. You can use it, as I did, by use of the SWAP command (see later p. 83) or if you only have a cassette system, by adding it to your programs as a subroutine.

```
9000 REM- SCREEN → PRINTER ROUTINE
9010 SS=53248: SL=40: SD=25
9020 DIM S(255)
9025 FOR I=0 TO 255: READ S(I): NEXT I
9030 FOR SR=0 TO SD-1
9040 S$=""
9050 FOR SC=0 TO SL-1
9060 SV=PEEK(SS+SC+SR*SL)
9070 S$=S$+CHR$(S(SV))
9080 NEXT SC
9090 PRINT#P S$
9100 NEXT SR
9104 REM ### TABLE OF ASCII EQUIVALENTS ###
9110 DATA 32,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83
9120 DATA 84,85,86,87,88,89,90,251,205,221,203,209,48,49,50,51,52,53
9130 DATA 54,55,56,57,45,61,59,47,46,44,229,231,236,218,227,226,215
9140 DATA 212,230,232,217,193,207,199,217,202,32,225,254,200,250,95
9150 DATA 248,241,247,63,103,219,220,233,245,58,94,60,91,243,93,64
9160 DATA 201,62,252,92,198,223,208,206,211,210,255,33,34,35,36,37,38
9170 DATA 39,40,41,43,42,222,246,235,234,195,197,239,240,228,231,238
9180 DATA 237,224,253,216,213,242,249,217,193,32,161,154,159,156,146
9190 DATA 170,151,152,166,175,169,184,179,176,183,158,160,157,164,150
9200 DATA 165,171,163,155,189,162,187,153,130,135,140,188,167,172,145
9300 DATA 147,148,149,180,181,182,174,173,186,178,185,168,177,131,136
9310 DATA 141,134,132,137,142,191,133,138,143,190,129,139,144,32,32
9320 DATA 32,32,32,32,32,96,97,98,99,100,101,102,103,104,112,113,114
9330 DATA 115,116,117,118,119,120,121,122,123,124,125,126,105,32,32
9340 DATA 32,32,32,32,32,106,107,108,109,32,110,111,32,112,32
9345 REM.... FOLLOWING ARE PIXEL APPROXIMATIONS
9350 DATA 123,123,115,123,114,223,117,123,222,113,118,116,119,120,112
```

4.3.2 Screen Handler Routines

In most BASIC programs, the user is prompted line-by-line for input:

```
WHAT IS THE VALUE OF X?  237
VALUE OF Y?  49
WHAT IS THE NAME OF THE FILE?  PAYROLL
```

In this short dialogue, the user responses are in bold type. The problems with this standard approach are:

What do you do if the data you keyed in five lines ago is wrong?

You only see one prompt at a time, not the whole picture.

What do you do about adjacent prompts, such as:

```
INPUT X : 500.3  INPUT Y : 492.7
```

In this latter case, the first response might be so long as to partly obliterate the second prompt.

So, what we need for this "form-filling" type of application is a technique that offers these facilities:

A screen-full of prompts can be presented simultaneously.

Any previously-entered data can be corrected.

The length of a user response can be controlled.

Programs offering such facilities are called "Screen Handlers". Alternatively, people talk about "Protected Fields".

The program below is an example of a very simple screen handler. The responses are stored in the string array EF$(NP), so that they can either be used as straight text or with the VAL function, converted to numeric values. As a general point, string input is much safer than numeric; the latter will "break" the program if a string is input.

Since even this simple example is a relatively complex program, here is a two-level description of its operation:

Overview

After displaying the user prompts, a cursor is placed against the first prompt. The cursor keys are used to select any particular prompt:

↑ move to a previous line
↓ move to the next line
→ move one space to the right
← move one space to the left

Corrections are made by over-typing any errors. When all of the prompts have been correctly answered press SHIFT and CLR to accept the screen. In the demonstration program, this simply causes the input data to be printed although you would normally use the data for some other purpose.

The layout of the screen is determined by the associated DATA statements. The first DATA specifies the number of fields. Subsequent ones are of the form:

```
DATA "prompt-string","length of reply"
```

Blank lines are specified by

```
DATA "0","0"
```

Detailed Design

The operation of the program is easiest to describe mainly in program description language, by reference to sections of numbered lines. The use of PEEK and POKE is described in section (3):

(1) Lines 1 to 1086

This is the initialisation section. Data entered by the user is stored in the array EF$ ( ).The 2-dimensional string array PF$(,) is filled from the DATA statements such that the element PF$(I,1) contains the prompt string and PF$(I,2) specifies the maximum length of the user input. For example, PF$(I,1) is "Name" and PF$(I,2) is "20", so that the first line to be typed by the user must not exceed 20 characters. The initialisation process is described by:

```
for i:=1 until field-max
  do
    read field(i), length(i)
       print field(i)
            for j:=1 until length (i)
              do
                print "-";
              end do
  end do
row:=1 ; char:=field(row)+1 ;dup:=char
max-char:=dup+length(row)-1
```

In this description, field (i) and length (i) denoted PF$(i,1) and PFS(i,2).

(2) Line 1090 to 1300

Using similar notation to the above, we have a loop structure:

```
while key ≠ CLR
    do
      if key  = [CR] or [↓] or [↑] or [←] or [→]
        then
             change row (RP) or char (CP)
               //and check the new values//
else
   //key is not a control character//
        call insert(key)
        print key
        char:=char+1
       //check the value of "char"//
end if
```

The range checking on RP and CP is carried out in lines 1210 - 1260.

(3) Lines 5000 - 5090

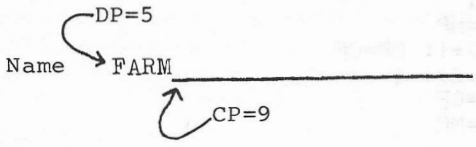
In this section, a cursor flashes at the present insertion position until the user presses a key. The symbols used include:

CC is 1 (cursor on) or -1 (cursor off)
SS is the start of video RAM
PT controls the on/off period
SL is the line length
symbol := current character //use PEEK//

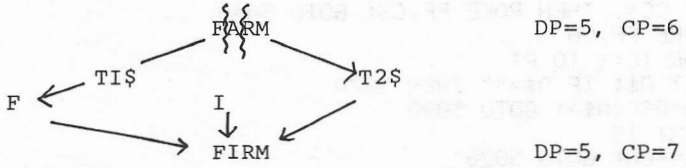
```
repeat
if cursor = TRUE then print " ■ "
                 else print symbol end if
cursor := FALSE

until key ≠ null
```

(4) Lines 6000 - 6050

This simply breaks down the present string EF$(RP) into two halves and inserts the new character at the present cursor position, allowing for the length of the prompt string. Remember that "dup" is a copy of the initial cursor position at each row. So, for the first row, if the user has typed FARM, we would have:



So, DP is fixed but CP is variable, depending on the use of the ← and → keys.
If the user backspaces to the letter A, we have DP=5 and CP=6. The action of lines 6000 - 6050 in response to a key-press of "I" is as follows:



The cursor moves to the letter R, and so on.

Program Listing

PROGRAM NAME: FORMS3

(The listing begins overleaf)

```
1 PRINT"■"
2 RESTORE
1000 READ NP: REM... NUMBER OF FIELDS
1002 REM-EF$ is the entry field, FF$ is the protected field
1005 DIM EF$(NP),PF$(NP,2)
1010 FOR I=1 TO NP
1015 READ PF$(I,1),PF$(I,2)
1020 S$="": EF$(I)=""
1025 FOR J=1 TO VAL(PF$(I,2))
1030 EF$(I)=EF$(I)+" "
1035 S$=S$+"_"
1040 NEXT J
1050 PRINT PF$(I,1);" ";S$
1080 NEXT I
1082 CURSOR 0,22: PRINT"The cursor keys are activated."
1083 PRINT"Shift-CLR/HOME accepts the screen."
1085 RP=1: CP=LEN(PF$(1,1))+1
1086 DP=CP: MP=DP+VAL(PF$(1,2))-1
1090 GOSUB 5000
1100 IF AA<>22 THEN 1120
1110 GOTO 9500
1120 IF (AA=102)+(A$="■") THEN RP=RP+1: GOTO 1210
1130 IF A$="■" THEN RP=RP-1: GOTO 1210
1140 IF A$="■" THEN CP=CP-1: GOTO 1250
1150 IF A$="■" THEN CP=CP+1: GOTO 1250
1155 REM- INSERT THE NEW CHARACTER
1160 GOSUB 6000
1200 CURSOR CP,RP: PRINT A$: CP=CP+1: GOTO 1250
1210 IF RP<1 THEN RP=1
1220 IF RP>NP THEN RP=NP
1230 CP=LEN(PF$(RP,1))+1: DP=CP
1240 MP=DP+VAL(PF$(RP,2))-1
1250 IF CP<DP THEN CP=DP
1260 IF CP>MP THEN CP=MP
1300 GOTO 1090
2000 REM*********************************
5000 REM- get character
5005 CC=1: PT=30: SS=53248: SL=40: CS=239
5010 PP=SS+CP+RP*SL
5015 CN=PEEK(PP)
5020 IF CC=1 THEN POKE PP,CS: GOTO 5040
5030 POKE PP,CN
5040 FOR IC=1 TO PT
5050 GET A$: IF A$="" THEN 5070
5060 AA=ASC(A$): GOTO 5090
5070 NEXT IC
5080 CC=-CC: GOTO 5020
5090 POKE PP,CN: RETURN
5100 REM*********************************
6000 REM- INSERT A$ AT NC INTO T$
6002 NC=CP-DP+1
6004 T$=EF$(RP)
6010 T1$="": T2$="": LT=LEN(T$)
6020 IF NC<>1 THEN T1$=LEFT$(T$,NC-1)
6030 IF NC<>LT THEN T2$=RIGHT$(T$,LT-NC)
6040 T$=T1$+A$+T2$
6045 EF$(RP)=T$
6050 RETURN
```

```
6060 REM*******************************
8900 DATA 8
9000 DATA "Name","20"
9010 DATA "Street","15"
9020 DATA "Town","20"
9030 DATA "County","20"
9040 DATA "Post code","8"
9050 DATA "","0","","0"
9060 DATA "Phone","12"
9500 REM-PRINT THE SCREEN
9502 PRINT"@"
9505 FOR I=1 TO NP
9510 PRINT EF$(I)
9520 NEXT I
```

The program listing is followed by an example run which could be used to provide input to a mailing label program. One advantage of a screen-driver such as this is that it helps a non-expert user to avoid errors or, when errors are made, to permit easy correction.

```
Name INTERNATIONAL KEYS
Street 25 HIGH LANE
Town COTINGHAM
County SURREY
Post code HU1 9NM
Phone 56712
```

The cursor keys are activated.
Shift-CLR/HOME accepts the screen.

5. SIMPLE COMPUTER GRAPHICS

In addition to performing calculations, your MZ-80K can:

Draw illustrations
Play games
Plot data

It really is very simple to get good results and - no - you do not have to learn machine code programming!

5.1 Composing Pictures

There are four main ways to draw a picture on the screen:

1. PRINT
2. POKE
3. CURSOR strings or CURSOR statements
4. SET and RESET

The first three are particularly useful for general drawing applications

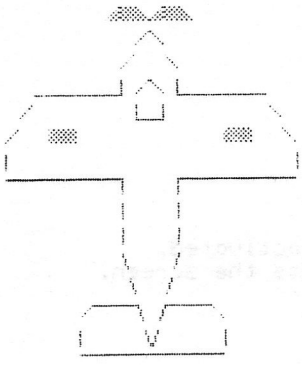
PRINT

This is simplest of all. You simply use the special graphics characters as you see them on the keyboard. For example,

```
10 PRINT "■"
20 PRINT "■"
30 PRINT "■"
```

More complex shapes are tedious to design like this, so a short-cut is to draw the picture on the screen with the various graphics characters and then add the print statements. Here is an example:

```
10 PRINT"■"
100 PRINT"
110 PRINT"
120 PRINT"
130 PRINT"
140 PRINT"
150 PRINT"
160 PRINT"
170 PRINT"
180 PRINT"
190 PRINT"
200 PRINT"
210 PRINT"
220 PRINT"
230 PRINT"
240 PRINT"
250 FOR I=1 TO 10:PRINT:NEXT I
260 GOTO100
```



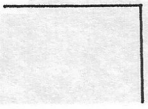
The aeroplane was drawn on the screen, When I was happy with it, I simply typed 100?", 110?" down the side of the picture and so on to change the picture into a program (? is shorthand for PRINT on the MZ-80K). If you RUN the program, the little plane moves up the screen, followed by another..... and another..... and so on.

An alternative to PRINT is POKE, described earlier. The justifications for this are that POKE gives a whole range of symbols and that PEEK, in conjunction with POKE, opens some new possibilities. We will see some of these later but let's see how far PRINT will take us.

Considerable flexibility is added when we use PRINT in conjunction with the cursor control characters. These enable you to build up a "cursor string" and then to print it anywhere, almost instantaneously. The cursor keys ↑ ↓→← must not be confused with the arrow symbols on the keyboard - which simply represent arrows! Cursor keys can be used in a string like this:

A$="--↓← | ↓ ←|"

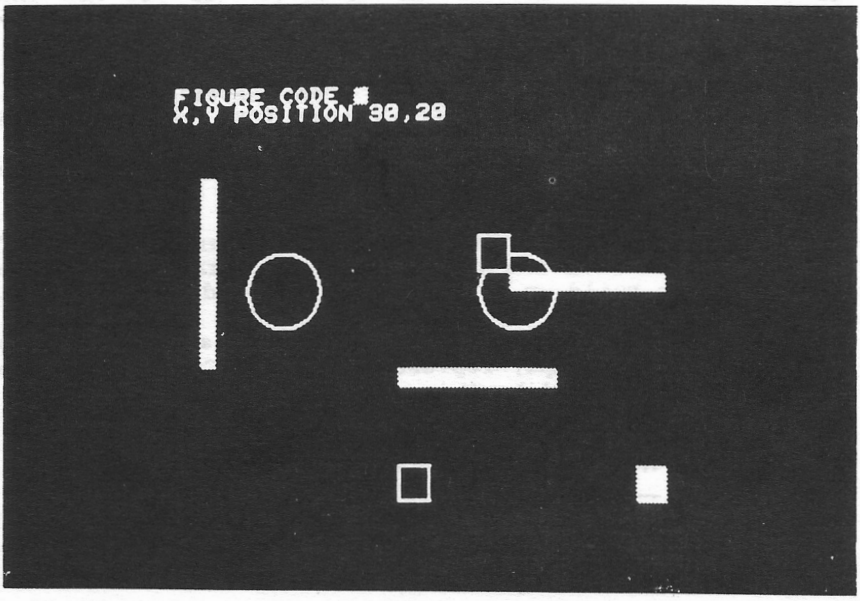
If you print A$, the diagram is:



The cursor itself moves to the right after printing each character, hence the need for the extra cursor-lefts in A$. The real use of cursor strings is in printing pictures just anywhere. For example, this litle program lets you print any one of 5 shapes anywhere on the screen:

```
1 PRINT "@"
10 READ A$
20 CURSOR 0,0: INPUT"FIGURE CODE ";FC
30 RESTORE
40 FOR I=1 TO FC: READ A$: NEXT I
50 CURSOR 0,1: INPUT"X,Y POSITION ";X,Y
60 CURSOR X,Y:PRINT A$
70 GOTO 20
100 DATA "/■■ ■■\■■■■■■■■■■ ■■ ■■/"
110 DATA "■■■■■■"
120 DATA"■■■■■■■■■■■■■■■■"
130 DATA"■■■■■■■■"
140 DATA"■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■"
```

Here is some typical output of all 5 shapes:



If you extend this library of shapes, you can easily assemble a complex picture from its smaller units. Also, unlike the fixed PRINT statements, you can add your library of shapes to every program you write with a great saving in effort. The only limitation is that no string can exceed 256 characters including cursor control characters. This precludes really large pictures.

5.2 Animation and Games

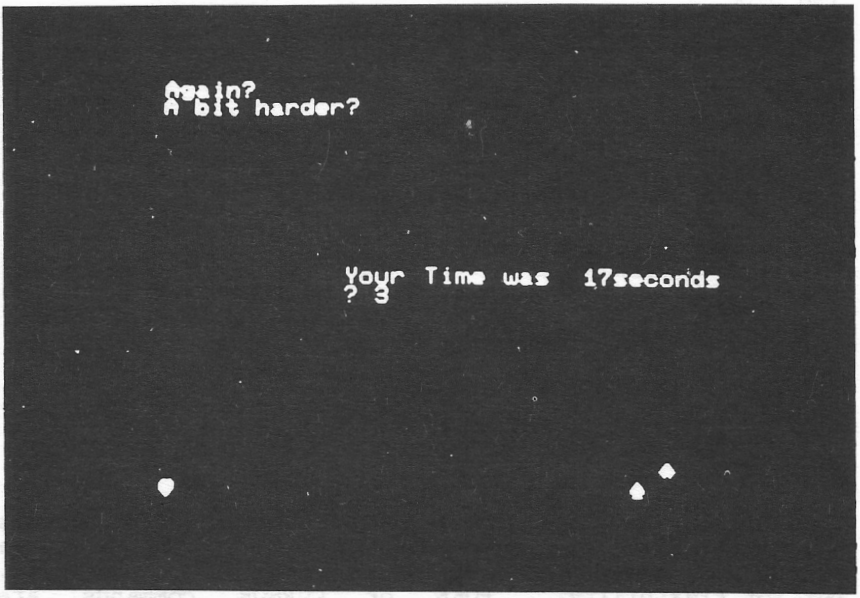
Convincing animation is the most difficult area of graphics, when we are restricted to BASIC. With small objects, there is no problem. For example, this program moves a star alternately back and forth across the screen, making use of the cursor control characters:

```
100 PRINT"■"
110 FOR I=0 TO 39
120 PRINT"■ *";
130 NEXT I
140 FOR I=0 TO 38
150 PRINT"■ ■■*";
160 NEXT I
170 ■■TO 110
```

And this is an equivalent program using CURSOR:

```
100 PRINT"■"
110 FOR I=1 TO 39
115 CURSOR I-1,10: PRINT" ";
120 CURSOR I,10:PRINT"*";
130 NEXT I
140 FOR I=39 TO 1 STEP -1
145 CURSOR I,10: PRINT" ";
150 CURSOR I-1,10: PRINT"*";
160 NEXT I
170 GOTO 110
```

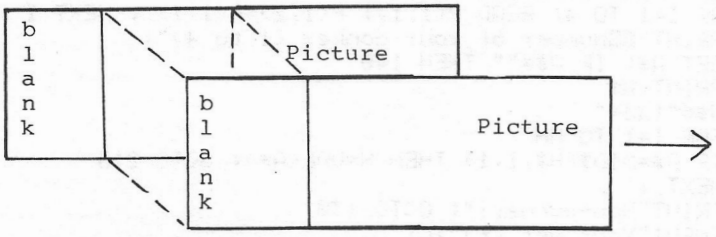
Although this method is limited, it will permit you to construct simple games. The following example draws a small car-like object (ASCII character 96) which moves one step to the right each time a simple sum is answered correctly. The object of the game is to move the car from the start to finish points in the shortest time possible. Various musical effects are included, but the simple animation is in lines 570-585. In the accompanying picture, our user has just completed the course:



```
1 PRINT "■"
2 S1=0:S2=0
3 INPUT "WHICH OPERATION (+ or -) ?";S$
4 IF (S$="+")+(S$="-") THEN 9
5 GOTO 3
9 S1=S1+5:S2=S2+5
10 PRINT"■"
20 TEMPO 4
22 MUSIC"CDEFG"
90 YC=20:Y0=10:X0=12
100 XC=0:X1=4:XE=30:XS=0
105 TI$="000000"
110 CURSOR XS,YC+1:PRINT"♥"
112 CURSOR XE,YC+1: PRINT"♠"
115 GOSUB 585
140 IF XC>=XE THEN 160
150 GOSUB 500
155 GOTO 140
160 MUSIC"CDEFGGFEDC"
161 CURSOR X0,Y0:PRINT"Your Time was ";VAL(TI$);"seconds"
165 CURSOR0,0:PRINT"Again? ";
166 GET A$: IF A$="" THEN 166
170 IF A$="N" THEN 190
181 CURSOR0,1:PRINT"A bit harder? ";
182 GET A$: IF A$="" THEN 182
183 IF A$="Y" THEN 4
184 GOTO 10
190 STOP
500 X=INT(S1*RND(1)):Y=INT(S2*RND(1))
505 IF S$="+" THEN Z=X+Y: GOTO519
506 IF X<Y THEN 500
507 Z=X-Y
519 Z$=STR$(X)+S$+STR$(Y)
520 CURSOR X0,Y0:PRINT"                "
521 CURSOR X0,Y0:PRINT Z$
525 CURSOR X0,Y0+1:PRINT"                "
530 MUSIC"G"
535 CURSOR X0,Y0+1:INPUT A
550 IF A=Z THEN 570
555 MUSIC"C"
560 GOTO 520
570 CURSOR XC,YC: PRINTCHR$(32)
580 XC=XC+X1
585 CURSOR XC,YC:PRINT CHR$(96)
700 RETURN
```

When we try to move more complex shapes, life becomes trickier. Vertical movement can be simulated to some extent just with PRINTs, as shown early in this chapter, but for flexibility, POKE or CURSOR commands are necessary.

In each case, the technique is to draw the picture in one position, then to re-draw it and erase all remnants of the old picture. For example, to move a picture from left to right, we can compose the picture inside a box with blanks down the left-hand side. Then, when the picture moves to the right, the blank edge erases the old left edge of the picture:



This diagram attempts to show the superimposition of two consecutive picture positions.

By way of demonstration, this next program uses this technique to draw the action in a "Donkey Derby". The picture of a donkey is represented in a two-dimensional array (H) of display codes, which are interpreted as shapes as shown below. Notice that the use of POKE is useful here as the symbol for a little man (🯅) to sit on the donkey is only available in the display code table:

```
                0   0   0   202  185  118  215
H (3,7)=        0  158  67   67   67    0    0 =
                0  216 112   0    0   215   0
```

Line 2040 contains the screen addresses at which the donkeys are initially drawn, using subroutine 1000. The heart of the programming is from lines 360 to 390 in which the donkeys are randomly moved from left to right. This is quite a favourite with my children!

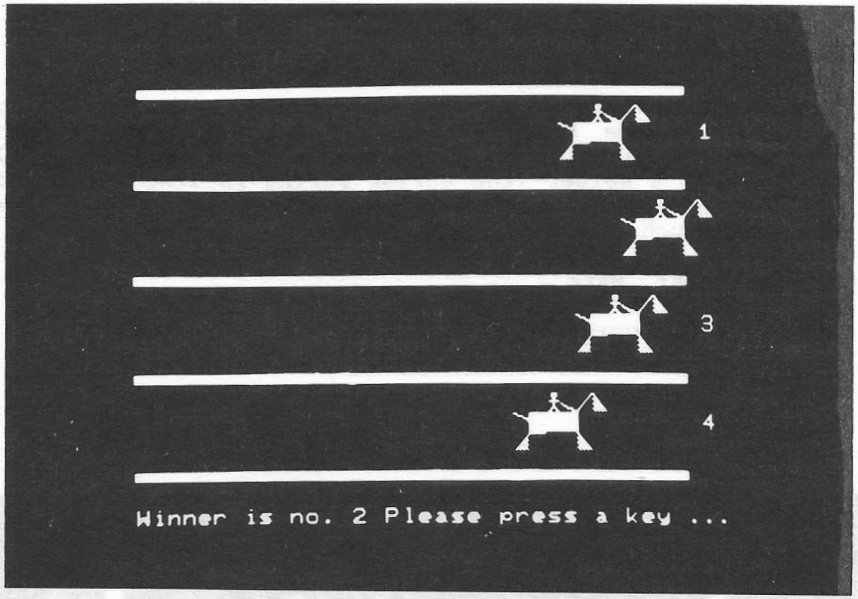
PROGRAM NAME: Donkey Derby

```
1 PRINT "£" : CURSOR 12,8 :PRINT"DONKEY DERBY!!!"
2 FOR I= 1 TO VAL(RIGHT$(TI$,2)) : A=RND(1): NEXT I
3 C=10:NH=4: REM... STARTING CASH AND NO OF DONKEYS
4 PRINT "You now have £";C
6 PRINT "Press 'R' for next race or 'S' to stop ";
8 GET A$: IF A$="" THEN 8
9 PRINT A$
10 IF A$="R" THEN 15
12 IF A$="S" THEN 500
14 GOTO 6
15 RESTORE
20 READ M,N
30 DIM H(M,N),P(4,2)
40 FOR I=1 TO M
45 FOR J=1 TO N
50 READ H(I,J)
60 NEXT J
70 NEXT I
```

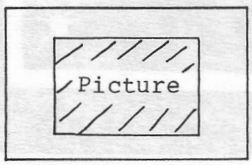
(Donkey Derby program contd)

```
80 REM- READ STARTING POSITIONS
90 FOR I=1 TO 4: READ P(I,1): P(I,2)=P(I,1) : NEXT I
130 PRINT"■■Number of your donkey (1 to 4)";
140 GET A$: IF A$="" THEN 140
145 PRINT A$
150 H$="1234"
160 FOR I=1 TO NH
170 IF A$=MID$(H$,I,1) THEN H=VAL(A$): GOTO 210
180 NEXT I
190 PRINT"Non-runner!": GOTO 130
210 INPUT"Your bet (£)";C1
220 IF C>=C1 THEN 240
230 PRINT"You don't have that much!": GOTO 210
240 REM-DRAW TRACK
245 PRINT"■"
250 FOR I=1 TO 5
260 FOR J=1 TO 35
270 PRINT"■";
280 NEXT J
290 IF I<>5 THEN PRINT"■■";I;"■■"
300 NEXT I
310 REM-START THE RACE
320 FOR H1=1 TO 4
330 PS=P(H1,1)
340 GOSUB 1000
350 NEXT H1
355 CURSOR 5,23: PRINT "THEY'RE OFF!"
360 H1=INT(4*RND(1))+1
370 P(H1,1)=P(H1,1)+1
380 PS=P(H1,1): GOSUB 1000
390 IF (P(H1,1)-P(H1,2))<30 THEN 360
394 IF H1=H THEN C=C+3*C1: GOTO 400
396 C=C-C1
397 IF C>0 THEN 400
398 PRINT"■";"■■■■■■■■YOU LOST ALL OF YOUR MONEY!"
399 MUSIC"9_D9_C": GOTO 500
400 CURSOR 0,23: PRINT "Winner is no.";H1;" Please press a key ..."
410 GET A$: IF A$="" THEN 410
420 PRINT"■" :GOTO 4
500 END
510 REM**********************************
1000 FOR I=1 TO M
1010 FOR J=1 TO N
1020 POKE  PS+(I-1)*40+J-1 ,H(I,J)
1030 NEXT J
1040 NEXT I
1050 RETURN
2005 DATA 3,7
2010 DATA 0,0,0,202,185,118,215
2020 DATA 0,158,67,67,67,0,0
2030 DATA 0,216,112,0,0,215,0
2040 DATA 53328,53528,53728,53928
```

(Donkey Derby example run)

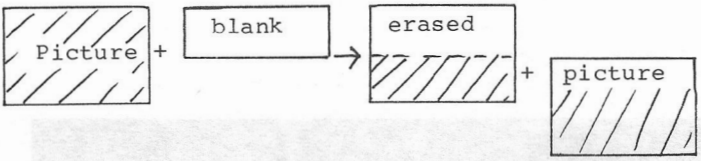


The next stage is to move pictures in any direction, not just left to right. This time, let us use the CURSOR command to position a cursor string. It is necessary to blank-out the remains of the picture after each of the four major possible directions of movement. One way to do this would be to surround the picture with a blank border:



Then, as the picture moves, the blank border erases the old, unwanted parts of the picture. This is a little wasteful of space, and a better method is to have four separate borders to use as required. For example, to move

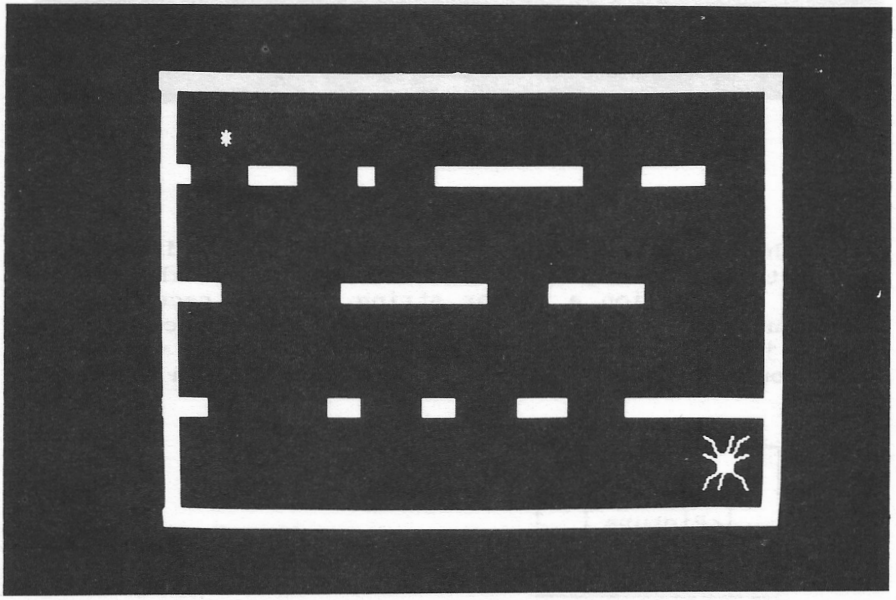
the picture downwards: erase the top segment then print the picture one space down:



This technique is used to good effect in the following interactive game for children, which gives practice in hand-eye coordination. A random maze is drawn (lines 10 to 70) then a spider (the cursor string in line 1010) and fly are drawn. The cursor keys are used to move the spider through the maze to the fly:

PROGRAM NAME: MAZE

Here is a photograph of the screen:



And here is a listing of the program:

Program Listing:

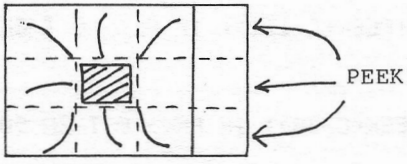
```
1 PRINT"■"
2 T=0.8
3 TEMPO 7
9 REM--- DRAW MAZE
10 FOR I=1 TO 3
15 A$=""
30 R=RND(1)
40 IF R>T THEN 60
50 A$=A$+"▒": GOTO 70
60 A$=A$+" "
70 IF LEN(A$)<=38 THEN 30
80 PRINT "▒▒▒▒▒▒"+LEFT$(A$,38)
90 NEXT I
100 FOR Y=0 TO 22
110 CURSOR 0,Y: PRINT"▒";
120 CURSOR 39,Y: PRINT"▒";
130 NEXT Y
140 FOR X=0 TO 39
150 CURSOR X,0: PRINT"▒";
160 CURSOR X,23: PRINT"▒";
170 NEXT X
200 REM- SET UP THE STRINGS
205 RESTORE
210 READ S$,UL$,UR$,HU$,HL$,F$
220 REM-POSITION SPIDER & FLY
230 X=2:Y=2: XF=35: YF=20: C=53330
240 CURSOR X,Y: PRINT S$;
250 CURSOR XF,YF: PRINT F$;
300 REM-MOVE SPIDER
305 A$=" "
310 P$=A$: GET A$
312 IF (A$="▘")+(A$="▝")+(A$="▖")+(A$="▗") THEN 316
314 A$=P$
316 M=ASC(A$)-16
330 ON M GOTO 340,360,380,400
335 GOTO 550
339 REM- move down
340 PK=PEEK(C+120)+PEEK(C+121)+PEEK(C+122): IF PK<> 0 THEN 500
350 CURSOR X,Y: PRINT HU$;
355 Y=Y+1: C=C+40: GOTO 520
359 REM- move up
360 PK=PEEK(C-40)+PEEK(C-39)+PEEK(C-38): IF PK<> 0 THEN 500
370 CURSOR X,Y: PRINT HL$;
375 Y=Y-1: C=C-40: GOTO 520
379 REM- move right
380 PK=PEEK(C+3)+PEEK(C+43)+PEEK(C+83): IF PK<> 0 THEN 500
390 CURSOR X,Y: PRINT UL$;
395 X=X+1: C=C+1: GOTO 520
399 REM- move left
400 PK=PEEK(C-1)+PEEK(C+39)+PEEK(C+79): IF PK<> 0 THEN 500
410 CURSOR X,Y:PRINT UR$;
415 X=X-1: C=C-1: GOTO 520
500 MUSIC"C": GOTO 550
520 CURSOR X,Y: PRINTS$;
550 IF (ABS(X-XF)<=3)*(ABS(Y-YF)<=3) THEN 570
555 GOTO 310
```

```
560 REM----- CAUGHT IT!
570 CURSOR XF,YF: PRINT" ";
580 MUSIC"CDEFG"
590 GOTO 1
1000 REM*******      SPIDER      ********
1010 DATA"\//▒▒▒▒-▒▒-▒▒▒▒//\"
1020 DATA" ▒▒ ▒▒ "
1030 DATA"▒▒ ▒▒ ▒▒ "
1040 DATA"    "
1050 DATA"▒▒    "
1060 DATA"*"
```

The blank borders Hu$, HL$, VL$ and VR$ are stored in the DATA lines 1020-1050. These strings erase the top, bottom, left or right of the picture. The main part of the program is in lines 300 to 570. It consists of a case construction:

```
repeat
      case of
            cursor down
            if no obstacle below then move spider down
            end if
            cursor up
            if no obstacle above then move spider up
            end if
            cursor right
            if no obstacle to the right then move spider
                                                  right
            end if
            cursor left
            if no obstacle to the left then move spider
                                                   left
            end if
      end case
until spider reaches fly
```

Obstacles are detected by PEEKing at the positions to be occupied by the spider. For example, before moving right, we PEEK like this:



The PEEK locations are +3, +43 and +83 locations from the top left hand corner of the spider.

You can make this game more of a challenge by randomly POKEing or PRINTing new barriers, and making holes appear in old ones. Quite frustrating!

5.3 Picture Storage

In the first part of this chapter, you saw how to design a simple picture and then print it out onto the

screen. It is also easy to store the pictures that you produce, either on tape or disk, and to play them back later.

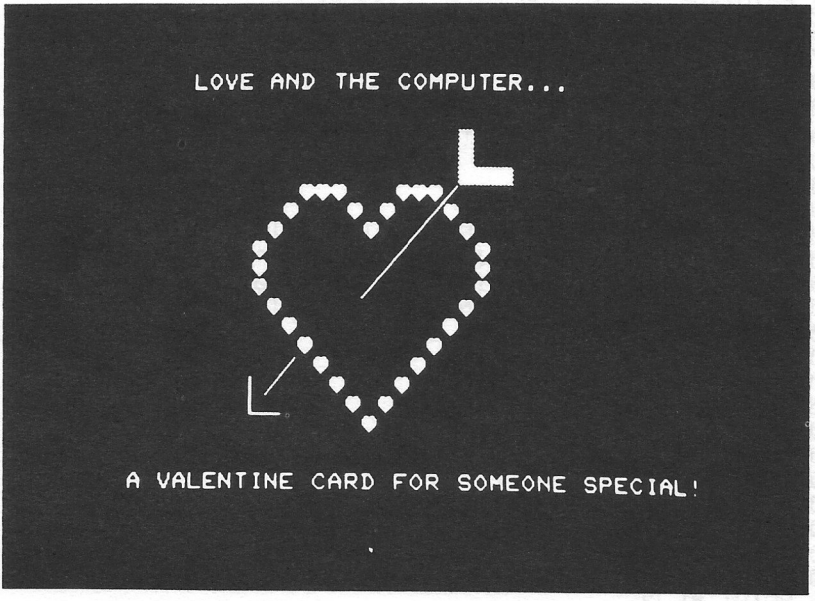
One way to do this is to design the picture, as usual, on the screen and then to PEEK at each location. The display codes returned with each PEEK are then recorded in a file. The problem is complicated a little by the fact that the cursor must be controlled in the program. All this means is that we have to distinguish the cursor codes from all other keys. Finally, we have to tell the program when the picture is complete. In the following program, we did this by choosing "א" as the "end of picture" key.

```
100 REM- PICTURE SKETCHING PROGRAM
101 PRINT"■";"THE PICTURE WILL BE STORED ON FILE DRIVE"
103 PRINT"NO. 2, IN THE FILE 'PICTURE'."
106 PRINT"Press any key to start:";
107 GET A$: IF A$="" THEN 107
108 PRINT"■";"■";
110 FOR I=1 TO 100:NEXT I:PRINT"▒";
120 GET A$: IF A$="" THEN 110
130 IF A$="ₘ" THEN 150
140 GOTO 220
150 PRINT "■ ■";
160 IF ASC(A$)=17 THEN PRINT"■";:GOTO 210
170 IF ASC(A$)=18 THEN PRINT"■";:GOTO 210
180 IF ASC(A$)=19 THEN PRINT"■";:GOTO 210
190 IF ASC(A$)=20 THEN PRINT"■";:GOTO 210
200 PRINT A$;
210 GOTO 110
220 PRINT"■ ";
225 ON ERROR GOTO 250
230 DELETE FD2,"PICTURE"
240 GOTO 260
250 RESUME NEXT
260 WOPEN #1,FD2,"PICTURE"
270 SS=53248: SE=54247
280 FOR I=SS TO SE
290 DC=PEEK(I)
300 PRINT #1,DC
310 NEXT I
320 CLOSE #1
330 END
```

The picture can be stored on tape or disk. The commands for file handling in Sharp BASIC are, in fact, almost identical in either case. This particular program uses the disk BASIC commands to write the screen contents to a sequential file called PICTURE. To re-play the picture, a very small second program reads the PICTURE file and POKEs the display codes back to the screen:

```
10 PRINT"■"
20 ROPEN #1,FD2,"PICTURE"
30 SS=53248: SE=54247
40 FOR I=SS TO SE
50 INPUT #1,DC
60 POKE I,DC
70 NEXT I
80 CLOSE #1
90 GOTO 90
100 REM-THE ABOVE LINE HALTS THE PROGRAM
```

And here is a sample output:



Just think of the money you'll save on Valentine cards!

5.4 Plotting Data and Functions

5.4.1 Which Way Round?

The easiest way to plot graphical representations of data is "sideways on". For example, a schoolteacher may have some percentage marks in DATA statements. All he needs to do is to scale them to a number not exceeding the screen width and then plot them as a character string, like this:

```
10 PRINT"■"
90 K$="■"
100 READ S
110 FOR J=1 TO S
120 READ M
130 N=M*25/100
140 GOSUB 3500
150 PRINT M;TAB(8);A$
160 PRINT
170 NEXT J
175 DATA 10,20,30,60,70,90,55,45,40,35,25
177 SWAP"SCREEN-DUMP"
180 GOTO 180
190 REM*********************************
3500 REM....STRING$
3505 A$="": IF N=0 THEN 3530
3510 FOR I=1 TO N
3515 A$=A$+K$
3520 NEXT I
3530 RETURN
```

As you can see, S tells us how many marks to expect, and the marks are then scaled to between 0 and 25 in line 130. They are printed out with the help of our STRING$ subroutine:

```
20      ■■■■■■
30      ■■■■■■■■
60      ■■■■■■■■■■■■■■■■■■
70      ■■■■■■■■■■■■■■■■■■■■
90      ■■■■■■■■■■■■■■■■■■■■■■■■■■
55      ■■■■■■■■■■■■■■■
45      ■■■■■■■■■■■■
40      ■■■■■■■■■■■
35      ■■■■■■■■■
25      ■■■■■■■
```

Similarly, you can plot a curve such as sin (x). This has values from -1 to +1 (not suitable at all for the MZ-80K), so scaling is essential before plotting. This time, we can use the TAB function:

```
10 PRINT"■"
20 INPUT"START, STEP VALUES ";A,SS
30 L=1: PRINT"■"
35 A=A+SS
40 PRINT"ANGLE"; A;
45 PRINT TAB(10);TAB(25+10*SIN(A*π/180));"*"
50 L=L+1: A=A+SS
60 IF L=20 THEN 80
70 GOTO 40
80 PRINT:PRINT"Press any key.....";
90 GET A$: IF A$="" THEN 90
100 GOTO 30
```

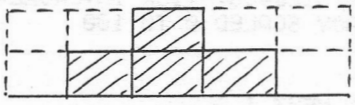
And here is some sample output from this program:

```
ANGLE 20                    *
ANGLE 40                      *
ANGLE 60                       *
ANGLE 80                        *
ANGLE 100                       *
ANGLE 120                      *
ANGLE 140                     *
ANGLE 160                   *
ANGLE 180                *
ANGLE 200             *
ANGLE 220          *
ANGLE 240        *
ANGLE 260       *
ANGLE 280       *
ANGLE 300        *
ANGLE 320          *
ANGLE 340             *
ANGLE 360                *
ANGLE 380                   *

Press any key.....
```

It is a little more difficult, but far more effective, to draw such pictures the correct way round. One solution is to use a "line buffer" - a string variable that is used for the temporary collection of output data.

For example, to draw a histogram of vertical bars, we

could start with a line buffer, A$, initially blank, and equal in length to the number of bars. The bars are constructed, with the line buffer, as horizontal slices:



Each slice is of course the line buffer itself. We add one character (e.g. a block) each time a block is to appear in a particular position. In the diagram above we could be looking at the tops of bars 2, 3 and 4. If we have a set of n values to be represented, by n bars, we can proceed as follows:

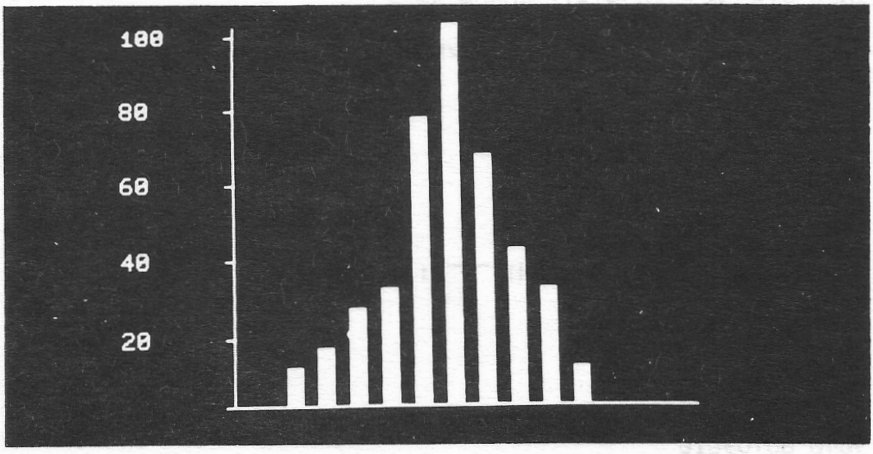
1. Find the highest value in the list and record its position , p, lying from 1 to n.

2. Insert a character, such as ▨ in the line buffer (i.e. in a string such as A$).

3. Look for the next lowest values and insert ▨ into A$, corresponding to their positions in the list.

4. Continue until the lowest values have been identified.

This next program uses this idea. It starts from the highest value and steps down towards the lowest, filling A$ as it goes. A small refinement is that the bars are 3 characters wide, to give better visual impact. In particular, notice the use of the string subroutines from section 2.3.2, in building up A$. Also, the y axis is annotated by using a plain bar or a -1 in lines 6080 - 6100.



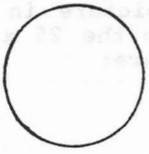
Histogram Program

```
190 REM- MAX. VALUE= VM: STEP SIZE=VS: TICK INTERVAL=VT
195 REM- THE VALUES ARE ALREADY SCALED 0 TO 100
200 READ VM,VS,VT,NB
215 DIM V(NB)
220 FOR I=1 TO NB: READ V(I): NEXT I
230 GOSUB 6000
250 DATA 100,5,20,10
260 DATA 12,16,25,32,77,100,67,40,30,10
300 GOTO 300
3500 REM
3505 A$="": IF N=0 THEN 3530
3510 FOR I=1 TO N
3515 A$=A$+K$
3520 NEXT I
3530 RETURN
4000 REM-STRING REPLACEMENT
4005 B=LEN(B$)
4007 AL=A-1: AR=LEN(A$)-A-B+1
4008 IF AL<0 THEN AL=0: AR=LEN(A$)
4009 IF AR<0 THEN AR=0
4010 A1$=LEFT$(A$,AL)
4020 A2$=RIGHT$(A$,AR)
4030 A$=A1$+B$+A2$
4040 RETURN
6000 REM- HISTOGRAM PLOTTER
6001 PRINT"■"
6002 N=3*NB: K$=" ":GOSUB 3500
6010 IF VM>0 THEN 6030
6020 GOTO 6160
6030 FOR I=1 TO NB
6040 IF V(I)>=VM THEN 6065
6050 GOTO 6070
6065 B$=" ▩ ":A=2*I+1: GOSUB 4000
6070 NEXT I
6080 IF ABS(VM-VT*INT(0.1+VM/VT))<0.001 THEN 6100
6085 PRINT TAB(8);"|";
6090 GOTO 6110
6100 PRINT VM;TAB(8);"┤";
6105 REM-PRINT BUFFER
6110 PRINT A$
6140 VM=VM-VS
6150 GOTO 6010
6160 PRINT TAB(8);
6165 FOR I=1 TO NB :PRINT"──"; : NEXT I
6170 PRINT
6200 RETURN
9000 A$="     "
9010 INPUT B$,A
9020 GOSUB 4000
9030 PRINT A$
9040 GOTO9010
```

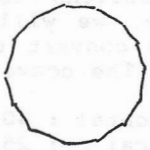
5.4.2 Approximation of Geometric Shapes

There are two main ways of producing regular geometric shapes: use the special graphics characters to **compose** a picture or use sequences of straight-line segments to **approximate** a picture. The first method was used in section 5.1 and we now see how to use the approximation scheme.

All geometric shapes, e.g. circles or rectangles, can be approximated by sequences of straight lines, as shown below:

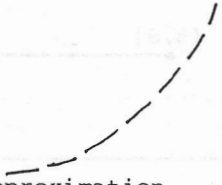


circle



approximation

curve



approximation

Using Solid Line Segments

Even horizontal or vertical straight lines are approximated. For example to draw a straight line, we could write:

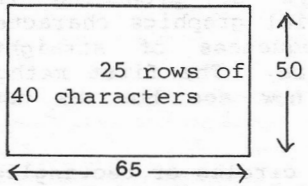
```
FOR I = 1 TO N : PRINT "-";: NEXT I
```

But this is only exact to one whole "-" character (display code 195), fractions are not allowed!

The attraction of this approach is that figures produced with characters such as "-" are continuous. But, to draw more complex shapes than a single line requires a little more thought.

The first problem is that the screen on the MZ-80K, like many other micros, is not square. The aspect ratio (height:width) is, in arbitrary physical units,

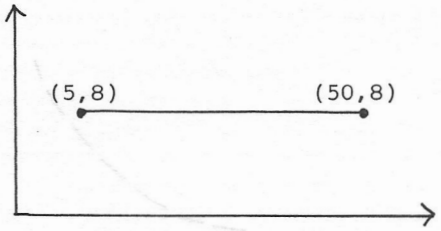
approximately 50:65, but the ratio of columns:rows is 25:40, which is not the same:



So, a conversion factor has to be applied from the physical screen dimensions to the actual printing position. In other words, we will design our picture in the 50 x 65 world, then convert the values into the 25 x 40 character positions. The conversion factors are:

```
horizontal : 40/65 = 0.615
vertical   : 25/50 = 0.5
```

So, to plot a line, we now refer solely to the design area, as if it were a sheet of graph paper:

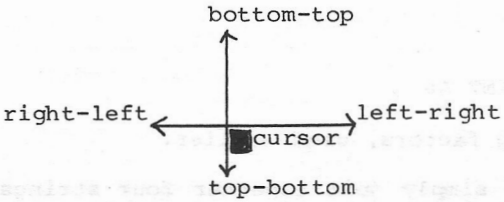


From now on, the TV screen will be called the PROBLEM SPACE and the array of printing positions, the DISPLAY SPACE. In this example, we can easily draw an approximation to the line:

```
10 F1=0.615 : F2=0.5
20 INPUT X1,Y1 : H1=F1*X1 : V1=F2*Y1
30 INPUT X2,Y2 : H2=F1*X2 : V2=F2*Y2
40 CURSOR H1,V1
50 FOR I=1 TO H2 : PRINT "-": NEXT I
```

Of course, this is of limited use, because it will only draw horizontal lines from left to right. In order to draw a rectangle, for example, considerably more care is required. One way of drawing the necessary vertical and horizontal lines is to concatenate a string as a sequence of actual characters and cursor movements. The figure is then represented by a single string which can be positioned anywhere. Restricting ourselves to straight lines, here is how we build up such strings.

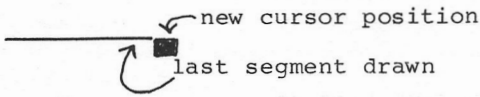
First of all, let us presume that the cursor is initially located as follows:



So, each of the 4 horizontal and vertical lines is positioned differently with respect to the cursor's start position. Now let's see how to draw each line.

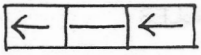
1. Horizontal line, left to right

Simple! Just use a character such as ⊟ since, as it is drawn, the cursor moves automatically to the right:



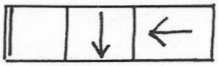
2. Horizontal line, right to left

More tricky! The first line segment is to be printed to the left of the starting position, so the cursor must be moved left then the character is printed. But because this is followed by the cursor moving to the right, yet another cursor-left is needed. So, the sequence is:



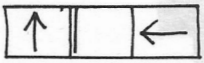
3. Vertical Line, top to bottom

Similarly to the above, we print the vertical segment, causing also an inadvertent shift to the right; then move the cursor down, and to the left i.e.



4. Vertical Line, bottom to top

No doubt you can figure this one out for yourself. The sequence is:



For example, to draw a vertical line of length L downwards from screen coordinates X, Y we could write:

```
10 INPUT "Length";L
20 LA = F2*L
30 INPUT "X and Y"; X,Y
40 X1 = F1*X : Y1 = F2*Y
50 A$ = ""
```

```
60 FOR I = 1TO LA
70 A$ = A$+"| ↓←"
80 NEXT I
90 CURSOR X1, Y1 : PRINT A$
```

F1 and F2 are the scaling factors, used earlier.

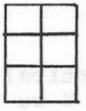
To draw a rectangle, you simply join together four strings correctly, taking care that the cursor does not obliterate any previous part of the figure. The following program contains a subroutine to do this, allowing you to draw any number of rectangles on the screen. The size of the rectangle is limited only by the upper limit of 256 bytes/string. This limitation can be overcome by, for example, using CURSOR to draw each side of the rectangle.

PROGRAM NAME: SOLIDLINES

```
10 PRINT"@"
20 F2=0.5: F1=0.615
90 CURSOR 0,0
100 INPUT"LENGTH & HEIGHT ";A9,B9
105 INPUT "X,Y ORIGIN ";X0,Y0
110 H2=F1*A9: L2=F2*B9
115 XC=F1*X0: YC=F2*Y0
120 L1$="| ": L2$="  "
130 H1$="¯": H2$="_ "
140 GOSUB 4000
150 GOTO 90
4000 REM- SOLID LINE RECTANGLES
4005 A$=""
4010 FOR I=1 TO L2:A$=A$+L1$: NEXT I
4020 FOR I=1 TO H2:A$=A$+H1$: NEXT I
4030 FOR I=1 TO L2:A$=A$+L2$: NEXT I
4035 A$=A$+" "
4040 FOR I=1 TO H2:A$=A$+H2$: NEXT I
4050 CURSOR XC,YC: PRINT A$;
4060 RETURN
```

5.4.3 Using SET and RESET

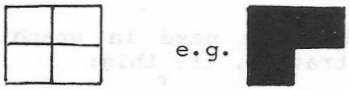
Some microcomputers provide a range of chunky graphics characters based on a 6x2 picture element:



e.g.

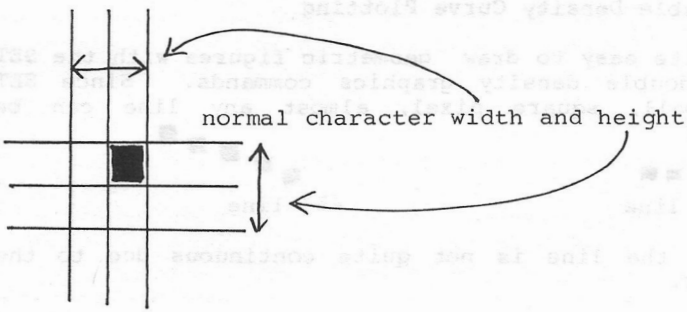
The height and width of this element is roughly the same as that of any other character. They are commonly used in Viewdata-type graphics and are referred to as "Pixel Characters", and the small squares are "Pixels".

The MZ-80K is slightly different, since its pixel characters are square:

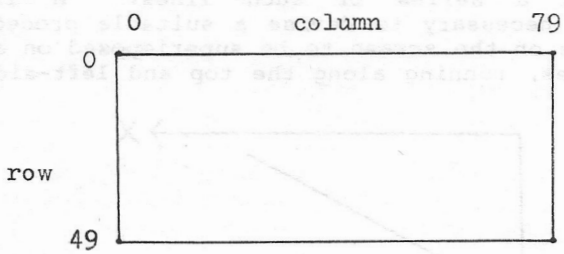


Also, each pixel character is surrounded by a very small border, so the pixel character is not uniformly white. You can see what these characters look like by POKEing the display codes 241 to 255 into the video RAM. There are no equivalent ASCII characters.

Although these pixel characters are useful in graphics, more flexibility is obtained by being able to position a single pixel at any position within a block. Since this requires twice as many horizontal and vertical screen locations than with the normal PRINTs or POKEs, this form of presentation is commonly referred to as "double-density graphics":



Therefore, the screen must now be addressed in terms of 80 columns of 50 rows:



To distinguish double-density from the single-density mode, two new commands are used - SET and RESET. For example:

```
SET 25,45
```

illuminates a pixel in row 25, column 45. While

```
RESET 25,45
```

extinguishes it.

In this chapter, SET and RESET are used in graphics programs. But, for a short illustration, try this:

```
1 PRINT C
10 X = INT(RND(80))
20 Y = INT(RND(50))
30 SET X,Y
40 GET A$: IF A$="" THEN 10
50 GET A$: IF A$="" THEN 50
60 GOTO 10
```

Eventually this will cause the screen to fill with pixels. Lines 40 and 50 are stop/restart switches such that the screen printing stop if a key is pressed, and pauses until a key is pressed again.

A serious side to this program is that it enables you to investigate just how random the RND function really is!.

5.4.3.1 Double Density Curve Plotting

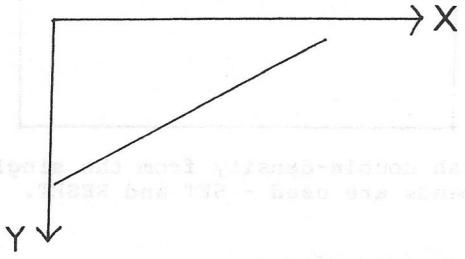
It is quite easy to draw geometric figures with the SET and RESET double density graphics commands. Since SET draws a small, square pixel, almost any line can be represented:

straight line

45° line

Notice that the line is not quite continuous due to the pixel border.

With this technique, the most important procedure is the drawing of a line in any direction, anywhere on the screen. If we can do this, any other figure can be approximated as a series of such lines. A little trigonometry is necessary to devise a suitable procedure. Imagine the line on the screen to be superimposed on a set of cartesian axes, running along the top and left-side of the screen:



Any point on the line is given by:

$$Y = mX+c$$

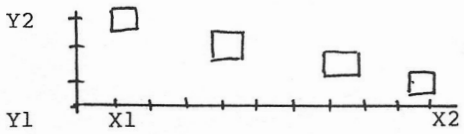
where m is the slope of the line, and C is its intercept on the Y axis. For any two points, we have:

$$(Y-Y1) = m(X-X1)$$
or, $$X=X1+ (Y-Y1)$$

m is computed from the start point (X1,Y1) and end point (X2,Y2) of the line to be:

$$m = (Y2-Y1)/(X2-X1)$$

So, to plot a line from X1,Y1 to X2,Y2 we could increment Y in steps of 1 from Y1 to Y2, calculate the new X at each point, and use SET(X,Y). This is very nearly correct but it is only accurate when (Y2-Y1) is of greater magnitude than (X2-X1). Otherwise, gaps appear along the line:



In this example, (Y2-Y1) equals 4 so that only 4 possible points can be calculated. Therefore, we must increment along the longer of the two directions, even though some points will be over-plotted. An algorithm incorporating this idea is as follows:

```
procedure draw (x1,y1;x2,y2)
      if start-point = end-point then return
      else
            step-x=sign(x2-x1); step-y=sign(y2-y1)
            if length (x2-x1)>length(y2-y1)
            then m:=(y2-y1)/(x2-x1);npts:=x2-x1
            else m:=(x2-y1)/(y2-y1);npts:=y2-y1
            end if
      end if
      while
            not at end point
            do
            if npts =x2-x1
              then y:=y+(step-y)
                   x:=x+m(y-y1)
              else
                   x:=x+(step-x)
                   y:=y+m(x-x1)
            end if
      SET(x,y)
      end do
return
```

A program based on this routine is shown below. The coordinate system is the "problem space" - an imaginary piece of paper 65 units wide by 50 units high. As for the solid line method, a conversion is necessary from the problem to display space. Since we are using double density graphics, with 80 pixels across and 50 vertically, the factors are 1.23 and 1.00. These are used in line 510.

Subroutine dashed-line

```
500 REM-OUTPUT ROUTINE
510 XX=F1*X: YY=F2*Y
520 SET XX,YY
530 RETURN
1000 REM- X1,Y1 TO X2,Y2
1001 X=X1: Y=Y1: GOSUB 500
1002 IF(X1=X2)*(Y1=Y2) THEN 1140
1005 SX=SGN(X2-X1): SY=SGN(Y2-Y1)
1010 YN=ABS(Y2-Y1): XN=ABS(X2-X1)
1020 N9=YN:IF XN>YN THEN N9=XN
1025 X9=X1: Y9=Y1
1040 IF XN>YN THEN M9=(Y2-Y1)/(X2-X1): GOTO 1050
1045 M8=(X2-X1)/(Y2-Y1)
1050 FOR I= 1 TO N9-1
1060 IF XN>YN THEN 1100
1070 Y9=Y9+SY
1080 X9=X1+M8*(Y9-Y1)
1090 GOTO 1110
1100 X9=X9+SX
1105 Y9=Y1+M9*(X9-X1)
1110 X=X9: Y=Y9: GOSUB 500
1120 NEXT I
1130 X=X2: Y=Y2: GOSUB 500
1140 RETURN
```

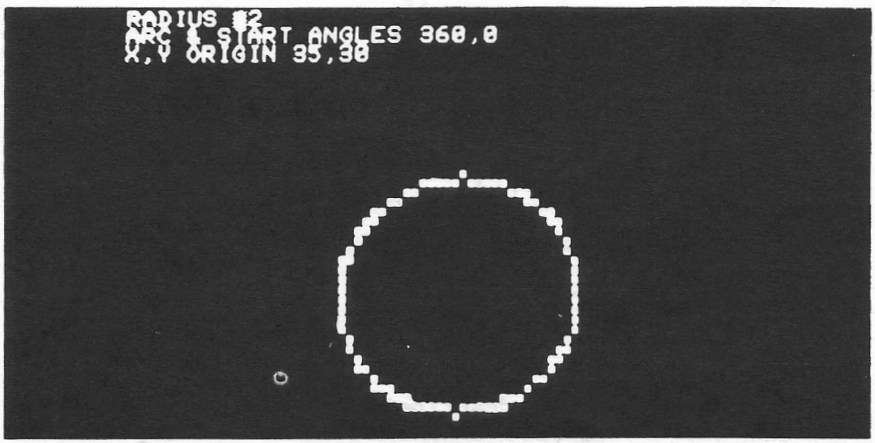
This routine can also be used to draw approximate curves, as demonstrated in this circle-drawing program:

PROGRAM NAME: CIRCLES

The straight-line segments are drawn in lines 2030-2070.

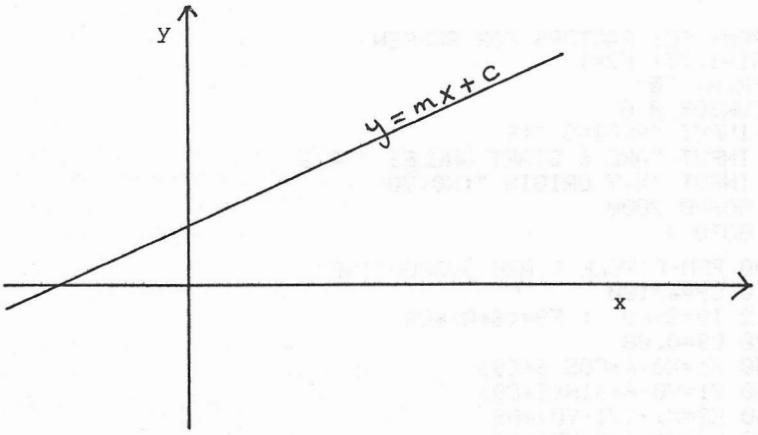
```
1 REM- SET FACTORS FOR SCREEN
2 F1=1.23: F2=1
3 PRINT "E"
4 CURSOR 0,0
10 INPUT "RADIUS ";R
20 INPUT "ARC & START ANGLES ";A,S
30 INPUT "X,Y ORIGIN ";X0,Y0
40 GOSUB 2000
50 GOTO 4

2000 REM-CIRCLE & ARC SUBROUTINE
2010 C9=π/180
2012 T9=S*C9  : F9=(S+A)*C9
2020 D9=0.08
2030 X1=X0+R*COS(S*C9)
2040 Y1=Y0-R*SIN(S*C9)
2060 X2=X1+(Y1-Y0)*D9
2070 Y2=Y1-(X2-X0)*D9
2080 GOSUB 1000
2085 T9=T9+D9
2090 IF T9>=F9 THEN 2110
2095 X1=X2: Y1=Y2
2100 GOTO 2060
2110 RETURN
```

As you can see from the specimen plot below, the results are far from perfect, but are certainly better than nothing!



5.4.3.2 Graph Plotting

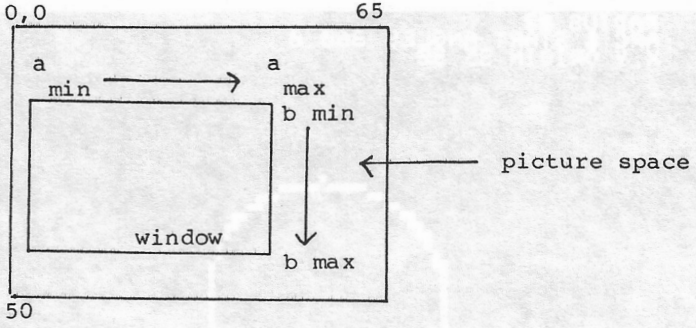
This is a really useful application of the double density techniques illustrated so far. Many people, especially in the scientific area, need to plot graphs similar to this:



For example, a least squares analysis may have been applied to some experimental data. In the simplest case, the equation of the line is:

y=mx+c

Now the values of x and y used by the experimenter will not bear any resemblance to the 65x50 picture space that we have used before. Also, we may not want to use the whole of the picture space for our graph, but just some window:



The window can be described by variables a and b for its width and height. This enables us to express the experimental x_e and y_e values in picture coordinates

x_p and y_p:

$$x_p = a_{min} + \left[\frac{a_{max}-a_{min}}{x_{e,max}-x_{e,min}} \right] . \left(x_e - x_{e,min} \right)$$

$$y_p = b_{max} - \left[\frac{b_{max}-b_{min}}{y_{e,max}-y_{e,min}} \right] . \left(y_e - y_{e,min} \right)$$

Although these equations look horrendous, they are just common sense. They enable you to transfer x,y data from the experimenter's world to a selected window on our picture space. In a moment, we will see that this method is used to plot a line between two particular x_e values, knowing also the gradient, m, and y_e- intercept,C.

The final stage is to draw the axes. This is easiest with solid line graphics. All this entails is the drawing of two straight lines, aligned such that:

the y_p- axis passes through $x_e = x_p = 0$

the x_p- axis passes through $y_e = y_p = 0$

These are useful observations since we can simply substitute $x_e = 0$ and $y_e = 0$ into the above equations and find immediately the values of a for the y-axis and b for the x-axis. Any self-respecting graph has tick-marks on the axes, and this is again easy if we simply say how many are required. To do this, we simply use the variables D1 and D2 (line 66) to divide up the axes (lines 5190-5500) required. Notice that the tick character can not be superimposed after the axis has been drawn since, for example, a character would erase part of the axis line leaving just the tick. Therefore, a complete character . must be used.

All of this is brought together in the following program with a test run following it. We have cheated a little by supplying the minimum and maximum values of x_e, together with the gradient and intercept. In a real-life application, you would have to calculate them.

Program Name: AXIS 2

```
1 REM- SET FACTORS FOR SCREEN
2 F1=1.23: F2=1: F3=0.615: F4=0.5
3 PRINT "E"
10 REM- DEFINE SCREEN AREA
20 AN=5 :AX=53
30 BN=0:BX=45
40 READ XN,XX,YN,YX
50 DATA -55,84  ,-30,98
60 GOSUB 4500
65 REM- SET D1,D2 X,Y DIVISIONS
66 D1=6: D2=5
70 GOSUB 5000
```

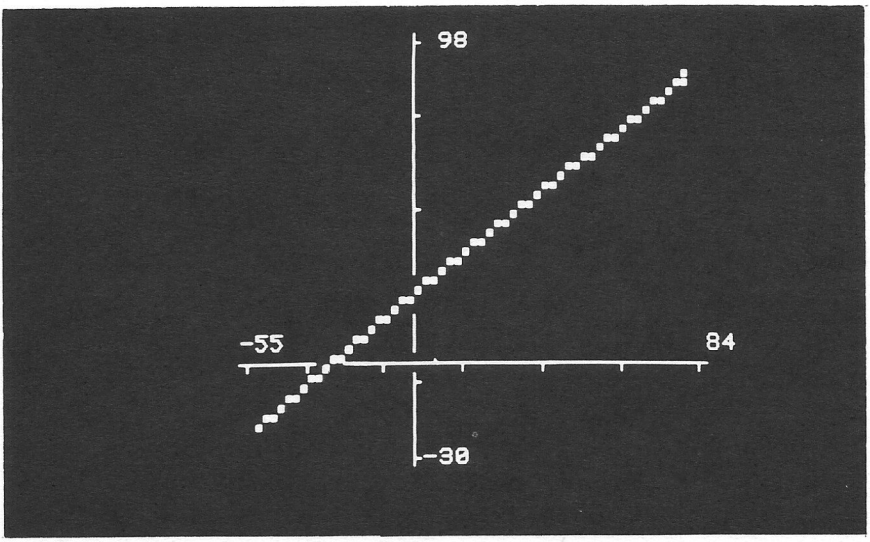
(Details of program AXIS2 continue here)

```
80 REM-USE X1=AN+SX*(X-XN)
85 REM-AND Y1=BX-SY*(Y-YN)
90 REM- TO DRAW A LINE Y=MX+C
100 M=0.8: C=20
105 X=-50
110 X1=AN+SX*(X-XN): Y1=BX-SY*(M*X+C-YN)
120 X=80
130 X2=AN+SX*(X-XN): Y2=BX-SY*(M*X+C-YN)
140 GOSUB 1000
150 GOTO 150
1000 REM- X1,Y1 TO X2,Y2
1001 H1=F1*X1:L1=F2*Y1:SET H1,L1
1002 H2=F1*X2:L2=F2*Y2
1003 IF(H1=H2)*(L1=L2) THEN 1140
1005 SH=SGN(H2-H1): SL=SGN(L2-L1)
1010 LN=ABS(L2-L1): HN=ABS(H2-H1)
1020 N9=LN:IF HN>LN THEN N9=HN
1025 H9=H1: L9=L1
1040 IF HN>LN THEN M9=(L2-L1)/(H2-H1): GOTO 1050
1045 M8=(H2-H1)/(L2-L1)
1050 FOR I= 1 TO N9
1060 IF HN>LN THEN 1100
1070 L9=L9+SL
1080 H9=H1+M8*(L9-L1)
1090 GOTO 1110
1100 H9=H9+SH
1105 L9=L1+M9*(H9-H1)
1110 SET H9,L9
1120 NEXT I
1130 SET H2,L2
1140 RETURN

4500 REM-SCALING ROUTINE
4510 SX=(AX-AN)/(XX-XN)
4520 SY=(BX-BN)/(YX-YN)
4530 REM- THIS COULD ALSO FIND MIN/MAX VALUES  & SCALE ALL X,Y VALUES.
4540 RETURN
5000 REM-AXIS ROUTINE
5010 IF XX<0 THEN 5050
5020 IF XN>0 THEN 5040
5030 A0=AN-SX*XN: GOTO 5060
5040 A0=AN: GOTO 5060
5050 A0=AX
5060 REM- DRAW Y AXIS
5070 FOR I=BN TO BX
5074 REM- USE THE SLIGHTLY-RIGHT BAR
5075 CURSOR F3*A0,F4*I: PRINT"|";
5080 NEXT I
```

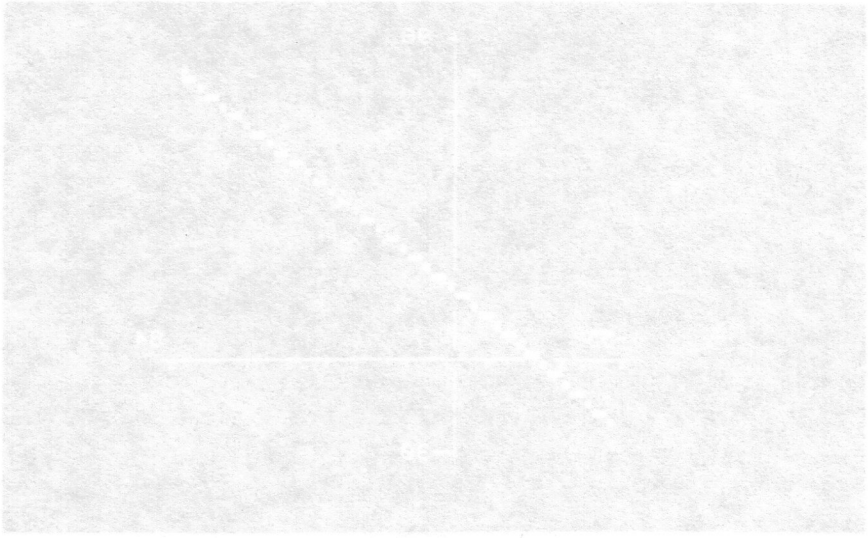
(Program concludes here)

```
5100 IF YX<0 THEN 5140
5110 IF YN>0 THEN 5130
5120 B0=BX+SY*YN: GOTO 5150
5130 B0=BX: GOTO 5150
5140 B0=BN
5150 REM- DRAW X-AXIS
5160 FOR I=AN TO AX
5170 CURSOR F3*I,F4*B0: PRINT"-";
5180 NEXT I
5190 REM- ADD TICK MARKS
5195 BY=(BX-BN)/D2
5200 FOR I=0 TO D2
5205 TY=BN+I*BY
5210 CURSOR F3*A0,F4*TY: PRINT"+";
5220 NEXT I
5230 AY=(AX-AN)/D1
5235 FOR I=0 TO D1
5240 TX=AN+I*AY
5250 CURSOR TX*F3,B0*F4: PRINT"+";
5260 NEXT I
5270 REM-ADD AXIS LABELS
5280 CURSOR A0*F3+1,BN*F4:PRINT YX;
5290 CURSOR A0*F3+1,BX*F4:PRINT YN;
5300 CURSOR AN*F3,B0*F4-1:PRINT XN;
5310 CURSOR AX*F3,B0*F4-1:PRINT XX;
5500 RETURN
```

TEST RUN



This is a very neat demonstration of the use of both single- and double- density graphics in one program. The only way to produce reasonable graphics is to practise doing it for yourself. As an exercise, try to plot the graph of a curve such as sin(x) or a quadratic equation, $ax^2 + bx + c$.



6. FILE HANDLING FUNDAMENTALS

So far, all of the data we have produced has disappeared when the power was turned off. But, the data can be permanently recorded onto either cassette or floppy disk, in much the same way that recordings of music are made on a hi-fi. In each case, the recording surface is a magnetizable medium such as ferric oxide.

There are two ways of organizing the recorded data, and in each case a collection of related data is called a "file". We can have either:

a sequential file: the data items are recorded one after another. So, you can only find a particular item by starting from the first one.

a direct access file: any item can be retrieved almost equally quickly by specifying some reference number!

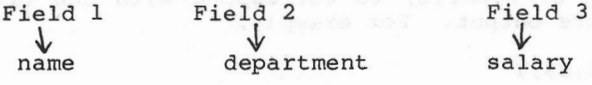
An example of a sequential system is the familiar cassette tape recorder. By comparison, a direct access method is used when selecting a track on a long-playing record.

6.1 Terminology

Files are often divided into sets of data called **records**. Each record consists of one or more data items. It is usually convenient to put sets of related data items into a record, such as employee name, department and salary. Then, when a record is obtained from a file, a complete set of usefully-related data is obtained:

	name	department	salary
record 1	Cookson	Sales	7000
record 2	Jones	Service	6000
record 3	Lambert	Sales	6500
record n			

This organisation is similar to that of a card index or filing cabinet. Usually the records would be arranged in a physical order that related to one particular aspect or **field** of the record. In our example there are 3 fields:



We might decide to arrange the records in alphabetical order of field 1: i.e., for the employee names as shown. The result is called a sequential file, and you simply examine each record until the required one is found. This type of file can easily be stored on either tape or disk. Alternatively, a record can be accessed by translating the contents of a field (e.g. a name) directly into a physical location. This is called a direct access file, and is only of use with disk storage.

6.2 Data Files on Tape and Disk

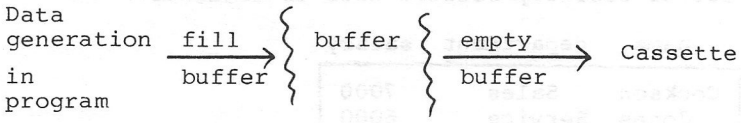
Before delving into how data files are organised for efficient usage, we need to know the fundamental commands that control the storage and retrieval of data on the two media. The cassette system is simplest to understand, but has some limitations. Fortunately, the more comprehensive disk system has many similar commands to the cassette system, therefore making it easy for the MZ-80K user to move from one to the other.

6.3 The Cassette System

The LOAD and SAVE commands for BASIC programs are probably well known to you. In order to process data files, there are the following additional commands:

(line-number) WOPEN"filename"

This allocates an area of temporary RAM storage, called a **buffer**, into which data is transferred and then stored onto cassette.



For example, 100 WOPEN"NEWFILE"

(Line number) PRINT/T (data list)

This statement outputs data, generated by the program, into the buffer. The buffer is emptied when it contains 128 bytes or when the CLOSE command is executed.

For example, 150 PRINT/T V1,A$,T

INPUT/T (data list)

This reads data from cassette, requiring the data types, string or numeric, to correspond with the order in which they were output. For example,

190 INPUT/T X,B$,Y

would be an acceptable way of reading the data that had been written by the preceding PRINT/T.

ROPEN (file name)

When this statement is executed, the tape is wound on until the computer locates the named data file, which is then able to be read by the INPUT/T statement e.g.

```
170 ROPEN "NEWFILE"
```

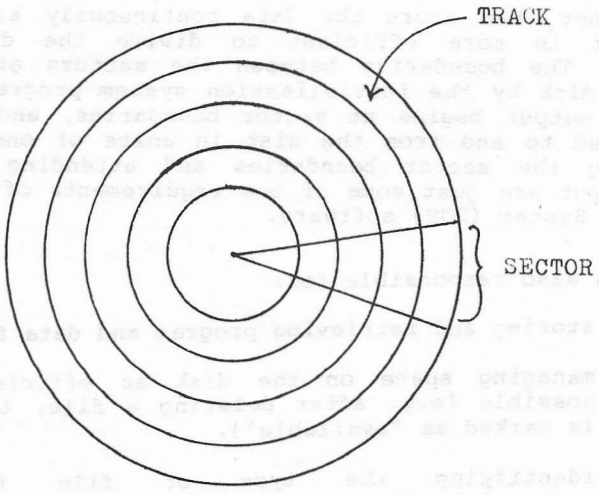
CLOSE

This must be used whenever a file has been written to or read from. In the former case, it has the additional property of emptying the file buffer.

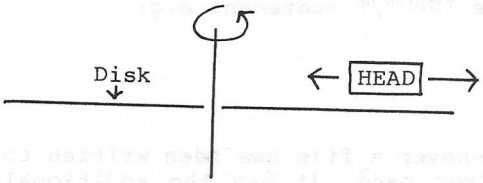
6.4 The Disk System

6.4.1 Disks and Disk Drives

A floppy disk is organised with the data recorded in concentric tracks:



The read/write head of the disk drive unit moves across the disk surface:



The head is positioned above the selected data track. On the MZ-80K each track can contain up to 2560 bytes of information, and there are 35 tracks available on each side of the disk, giving 179,200 bytes of storage per disk drive. The diameter of the disk is 5¼". A disk with this amount of storage is said to be: Double sided, single density.

Rather than store the data continuously along each track, it is more efficient to divide the disk into sectors. The boundaries between the sectors are placed onto the disk by the initialisation system program. Data input or output begins at sector boundaries, and data is transferred to and from the disk in units of one sector. Allocating the sector boundaries and attending to data input/output are just some of the requirements of the Disk Operating System (DOS) software.

The DOS is also responsible for:

storing and retrieving program and data files.

managing space on the disk as efficiently as possible (e.g. after deleting a file, the space is marked as "available").

identifying the type of file (program, sequential, direct access or system).

security of file access.

Of course, you could write separate programs yourself that would accomplish all of these tasks. The purpose of using DOS, however, is is to provide the user with a comprehensive set of utilities without having to bother about the nitty-gritty of disk organisation.

Before DOS can be used, it has to be loaded from the system disk, supplied with the disk drive, into RAM. This is done under the control of the in-ROM monitor program. So, when the computer is first turned on, you will see:

```
** MONITOR SP - 1002**
*
```

Since the DOS is the first program to be loaded, the process of taking it from disk to RAM is called "boot strapping" since, so to speak, it lifts itself by its own boot laces. (Why not "boot lacing"?.....I've no idea!).

DOS is loaded or "booted" complete with the disk BASIC interpreter as follows:

Insert the Master disk into a disk drive.

Type "FD", then in response to

BOOT DRIVE? Type 1 or 2 since the Master disk can be inserted into either drive 1 or 2. Drive 1 is assumed if you simply press the CR key.

The system responds with:

```
* SHARP BASIC SP-6015
27188 BYTES
READY
```

Notice that the remaining RAM is reduced (from 48000 bytes in the example) due to the lengths of the files which constitute DOS and BASIC.

6.4.2 Disk Utilities

The DOS supports a number of essential programs called utilities. These are connected with:

Initialisation (S-Diskette INIT)

Before a disk can be used, the sector boundaries must be recorded, together with a minimum of system information such as the location of the directory. Initialisation is often referred to as "formatting".

Copying (DISKETTE COPY)

For security, at least one copy of your programs should be made. If you are using a new diskette, it must be initialised. An old disk (containing data) may be used but the DOS may initially reject it, even though it is formatted. If this happens, demagnetize the disk with a bulk eraser, and re-initialise it.

Tape to Disk (FILING-CMT)

Cassette programs can be transferred from cassette tape to disk and, because the dialects of BASIC are so similar, you run these programs just as any other BASIC program. But, unlike BASIC programs, you must return to the monitor (by typing "!") and re-boot the system before you can continue.

6.4.3 Disk BASIC

There are very few differences between the computational abilities of cassette and disk BASIC. There are four new or updated commands within the BASIC language:

1) CURSOR X,Y
2) Up to 10 USR machine code functions are allowed
3) RESTORE is allowed to refer to a line number. RESTORE now moves the DATA pointer to a particular line of data
4) INP and OUT are followed by @, rather than #, since # now refers to a buffer associated with a file (or with a USR function)

There are also many commands specifically concerned with program-files or data-files on disk:

6.4.3.1 Direct Mode Commands

These are used in conjunction with BASIC program files:

DIR[d][/P]

With this and the other commands, square brackets indicate that the enclosed parameters are optional. DIR lists a directory of the names of the files on a disk onto the screen. If a disk drive (d) is specified, such as DIR2, the directory for the file in drive 2 is shown. If d is omitted, drive 1 is assumed. DIR with the /P gives more information:

```
VOL.MASTER(S,673)
OBJ*   "S-DISKETTE INIT"          19 SECT
OBJ    "FILING + CMT"             23 SECT
OBJ*   "DISKETTE COPY"            15 SECT
OBJ*   "BASIC SP-5025"            92 SECT
BTX    "PROLOGUE"                 19 SECT
BTX    "FASHION"                  13 SECT
BTX    "CLOCK"                    23 SECT
BTX    "TAKE-DOWN GAME"           19 SECT
```

Notice that:

After displaying the directory on the screen, you can position the cursor against a selected program file and over-type BTX with, for example, RUN, LOAD or DELETE.

The display stops when a screen-full of filenames has been listed. The next screen-full is listed after any key is pressed.

After a DIR or DIR/P has been executed, the system presumes that files will be selected from the drive number you used, unless you over-ride this condition (see next section).

With the exception of SAVE, the following commands can be used on their own, or they can be typed against a selected program on a directory listing:

SAVE [FDd@V,1]"program name"

This command stores the BASIC program currently in RAM to disk. If the first two parameters are omitted:

SAVE "TEST"

then the program will be stored on the disk most recently referenced by a DIR command. Thus, if you have previously typed DIR2, program TEST will stored on drive 2. If a program of the same name exists the error message ER42 is returned which, as you will see from the list at the end of this chapter, means that the file has already been "registered". Therefore, you can not accidentally over-write an old file. The file drive number and volume number (v) are optional and are described at the end of this chapter.

NOTE
if you save a blank file name i.e. SAVE"" you will DESTROY access to all files on the diskette.

LOAD [FDd@V,]"program name"

This is the reverse of SAVE: it transfers a previously-saved program into RAM. Remember that if you omit the FD parameter, the system will automatically search for the file on the directory most recently listed with a DIR.

RUN [FDd@V,] ["program name"]

RUN with no other parameters will execute the program currently in RAM. If a file name is specified, the program specified by the file name will be LOADed and then RUN.

6.4.3.2 Statement and/or Direct Mode Commands

This next group can be executed either as a numbered line in a BASIC program or in direct mode. Whilst some (such as DELETE) can be applied to data or program files, some (such as SWAP) can only be used with program files:

DELETE [FDd@V,] "filename"

The file is deleted from the disk and the file name disappears from the directory. The space occupied by the file is returned to a free storage area so that new files can be stored.

RENAME [FDd@V,] "filename 1","filename 2"

The contents of a file are unaltered, but its name is changed from the first file name to the second.

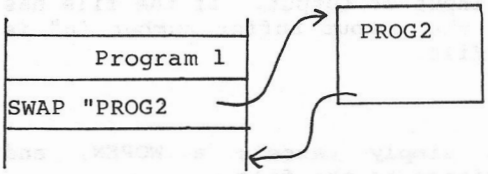
CHAIN [FDd@V,],"program name"

The current program transfers control to the beginning of a new program identified by "file name". This over-writes the old program, but the values of all variables are preserved so that they can be used by the new program. It can be used when a program is too large for the RAM, and can be split into smaller, connected parts. Obviously, the execution time is increased because of the physical disk access required.

SWAP [FDd@V],"program name"

This is similar to CHAIN except that, when the new

BASIC program has been executed, control returns to the statement after the SWAP in the original program



So, this is the disk version of a subroutine. But, unlike a subroutine, the SWAPped program may not itself SWAP with another program. Both CHAIN and SWAP can be used to load a program in the middle of some other program. This is useful since LOAD can only be executed as a direct mode command.

LOCK [FDd@V,]"file name" UNLOCK [FDd@V,]"file name"

These permit a small degree of security to be added:

```
LOCK  FD2"TEST"
```

Program TEST on drive 2 can no longer be deleted or changed, until we type

```
UNLOCK FD2"TEST"
```

A locked file has an asterisk against it in the directory listing.

Since LOCK and UNLOCK can be used as statements, they can be used in conjunction with a password to add a further level of security.

6.4.3.3 Data File Commands

The third and final group are concerned solely with data files. Their operation is discussed in detail in the next two chapters, but, briefly, there are 5 sequential file commands:

WOPEN#n,[FDd@V,]"file name"
Opens the file output and places the file name on the directory

ROPEN#n,[FDd@V,]"file name"
Open the file for input

PRINT#n,(list of variables)
Prints the values of the variables onto the file

INPUT#n,(list of variables)
Reads variables from the file

CLOSE[#n]

Closes the file for input or output. If the file has been WOPENed, any data in the output buffer number "n" is physically written to the file.

KILL[#n]

Unlike CLOSE, this simply cancels a WOPEN, and prevents any data being written to the file.

If #n is omitted, CLOSE or KILL applies to all files currently open.

There are 3 direct access file commands:

XOPEN#n,[FDd@V,]"file name"

This opens a specified direct access file for reading or writing.

PRINT#n(r),(list of variables)

Outputs the variables to the direct access file XOPENed with buffer n. The first variable is output into record r, the next into record (r+1) and so on.

INPUT#n(r),(list of variables)

Reads the values of a list of variables, starting with the value in record r.
For both PRINT and INPUT, r can be a numeric integer or an expression. KILL and CLOSE can also be used with direct access files.

6.4.3.4 File Condition Commands

These are concerned with detecting particular conditions that occur with file processing:

End of file detection

IF EOF(#n) THEN line-number

This transfers control to a specified line number when the end of a file with buffer number n is encountered.

Error processing

A general error condition can be detected by:

ON ERROR GO TO line number

A specific error is handled by:

IF ERN=error number THEN line number
or IF ERL=line number THEN line number

In either case, the line to which we jump will take some action to correct the error. One general statement for error correction is:

```
RESUME[NEXT or line number or 0]
```

If no parameter is specified, the program returns to the line in which the error occurred. NEXT causes the subsequent line to be executed. A line number transfers control to a specific line, while 0 causes control to be passed to the beginning of the program.

6.5 The FD and @ parameters

Disk file commands in this chapter have been of the form:

```
command[FDd@V],"filename"
```

where d = number of the drive (1 to 4) containing the required disk
V = a reference "volume" number from which we can identify a particular disk

These optional parameters enable you to direct output to or from a particular disk drive, and ensure that a specified disk is being used, if necessary.

As you can see from the form of the commands, both the FD and @ parameters are optional. Also you can omit:

the comma
the second quotation mark

So, the simplest form of this family of commands is:

```
command"filename
```

Normally, the file name is enclosed in a program, this means that you must explicitly refer to a filename such as:

```
SWAP FD2,"NEWPROG"
```

But what if the program was not always NEWPROG? In that case, you would prefer to input the file name and file drive

```
100 INPUT"New file name";F$
110 INPUT"File drive";DF
```

Then, you must specify the file drive with a simple POKE sequence:

```
120 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
```

The $ indicates that 49AC, for example, is a hexadecimal number. You can now type:

```
130 SWAP F$
```

This enables you to select any file without changing the program.

Of course, the POKE sequence is omitted if a directory has been listed and you simply want a file from that directory. You can then type:

```
100 INPUT"File name";F$
110 SWAP F$
```

But, this needs to be used with care. For example, consider this sequence:

1. The user lists the directory on drive 1, which contains a write-protected Master disk.
2. He loads and runs a program on disk drive 2....
3. He then selects a new program, using SWAP, such as:

```
SWAP FD2,"SCREEN-DUMP".
```

All goes smoothly, until the system attempts to store a copy of the program loaded in step 2 onto the currently-active disk, which, because of step 1 is drive 1. The disk in this drive is write-protected so, disaster! The manual does not warn you about such pitfalls so, the safe way is to specifically select the disk drive (using the above POKEs) **every** time!

7. SEQUENTIAL FILES on TAPE and DISK

Sequential files are organised with the data written in the same order as it must be read. In the case of cassette tape, there is no real alternative. With disks, there is a choice of sequential or direct-access files, but the sequential organisation is much simpler to understand, before getting to grips with the mysteries of direct-access addressing problems.

7.1 Cassette Data Files

Cassette data files are created by the following sequence:

1) Open a named file with the WOPEN (write-open) command.

2) Use the PRINT/T command to output the data to the file.

3) Close the file using CLOSE.

For example, this short program stores the employee data from section 6.1 onto a file called STAFF:

```
100 WOPEN"STAFF"
110 PRINT"■"
120 INPUT"Name (ZZZ to finish): ";N$
125 IF N$="ZZZ" THEN 170
130 INPUT"Dept: ";D$
140 INPUT"Salary: ";S$
150 IF VAL(S$)<=0 THEN PRINT "Invalid salary .. try again": GOTO 140
160 PRINT/T N$,D$,S$
165 PRINT"■■": GOTO 120
170 PRINT/T "ZZZ","",""
175 CLOSE
180 END
```

After typing RUN, the screen displays:

↓ RECORD PLAY (i.e. press both the record and play buttons on the cassette recorder).

the cassette starts and we see:

WRITING STAFF (a "leader" is being written on the tape so that the file can be found later)

The cassette stops and we are asked to input the name, department and salary.

Notice that the program checks that the salary is numeric and positive, with the VAL statement.

The cassette remains stationary until the CLOSE statement. This is because the data is stored in a temporary buffer; CLOSE causes the data to be output . This is very efficient, and avoids the long gaps that would exist on the tape if each data item were immediately output to the file. The buffer was described in the previous chapter.

We now have a sequential file, safely stored on tape. We can think of it as a representation of the employee data in records:

[name, department, salary]

although, on cassette tape, the data items follow each other and there are no actual physical records.

To read this data back, we need a very similar program. The data could be read into string arrays or, as we have done, simply displayed on the screen:

```
100 ROPEN"STAFF"
110 PRINT"■"
112 INPUT#T N$,D$,S$
115 IF N$="ZZZ" THEN 175
120 PRINT"Name : ";N$
130 PRINT"Dept: ";D$
140 PRINT"Salary: ";S$
165 PRINT"■■": GOTO 112
175 CLOSE
180 END
```

When the program is running, we see:

```
FOUND STAFF
LOADING STAFF
```

The original data is then displayed on the screen.

Although this sounds simple, a word of caution:

Be sure to store your data files on a different cassette from your program files, since the computer does not check to see if the cassette already contains data. There is a real risk of losing your data, and it is unwise to use cassette data files for important applications. Let us now look at the greater number of facilities offered on a disk system, but still retaining the simplicity of sequential files.

7.2 Sequential Disk Files

As a bare minimum, the user of any sequential file needs to be able to:

set up a new file, and put records into it
append records to an old file
delete records from a file
search for a particular record
list a file

With a disk system, these operations can be implemented easily. We will deal with each of these topics, pointing out the disk BASIC commands needed, and then will demonstrate a complete program.

Setting up a new file

To add records to a file which does not yet exist, the sequence is simple:

1) Open a named file for output
e.g. 20 WOPEN#1, "NEWFILE"
or, if the file drive has been selected with the POKE method of selection
e.g. 100 WOPEN#4,F$
where F$ is a string variable.
The integer following # is a buffer number; as with cassette files, a buffer is a small area of temporary storage. Normally, one buffer is associated with each file, and data is initially transferred to the specified buffer (up to 10 are allowed). The transfer of data from the buffer area to the physical file only happens after a CLOSE or when the buffer is full.

2) Write the data records to the buffer:

PRINT#n,(list of variables)

For example, if each record on the file consists of one numeric variable and one string variable, and if we have already WOPENed a file with buffer number 3, we could write:

50 PRINT#3,V1,B$

The disk drive will remain stationary because the data is being written to a buffer, not to the file.

3) Close the file:
line-number CLOSE [#buffer-number]
The disk drive will start, and the contents of

the buffer are transferred to the file associated with the buffer.

e.g. 50 CLOSE#2

If no buffer is specified, all files are closed. Until a CLOSE is executed, any new file in a preceding WOPEN will not appear on the directory. After a CLOSE, any new file is put onto the directory and any data in a buffer is physically written into that file. Finally, an end-of-file (EOF) marker is placed at the end of the file.

Appending records to an old file

You might imagine that this would enable you to add records to the end of an already-existing file:

1) WOPEN the file
2) Find the end of it
3) Write the new records at the end
4) Close it

Unfortunately, step 2 requires us to read data from the file, but step 1 has only enabled us to write data to it. To read data from a file, you need the ROPEN statement, as in:

```
200 ROPEN#1,"NEWFILE"
```

But step 3 requires the file to be WOPEN! Since a file can not be simultaneously ROPEN and WOPEN, a different solution is needed. In terms of a design, here is one possible solution:

1) Open the file for reading, using ROPEN:
```
100 ROPEN#1,F1$
```

2) Open a temporary file for writing:
```
110 WOPEN#2,F2$
```

3 Copy the file, (in this case F1$) into the temporary file, F2$. Do this by reading from F1$ until the end-of-file marker is detected with the EOF statement. Note that the type of variable (string or numeric) in an INPUT statement must correspond to the type of data on the file. In this example, a file was originally written with one numeric and one string variable per record.

This loop will carry out the copying:

```
110 ROPEN#1,F1$
110 WOPEN#2,F2$
```

```
120 INPUT#1,A,B$
130 IF EOF(#1) THEN 160
140 PRINT#2,A,B$
150 GO TO 120
160 REM.....CONTINUE
```

4) Since the temporary file is WOPEN, write the new records to this file:

5) CLOSE both files.

6) Delete F1$

7) Rename F2$ as F1$

So, the temporary file F2$ has now replaced the original file F1$. It contains the original data plus the new records.

Searching for a record

Simply open the file for reading:

```
100 ROPEN#1,F1$
```

and examine each record with a suitable algorithm. In the simplest case, this will be an examination of a numeric key; when the search identifies the key , stop, since there is a unique answer. If the records are not unique (e.g. several with the same numeric key), continue searching. In either case, keep checking for the end-of-file marker as no error condition arises when input proceeds past the end of the file.

Deleting a record

To save space on the file, unwanted records must be deleted. You can do so by a procedure like this:

```
input search-key
repeat
  input (from main file)
  call check-record (search-key,hit)
  //hit="true" if record contains search-key //
  if hit=false then print (to temp-file) end if
until end-of-file
```

The subroutine **check-record** simply finds if a record

contains a particular piece of information, called the "search key". The temporary file now contains only the required records. This file is re-named as the main file, exactly as we did when adding records to a file.

Listing a file

After all of the complications of addition and deletion, this is simple! You just ROPEN the file, read the records and print them. You only have to decide if the screen or printer are to be used. And, of course, you must check for the end-of-file marker.

7.2.1 Filing your Holiday Photographs: an example application

For some time, I have tried to organise my large collection of colour slides. Now, with the MZ-80K, I seem to have solved it! Every slide can be given a one-line description plus a unique identification number, such as:

The children on the beach [1980/49]

The reference number tells me that the slide is number 49 in the 1980 box.

Here is a program which stores such records in a sequential file, and enables me to add new items, delete old ones, find a particular title, or list the whole lot. There is an exact correspondence to the various steps described in detail above. Some particular points are:

Lines 100-267: control module to select subroutines (see section 4.1). On returning from a subroutine, line 266 "freezes" the screen until a key is pressed. Lines 101-105 select the file and disk drives to be used. So, you are not limited to a single file. If one disk contains both the program and data files, the POKE can be omitted.

Lines 290-520: module to add items to a new or existing file. If it is a new file, line 355 will cause error number 40 (non existing file). Line 330 detects this condition, causing lines 360 to 390 (which copy the main file to a temporary file) to be skipped. Also, a variable F is changed from 2 to 1, so that the new records are directed immediately to the newly-formed main file.

Finally, if an old file is being updated, subroutine 8000 is called to rename the temporary file.

Lines 600-760: deletion module; this uses subroutine 5000, which is the INSTRING routine of section 2.3.2 to examine each record for a keyword. Error detection is used in case of, for example, a user trying to delete from an empty file.

The rest of the program is very straightforward.

PROGRAM NAME: SEQ-DEMO

```
100 PRINT"■";"■■■  FILE MAINTENANCE PROGRAM  ■■■"
101 INPUT"Which disk drive? ";DF
102 INPUT"Which file? ";F$
103 T$="TEMP-FILE"
104 REM- SELECT DATA FILE DRIVE.
105 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
106 PRINT"■"
108 CURSOR 0,10
110 PRINT"Select 1 to add new items"
120 PRINT"       2 to delete"
130 PRINT"       3 to search"
140 PRINT"       4 to list"
150 PRINT"       5 to end"
160 CURSOR 0,15: PRINT"Option? ";
180 GET A$: IF A$="" THEN 180
190 PRINT A$;
200 M$="12345"
210 FOR I=1 TO LEN(M$)
220 IF A$=MID$(M$,I,1) THEN 260
230 NEXT I
240 PRINT"■ ";: GOTO 160
260 ON I GOSUB 300,600,900,1200,9999
265 PRINT"■■Press any key to return to the menu:"
266 GET A$: IF A$="" THEN 266
267 GOTO 106
290 REM**********************************
300 REM- APPEND SUB-MODULE
310 PRINT"■";"■■▒▒▒▒ ADDITION MODULE ▒▒▒▒■■"
330 ON ERROR GOTO 400
350 WOPEN #2,T$
355 ROPEN #1,F$
360 INPUT #1,R$
370 IF EOF(#1) THEN 420
380 PRINT #2,R$
390 GOTO 360
400 PRINT"■■File ";F$;" is currently empty■■"
410 CLOSE #1: WOPEN #1,F$: REM- register F$
420 PRINT"■■New item information."
422 PRINT"                        "
425 PRINT"■Description: "
430 INPUT D$
435 PRINT"■Reference: "
440 INPUT P$
450 R$=D$+"  <REF: "+P$+">"
470 PRINT #2,R$
480 PRINT"■■Any more (Y/N)?";
490 GET A$: IF A$="" THEN 490
492 PRINT A$
```

```
494 IF A$="Y" THEN 420
496 IF A$="N" THEN 500
498 GOTO 480
500 CLOSE
502 REM-COPY BACK
505 GOSUB 8000
520 RETURN
550 REM**********************************
600 REM- DELETION SUB-MODULE
601 PRINT"▣";"▦▦▒▒▒▒ DELETION MODULE ▒▒▒▒▦▦"
605 ON ERROR GOTO 760
640 ROPEN #1,F$
650 WOPEN #2,T$
655 INPUT"Keyword or phrase: ";K$
660 INPUT #1,R$
670 IF EOF(#1) THEN 740
675 GOSUB 5000
680 IF AB=0 THEN 720
690 PRINT"▦▦▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬"
695 PRINT R$,"▦▦"
700 PRINT "Is this to be deleted (Y/N)? ";
705 GET A$: IF A$="" THEN 705
710 PRINT A$: IF A$="Y" THEN PRINT"▦▦Item deleted▦▦": GOTO 730
720 PRINT #2,R$
730 GOTO 660
740 REM- COPY BACK
745 PRINT"▦▦Reached end of file."
747 CLOSE
750 GOSUB 8000
760 RETURN
800 REM**********************************
900 REM- SEARCH MODULE
905 PRINT"▣";"▦▦▒▒▒▒ SEARCH MODULE ▒▒▒▒▦▦"
910 ROPEN #1,F$
920 INPUT"Keyword or search phrase: ";K$
935 ON ERROR GOTO 1140
960 INPUT #1,R$
970 IF EOF(#1) THEN 1130
980 GOSUB 5000
990 IF AB=0 THEN 960
1000 PRINT"▦▦▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬"
1010 PRINT R$,"▦▦"
1100 PRINT"Press a key to continue ..."
1110 GET A$: IF A$="" THEN 1110
1120 GOTO 960
1130 PRINT"▦▦▒▒▒▒▒▒ END OF FILE ▒▒▒▒▒▒▦▦"
1140 CLOSE
1150 RETURN
1160 REM**********************************
1200 REM-LIST THE FILE
1201 PRINT"▣";"▦▦▒▒▒▒ LIST MODULE ▒▒▒▒▦▦"
1202 ON ERROR GOTO 1410
1208 ROPEN #1,F$
1210 PRINT"Output to printer (Y/N)?";
1220 GET A$: IF A$="" THEN 1220
1230 PRINT A$
1240 P=0: IF A$="Y" THEN P=1
1260 INPUT #1,R$
1270 IF EOF(#1) THEN 1400
1280 PRINT R$: IF P=1 THEN PRINT/P R$
```

```
1290 PRINT
1300 GOTO1260
1400 CLOSE
1410 RETURN
1450 REM*********************************
5000 REM- INSTRING FUNCTION
5010 A=LEN(R$): B=LEN(K$)
5020 AB=0
5030 FOR I=1 TO A-B+1
5040 IF MID$(R$,I,B)=K$ THEN 5050
5045 GOTO 5060
5050 AB=I: GOTO 5070
5060 NEXT I
5070 RETURN
6000 REM*********************************
8000 REM- FILE CHANGER
8005 PRINT"Changing files .... please wait ...."
8100 DELETE F$
8200 RENAME T$,F$
9000 RETURN
9999 END
```

Here are some example uses of the program.

The original file:

```
A RAINY DAY IN MANCHESTER  <REF: 1981/98>
OUR SUMMER HOLIDAY IN THE LAKE DISTRICT  <REF: 1982/55>
THE CHILDREN ON THE BEACH  <REF: 1980/49>
A PICTURE OF OUR NEW HOUSE  <REF: 1981/56>
OUR NEW ARRIVAL  <REF: 1979/34>
A VIEW OF EDINBURGH  <REF: 1981/72>
```

Searching for a record containing "HOUSE":

```
▒▒▒▒ SEARCH MODULE ▒▒▒▒

Keyword or search phrase: HOUSE

A PICTURE OF OUR NEW HOUSE  <REF: 1981/5
6>

Press a key to continue ...
```

Deleting the first record to contain 1980 in its reference:

```
▒▒▒▒ DELETION MODULE ▒▒▒▒

Keyword or phrase: 1980

THE CHILDREN ON THE BEACH  <REF: 1980/49
>

Is this to be deleted (Y/N)? Y

Item deleted
```

You could make several changes to this program:

1) If your file is small enough, read the records into a string array such as R$(100). Storage will quickly be used, but response times will be improved.

2) If many records have to be deleted, include a bulk erase routine. To do this, add an extra "flag" character at the beginning of a record:

flag data

You can set "flag" to 0 for a normal record, or to 1 if the record is to be deleted. Your program could then whisk through the file, deleting all records with "1" as the first character.

7.3 Keyed Sequential Files

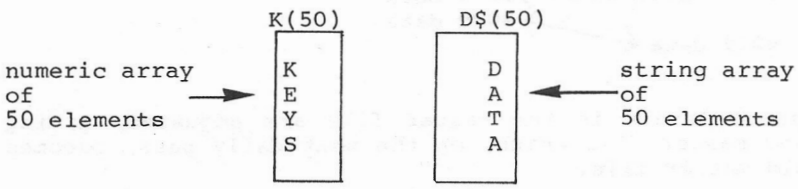
In the preceding section the order of the records was unimportant. But, in many applications, the records must be in strict ascending or descending order as dictated, for example, by a part number. This can make life very much easier when confronted with a question such as:

"List the products with part numbers between 5001 and 5420"

You could use an unordered file, but you would have to make 420 separate queries! If the file were ordered, you would simply specify the range of part-numbers:

```
input start,end  //e.g. 5001 and 5420//
repeat
      read record
until part-number>=start
      print record
        repeat
          read record; print record
        until part-number>=end or end-of-file
```

An ordered file of this type, with associated numeric keys, is called a keyed sequential file. Subject to storage limitations, it can be read into two or more arrays, representing a "key-table", as shown below:



To find a particular record, you can either:

1) Scan through the array K( ) until you find the desired key.

Or 2) Use a bisection method: read the middle key; if this is less than the desired key read the middle key of the upper half, and so on.

Sorting the file

It is relatively easy to construct a master file with the keys, such as part numbers, in ascending order. But, imagine that a company is despatching and receiving parts with these numbers:

part number	quantity	date
1016	25	6/9/82
4029	-4	7/9/82
3071	-82	7/9/82
1782	73	8/9/82

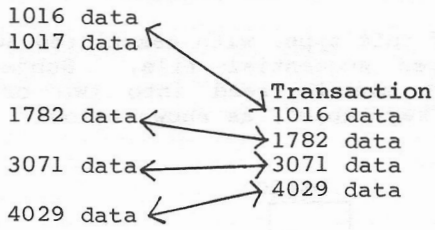
If this data is recorded onto a file, it will obviously be jumbled in terms of part numbers. Such a file is called a transaction file, and it is much more useful if it is sorted into the same order as the master file, which, in

this example, records the total stock situation:

Old Master File
Daily Transaction File } merge→ New Master File

The "merge" operation boils down to matching the ordered Master and transaction files against each other:

Master



The stock levels in the master file are adjusted, giving the new master file which, on the next daily pass, becomes the old master file.

There are many ways of sorting the transaction file. First, let us assume that the volume of data is such that the important data in the transaction file can be read into a key table in RAM. Arrays are limited on the MZ-80K to a maximum length of 256, so this is a slightly limiting factor. Problems of string storage can be eliminated by only storing numeric data; the text description can remain on file.

Therefore, all we have to do is sort the key table, giving us the sorted transaction file. It is only necessary to sort the keys themselves, so long as the algorithm will remember which data are associated with which keys. This is called a "tag" sort, since each key is tied to a particular piece of data. There are many forms of tag sort, but a simple one is based on the well-known "bubble" sort:

```
procedure bubble-tag//for an array "key" of length n//
   j:=n
      repeat
        mk:=0
        for i:=1 until j-1
        do
            if key (tag(i))>key(tag(i+1))
            then mk:=1 ; copy:=tag(i)
              tag(i):=tag(i+1) ; tag(i+1):=copy
            j:=j-1
```

```
      end if
    end do
    until mk=0 or j=2
```

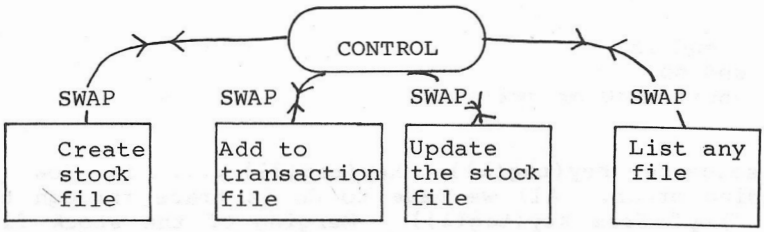
The sequence key(tag(1)), key(tag(2)),.... is now in ascending order. All we have to do is trace through the array "key" from key(tag(1)). Merging of the stock file with the sorted transaction is straightforward, but it needs careful control over the sequence of processing the two sets of data. In the following program design, two pointers are used: i refers to the number of a record in "stock-file" and j is the index of the tag, i.e. tag(j), which locates the correct transaction datum. It can be thought of as a pointer to the j'th record in a sorted transaction file:

```
procedure
i:=1 ; read stock-key // from file "stock"//
j:=1 ; read trans-key // from file "trans"//
while not end-of "stock"
  do
  if not end-of "trans"
  then if trans-key>stock-key
         then output stock-record //unchanged//
            i:=i+1 ; input stock-record
  else
       if trans-key = stock-key
       then update stock-record ; output stock record
                 j:=j+1 ; input trans record
       else print "error in part number"
       end if
  end if
end do
```

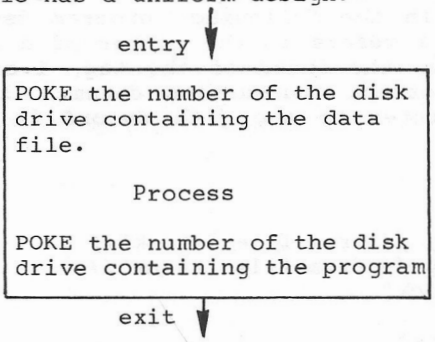
7.3.1 Stock Control: An example of a keyed sequential file

The above techniques will now be illustrated in a simple stock control program. This is NOT intended for a real commercial application, but it does illustrate a number of features. An important aspect is that it uses the SWAP command so that only the bare minimum of BASIC program is in RAM at any one time. This maximises the possible size of the key-table and gives a realistic application of SWAP. Since this command involves disk accesses, it does slow the response time and you may wish to re-design the program without SWAPping, so long as the key-table size is carefully tailored.

The program consists of a control module and four process modules:



Each module has a uniform design:



The POKEs are essential if, as is likely, the data files (stock master file, transaction file) are on a different disk from the one containing the program.

Each record is of the form:
<Part number, description, quantity, date>

Whenever the transaction file refers to a particular part, the date is also changed to show when a change was made.

The various modules are listed below with brief descriptions:

The Control module (STOCK-MENU)

This presents a menu of choices, and routes the user to the correct program. Line 140 selects the disk drive containing the program files:

```
100 REM- MENU FOR STOCK CONTROL
105 PRINT"■"
110 REM- DP IS DRIVE NO. FOR 'FILES' DISK
115 DF=1:DP=2
117 REM- NT IS MAX. NO. OF TRANSACTIONS
118 NT=200
120 F1$="STOCK": F2$="TRANS": F4$="SCRATCH"
135 CURSOR 0,5
140 POKE $49AC,DP-1:POKE $49AB,0:POKE $62BC,DP-1
150 PRINT"1. CHANGE THE STOCK FILE"
160 PRINT"2. CHANGE THE TRANSACTION FILE"
170 PRINT"3. UPDATE THE STOCK FILE"
```

```
180 PRINT"4. EXAMINE A FILE
190 PRINT"5. STOP
195 CURSOR 0,15
200 PRINT "Choose a number (1 to 5): ";
280 GET A$: IF A$="" THEN 280
290 PRINT A$;
300 M$="12345"
310 FOR I=1 TO 5
320 IF A$=MID$(M$,I,1) THEN 350
330 NEXT I
340 PRINT"■"+CHR$(32); : GOTO 195
350 ON I GOTO 370,380,390,400,500
370 SWAP"STOCK-CREATE":GOTO 450
380 SWAP"TRAN-CREATE": GOTO 450
390 SWAP"UPDATE": GOTO 450
400 SWAP"FLIST"
450 PRINT"PRESS ANY KEY TO DISPLAY MENU";
460 GET A$: IF A$="" THEN 460
470 GOTO 105
500 END
```

STOCK and TRANS are the stock and transaction files. SCRATCH is a file used for temporary storage.

File lister

This examines a file, either for a particular entry, or for its entire contents. Lines 142-146 search for a matching key and, because of the variable lengths of the files in the records, blanks are added (line 144) so that the records are listed as neat columns, to fit onto a 40 character screen:

```
100 REM- LIST ANY FILE
101 POKE $49AC,DF-1: POKE $49AB,0: POKE $62BC,DF-1
102 REM- SET UP BLANK FORMAT STRING
103 B$="                    ": LB=10
104 REM- ND IS THE NO. OF FIELDS
106 ND=4
108 ON ERROR GOTO 200
110 PRINT"■"
120 INPUT"Name of file or 'END'";F$
125 IF F$="END" THEN 300
126 PRINT"Output to printer (Y/N)?";
127 GET A$: IF A$="" THEN 127
128 IF (A$="Y")+(A$="N") THEN 130
129 GOTO 127
130 PRINT A$
131 INPUT"Keyword (or * for all) ";S$
135 ROPEN #1,F$
140 R$="": F=0
141 FOR I=1 TO ND: INPUT #1,A1$
142 A1$=LEFT$(A1$,LB): A2$=RIGHT$(A1$,LEN(S$))
143 IF (A2$=S$)+(S$="*") THEN F=1
144 LA=LEN(A1$): R$=R$+A1$+RIGHT$(B$,LB-LA)
146 NEXT I
```

```
150 IF EOF(#1) THEN 190
155 IF A$="Y" THEN 168
160 IF F=1 THEN PRINT R$
165 GOTO 170
168 IF F=1 THEN PRINT/P R$
170 GOTO 140
190 PRINT:  PRINT"%%% END OF FILE %%%": PRINT
195 CLOSE
197 GOTO 210
200 PRINT"ERROR IN FILE NAME"
204 REM- NOTE THAT 'KILL' NOT 'CLOSE' IS USED
205 KILL #1
210 GOTO 120
300 POKE $49AC,DP-1:POKE $49AB,0:POKE $62BC,DP-1
310 END
```

If an invalid file name is selected, the error detection statement in line 108 transfers control to lines 200-210 in which the KILL cancels the current ROPEN statement.

A typical file listing is shown below:

```
Name of file or 'END'STOCK
Output to printer (Y/N)?N
Keyword (or * for all) *
 210      LEICA      5        18/1/82

 221      PENTAX-ME  20       18/1/82

 225      KONICA FC- 10       21/2/82

 232      FUJICA AX1 16       25/2/82

 241      MAMIYA-ZE  10       18/1/82

 250      CANON AE-1 15       18/1/82

 255      NIKON EM   25       18/1/82

 270      MINOLTA XD 4        20/1/82

%%% END OF FILE %%%
```

Stock Creation module. Program name: (STOCK-CREATE)

This is very similar to the unkeyed file maintenance program presented previously. If you have not already done so, you should read its description and compare it with this program. It contains submodules for:

1) **Appending new items (lines 300 to 520)**
   If the stock file is not empty, this checks that new items are typed in sequence, thereby ensuring that the stock file is in ascending order.

2) **Deletion (lines 600 to 760)**
   the user chooses an item for deletion by its part number. If the item is present, its description is presented, and the user **must** verify that it is to be deleted.

3) **Insertion (lines 900 to 1120)**
   New items can be inserted with part numbers before existing numbers. The insertion date is requested, so that the user can see when additions were made. The insertion mode is intended for the occasional new item, whereas appending (step 1) is intended for a batch of new items.

Note that numeric input is never used in this program; instead, character strings are input and, if they should be numeric, they are converted with the VAL function. A resultant value of zero or less will always be unacceptable, and such an error is queried. (See, for example, lines 425 and 426):

(Details of program start here)

```
50 REM- STOCK CREATE MODULE
90 REM- SET MAX PART NUMBER
100 PM=9999
102 PRINT"■";"STOCK FILE MODULE"
104 REM- SELECT DATA FILE DRIVE.
105 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
108 CURSOR 0,10
110 PRINT"Select 1 to append"
120 PRINT"       2 to delete"
130 PRINT"       3 to insert"
140 PRINT"       4 to finish"
150 PRINT
160 CURSOR 0,15: PRINT"Option? ";
180 GET A$: IF A$="" THEN 180
190 PRINT A$;"■■"
200 M$="1234"
210 FOR I=1 TO 4
220 IF A$=MID$(M$,I,1) THEN 260
230 NEXT I
240 PRINT"■ ";: GOTO 160
260 ON I GOSUB 300,600,900,2000
270 GOTO 102
300 REM- APPEND SUB-MODULE
320 P1=0
330 ON ERROR GOTO 400
350 WOPEN #2,F4$
355 ROPEN #1,F1$
```

```
360 INPUT #1,P,D$,Q,T$
370 IF EOF(#1) THEN 410
375 P1=P
380 PRINT #2,P,D$,Q,T$
390 GOTO 360
400 PRINT"The Stock File is Currently empty"
405 GOTO 420
410 PRINT"The Current highest part no. is ";P1
420 INPUT"Please type today's date: ";T$
425 INPUT"Part number (9999 to stop) ";P$
426 P=VAL(P$): IF P<=0 THEN 425
427 IF P>PM THEN 429
428 GOTO 430
429 PRINT"MAX. PART NUMBER IS ";PM: GOTO 425
430 IF P<>9999 THEN 435
432 GOTO 500
435 IF P>P1 THEN 450
440 PRINT"OUT OF SEQUENCE": GOTO 425
450 INPUT"Description ";D$
460 INPUT"Quantity ";Q$
462 Q=VAL(Q$): IF Q<=0 THEN 460
470 PRINT #2,P,D$,Q,T$
480 P1=P
490 GOTO 425
500 CLOSE
502 REM-COPY BACK
505 GOSUB 8000
520 RETURN
600 REM- DELETION SUB-MODULE
605 ON ERROR GOTO 760
610 INPUT"■■Number for deletion (9999 to stop)";P$
612 P=VAL(P$): IF P<=0 THEN 610
620 IF P<>9999 THEN 640
630 GOTO 760
640 ROPEN #1,F1$
650 WOPEN #2,F4$
660 INPUT #1,P1,D$,Q,T$
670 IF EOF(#1) THEN 740
680 IF P<>P1 THEN 720
690 PRINT"Description: ";D$
700 INPUT"Is this to be deleted? ";A$
705 AL$=LEFT$(A$,1)
710 IF AL$="Y" THEN 730
720 PRINT #2,P1,D$,Q,T$
730 GOTO 660
740 CLOSE
745 PRINT"End of stock file"
750 GOSUB 8000
760 RETURN
900 REM-INSERT SUB-MODULE
905 PRINT
910 INPUT"New part number, or 9999 to stop";P$
912 P=VAL(P$): IF P<=0 THEN 910
915 IF P>PM THEN 917
916 GOTO 920
917 PRINT"MAX. PART NUMBER IS ";PM: GOTO 910
```

```
920 IF P<>9999 THEN 940
930 GOTO 1120
935 ON ERROR GOTO 1100
940 ROPEN #1,F1$
950 WOPEN #2,F4$
960 INPUT #1,P1,D$,Q,T$
970 IF EOF(#1) THEN 1100
980 IF P1>P THEN 1000
990 GOTO 1060
1000 INPUT"Description of new item ";D1$
1010 INPUT"Quantity ";Q$
1015 Q1=VAL(Q$): IF Q1<=0 THEN 1010
1020 INPUT"Today's date ";T1$
1030 PRINT #2,P,D1$,Q1,T1$
1040 REM- RESET P TO MAXIMUM SO REST OF FILE IS COPIED
1050 P=PM
1060 PRINT #2,P1,D$,Q,T$
1070 GOTO 960
1100 CLOSE
1105 PRINT"End of stock file"
1110 GOSUB 8000
1120 RETURN
2000 POKE $49AC,DP-1: POKE $49AB,0: POKE $62BC,DP-1
2010 END
8000 REM- renaming
8002 ON ERROR GOTO 8200
8005 PRINT"■■..... please wait .....■■"
8100 DELETE F1$
8200 RENAME F4$,F1$
9000 RETURN
```

The initial menu is presented as follows:

```
STOCK FILE MODULE



Select 1 to append
       2 to delete
       3 to insert
       4 to finish

Option? 4
```

You would use this module to produce the initial file of stock items. We have used STOCK as the file name of F1$. An example file was printed out in the description of the previous module. You now need to produce a transaction file to show the quantity of each item bought or sold during some period of trading. The production of this is

described below.

The Transaction Module (program name: TRANS-CREATE)

This accepts the transactions in any order, and stores them in file F2$. The name TRANS was chosen for F2$. Both F2$ and F1$ have the same record format but, for user convenience, neither the item description nor the date are input by the user. Here is a listing of the program. Notice that lines 235-242 produce a flashing cursor, since the GET function is used to respond to line 230:

```
100 REM-CREATE TRANSACTION FILE
110 PRINT"■"
120 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
130 PRINT"CREATE TRANSACTION FILE"
140 ON ERROR GOTO 550
180 WOPEN #1,F2$
190 CURSOR 0,8
200 INPUT "■■Part number (9999 to stop)";P$
205 P=VAL(P$): IF P<=0 THEN 200
210 IF P<>9999 THEN 225
220 GOTO 600
225 INPUT"Quantity ";Q$
226 Q=VAL(Q$): IF Q<=0 THEN 225
230 PRINT"Buy or Sell (B or S)?";
235 CC=1
237 IF CC=1 THEN PRINT"%■";
238 IF CC=-1 THEN PRINT" ■";
242 CC=-CC: GET A$: FOR I=1 TO 200: NEXT I: IF A$="" THEN 237
250 PRINT A$
260 IF A$="B" THEN 299
270 IF A$="S" THEN 290
280 GOTO 230
290 Q=-Q
298 REM- USE DUMMIES FOR FORMATTING
299 D$="DUMMY"
300 PRINT #1,P,D$,Q,D$
340 GOTO 200
550 KILL #1
590 CURSOR 0,15
591 PRINT"A TRANSACTION FILE ALREADY EXISTS"
592 PRINT"Do you want to erase it (Y/N)?";
593 GET A$: IF A$="" THEN 593
594 PRINT A$
595 IF A$="Y" THEN 597
596 GOTO 700
597 DELETE F2$
598 PRINT"Transaction file deleted."
599 GOTO 700
600 CLOSE
700 POKE $49AC,DP-1: POKE $49AB,0: POKE $62BC,DP-1
710 END
```

A typical dialogue with this program is as follows:

```
Part number (9999 to stop)255
Quantity 5
Buy or Sell (B or S)?B
Buy or Sell (B or S)?S


Part number (9999 to stop)221
Quantity 6
Buy or Sell (B or S)?B


Part number (9999 to stop)221
Quantity 10
Buy or Sell (B or S)?S


Part number (9999 to stop)241
Quantity 2
Buy or Sell (B or S)?S


Part number (9999 to stop)9999
```

And the resultant file looks like this:

```
Name of file or 'END'TRANS
Output to printer (Y/N)?N
Keyword (or * for all)
? *
 255       DUMMY       -5          DUMMY

 221       DUMMY        6          DUMMY

 221       DUMMY      -10          DUMMY

 241       DUMMY       -2          DUMMY

%%% END OF FILE %%%
```

Stock File Update

This requires the sorted transaction file to be merged with the current stock file. The sorting is performed in lines 160 to 330 using the tag sort previously described. The part numbers (keys) are stored in the first column of the array A(NT,2) ; the second column of the array stores the transaction quantities. The design previously presented is followed faithfully:

```
(
  100 REM-UPDATE & SORT MODULE
  105 PRINT"■";"       SORT & UPDATE STOCK FILE"
  107 INPUT"■■ Please type today's date ";T1$
  112 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
  115 DIM O(NT),A(NT,2)
  120 ON ERROR GOTO 8000
  130 WOPEN #2,F4$
  140 ROPEN #1,F2$: ROPEN #3,F1$
  150 NF=0
  155 PRINT"■■Sorting transaction file...■■"
  160 INPUT #1,P,D$,Q,D$
  170 IF EOF(#1) THEN 190
  175 NF=NF+1: A(NF,1)=P: A(NF,2)=Q
  180 GOTO 160
  190 REM- SORT USING A(I,1)
  200 FOR I=1 TO NF: O(I)=I: NEXT I
  210 J=NF
  220 M=0
  230 FOR I=1 TO J-1
  240 IF A(O(I),1)>A(O(I+1),1) THEN 280
  250 GOTO 300
  280 C=O(I): O(I)=O(I+1): O(I+1)=C: M=1
  300 NEXT I
  310 IF M=0 THEN 340
  320 IF J=2 THEN 340
  330 J=J-1: GOTO 220
  340 REM- LIST NOW SORTED
  400 REM- UPDATE THE FILE. F1 & F2 ARE FLAGS
  402 PRINT"■■Merging with stock file...■■"
  405 T=0
  408 GOSUB 1000
  410 GOSUB 2000
  412 IF F2=1 THEN 640
  415 REM- CHECK FOR END OF TRANSACTIONS LIST.
  420 IF F1=1 THEN 450
  430 IF A(O(T),1)>P THEN 450
  440 GOTO 490
  450 PRINT #2,P,D$,Q,T$
  460 GOSUB 2000
  480 GOTO 600
  490 IF A(O(T),1)=P THEN 510
  500 GOTO 550
  510 Q=Q+A(O(T),2): T$=T1$
  520 GOSUB 1000
  530 GOTO 600
  540 REM- IF TESTS FAIL, P CAN NOT BE IN FILE
  550 PRINT"Error in transaction file."
  555 PRINT"Part no. ";A(O(T),1);" invalid."
  560 GOSUB 1000
  600 GOTO 412
  640 REM- FILE COPIER
  645 PRINT"■■.....please wait....■■"
  650 CLOSE
  660 DELETE F1$
  665 RENAME F4$,F1$
  730 GOTO 9000
```

```
1000 REM- INCREMENT TRANSACTION POINTER
1010 T=T+1
1020 F1=0
1030 IF T>NF THEN F1=1
1040 RETURN
2000 REM- INCREMENT STOCK RECORD
2010 F2=0
2020 INPUT #3,P,D$,Q,T$
2030 IF EOF(#3) THEN F2=1
2040 RETURN
8000 PRINT" *** UPDATE ABANDONED ***"
8010 PRINT"* Check all files are OK *"
8020 PRINT "ERROR ";ERN;" IN LINE ";ERL
8900 CLOSE
9000 POKE $49AC,DP-1:POKE $49AB,0: POKE $62BC,DP-1
9010 END
```

Whilst the program is running, messages are displayed to tell you that the transaction file is being sorted, that merging is occurring and, finally, that the temporary file F4$ is being renamed as F1$. This now contains the updated stock records. Any item affected by this update has a new date allocated to it. So, by examining printed copies of the stock file, you can see at a glance which items are moving quickly. Also, you will have accurate reports of current stock levels. This would be useful if our hypothetical camera shop had a number of branches, with stock records maintained on the "head office" computer.

If you wish to use this as the basis for a real-life stock control program, the following improvements could be included:

1) Use a screen handler routine (see section 4.3.2) to input data.

2) Make improvements on the data-vetting - in particular, ensure that pre-set ranges are not exceeded. For example, you must not be able to buy more than say, 100 cameras without the system querying it.

3) Stock values could be included, so that the listing of a file is accompanied by a statement of the present total stock value.

8. DIRECT ACCESS FILES

Direct access enables you to select any record equally quickly, wherever it is located in the file. This is in contrast with sequential files, when each record must be processed until the required one is found. In this chapter, we survey some of the methods of file handling that require direct access. Each of these is concerned with a balance of 2 factors:

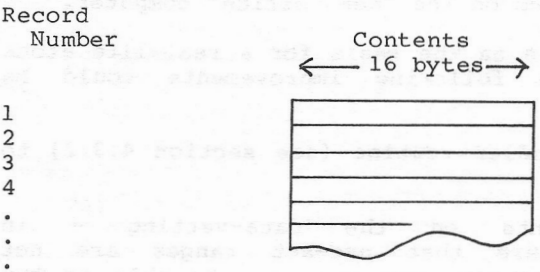
Speed of response

Ease of amendment

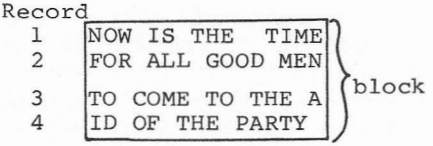
In each case, these factors will be compared. But first of all, we examine how direct access files are stored, and what BASIC commands are needed to manipulate them.

8.1 File Organisation

On the MZ-80K, records in a direct access file are limited to a maximum length of 16 bytes. Since one byte represents one character, this means that only 16 characters can be stored in in one record. Alternatively, one numerical data item can be recorded:



On many other microcomputers, the record length is greater. For example, the TRS-80 has a fixed record length of 256 bytes. But this is often a disadvantage, as it can lead to wasted space if, for example, the records are only filled with 10 byte strings. The shorter record length of the MZ-80K leads to more efficient disk storage. Longer strings are easily accommodated simply by joining together the requisite number of records. For convenience, such a sequence will be called a "block". For example, the string below can fit into into a block of 4 records:



In order to produce a file that contains information of this type, the following sequence is needed:

1) **Open the file**

```
XOPEN #n,[FDd@V,]"filename"
```

where n is a buffer number, and the other parameters are as described in the previous chapters. For example,

```
100 XOPEN #1,FD2,"RANDOM"
```

This associates file RANDOM with buffer number 1.

2) **Print the records**

```
PRINT #n,(expression),list of variables
```

The first variable will be written to the record pointed at by the value of the expression; the second variable goes into the next record and so on. For example:

```
200 PRINT #1,(20),A$,B$
```

This will place the contents of string A$ into record 20, and of B$ into record 21. Similarly,

```
230 X=10
240 PRINT #2,(X+1),"NOW","AND AGAIN"
```

will store "NOW" in record 11 and "AND AGAIN" in record 12.

3. **Close the file**

In an identical manner to sequential files we type

```
CLOSE[#n]
```

where #n is optional, and refers to a particular buffer. This causes any buffer(s) to be emptied and for the file name to be registered on the appropriate directory. A direct access file is identified by "BRD" in the directory listing.

To read from a direct access file, a similar sequence is followed:

1) **Open the file**
Use XOPEN, exactly as above

2) **Read a record**
INPUT #n(expression),list of variables

For example:

```
300 INPUT #1(10),A$
```

This inputs the contents of record 10, in the file associated with buffer 1, into A$. Similarly,

```
305 X=5
310 INPUT #1(X),A$,B$,C$
```

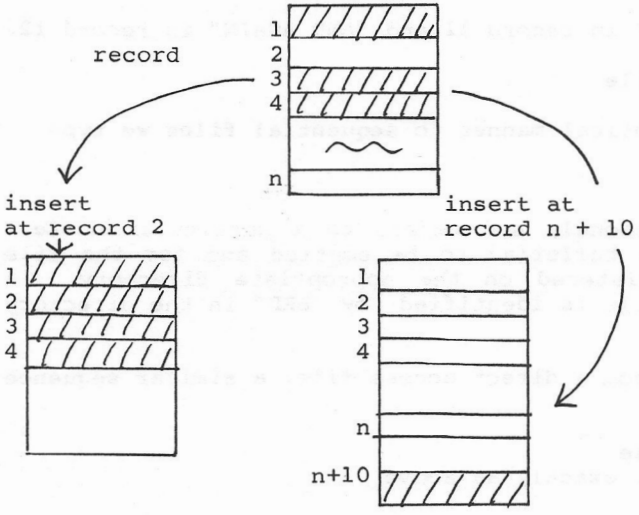
This inputs the contents of records 5, 6 and 7 into A$, B$ and C$.

You can continue to INPUT from an XOPENed file. Also, you can freely mix the file INPUT and PRINT statements - you do not have to close the file as you did with a sequential file. There are, however, some points to watch out for:

1) The file name is not registered on the directory until a CLOSE is executed.

2) The types of variables must correspond in both INPUT and PRINT. You can not read a record containing string data into a numeric variable.

3) There is no end-of-file marker. So, you can not detect if you are attempting to input from beyond the end of the file. In fact, new records can be inserted both into the existing range of records, or beyond the highest numbered one:



In this diagram, filled records are shaded. As you can see, insertion at (n+10) simply extends the file.

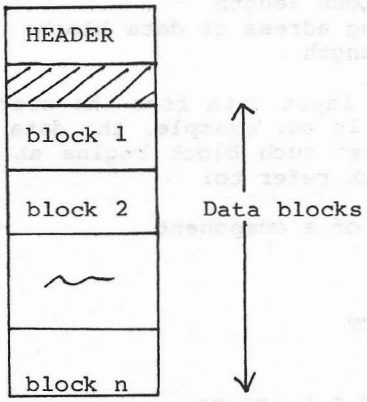
8.2 Simple Access Methods

We can not expect a user to know the exact location of each record in a direct access file. Also, it is likely that "blocks" of records will be used, so the retrieval of data from a file is not straightforward.

The first step in setting up an easily-useable file is to construct a **header block**. This is a block of records that describes the file; it will normally be read at the beginning of any direct-access program to make subsequent file-handling easier. This is an example header:

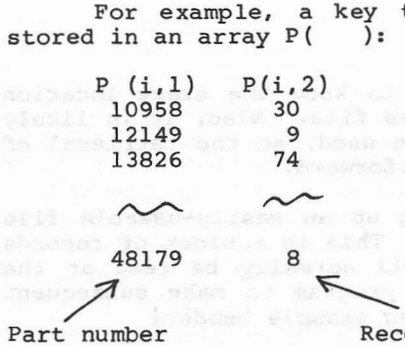
records			
	1	7	length of header
	2	PAYROLL	file description
	3	1000	maximum length (in blocks)
	4	250	actual length
	5	4	records per block
	6	29/1/82	date when last changed
	7	10	first data record

Whilst great flexibility is possible in header design, the first record is useful if the header is more than 1 record in length. The last record tells us where the data record is located, if it is not immediately after the header:



The easiest method of direct access file maintenance for small files (up to say, 200 items) is by the use of a key table. This is a simple device to convert user-oriented data (e.g. a name or part number) into a block or record number. The key table is usually small enough to reside in an array in main memory, though it will be stored in permanent form as a file. Compare this with the use of key tables for sequential files, section 7.3.

For example, a key table of part numbers might be stored in an array P( ):



So, given a part number, we simply scan through P(i,1) until we find it and, immediately, P(i,2) tells us which record in the main file to inspect. This record might contain the quantity in stock. If more information than this was needed, P(i,2) would point to a block of records.

We can easily write a program to use a key table for retrieval, but firstly we must construct the direct access file. A short program will suffice wich simply puts new data blocks at the end of the current file, as required. Here is the program description:

In lines 1000 to 1080 a file header is created of the form:

record 1 : file description
2 : data block length
3 : starting adress of data blocks
4 : file length

Lines 1500 to 1670 take input data from the user and write it to the data blocks. In our example, the data blocks are of length 3 and the first such block begins at record 10. The records in each block refer to:

(1) description of a component
(2) part number
(3) quantity

Program Name: MAKE-DIRECT

```
100 REM-CONTROL BLOCK FOR D.A. FILES
120 PRINT"■"
130 INPUT"Name of file ";F$
140 INPUT"Number of disk drive ";DF
150 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
190 CURSOR 0,6
200 PRINT"1: Initialise the file"
210 PRINT"2: Add to it"
220 PRINT"3: List it"
230 PRINT"4: Stop"
240 CURSOR 0,12
250 PRINT"Choose a number ";
```

```
260 GET A$: IF A$="" THEN 260
270 PRINT A$;
280 S$="1234"
290 FOR I=1 TO LEN(S$)
300 IF A$=MID$(S$,I,1) THEN 330
310 NEXT I
320 PRINT"■ ": GOTO 240
330 ON I GOSUB 1000,1500,1700,2000
400 PRINT"Press any key to continue"
410 GET A$: IF A$="" THEN 410
420 PRINT"■"
430 GOTO 190
500 REM**********************************
1000 REM-INITIALISE
1010 PRINT"■";"INITIALISATION";"■■■■"
1015 INPUT"File description: ";DF$
1020 INPUT"Block length: ";BL$
1025 BL=VAL(BL$): IF BL<1 THEN 1020
1030 INPUT"Starting address: ";SA$
1035 SA=VAL(SA$): IF SA<5 THEN 1035
1037 XOPEN #1,F$
1040 PRINT #1(1),DF$,BL,SA,0
1070 CLOSE
1080 RETURN
1100 REM**********************************
1500 REM- ADD TO FILE
1510 PRINT"■";"ADD MODULE";"■■■■"
1520 XOPEN #1,F$
1530 INPUT #1(1),DF$,BL,SA,FL
1540 PRINT"Addition?";
1560 GET A$: IF A$="" THEN 1560
1570 PRINT A$
1580 IF A$="N" THEN 1650
1585 FL=FL+1
1590 FOR I=1 TO BL
1600 PRINT"Line ";I;
1610 INPUT A$
1620 PRINT #1(SA+(FL-1)*BL+I-1),A$
1630 NEXT I
1640 GOTO 1540
1650 PRINT #1(4),FL
1660 CLOSE
1670 RETURN
1690 REM**********************************
1700 REM-LIST A FILE
1710 PRINT"■";"LISTING OF FILE ´";F$;"´■■■■"
1720 XOPEN #1,F$
1730 INPUT #1(1),DF$,BL,SA,FL
1740 FOR J=1 TO FL
1750 FOR I=1 TO BL
1760 INPUT #1(SA+(J-1)*BL+I-1),A$
1765 PRINT A$;"■■";
1770 NEXT I
1780 PRINT "■■"
1790 NEXT J
1800 CLOSE
1810 RETURN
2000 REM-STOP
2010 PRINT"■■";"END OF PROGRAM"
2020 END
```

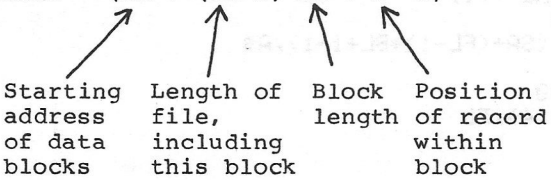
Notice that the data is stored as string text in each case. This avoids data entry errors. Finally, lines 1700 to 1810 list the file contents as shown below:

component	part number	quantity
INTERFACE	1109	8
KEYBOARD	1251	9
CABLES	1253	20
PAPER	1312	54
DISKETTES	1421	97
POWER PACK	1429	3

You may wish to add further features that will enable you to edit or delete from the file.

In order to calculate the correct record number, the following equation is used:

record number = (SA + (FL-1)*BL + I -1)

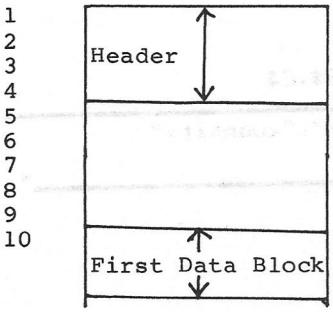


A little experimentation will prove this formula. For example, if

SA = 10
FL = 1 (i.e. this is the first block)
BL = 3
I = 1 (the first record in the block)

then, the record number of the first record in the first

block is 10:



We can now write the key-table program. This is extremely simple:

Lines 120 - 140: select the file required

Lines 1000 - 1020: read the header, telling us the block length (BL) starting address (SA) and file length (FL)

Lines 1030 - 1080: read the part-numbers into the array elements P(I,1) and set P(I,2) to the block number (I)

Lines 1100 - 1220: compare an input part-number with the entries in P(I,1) and if a match is found, read the contents of block I.

Here is the program:

```
100 REM- SELECT DISK DRIVE
110 PRINT"■"
120 INPUT"Name of file: ";F$
130 INPUT"Which disk drive: ";DF
140 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
900 PRINT"■■■DEMONSTRATION OF KEY TABLES■■■"
999 REM*********************************
1000 REM- READ HEADER INFO.
1010 XOPEN #1,F$
1020 INPUT #1(1),DF$,BL,SA,FL
1030 REM-- READ IN KEY TABLE
1040 DIM P(FL,2)
1050 FOR I=1 TO FL
1060 INPUT #1(SA+(I-1)*BL+1),P(I,1)
1070 P(I,2)=I
1080 NEXT I
1090 REM*********************************
1100 INPUT"■■■Part no. (-9999 to stop): ";PN
1105 IF PN=-9999 THEN 2000
1110 FOR I=1 TO FL
1120 IF PN=P(I,1) THEN BN=P(I,2): GOTO 1200
1130 NEXT I
```

```
1140 PRINT"■■No such number."
1150 GOTO 1100
1190 REM..... GET THE RECORD
1200 INPUT #1(SA+(BN-1)*BL)A$,B$,C$
1205 PRINT"■■                                                    "
1207 PRINT"  name","part number","quantity"
1210 PRINT A$;B$;C$
1215 PRINT"                                                      "
1220 GOTO 1100
2000 END
```

And this is an example dialogue:

```
Name of file: STOCK-LIST
Which disk drive: 1

DEMONSTRATION OF KEY TABLES


Part no. (-9999 to stop): 1251

  name      part number          quantity
KEYBOARD         1251               9
```

You could easily enhance this program to provide updating facilities and other features. Also, you could include the file creation program simply as an extra option. If you run this program, you will find the response speed to be almost equally fast for any part number, so long as the file is small (up to 100 or so items). In fact, the maximum length of file is 255 blocks since this is the maximum permitted size of an array subscript. For larger files, the response speed worsens, and there are 4 possible methods to improve on this:

(1) With only one array (up to 255 items) order the part numbers in sequence so that a "bisection" method can be applied.

(2) Use more than one array, such that each array refers to a sub-file of the main file.

(3) Store the key-table in a file, rather than an array.

(4) Use a "hashing" method. This is discussed in the next section.

8.3 Hashing Methods

As you have seen, key tables are fine for small files, but their performance degrades as the files lengthen. An alternative is the "hashing" or "randomising" method which, although slightly more complex, does have the following advantages:

(1) No searching is required, hence very fast response is achieved.

(2) Insertion is just as easy as deletion.

(3) Storage capacity is limited only by the allowed size of a file.

(4) Names, rather than numbers, can be used for easy retrieval.

In a hashing scheme, the key (e.g. part number, or text name) is converted into a numerical value. For example, if the key is K$, one method is to sum the ASCII values of its component characters:

```
100  H=0
110  FOR I=1 to LEN(K$)
120  H=H+ASC(MID$(K$,I,1)
130  NEXT I
```

The idea is then to use H (the hash value) as the address of a record (or block) witin a direct access file. The problem with our simple scheme, above, is that different keys will often sum to the same value of H (e.g. STOP gives the same value as POTS). A modificaton is to change line 120 to:

```
120 H=H+I*ASC(MID$(K$,I,I)
```

which weights the ASCII values depending on their position in K$.

The next step is to use the hash value to point to a valid file location. In the hashing scheme, it is desirable to ensure that input keys select uniformly from the allowed range of file addresses. There are many ways to do this (see references at the end of this chapter) but one simple scheme, using the RND function on the MZ-80K is as follows:

1) Initialize the RND function using a zero or negative argument.

2) Select a random number in the range required.

For example, having obtained H, we can compute a file address, ID, in a file of length HT as follows:

```
140 R=RND(0)
150 FOR I=1 to H-1: R=RND(1): NEXT I
160 ID= INT(HT*RND(1))+1
```

The beauty of this is that, although the RND function is used, the value of ID is entirely predictable for a given value of H and HT. Also, ID values are uniformly spread from 1 to HT , which is important, as you will see later.

As with any type of file processing, we need to add new items, delete old ones and search for particular information. Under a hashing scheme, these actions are handled in a consistent manner:

1) Hash the input key to a file location, and examine the information found at that location.

2) If the information stored at that location matches the input key, go ahead and display the contents of the block, or if deletion is required, erase the block.

3) If the location is empty, the new input key can be stored there, together with other data in the subsequent records of the block.

The only problem is that two non-identical keys may hash to the same file location. So, in case (2), we find that the stored key in the file does not match the input key. Similarly, in case (3), when trying to add a new item to the file, the location is found to be occupied. This situation is called a "collision" and many schemes have been devised to avoid the problem and to solve it when collisions occur. You can now see why a uniform random scheme was chosen, so that input keys would hash to points all over the file. If, after doing this, collisions still occur, all we can do is to abandon the first location and try another. The simplest method is to step onto the next location, but many better schemes have been devised, as exemplified in the next program.

PROGRAM NAME: HASHING3

In this program, we bring together the above ideas. Referring to the listing, lines 9020 to 9078 select the initial hashing value. The only improvement on the above scheme is, in line 9070, to compute the remainder of H divided by HT, so that line 9077 is not delayed by very large values of H. Lines 9080-9550 use a scheme to handle the collision problem which occurs:

1) If deletion or display of an item is needed (OP=1) and the input key does not match the stored data

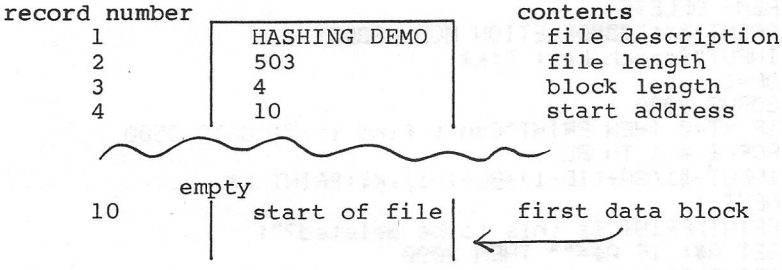
or 2) An item is being added (OP=2) and the file contains data at that location.

Briefly, the scheme used is a highly-efficient "non-linear offset" scheme described in one of my previous books (see references). Though I take no credit for it, one advantage is that it examines all locations other than the original, once only. In order for this to happen, the length of the file should be a prime number of the form 4j+3.

Some values are:

j	Prime
31	127
62	251
125	503
254	1019

The rest of the program is straight-forward, using blocks of data records as described earlier. The file has a header block which, in our case, was:



Program: HASHING3

```
100 REM- CONTROL BLOCK
110 REM- HASHING DEMONSTRATION
120 PRINT"■"
125 REM FOR TESTING,USE FILE 'HASH2'
130 INPUT"Name of file?";H$
160 INPUT"Number of disk drive?";DF
170 POKE $49AC,DF-1: POKE $49AB,0: POKE $52BC,DF-1
174 XOPEN #1,H$
180 INPUT #1(1),DF$,HT,BL,SA
185 IF DF$="" THEN PRINT"■■THIS IS A NEW FILE: use option 5:■■"
190 CURSOR 0,6
200 PRINT"1: Add a new item"
210 PRINT"2: Delete an item"
220 PRINT"3: Display an item"
230 PRINT"4: Stop the program"
235 PRINT"5: Create or change header"
```

(Program HASHING3 continues)

```
240 CURSOR 0,12
250 PRINT"Choose a number";
260 GET A$: IF A$="" THEN 260
270 PRINT A$;
280 S$="12345"
290 FOR I=1 TO LEN(S$)
300 IF A$=MID$(S$,I,1) THEN 330
310 NEXT I
320 PRINT"■ ":GOTO 240
330 ON I GOSUB 1000,2000,3000,4000,5000
400 PRINT"Press any key to continue"
410 GET A$: IF A$="" THEN 410
420 PRINT"■"
430 GOTO 190
500 REM**********************************
1000 REM-ADD
1005 PRINT"■";"■■■ADDITION MODULE■■■"
1010 INPUT"■■Input key: ";K$
1015 OP=2
1020 GOSUB 9000
1030 IF XT=0 THEN PRINT"File full.":GOTO 1500
1035 PRINT #1(SA+(ID-1)*BL),K$
1040 PRINT"Type description (";BL-1;" lines):"
1050 FOR I=1 TO BL-1
1060 PRINT"Line ";I;": ":: INPUT K$
1070 PRINT #1(SA+(ID-1)*BL+I),K$
1080 NEXT I
1500 RETURN
1600 REM**********************************
2000 REM- DELETE
2010 PRINT"■";"■■■DELETION MODULE■■■"
2020 INPUT"Search key: ";k$
2025 OP=1
2030 GOSUB 9000
2040 IF XT=0 THEN PRINT"Can't find it!": GOTO 2500
2050 FOR I = 1 TO BL
2060 INPUT #1(SA+(ID-1)*BL+I-1),K$:PRINT K$
2070 NEXT I
2080 PRINT:PRINT"Is this to be deleted?";
2090 GET A$: IF A$="" THEN 2090
2095 PRINT A$
2100 IF A$="Y" THEN PRINT #1(SA+(ID-1)*BL)," "
2500 RETURN
2600 REM**********************************
3000 REM-SEARCH
3010 PRINT"■";"■■■SEARCH MODULE■■■"
3020 INPUT"Search key: ";K$
3025 OP=1
3030 GOSUB 9000
3040 IF XT=0 THEN PRINT"Can't find it": GOTO 3500
3050 FOR I=1 TO BL
3060 INPUT #1(SA+(ID-1)*BL+I-1),K$:PRINT K$
3070 NEXT I
3500 RETURN
3600 REM**********************************
4000 REM-CLOSE DOWN
4060 CLOSE
4080 END
4100 REM**********************************
```

```
5000 REM-INITIALISE
5010 PRINT"■";"■■■INITIALISE A FILE■■■"
5040 INPUT"File description: ";DF$
5050 INPUT"Maximum length (blocks): ";HT$
5060 HT=VAL(HT$): IF HT<1 THEN 5050
5070 INPUT"Block length (records): ";BL$
5080 BL=VAL(BL$): IF BL<1 THEN 5070
5090 INPUT"Start address (record): ";SA$
5100 SA=VAL(SA$): IF SA<5 THEN 5090
5120 PRINT #1(1),DF$,HT,BL,SA
5124 PRINT"■■Please wait..... initialising.....■■■"
5126 PRINT #1(SA+HT*BL)," "
5130 RETURN
5500 REM*********************************
9000 REM-HASHING FUNCTION
9010 XT=0
9020 N=LEN(K$)
9030 H=0
9040 FOR I=1 TO LEN(K$)
9050 H=H+I*ASC(MID$(K$,I,1))
9060 NEXT I
9070 H1=H/HT:  H=INT(HT*(H1-INT(H1)))
9075 R=RND(0)
9077 FOR I=1 TO H-1: R=RND(1): NEXT I
9078 ID=INT(HT*RND(1))+1
9080 IN=-HT
9085 REM- NOW USE ID TO FIND THE RECORD
9090 IF IN<HT THEN 9100
9092 GOTO 9600
9100 INPUT #1(SA+(ID-1)*BL),I$
9105 ON OP GOTO 9200,9400
9110 PRINT"OPTION ERROR": RETURN
9200 IF K$=LEFT$(I$,N) THEN XT=1:GOTO 9600
9300 GOTO 9500
9400 IF LEFT$(I$,1)=" " THEN XT=1:GOTO 9600
9500 IN=IN+2
9510 ID=ID+ABS(IN)
9520 IF ID>HT THEN ID=ID-HT
9550 GOTO 9090
9600 RETURN
```

Here is an example dialogue with the deletion module:

```
DELETION MODULE

Search key: CARTWRIGHT
12 MT PLEASANT
NEWCASTLE
STAFFS

Is this to be deleted?N
```

Notice that initialisation (subroutine 5000) extends an empty file to the maximum required length. This makes addition (subroutine 1000) much faster than it would be if

the file had to be extended (often by hundreds of records) with each addition.

When you run this program, try inputting names and addresses as part of a data bank. Notice that any item is added/retrieved/deleted equally quickly. You could use a program like this to store client records or names and phone numbers. The only points to note are:

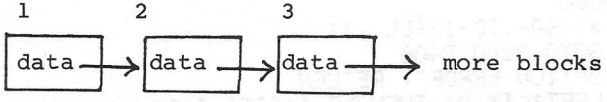
1) A hash scheme does not allow you to retrieve sets of data (e.g. all names that begin with B)

2) You have to use exactly the same key to retrieve or delete an item as the one you used to add it to the file. This can be assisted by keeping a directory of all of the keys on some other small file (possibly just a sequential file) in case you forget the keys!

8.4 Linked-List Files

In complete contrast to hashing, we turn to linked-list files. These belong to the larger group of "dynamic data structures" in which each item is related to every other by some linking mechanism.

The linked list can be thought of as a series of connected boxes, each of which represents a data block:



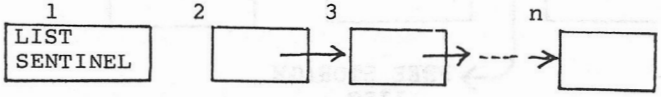
Each block contains a **pointer record** in addition to the data records. If we begin at block 1 and follow the pointers, the entire list will be encountered. The usual purpose of a linked list is to keep a file organised according to some sequencing criterion, such as alphabetical order. This can be applied to a mailing list: each data block can be a name and address and, by following the pointers, the entire mailing list is produced in aphabetical order.

This is a very attractive scheme, especially since insertion and deletion are handled in such a way that the ordering is always maintained, no matter how much we mess around with the file. The only drawback is that, to find a particular item, you start from the beginning of the list in a similar fashion to a sequential file. Since the linked list is so attractive, let us see briefly how to organise one, following one set of rules:

1) Use a direct access file of, say, n blocks.

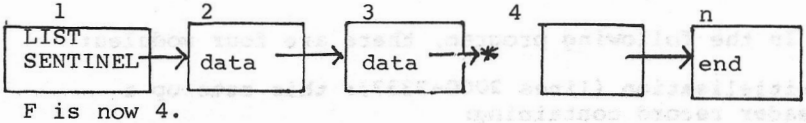
2) Initialise the file as two linked lists: the (so

far) empty data list and a fully linked "free-storage" list from which new blocks are obtained. Initially, the set up is like this:



The "list sentinel" is not used for data storage. Instead, data blocks will be attached, as needed, to the sentinel block. A free-storage pointer (F) is needed to identify the first available free block. As shown, F is equal to 2.

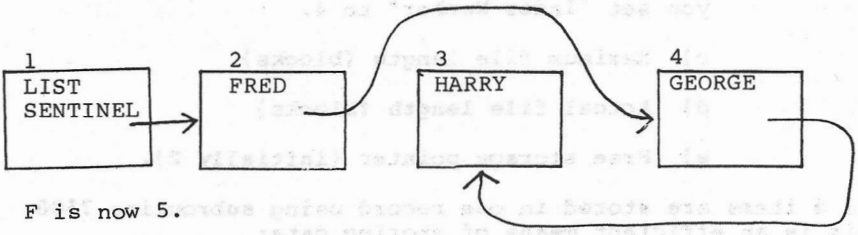
3) To add blocks to the data list, connect the sentinel block to the data blocks by a pointer, and change F accordingly. So, after connecting 2 blocks we have:



F is now 4.

4) Output the data file by starting at block 1 and following the pointer until a "null" pointer is found. In the above list, the null is represented by ⟶ *, telling us to stop.

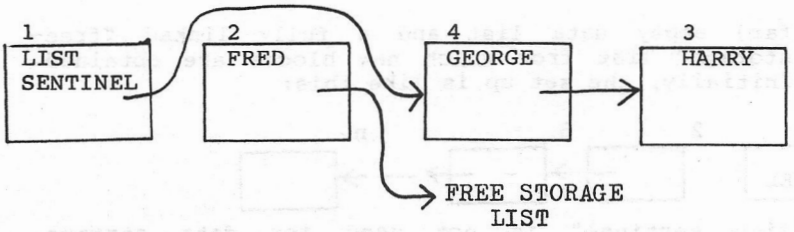
5) To insert a new data block in its correct position, get a block from free storage, find where it should be inserted, and change the pointers to include the new block:



F is now 5.

This shows block 4 being linked in alphabetically, from which you will see that the physical sequence of blocks bears no necessary resemblance to its logical order.

6) To delete a block, connect it to the beginning of the free storage chain and adjust the pointers. For example, to delete FRED:



You can now see why this is called a "dynamic" structure: unwanted blocks are returned for immediate re-use, and the system is completely self-maintaining. F is now 2, so block 2 is the first available block for future insertions.

It is easy to follow these rules to produce a useful linked list program in BASIC.

PROGRAM: LINKER2

In the following program, there are four modules:

1) **Initialisation** (lines 2000-2337): this sets up a header record containing:

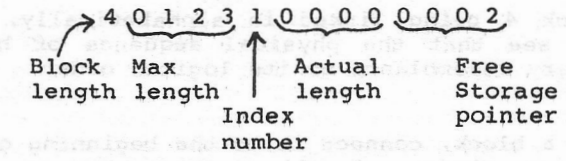
   a) Number of records/block (up to 8)
   b) Index number: if the file contains names an d addresses, you could index the file on the name, so the index number would be 1 (for line 1). But, instead, you can index on any part of the address if you wish. For example, if your data block contained the town on line 4 (i.e. record 4), you would produce a file alphabetically ordered in terms of towns, if you set "Index Number" to 4.

   c) Maximum file length (blocks)

   d) Actual file length (blocks)

   e) Free storage pointer (initially 2).

All 5 items are stored in one record using subroutine 7500. This is an efficient means of storing data:



The free storage list is created in lines 2190-2240 by placing a pointer at the end of each block. The end of list character (-1) is inserted into the free storage chain in line 2310 and in the sentinel block in line 2320.

2) **Output** (Lines 3000-3139) This simply follows the data list through its pointers, either outputting all of the blocks, or just those that contain a particular keyword in the index record.

3) **Insertion** (lines 4000-4590) This finds the insertion position (lines 4025-4090) and then gets a block from free storage (lines 4100-4220). The header block is updated.

4) **Deletion** (lines 5000-5710) When the data block is found, the user is asked to confirm that it is to be deleted.

This is a fairly complex program, although it is very simple to use. The program is listed below, with some example output:

Listing of Linker2

```
100 REM-CONTROL BLOCK
105 PRINT"■"
110 INPUT "File name?";F$
120 INPUT"Number of disk drive ";DF
130 POKE $49AC,DF-1:POKE $49AB,0: POKE $62BC,DF-1
140 ON ERROR GOTO 900
145 CURSOR 0,6
150 PRINT"1: Initialise the file"
160 PRINT"2: Output the file"
170 PRINT"3: Insert an item"
180 PRINT"4: Delete an item"
190 PRINT"5: Stop"
195 CURSOR 0,12
200 PRINT:PRINT"Choose a number (1 to 5) ";
210 GET A$: IF A$="" THEN 210
220 PRINT A$;
230 S$="12345"
240 FOR I=1 TO LEN(S$)
250 IF A$=MID$(S$,I,1) THEN 280
260 NEXT I
270 PRINT"■ ":GOTO 195
280 ON I GOTO 2000,3000,4000,5000,1100
900 PRINT"ERROR IN CONTROL BLOCK"
910 KILL #1
915 PRINT"ERROR ";ERN;" IN LINE ";ERL
1000 PRINT"Press any key to continue"
1010 GET A$: IF A$="" THEN 1010
1020 PRINT"■"
1030 GOTO 145
1100 END
1999 REM**********************************
2000 REM-INITIALISATION
2001 ON ERROR GOTO 2335
2005 XOPEN #1,F$
2010 PRINT"■"
2012 PRINT"WARNING - initialisation will erase any contents"
2014 PRINT"Do you want to continue?";
2016 GET A$: IF A$="" THEN 2016
2017 PRINT A$: IF A$="Y" THEN 2020
```

```
2019 GOTO 2325
2020 PRINT:INPUT"Data records per block? ";L$
2021 L=VAL(L$): IF (L>0)*(L<9) THEN 2025
2022 PRINT"No more than 8.....":GOTO  2020
2025 LB=L+1
2040 INPUT"File length (blocks)? ";MF$
2050 MF=VAL(MF$): IF MF<=9999 THEN 2080
2060 PRINT"No more than 9999....": GOTO 2040
2080 INPUT"Number of index record? ";SF$
2085 SF=VAL(SF$): IF (SF)0)*(SF<=L) THEN 2120
2087 PRINT"Must be from 1 to ";L: GOTO 2080
2120 AF=0
2130 FP=2
2160 REM- USE STRING HEADER SUBROUTINE
2170 GOSUB 7500
2190 REM- INSERT LINKS
2210 FOR I=2 TO MF
2220 PRINT #1(I*LB),(I+1)
2240 NEXT I
2300 REM- CLOSE THE CHAINS
2310 PRINT #1 ((MF+1)*LB),-1
2320 PRINT #1 (LB),-1
2330 CLOSE: GOTO 1000
2335 PRINT"ERROR DURING INITIALISATION"
2337 GOTO 915
2340 REM*********************************
3000 REM-OUTPUT MODULE
3001 ON ERROR GOTO 3135
3002 XOPEN #1,F$
3005 PRINT"C";"Output from file ";F$
3010 PRINT"Output to printer?";
3015 GET A$: IF A$="" THEN 3015
3016 PRINT A$
3017 PF=0: IF A$="Y" THEN PF=1
3020 GOSUB 7000
3022 PRINT"Current length: ";AF;" Max. :";MF
3023 PRINT
3024 PRINT"Please type the keyword in"
3025 PRINT"Line no. ";SF;" of the items to be listed"
3026 INPUT"or type '*' to list them all.";B$:BL=LEN(B$)
3030 PL=1
3040 GOSUB 9000
3110 IF PT=-1 GOTO 3120
3115 PL=PT: GOTO 3040
3120 PRINT"End of listing"
3125 PF=0
3130 CLOSE: GOTO 1000
3135 PRINT"ERROR IN OUTPUT MODULE"
3137 KILL
3139 GOTO 915
3140 REM ****************************
```

```
4000 REM- INSERTION
4001 PRINT"E";"Insertion Routine":PRINT
4002 ON ERROR GOTO 4300
4003 XOPEN #1,F$
4005 GOSUB 7000
4012 IF FP<>-1 THEN 4020
4014 PRINT"No more space!": GOTO 4260
4020 DIM T$(LB)
4021 PRINT"Type new item (";LB-1;" lines)
4022 FOR I=1 TO LB-1
4023 PRINT"LINE ";I;": ";: INPUT T$(I)
4024 NEXT I
4025 PL=1
4030 INPUT #1(PL*LB),PT
4040 IF PT<>-1 THEN 4060
4050 GOTO 4110
4060 INPUT #1((PT-1)*LB+SF),A$
4065 B$=T$(SF)
4070 GOSUB 4500
4075 IF SP=2 THEN 4090
4080 GOTO 4100
4090 PL=PT: GOTO 4030
4100 REM- GET A FREE NODE
4110 W=FP
4120 INPUT #1(FP*LB),NF
4125 REM-UPDATE HEADER
4130 AF=AF+1: FP=NF: GOSUB 7500
4155 FOR I=1 TO LB-1
4160 PRINT #1((W-1)*LB+I),T$(I)
4170 NEXT I
4190 REM-CONNECT TO REST OF CHAIN
4200 PRINT #1(W*LB),PT
4210 REM- CONNECT FIRST PART OF CHAIN
4220 PRINT #1(PL*LB),W
4230 PRINT"Any more?";
4240 GET A$:IF A$="" THEN 4240
4245 PRINT A$
4250 IF A$="Y" THEN 4012
4260 CLOSE: GOTO 1000
4300 PRINT"ERROR DURING INSERTION"
4310 KILL
4320 GOTO 915
4500 REM- STRING COMPARISON
4515 XX$="                ": REM.. 16 BLANKS
4520 XX=16
4522 A1$=A$+RIGHT$(XX$,XX-LEN(A$))
4524 B1$=B$+RIGHT$(XX$,XX-LEN(B$))
4525 IF A1$=B1$ THEN SP=3: GOTO 4550
4528 FOR ZZ=1 TO XX
4530 IF ASC(MID$(A1$,ZZ,1))<ASC(MID$(B1$,ZZ,1)) THEN SP=2: GOTO 4550
4535 IF ASC(MID$(A1$,ZZ,1))>ASC(MID$(B1$,ZZ,1)) THEN SP=1: GOTO 4550
4540 NEXT ZZ
4550 RETURN
4600 REM**********************************
```

```
5000 REM-DELETION
5005 ON ERROR GOTO 5700
5010 PRINT"@";"Deletion Routine":PRINT
5020 XOPEN #1,F$
5030 GOSUB 7000
5045 PRINT"Please type the keyword in"
5050 PRINT"Line no. ";SF;" of the item to be"
5060 INPUT"Deleted :";B$
5070 BL=LEN(B$)
5072 PL=1
5075 REM-9000 RETURNS WITH PT & HT
5080 GOSUB 9000
5085 IF PT=-1 THEN 5100
5090 IF HT=1 THEN 5150
5095 PL=PT: GOTO 5080
5100 PRINT"Can't find it.": GOTO 5600
5150 PRINT"Is this to be deleted?";
5155 GET A$: IF A$="" THEN 5155
5160 PRINT A$: IF A$="Y" THEN 5170
5165 PRINT"Continuing.....":PL=PT: GOTO 5080
5170 PRINT"Deleting......"
5180 INPUT #1(PT*LB),PR
5190 PRINT #1(PL*LB),PR
5200 PRINT #1(PT*LB),FP
5210 AF=AF-1: FP=PT: GOSUB 7500
5240 PRINT"Item deleted."
5600 CLOSE: GOTO 1000
5700 PRINT"ERROR DURING DELETION"
5705 KILL
5710 GOTO 915
5750 REM*********************************
7000 REM-READ HEADER
7010 INPUT #1(1),R$
7020 LB=VAL(LEFT$(R$,1))
7030 MF=VAL(MID$(R$,2,4))
7040 SF=VAL(MID$(R$,6,1))
7050 AF=VAL(MID$(R$,7,4))
7060 FP=VAL(MID$(R$,11,4))
7070 RETURN
7080 REM*********************************
7500 REM-PRINT HEADER
7510 LB$=STR$(LB): SF$=STR$(SF)
7520 D$="0000"
7530 MF$=STR$(MF): MF$=LEFT$(D$,4-LEN(MF$))+MF$
7540 AF$=STR$(AF): AF$=LEFT$(D$,4-LEN(AF$))+AF$
7550 FP$=STR$(FP): FP$=LEFT$(D$,4-LEN(FP$))+FP$
7560 R$=LB$+MF$+SF$+AF$+FF$
7570 PRINT #1(1),R$
7580 RETURN
8000 REM*********************************
9000 REM-STRING SEARCH
9010 HT=0:  REM.... 'HIT' MARKER
9020 INPUT #1(PL*LB),PT
9030 IF PT<>-1 THEN 9045
9040 GOTO 9500
9045 IF B$="*" THEN HT=1: GOTO 9160
9050 INPUT #1((PT-1)*LB+SF),A$
9060 AL=LEN(A$): DL=AL-BL+1
```

```
9070 FOR K=1 TO DL
9080 S$=MID$(A$,K,BL)
9090 IF S$=B$ THEN HT=1: GOTO 9160
9140 NEXT K
9145 GOTO 9500
9160 FOR I=1 TO LB-1
9165 INPUT #1((PT-1)*LB+I),A$
9170 PRINT A$
9175 IF PF=1 THEN PRINT/P A$
9180 NEXT I
9185 PRINT: IF PF=1 THEN PRINT/P
9500 RETURN
```

A small set of output is shown below:

```
Output from file MAIL-LIST
Output to printer?Y

Current length:  2 Max. : 100


Please type the keyword in
Line no.  1 of the items to be listed
or type '*' to list them all.*
ALBERTO
BANK SQ
TWITTINGTON
HAMPS

CARTWRIGHT
16 MT PLEASANT
NEWCASTLE
STAFFS


End of listing
```

The data file was stored on drive 1, with the name MAIL-LIST. The maximum file length was set to 100 blocks, each of length 4. The index record was line 1, giving a list sequenced on the first (name) entry.

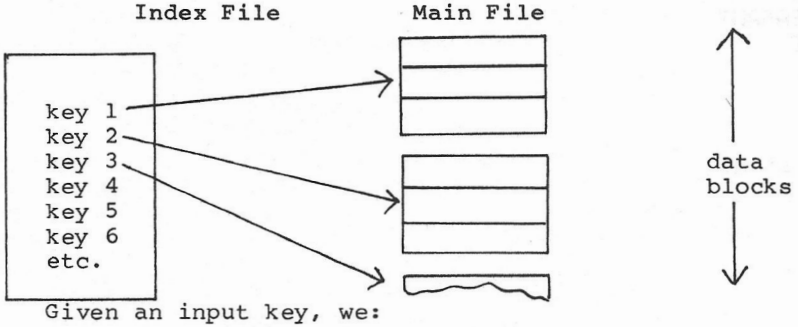
8.5 Index Sequential (ISAM) Files

Of the methods discussed so far, hashing gives the fastest response, while the linked list was slower, but at least it kept the items in the correct order. A compromise scheme which gives fast response and keeps the items in order is called the "Index Sequential Access Mode" or ISAM method.

A simple example of an ISAM is provided by KOLDERUP (see references). He forms an alphabetic index for each letter of the alphabet: the A entry in the index points to the first record that begins with A; the "B" entry to the first record beginning with "B" and so on. Then to find any entry, you simply scan through the index to find the initial letter of your keyword, follow its pointer, and examine just one small part of the main file. An alternative, which overcomes the 16 byte record length limitation, is to divide the main file into small sequential files; in this case you simply LOAD the small file required.

The only problem is that these simple ISAM's assume that the file can be sorted in main memory. Unfortunately, in-memory sorts are only applicable to a hundred or so items, principally because of limitations of array space. Therefore, if the file is small enough for an in-memory sort, you would be better to choose a simpler method than ISAM. So, let us examine the basis of a realistic ISAM.

Any ISAM uses an index to determine the block address in an ordered direct access file from which to start. This is visualised below:



Given an input key, we:

1) Scan the index file until a key (K_i) is found which is greater than the input key.

2) Go back to the preceding key (K_{i-1}) in the index file, and follow its index pointer to the main file.

3) Inspect just that part of the main file between K_i and K_{i-1}.

Therefore, only a small part of the file has to be searched.

The index file is constructed as follows:

1) Sort the main file into the desired order.

2) For every n'th block in the main file, write both the key and the block number to the index file. Block 1 must be the first entry in the index file, otherwise the first n-1 blocks are inaccessible. Also the final entry in the index file has a key much geater than any expected key (such as zzzzzzzz.....).

If the file is very large, making a good candidate for an ISAM, we must devise a means of sorting the main file. Many methods exist, although most of them involve several passes through the main file. A scheme which is admittedly slow, but which needs only one pass to sort any file, is the **threaded binary tree** sort. This is easier to picture than to explain:

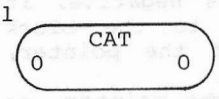
Each key from the main file is allocated a block of two records: one to store the key itself and one to store left and right pointers:

key	
left-pointer	right-pointer

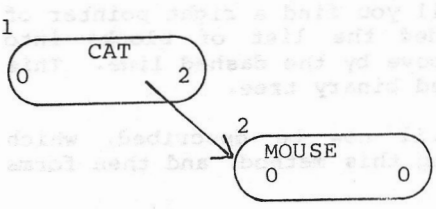
Now, imagine that the keys to be sorted are as follows:

CAT, MOUSE, APE, ELEPHANT, HIPPO

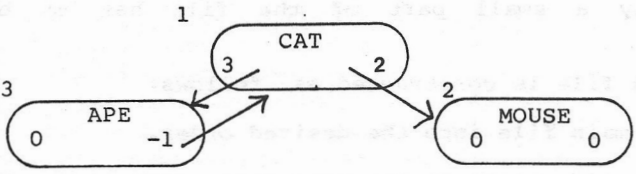
The first key is entered into block 1 and its pointers are each set to zero:



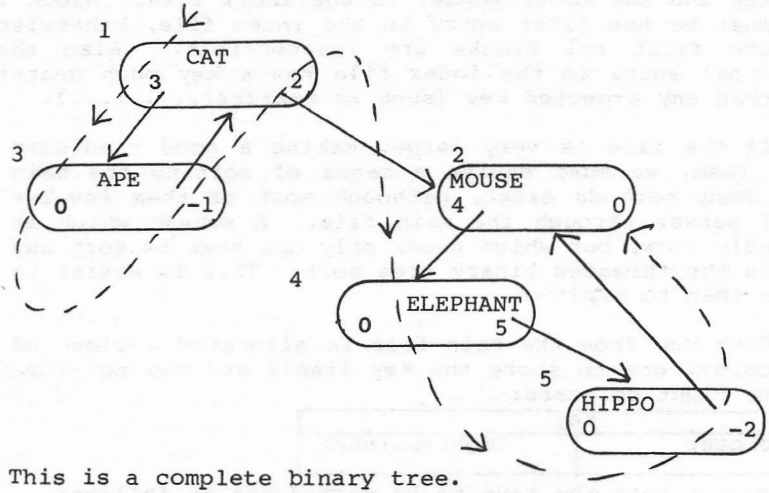
The second key is aphabetically greater than CAT, so the right-pointer of CAT is changed to 2:



APE precedes CAT, so the left pointer of CAT is set to 3. Also, we need to know which block follows APE, so we must set the right pointer also. But, to indicate that CAT is at a higher level than APE, we set the pointer to a negative value, to indicate that it is a **back-track:**



The next two items are added, to give:



This is a complete binary tree.

The beauty of this is that if:

1) You start at block 1 and
2) Follow the left pointers as far as possible then you have found the first alphabetic item.
3) Examine the right pointer. If it is negative, it must be a **back-track**, so go back to the block identified by the absolute value of the pointer, otherwise:
4) If it is positive, it is simply a right pointer, so follow it to the next block and go back to step 2.

Continue in this fashion until you find a right pointer of zero. You have then decoded the list of blocks into alphabetic order, as shown above by the dashed line. This is why it is called a **threaded** binary tree.

A complete program will now be described, which sorts any length of file using this method, and then forms an ISAM index to the file.

PROGRAM NAME: ISAM2

The program consists of these main modules:

Sort module (lines 1000-1430)

A direct access file, F$, of unsorted blocks is opened. A new file, T$, is opened in line 1045 with the name composed of F$ concatenated with IND. It is assumed that F$ has already been produced with a program such as MAKE-DIRECT from section 8.2. File F$ has a 3-line header specifying:

Block length
Current file length
Starting address

There are two subroutines, 5500 and 6000, to retrieve information on any block in T$ and to create a new block.

Indexing Module (lines 2000-2160)

This creates the index file, TT$, by producing 2-line blocks of the form:

(1) key
(2) address in main file

for every ID'th record. ID is the "indexing density"; e.g. if ID = 10, every tenth record is indexed.

Search Module (lines 4000-4310)

This compares a query key with the entries in file TT$, the contents of which are stored in array I$(IL,2). This gives an index size of 255, which is adequate. For example, with an indexing density of 10, the index would refer to a main file of over two and a half thousand blocks. When an entry in the array is found which is greater than the query key, lines 4160-4270 cause part of the threaded binary tree file, T$, to be inspected for the desired key. Notice that entries are found even when only part of the key is known.

Here is a complete listing of the program:

Program: ISAM2

```
100 REM-RANDOM ACCESS SORT
110 PRINT"■";"■■■RANDOM ACCESS SORT■■"
120 INPUT"Master file name? ";F$
130 INPUT"Disk drive ";DF
140 POKE $49AC,DF-1:POKE $49AB,0:POKE $62BC,DF-1
145 CURSOR 0,10
150 PRINT"1: Sort the file"
160 PRINT"2: List it or index it"
170 PRINT"3: Search it"
175 PRINT"4: Stop"
180 CURSOR 0,14
```

```
190 PRINT"Choose a number ";
200 GET A$: IF A$="" THEN 200
205 PRINT A$;
210 S$="1234"
220 FOR I=1 TO LEN(S$)
230 IF A$=MID$(S$,I,1) THEN 260
240 NEXT I
250 PRINT"■ ": GOTO 180
260 ON I GOSUB 1000,2000,4000,300
270 PRINT"Press any key to continue..."
280 GET A$: IF A$="" THEN 280
290 PRINT"■": GOTO 145
300 REM-FINISH
310 PRINT:PRINT"■■■End of program."
320 END
330 REM**********************************
1000 REM-SORTING
1010 PRINT"■";"Sorting Module."
1020 INPUT"■■Number of indexing record? ";IR$
1025 IR=VAL(IR$): IF IR=0 THEN 4020
1030 T$=F$+STR$(IR): REM- ASSIGN TREE NAME
1040 TT$=T$+"IND"
1045 XOPEN #1,F$: XOPEN #2,T$
1050 REM- READ HEADER
1055 INPUT #1(1),BL,FL,SA
1160 PRINT"■■Sorting ";FL;" records...."
1200 INPUT #1(SA+IR-1),K$
1212 LP=0: RP=0: T=1
1216 GOSUB 6000
1220 FOR I=2 TO FL
1225 RI=0: LI=0
1230 INPUT #1(SA+BL*(I-1)+IR-1),AI$
1235 J=1
1240 GOSUB 5500
1250 A$=AI$: B$=AJ$
1260 GOSUB 4500
1270 IF SP=1 THEN 1350
1280 IF LJ=0 THEN 1310
1290 J=LJ: GOTO 1240
1310 RP=-J: LP=RI: K$=AI$: T=I
1315 GOSUB 6000
1320 RP=RJ: LP=I: K$=AJ$: T=J
1325 GOSUB 6000
1340 GOTO 1410
1350 IF RJ>0 THEN 1360
1355 GOTO 1380
1360 J=RJ: GOTO 1240
1380 RP=RJ: LP=LI: K$=AI$: T=I
1390 GOSUB 6000
1400 RP=I: LP=LJ: K$=AJ$: T=J
1405 GOSUB 6000
1410 NEXT I
1420 CLOSE
1430 RETURN
1460 REM**********************************RUN
```

```
2000 REM-STRIP THE TREE & FORM INDEX
2001 PRINT"@Listing & Indexing module■■■"
2002 PRINT"@":INPUT"■■Number of indexing record? ";IR
2004 T$=F$+STR$(IR): TT$=T$+"IND"
2005 XOPEN #1,F$: XOPEN #2,T$: XOPEN #3,TT$
2006 INPUT #1(1),BL,FL,SA
2007 INPUT #3(1),A$: IF LEFT$(A$,1)="" THEN PRINT"■■NOT YET INDEXED■■■"
2008 PRINT"Is the file to be indexed?";
2009 GET A$: IF A$="" THEN 2009
2010 PRINT A$
2011 IF A$="N" THEN 2018
2013 INPUT"Indexing density? ";ID
2014 PRINT"File now being indexed ■■■"
2015 IR=2: K=0
2018 PRINT "■■Listing of all ";FL;" records in ";F$;"/■■"
2020 J=1
2060 GOSUB 5500
2070 IF LJ>0 THEN J=LJ: GOTO 2060
2080 K=K+1: REM - INCREMENT RECORD COUNT
2085 FOR I=1 TO BL
2090 INPUT #1(SA+BL*(J-1)+I-1),AF$:PRINT AF$
2095 NEXT I
2097 PRINT
2100 IF A$="N" THEN 2110
2102 IF (K<>1)*(K/ID<>INT(K/ID)) THEN 2110
2104 PRINT #3(IR),AJ$
2105 J$=STR$(J): PRINT #3(IR+1),J$: IR=IR+2
2110 IF RJ>0 THEN J=RJ: GOTO 2060
2120 IF RJ=0 THEN 2142
2130 J=-RJ: GOSUB 5500
2140 GOTO 2080
2142 IF A$="N" THEN 2150
2143 PRINT #3(IR),"ZZZZZZZZZZZZZZZ": PRINT #3(IR+1),"0"
2144 J$=STR$(IR/2): PRINT #3(1),J$
2150 CLOSE
2160 RETURN
2170 REM**********************************
4000 REM-INDEX SEARCH
4010 PRINT"@";"Key search module■■■"
4020 INPUT"■■Number of indexing record? ";IR$
4025 IR=VAL(IR$): IF IR=0 THEN 4020
4030 T$=F$+STR$(IR): REM- ASSIGN TREE NAME
4040 TT$=T$+"IND"
4050 XOPEN #1,F$: XOPEN #2,T$: XOPEN #3,TT$
4060 INPUT #1(1),BL,FL,SA
4070 INPUT #3(1),IL
4080 DIM I$(IL,2)
4090 FOR I=1 TO IL
4100 INPUT #3(2*I),I$(I,1)
4110 INPUT #3(2*I+1),I$(I,2)
4130 NEXT I
4132 INPUT"■■■Query key?";Q$
```

```
4133 LQ=LEN(Q$)
4135 I=1
4140 A$=Q$: B$=I$(I,1): GOSUB 4500
4142 IF SP=1 THEN I=I+1: GOTO 4140
4150 IF SP=3 THEN J=VAL(I$(I,2)): GOTO4240
4155 REM- NOW FOUND A VALUE > Q$
4160 IF I=1 THEN 4270
4164 J=VAL(I$(I-1,2))
4166 REM-FIND RIGHT POINTER
4168 GOSUB 5500
4170 J=ABS(RJ)
4180 GOSUB 5500
4190 IF LJ>0 THEN J=LJ: GOTO 4180
4195 B$=LEFT$(AJ$,LQ): GOSUB 4500
4200 IF  SP=3 THEN 4240
4205 IF SP=2 THEN 4270
4210 IF RJ=0 THEN 4270
4220 IF RJ>0 THEN J=RJ: GOTO 4180
4230 J=-RJ: GOSUB 5500: GOTO 4195
4240 PRINT"■■■ITEM FOUND...■■"
4250 FOR I=1 TO 4:INPUT #1(J*BL+I),LI$: PRINT LI$: NEXT I
4265 PRINT"■■Is this the item (Y/N)?";
4266 GET AA$:IF AA$="" THEN 4266
4267 IF AA$="Y" THEN 4280
4268 PRINT"■■Continuing....■■": GOTO 4210
4270 PRINT"■■ ITEM NOT FOUND"
4280 CLOSE
4290 RETURN
4300 PRINT"!!!! INDEXING ERROR !!!!"
4305 CLOSE
4310 RETURN
4320 REM**********************************
4500 REM-STRING COMPARISON
4515 XX$="                                        ": REM- 40 BLANKS
4520 XX=40
4522 A1$=A$+RIGHT$(XX$,XX-LEN(A$))
4524 B1$=B$+RIGHT$(XX$,XX-LEN(B$))
4525 IF A1$=B1$ THEN SP=3: GOTO 4550
4528 FOR ZZ=1 TO XX
4530 IF ASC(MID$(A1$,ZZ,1))<ASC(MID$(B1$,ZZ,1)) THEN SP=2: GOTO 455
4535 IF ASC(MID$(A1$,ZZ,1))>ASC(MID$(B1$,ZZ,1)) THEN SP=1: GOTO 455
4540 NEXT ZZ
4550 RETURN
4560 REM**********************************
5500 REM - RETRIEVE INFO ON ITEM J
5510 INPUT #2(J*2-1),AJ$,R$
5530 LJ=VAL(LEFT$(R$,8)): RJ=VAL(RIGHT$(R$,8))
5540 RETURN
5550 REM**********************************
6000 REM- PRINT TO THE TREE
6010 LP$=STR$(LP): RP$=STR$(RP)
6020 D$="00000000"
6030 LP$=LEFT$(D$,8-LEN(LP$))+LP$
6040 RP$=LEFT$(D$,8-LEN(RP$))+RP$
6050 R$=LP$+RP$
6060 PRINT #2(T*2-1),K$,R$
6070 RETURN
6080 REM**********************************
```

This is part of the unsorted file (I used a list of Sharp dealers:

```
BCS EQUIP LTD
BRISTOL
AVON
0272-425338

DECIMAL LTD
BRISTOL
AVON
0272-294591

ISHERWOODS
LUTON
BEDS
0582-416202

BCG EQUIP LTD
READING
BERKS
0734-54015

NEWBEAR LTD
NEWBURY
BERKS
0635-30505

INTERFACE LTD
AMERSHAM
BUCKS
02403-22307

CASH REG SERVICE
CHESTER
CHES
0244-317549
```

And here is part of the sorted list:

```
A & G KNIGHT
ABERDEEN
SCOTLAND
0224-630526

ARDEN DATA PROC.
LEICESTER
LEICS
0533-22255

BCG EQUIP LTD
READING
BERKS
0734-54015

BCG SHOP EQUIP.
PAIGNTON
DEVON
0803-557711

BCS EQUIP LTD
BRISTOL
AVON
0272-425338

BITS & PCS
WETHERBY
YORKS
0937-63744
```

With an indexing density of 5, the response time was 5 seconds. With a density of 25 (every 25'th dealer indexed) the response time rose to 10 seconds, and so on.

Here is an example of how the program responded to the query key "NEW":

```
Query key?NEW


ITEM FOUND...


NEWBEAR LTD
CHEADLE HEATH
STOCKPORT
061-491-2290


Is this the item (Y/N)?
```

Continuing....

This is an interesting example since there two Newbear stores so, having found a match in one block, the system is now continuing to examine subsequent blocks in the threaded binary tree.

Conclusion

This is the most advanced method in this chapter. In its present form, the sorting module is VERY slow (but dependable). You can speed it up appreciably by outputting blocks to T$ only when both pointers are positive, as they will never change subsequently. This avoids frequent disk accesses to a rather slow device.

I hope that this chapter - indeed the whole book - has given you food for thought. If I have one message, it is this: "There is never a single answer to a computing problem. Computer science is common sense, and your answers are probably as good as mine!".

Further Reading on Disk Files

1. For a general background, read **Successful Software for Small Computers** by G. Beech (Sigma Technical Press, 1980), section C2: "Data Files"

2. For a quick review, try "How to decide on a suitable file organisation method", by M. Collier in **Practical Computing** July 1979, p44

3. The hashing method is described in "Hashing revisited", by R. T. Vizzone in **Microcomputing** May 1980, p78

4. A good article on linked lists is (despite the title), "Saving disk space with control programs", by B. White in **Practical Computing**, December 1979, p91

INDEX

ABOUT THIS BOOK

The Sharp MZ-80K is a very popular personal computer, offering excellent value for money. This book enables users of this - and other machines - to:

* Produce computer games and other graphics software
* Understand how to store and process data on cassette or disk.

These techniques are seldom covered in standard programming books.

Graham Beech is the author of several books, and is the owner of Sigma Technical Press - a fast growing publisher in the computer software area.

About our other books

Living with the Micro, by M Banks	£4.50
Successful Software for Small Computers, by G Beech	£5.95
Computer Programs that work, by J D Lee, (3rd Edn) G Beech and J D Lee	£4.95
BYTEING DEEPER INTO YOUR ZX81 by M Harrison	£4.95
PRACTICAL PROGRAMS: for the BBC Computer and Acorn Atom by D Johnson-Davis (December 1981)	£5.95
PRACTICAL PASCAL FOR MICROCOMPUTERS by R Graham (January 1982)	£6.50
SHARP SOFTWARE TECHNIQUES: Programming the MZ-80K by D Trowsdale (February 1982)	£5.95
THE BROADWATER ECONOMICS SIMULATIONS (Software package) by G Addis (January 1982)	£25.00
CP/M- the Software Bus: by A Clarke & D Powys-Lybbe (April 1982)	£8.50
Z80 Instant Programs - SECOND, REVISED EDN by J Hopton (February 1982)	£7.50

Sigma Technical Press
5 Alton Road Wilmslow
Cheshire SK9 5DY UK

ISBN: 0 905104 15 3