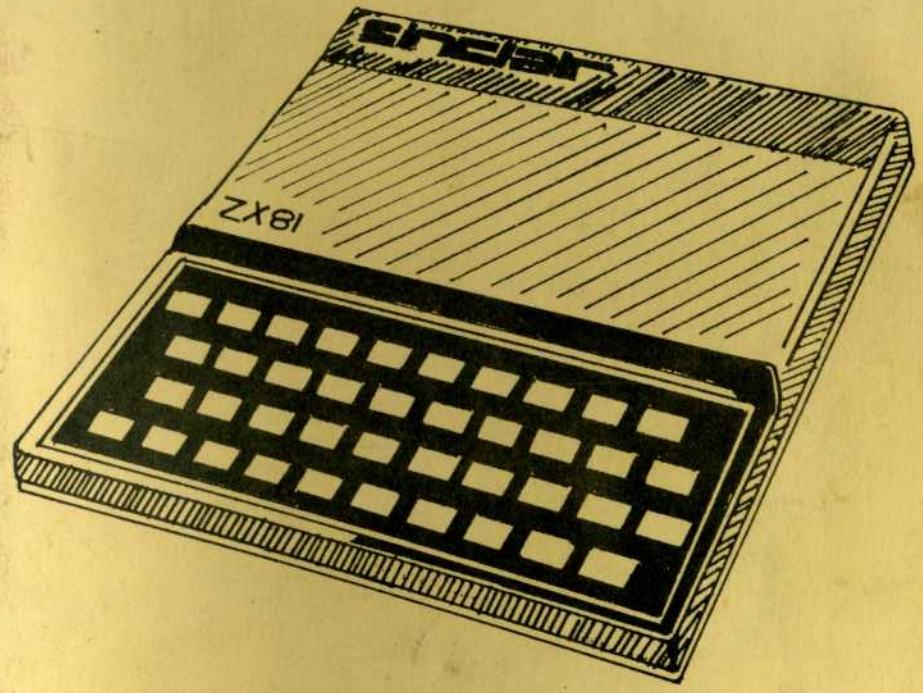
HINTS & TIPS for the



ZX81

HC

HINTS & TIPS

for the

ZX81

Andrew D Hewson

Copyright © Hewson Consultants 1981 All rights reserved

PREFACE

Following the runaway success of the Sinclair ZX80 released in April 1980 the manufacturers have brought out their 1981 computer - the Sinclair ZX81. The new machine is more sophisticated than its predecessor in that it can provide a continuous "no flicker" display, uses floating point arithmetic and carries a larger range of arithmetic, logical and display instructions.

This book is also based on a previous product * and contains many improvements over its predecessor having been entirely rewritten for the new machine. Some of the techniques described previously such as those to SCROLL the display or to generate an ACTIVE DISPLAY have been made superflouous. Others such as the method of protecting variables selectively cannot be implemented on the ZX81. However, all the material in the book has been tried, developed and tested on the ZX81 to make sure that it is both correct and useful.

An unexpanded ZX81 contains only 1K bytes of RAM of which about a fifth is used by the system in one form or another. Even with the add-on memory the user has only 16K bytes of RAM. A major aim of this book is to show how the limited space available can best be exploited. All the programs in the book run on the 1K machine although in many cases variations are described for use on the 16K machine.

The ZX81 Operating Manual supplied with each machine is much more comprehensive than its ZX80 companion and so a lot of the more straightforward

* HINTS & TIPS for the ZX80 by A D & J S Hewson

material in HINTS & TIPS for the ZX80 has been omitted. This has enabled the chapter on machine code programs to be much expanded in response to the requests of reviewers and readers alike,

The ZX81, like the ZX80, is a cheap and remarkably sophisticated microcomputer which brings the joys and frustrations of home computing within the reach of Everyman. Sadly, the modus operandi of a computer is as unreal to the uninitiated as are the shadows in a darkened room. If this book can help to kindle the torch of understanding in the minds of its readers it will have served its purpose. If it can lead the reader towards better, more enjoyable, more entertaining use of his ZX81 then its author will be a happy man.

Andrew D Hewson

Blewbury, June 1981

CONTENTS

SAVING SPACE

Economical use of memory is an important consideration when writing programs for large computers. It is of vital importance when writing programs for the ZX81 which in its minimum configuration has only 1K of RAM, that is only 1024 bytes or memory locations, available to the user.

The obvious way to save space is to write all or part of a program in machine code and call the machine code into use by using the USR instruction. This approach can lead to a great saving in space if the job you want the computer to do is not conveniently programmable in BASIC. For example you may wish to move material around in RAM perhaps altering the form of the display. A BASIC program of this nature would probably require many PEEKs and POKEs. A POKE instruction typically requires as many as 30 bytes in which to store it whereas the equivalent machine code requires about half a dozen.

Not all tasks are better written in machine code. For example if you want to multiply two variables A and B together it is sensible to use something like

```
LET A = A * B
```

thus letting the BASIC interpreter translate and execute the task. You could instead call your own machine code routine which in turn called the appropriate routines in ROM but you would use more of your valuable space in RAM in so doing.

* * * TIP * * *
If the task you wish to program is not conveniently programmable in BASIC try writing a machine code program to do it instead.

Unfortunately writing lengthy programs in machine code is rather like solving The Times crossword. It is a pleasant intellectual challenge once you get the hang of it but until that time it can be a time consuming and frustrating business. As in all things practice makes perfect. The chapter on Machine Code Programs introduces the main concepts and so the remainder of this chapter is devoted to streamlining BASIC programs.

COUNTING THE LENGTH OF PROGRAM LINES

The first step towards saving space is to develop an awareness of how much space your program is consuming in RAM. You will have learnt from the Operating Manual, page 171, that the RAM is divided into different areas i.e. system variables area, program area, display file etc and that these areas vary in size depending on the requirements of the program at each stage of its execution. Clearly the more economical of space your program lines are, the more lines you will be able to squeeze in.

The length of a program line in the program area is not the same as its length in characters when it is listed on your TV screen. For example there are 11 characters and spaces in the line :

```
9999 GOTO 1
```

and so the line occupies 11 bytes when listed in the display file plus another, hidden byte containing the newline code, 118 making 12 in all. In the program area the line occupies 13 bytes as you can see by entering NEW and

then loading and running the following program :

```
10    FOR I=1 TO 13
20    PRINT PEEK(16572+I);",";
30    NEXT I
40    STOP
9999  GOTO 1
```

Lines 10 to 40 print the contents of addresses 16573 to 16585 inclusive which contain 9999 GOTO 1 as follows :

```
39,15,9,0,236,29,126,129,0,0,0,0
,118,
```

The first two bytes hold the line number 9999= 39*256+15 and the next two show that the line uses a further 9+0*256=9 bytes. The fifth byte contains 236 - the token for GOTO. Byte numbers 6 to 12 hold two different representations of 1. The first is the character form with code number 29; the second is the numeric form 129,0,0,0,0 and there is a byte in between containing 126. The line is terminated by the newline code, 118.

Hence you can see that program lines are held in the program area in a rather complicated form. You can work out program line lengths by remembering that :
(a) the first two bytes contain the line number;
(b) the next two bytes hold the length in bytes of the remainder of the line;
(c) keywords, symbols and tokens are each stored in one byte;
(d) numbers are stored twice separated by a byte holding 126, first they are stored digit by digit with one digit per byte and then they are stored in their five byte numeric form.

If you are unsure of the length of a program line then use lines 10 to 40 of the routine above to print the

line out. Your line should follow immediately after the routine and it may be necessary to increase the maximum value of I from 13 to some higher value if your line is a long one.

The CONTENTS OF RAM program on page 61 will give you more insight into the structure of the program area.

The use of numerical constants in programs should be avoided wherever possible to save space because six "hidden" bytes, which do not appear in the program listing, are used to hold the separator code (126) and the five byte numeric form as explained above. For example if a 3 digit number is to be used more than twice in a program a variable should be declared to hold the value rather than including the number explicitly in the program. Suppose you want to convert three decimal fractions held in A,B,C to percentages by multiplying by 100 then the obvious routine :

```
500   LET A=A*100          19
510   LET B=B*100          19
520   LET C=C*100          19
```

takes up more space, including the six bytes required in the variables area to hold the value of Z, than :

```
490   LET Z=100            17
500   LET A=A*Z            11
510   LET B=B*Z            11
520   LET C=C*Z            11
```

The number of bytes required by each line is shown in the column to the right.

* * * TIP * * *

Avoid using numerical constants wherever possible.

It is clear that the length of the program when

listed on the screen is not an infallible guide to the size of the program area required to hold it. There are always several forms which a program designed for a given purpose can take and it is worth trying different versions during program development and keeping a note of the address of the end of the program area. This address is one less than the value held in the system variable, D-FILE, and it can be obtained by entering

```
PRINT PEEK 16396 + 256*PEEK 16397 -1
```

from the keyboard. Alternatively the following routine can be kept at the end of the program area

```
9998 STOP
9999 PRINT PEEK 16396 + 256*PEEK 16397 -57
```

and called from the keyboard by entering

```
GOTO 9999        or        RUN 9999
```

The routine allows for the 56 bytes used to hold the routine itself.

* * * TIP * * *

Write different versions of the same program and choose the one which is most compact.

Of course different versions will make varying demands on the display file, the variables area and the stacks. The user can do little to control the size of the calculator and machine stacks but careful programming can limit the size of the GOSUB stack. Each GOSUB call puts a two byte return address on the GOSUB stack which is then cleared when a RETURN is made and therefore several GOSUBs without intervening RETURNs are more wasteful of space than are alternating GOSUBs and RETURNs. It should also be noted that short subroutines are uneconomical because each pair of GOSUB and RETURN program lines use at least 19 bytes.

* * * TIP * * *
Do not nest subroutines unnecessarily. Avoid short subroutines.

As an example of what can be achieved by tailoring a program to suit the ZX81 look at the STATISTICS program on page . This program uses only one numerical constant explicitly - the number 999 in line 60. At all other points where numerical constants might be used either the constant is constructed from other variables with known values or the CODE function is used. For example at line 2,J is set equal to one not by declaring one explicitly - requiring seven bytes but by using CODE "■" which only requires four bytes. Similarly at line 4, A is set to zero by equating it to J-J using just three bytes rather than the seven bytes required by numeric zero. Notice also the use of a carefully constructed GOTO calculation at line 32 requiring seven bytes for the body of the line. The more obvious form :

IF Z = 55 THEN GOTO 36

uses 21 bytes.

* * * TIP * * *
Use the CODE function to generate integer constants.

There are occasions such as when performing complex, one line numerical calculations when the use of the CODE function is less economical of space than the usual alternative because many more intermediate calculations become necessary. Intermediate results are stored in the stack and therefore it is worth checking on the value of the STKEND system variable after testing a routine to see how big a stack it used. This check can be made from the keyboard by entering

PRINT PEEK 16412 + 256*PEEK 16413

or by loading and running


```
9998 STOP
9999 PRINT PEEK 16412 + 256*PEEK 16413
```

The value returned depends not only on the size of the calculator stack but also on the size of the program area, the display file and the variables area.

MAKING FULL USE OF VARIABLES

The diagrams on pages 172 - 174 of the Operating Manual show how the different types of variables are stored in the variables area of the RAM. Six types are recognised by the BASIC interpreter and their characteristics are shown in the table overleaf.

In the table the column headed "program area length" gives the number of locations occupied by a reference to the variable in the program area. Similarly the column headed "variable area length" gives the number of locations required to store the variable in the variables area. The "example stored as" column shows what the example appears as in the variables area.

Numeric variables with multiple character names possess no redeeming features to excuse the extra space which they consume in both the variables area and in the program areaand so they should not be used if space is at a premium.

* * * TIP * * *
Use single character names to save space.

Of course every effort should be made to reuse variables and avoid declaring new ones but sometimes this effort can be counterproductive. The interpreter allows numeric variables to be reused as loop controlling

VARIABLE TYPE	PROGRAM AREA LENGTH	VARIABLE AREA LENGTH	EXAMPLE	EXAMPLE STORED AS
Numeric (single character name)	1	6	LET A=15	102,132,112,0,0,0
Numeric (multiple character name)	Name length	5+name length	LET TOTAL=15	185,52,57,38,177,132,112, 0,0,0
Numeric array	3+subscript symbol lengths	4+2*number of dimensions+5* number of elements	DIM B(2,2)	135,25,0,2,2,0,2,0,0,0,0, 0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0
Control variable of a FOR-NEXT loop	1	18	10 FOR J=1 TO 7	239,132,0,0,0,0,131,96,0, 0,0,129,0,0,0,0,11,0
String	2	3+string length	LET Z$="GO"	95,0,2,44,52
Character array	4+subscript symbol lengths	4+2*number of dimensions+number of elements	DIM A$(2,2)	198,9,0,2,2,0,2,0,0,0,0,0

Table of variable types and the number of bytes required to hold them in the program and variables area.

variables and vice versa but once a variable has been used to control a loop it retains 18 locations in the variables area for the remainder of the program. In most programs it is more economical to set aside certain variables to control loops and to use no others for this purpose.

* * * TIP * * *

Think carefully before reusing a numeric variable to control a loop

PRINTING TRICKS

It is explained on page 172 of the Operating Manual that when less than 3¼K of RAM is available the size of the display file depends on the amount of material written to it. Therefore much space can be saved when using the 1K machine simply by keeping the amount of output to a minimum. PRINT statements occupy space in both the program area and eventually in the display and so it makes sense to omit or truncate long messages whenever possible. You should aim to write comprehensive instructions on running your programs instead.

* * * TIP * * *

Keep PRINTed messages short.

Clearing the display causes the display file to shrink to its minimum size (on the 1K machine) and so using CLS frequently is a good idea when space is at a premium. Each line of the display is constructed from left to right and so displaying material on the left hand side of the screen rather than in the middle or on the right will save considerable space.

* * * TIP * * *

Make frequent use of CLS. Construct displays on the left hand side of the screen.

All letters, digits, punctuation marks and graphics characters are accessible directly from the keyboard as explained on page 77 of the Operating Manual and can be incorporated directly into PRINT or REM statements. It is also possible to use tokens in these statements either by POKEing them into place or by careful manipulation of the ZX81 EDIT facilities.

For example suppose you wish to enter the line :

```
10    PRINT "INPUT THE NEXT VALUE"
```

This line would normally occupy 26 bytes in the program area. However the words INPUT and NEXT can be held as tokens (code numbers 238 and 243) if the PRINT statement is constructed as follows :

1) Enter `1  PRINT "ZTHEZVALUE"`
2) Enter `POKE 16513, 238`
3) Enter `POKE 16517, 243`
4) Edit line 1 to become line 10 in the usual way.

Using line 1 ensures that the line is the first line in the program and so the position of the Z markers can be worked out by remembering that the program area starts at 16509. The Zs are used to reserve a byte to hold the tokens. Notice that the interpreter expands the tokens to their full form both in the program and, when the program is RUN, in the display.

A more elegant method of inserting the tokens is to work from right to left manipulating the facilities of the BASIC system. Enter the line as follows :


```
1;0;NEXT;V;A;L;U;E;";b;b;b;b;b;b;b;
INPUT;T;H;E;b;b;b;b;PRINT;";
```

Here the semicolon separates the keystrokes and the lower case b represents a backspace. Notice that NEXT, INPUT and PRINT are entered in one keystroke ie. in their token form.

CLEAR

The CLEAR causes all current variables to be deleted thereby releasing space for new variables or for an enlarged display file. If used judiciously the command can be very useful but unfortunately it is not possible to delete some variables and keep others with this command.

A simple technique can be used to obviate this restriction if the value or values to be retained is an integer between 0 and 255. The following routine illustrates the technique which consists of POKEing the value into the display file for later recovery.

```
10    SLOW
20    PRINT "NOW"
30    PAUSE 50
40    LET J=60
50    GOSUB 130
60    POKE W,J
70    CLEAR
80    GOSUB 130
90    LET J=PEEK W
100   PRINT
110   PRINT "J = ";J
120   STOP
130   LET W=1+PEEK 16396 + 256*PEEK 16397
140   RETURN
```

The value of J is set at line 40, POKEd into the display file at line 60, CLEARed at line 70, retrieved from the

display file at line 90 and PRINTed at line 110. The subroutine at line 130 calculates the address of the first byte of the display after the initial newline character. This byte is set inially to 51 (the code for N in NOW) and is changed to 60 (the code for W) at line 60.

The routine is cumbersome because the address of the display file is determined and then the value is lost when the variables are CLEARed and has to be determined again. An alternative routine which uses less space is as follows :

```
10   SLOW
20   PRINT "NOW"
30   PAUSE 50
40   LET J=60
50   POKE (1+PEEK 16396+256*PEEK 16397),J
60   CLEAR
70   LET J=PEEK (1+PEEK 16396+256*PEEK 16397)
80   PRINT
90   PRINT "J = ";J
```

A second technique which works for any variable is to copy the appropriate part of the variables area into the display file as shown by this routine :

```
10   PRINT "THIS STATEMENT RESERVES SOME
SPACE IN THE DISPLAY FILE"
20   LET A$="STRING TO BE KEPT"
30   GOSUB 200
40   FOR I=0 TO 16
50   POKE U+I, PEEK (V+I)
60   NEXT I
70   CLEAR
80   LET A$="WILL BE WIPED OUT"
90   GOSUB 200
100  FOR I=0 TO 16
110  POKE V+I, PEEK (U+I)
```



```
120  NEXT I
130  PRINT A$
140  STOP
200  LET U=1+PEEK 16396 + 256*PEEK 16397
210  LET V=3+PEEK 16400 + 256*PEEK 16401
220  RETURN
```

A$ is declared at line 20, transferred byte by byte to the display file in lines 40 to 60, CLEARed at line 70, retrieved in lines 100 to 120 and PRINTed at line 130. The subroutine at line 200 calculates the address of the beginning of the display file and the address of the beginning of A$ in the variables area allowing one byte for the identifying letter, A, and two bytes to show its length.

Transferring data into and out of the display file needs very careful consideration because it is all too easy to overwrite one of the newline characters on which the compute and display software depends for its functioning. A detailed study of the display file and methods by which it can be manipulated is described in the next chapter.

UNDERSTANDING THE DISPLAY FILE

All computers have so-called peripheral devices attached to them through which they receive information from, and transmit information to, the outside world. The ZX81 can receive information via the keyboard and the cassette "ear" socket and can transmit information via the cassette "mic" socket and the TV socket. This chapter is concerned with the way in which the contents of some locations in RAM are copied, marshalled and passed by the Z80 processor to the UHF modulator which superimposes a TV signal and passes the resultant to the TV socket.

The area of RAM which is copied to the TV screen is called the display file and the addresses of the first and last byte of this area are held in the D-FILE and VARS system variables respectively. When there is no program running the BASIC interpreter copies the contents of all or part of the program area into the display file, converting line numbers from their two byte integer form into character codes, expanding tokens into keywords and ignoring 5 byte numeric values following 126 numeric separators. Thus the display file only contains character codes in the range 0 to 63, the newline code 118, or in the range 128 to 191. If any other codes are encountered in material to be displayed the interpreter converts them to the appropriate character form.

For example suppose the token for INPUT is put into a PRINT statement using the technique described in the previous chapter ie. by entering :

```
1;0;INPUT;";b;b;PRINT;";
```

where a semicolon separates keystrokes and b represents backspace. The following routine shows that INPUT is

held in token form in the program area (represented by decimal code 238) whereas it is spelt out in characters in the display file (codes 0,46,51,53,58,57,0).

```
10  PRINT " INPUT "
20  PRINT "THE CODE IN THE PROGRAM AREA
IS "; PEEK 16515
30  PRINT AT 3,0;"THE CODES IN THE DISPLAY
FILE ARE ";
40  LET I=1 + PEEK 16396 + 256*PEEK 16397
50  FOR J=I TO I+6
60  PRINT PEEK J;",";
70  NEXT J
```

Notice that the interpreter puts spaces (ie. bytes holding zero) on either side of the expanded form to make the display of the program more readable.

In the 1K machine a blank display consists of 25 bytes containing 118 that is an initial 118 and 24 empty lines terminated by 118. This can be seen by entering and running the following routine :

```
10  LET I = PEEK 16396 + 256*PEEK 16397
20  LET J = PEEK 16400 + PEEK 16397 * 256
30  DIM B(J-I)
40  FOR K=1 TO J-I
50  LET B(K) = PEEK (K+I-1)
60  NEXT K
70  PRINT "ADDRESS OF FIRST BYTE IS ";I
80  PRINT "ADDRESS OF FINAL BYTE IS ";J-1
90  FOR K=1 TO J-I
100 PRINT B(K),
110 NEXT I
```

If $3\frac{1}{4}$K or more RAM is available the ZX81 "pads out" the display file using blank bytes containing decimal zero to its maximum size of 24 x 32 displayed bytes plus 25 bytes containing newline codes making 793 bytes in total. The program above can be used to look at the full

size file by adding the line :

```
105   IF K > 38 AND K=2* INT (K/2) THEN SCROLL
```

but be warned that the program takes about 2 minutes to run.

If the contents of the display file are PRINTed immediately rather than being stored temporarily in an array as in the above program then it is the decimal codes of the characters previously PRINTed which are displayed as shown by the following routine :

```
10    LET I= PEEK 16396 + 256* PEEK 16397
20    PRINT "DISPLAY FILE STARTS AT ";I
30    FOR J=I-10 TO I
40    PRINT J, PEEK J
50    NEXT J
```

This program generates the following display :

```
DISPLAY FILE STARTS AT 16630
16620      211
16621      47
16622      118
16623      0
16624      50
16625      3
16626      0
16627      243
16628      47
16629      118
16630      118
16631      41
16632      46
16633      56
16634      53
16635      49
```

Bytes 16623 to 16629 in the program area hold the last program line viz. 50 NEXT J. The 118 which terminates the program line is followed by another 118 - the first

character of the display. Bytes 16631 to 16635 contain the codes for D,I,S,P,L - the first five characters at the top of the screen.

CLEARING A PART OF THE DISPLAY

The compute and display software picks up the value of D-FILE and passes all data from that address onwards between the first and the twentyfifth newline character to the modulator. The CLS function complements this action by counting down twentyfive newline codes from the end of the display file (ie. from the address held in VARS). When it finds the twentyfifth it counts back up twentyfour addresses, inserting 118 into each byte and then copies the variables area down the RAM and resets VARS and other system pointers. Therefore to retain the first, say, three lines of the display and clear the remainder it is only necessary to POKE in three extra newline characters, clear the screen (CLS) and POKE the three bytes back to their original state. The following program illustrates the principle by retaining the first line :

```
10    LET I = 25 + PEEK 16396 + 256*PEEK 16397
20    PRINT "HC^^^^^ THIS IS A HEADING*"
30    PRINT AT 3,2;"THIS MESSAGE WILL BE
CLEARED"
40    PRINT AT 5,7;"WHEN YOU HIT A KEY"
50    PAUSE 32768
60    POKE I,118
70    CLS
80    PAUSE 100
90    PRINT AT 3,9;"NOW HIT A KEY"
100   PRINT AT 5,4;"TO CLEAR THE HEADING TOO"
110   PAUSE 32768
120   POKE I,28
130   CLS
```

A byte is reserved for the extra newline character - in this example an asterisk is added at the end of line 20 for clarity but a blank, or any other character could be used. The asterisk is the twentyfifth character in the PRINT statement (including blanks) and so I is set to point to its address by adding 25 to the value held in D-FILE.

If line 120 is omitted the display file will continue to contain twentysix newline characters and the heading will remain and appear in all susequent program listings and displays. Such a situation can be retrieved by entering POKE I,28 from the keyboard.

Unfortunately it is not possible to include newline characters directly in PRINT statements because they cause weird effects in program listings and the BASIC interpreter refuses to execute the statements. Manipulating the newline characters is a hazardous business particularly if the display file contains less than twentyfive of them as a result. This can cause crazy displays at best and at worst the ZX81 ceases to function and must be turned off. Hence it is a good idea to decide on the manipulations which you wish to make using a small test program which can be easily modified and reloaded and then to add the remainder of the program when you are satisfied that your routine will not fail.

PRINT AT TAB PLOT AND UNPLOT

When executing these instructions the 1K ZX81 enlarges the display file by the minimum amount to display at the requested location. Lines are constructed from left to right and so for example to

```
PRINT AT 0,5;
```


the ZX81 moves the display up the RAM and puts five bytes containing zero after the first newline character. When TABbing to the next line it terminates the current line immediately. Similarly

```
PRINT 2.5,2.5
```

will result in 20 bytes containing :

30,27,33,0,0,0,0,0,0,0,0,0,0,0,0,0,30,27,33,118

If a comma is placed after the second number as in

```
PRINT 2.5,2.5,
```

one extra byte will be allocated :

30,27,33,0,0,0,0,0,0,0,0,0,0,0,0,0,30,27,33,0,118

In general each display causes the display file to grow (on the 1K machine) and there is no mechanism for eliminating blank lines. The following routine demonstrates how the display remains enlarged even when there is no need for it to remain so :

```
10   LET I=1+PEEK 16396 + 256*PEEK 16397
20   PRINT 2.5,2.5,
30   GOSUB 200
40   PAUSE 500
50   PRINT AT 0,0;"20 blank characters"
60   GOSUB 200
70   STOP
200  FOR J=I TO I+20
210  PRINT PEEK J;",";
220  NEXT J
230  RETURN
```

Line 50 overwrites the top line of the display with blanks (decimal zero) and for all the difference it would make to the display, the ZX81 could discard the contents of these bytes and shorten the display file appropriately. Unfortunately it does not do so. This makes it difficult to write programs for the 1K machine which make frequent changes to the display because the display file grows to an unacceptable size.

CONVERTING ZX80 PROGRAMS

The design of the ZX81 bears more than a passing resemblance in both hardware and software to that of its progenitor, the ZX80. In the relatively short period for which the ZX80 was available a large number of programs were written and published and many of these programs can be converted to run on the ZX81. The purpose of this chapter is to outline the similarities in the BASIC syntax of the two machines and to draw attention to the differences so that the ZX81 owner can tackle the conversion of ZX80 programs with confidence.

SIMILARITIES BETWEEN THE ZX80 AND THE ZX81

1) Both machines are designed around a Z80A microprocessor and have a separate ROM containing the code for translating BASIC programs into a form which the Z80A can understand.
2) RAM is divided into system variables area, program area, display file, variables area and stack. The addresses of the boundaries between the area are held in the system variables area.
3) The size of each area is altered automatically both whilst a program is being entered or editted and whilst a program is running so as to make the maximum use of the limited RAM available.
4) If there are no more pressing tasks the Z80A passes the current contents of the display file to the TV 50 times per second.
5) The maximum display is 32 columns by 24 lines. The bottom two lines are not available to the user. The size of the display file is never larger than it need be except in the 16K ZX81. The display is not scrolled.
6) The program and current values of all variables can

be read to and from cassette. There are no DATA, READ or RESTORE commands.
7) The graphics characters are identical although their codes are not the same. The ZX80 has no PLOT and UNPLOT commands.
8) Characters, keywords and tokens occupy one byte only in the program area of each machine but CHR$, PEEK, USR, STR$, CODE, ABS, TL$ and RND must be spelt out on the ZX80.

DIFFERENCES BETWEEN THE ZX80 AND THE ZX81

All the software facilities available on the ZX80 can be mimicked on the ZX81 but as a consequence of its greater sophistication the ZX81 is less economical of RAM than the ZX80. Hence careful use of space is even more important when using the ZX81 than when using the ZX80. The ZX81 use 85 bytes more in the systems variables area than does the ZX80 and it also requires a calculator stack and so the basic machine has about 10% less space available to the user. It also requires 5 bytes to hold the value of a variable (thus permitting the use of non-integer numbers) compared with the 2 bytes used by the integer-only ZX80. Constants in a program are held both in their character and in their numeric form so that an extra 6 bytes per constant are required compared to the ZX80. Finally two bytes per program line are used by the ZX81 to hold the length of that line and there is no equivalent usage by the ZX80. Hence it will often be necessary to prune a ZX80 program in order to load it into a ZX81.

Many more functions are available on the ZX81 than on the ZX80. All ZX80 functions have exact counterparts on the ZX81 except as follows. The TL$ ZX80 function is replaced by (2 TO) on the ZX81, thus:

ZX80	ZX81
LET A$=TL$(A$)	LET A$=A$(2 TO)

Array indices start at one on the ZX81 and at zero on the ZX80 and so

```
DIM B(5)
PRINT B(0)
```

is only meaningful on the ZX80. The easiest way of converting arrays is to add one to array subscripts at every occurrence.

If a ZX80 program makes use of integer division then this will need to be stated explicitly when converting it to run on the ZX81. Thus both the following :

ZX80	ZX81
IF J=2*(J/2)	IF J=2*INT(J/2)

will be true if J is even and false if J is odd.

The random number generator on the two machines is different. On the ZX80 RND(N) generates a random integer between 1 and N inclusive whereas on the ZX81 RND generates a random number in the range 0 to 1. The following statements are equivalent :

ZX80	ZX81
LET A=RND(N)	LET A=1+INT(N*RND)

CHARACTER CODES AND SYSTEM VARIABLES

The character codes used by the ZX81 are listed in Appendix A of the Operating Manual. The codes for both letters and numbers in both normal and inverse video are the same as those used by the ZX80 but the remainder are mostly different. The table opposite lists the ZX81 equivalents for the non-alphabetic or numeric codes used by the ZX80. Reference should be made to this table when converting CHR$ or CODE commands.

ZX80	ZX81	ZX80	ZX81	ZX80	ZX81
0	0	132	129	220	22
1	11	133	130	221	21
2	5	134	132	222	23
3	131	135	7	223	24
4	1	136	134	224	218
5	2	137	136	225	217
6	4	138	137	226	216
7	135	139	138	227	20
8	6	140	140	228	18
9	8	141	141	229	19
10	9	142	142	230	240
11	10	143	143	231	254
12	12	144	144	232	251
13	13	145	145	233	233
14	14	146	150	234	248
15	15	147	149	235	235
16	16	148	151	236	236
17	17	149	152	237	244
18	22	150	148	238	238
19	21	151	146	239	249
20	23	152	147	240	241
21	24	153	153	243	243
22	20	154	154	244	245
23	18	155	155	246	230
24	19	212	11	247	247
25	25	213	222	248	227
26	26	214	223	249	232
27	27	215	25	250	250
128	128	216	26	251	237
129	139	217	17	252	239
130	133	218	16	253	253
131	3	219	215	254	234

Table of ZX80 character codes and their ZX81 equivalents.

The ZX80 uses fewer system variables than the ZX81 and most of them have direct equivalents as listed in the table opposite. Note that the variables area occurs below the display file in the ZX80 but above it in the ZX81 and so conversion of a program which manipulates the VARS, D-FILE and DF-CC variables is particularly difficult. On the ZX80 the FRAMES counter is incremented 50 times per second whereas on the ZX81 it is decremented (except bit 15) and this difference should be taken into account during program conversion.

CONVERSION CHECKLIST

1) Add one to every array subscript.
2) Make appropriate alterations to statements containing CHR$, CODE, TL$ or RND.
3) Check for the use of integer division and make use of the INT function if necessary.
4) Check any PEEKs and POKEs of system variables paying particular attention to VARS, D-FILE and DF-CC.
5) Be ready to prune the program so as to squeeze it into the ZX81.

ZX80 address	No of bytes	ZX81 address	ZX81 name
16384	1	16384	ERR-NR
16385	1		
16386	2	16391	PPC
16388	2		
16390	2	16394	E-PPC
16392	2	16400	VARS
16394	2	16404	E-LINE
16396	2	16396	D-FILE
16398	2	16398	DF-CC
16400	2	16400	VARS
16402	1	16418	DF-SZ
16403	2		
16405	2	16408	X-PTR
16407	2	16427	OLDPPC
16409	1		
16410	2	16432	T-ADDR
16412	2	16434	SEED
16414	2	16436	FRAMES
16416	2		
16418	2		
16420	1	16441	S-POSN
16421	1	16442	
16422	2		

Table of ZX80 system variables and their ZX81 equivalents.

CHAINING PROGRAMS

A common requirement in the commercial world is for a computer to examine, manipulate and update data using several different programs. Suppose for example a company maintains computer records of its sales of various products. The company might require one program to update the records day by day or week by week, another program to calculate income from sales, another to decide what new parts and spares need to be made to maintain stocks and perhaps another to show which products are selling well. In order for the computer to function in this manner the data and the programs must be stored separately. Very often the programs need to be able to call one another in succession. All that the user need do is to load a program into RAM and run it and then the program looks after itself.

There are no facilities immediately available on the ZX81 to allow it to work in this manner but with a little thought and effort the programmer can provide them for his particular application. This chapter describes the basic technique for passing data between programs, calling subroutines from cassette and establishing data files.

PASSING DATA BETWEEN PROGRAMS

Suppose there is a program in RAM and that the first line of the program declares an array B of size 10. How can the values in the array be passed to another program when the first has completed its tasks ? One tedious and time-consuming method would be for the first program to display the values for the user to write

down and then pass to the second program via the keyboard. A much more elegant method is to establish a vacant area above RAMTOP before loading the first program and for the program itself to copy the data into the area. The second program declares a suitable amount of space in the variables area and then copies the data back from above RAMTOP into the variables area.

It is necessary to use this rather circuitous method because the location of the variables area in RAM is dependent on the length of the current program and in general two programs use different locations for the variables area. The second program could attempt to move data from the old variables area directly to the new area by PEEKing and POKEing the appropriate locations but it would necessarily use the calculator stack in so doing. The calculator stack wanders up and down RAM and so in most cases the old variables area would be overwritten before the data had been safely retrieved.

In contrast once an area has been set aside above RAMTOP the BASIC system will not write to it even during LOAD operations and therefore there is no risk of inadvertently corrupting the data. An area is declared as described on page 168 of the Operating Manual by POKEing new values into locations 16388 and 16389 and entering NEW. On power up these locations contain 0, 68 (1K machine) or 0, 128 (16K machine). The following equations can be used to calculate the values to be POKEd to reserve a given number of bytes, say N, for the area :

	16388	16389
1K	256*(1+INT(N/256))-N	68-INT(N/256)
16K	256*(1+INT(N/256))-N	128-INT(N/256)

The following pair of programs illustrate the method. Before loading the first program move RAMTOP down 256 locations by POKEing 67 into 16389 (127 on the 16K machine). It is not necessary to reset 16388 because it will already contain the required value ie 0.

```
10    DIM B(10)
20    FOR I=1 TO 10
30    LET B(I)=I
40    NEXT I
50    LET J= PEEK 16400 + 256*PEEK 16401
60    LET K= PEEK 16388 + 256*PEEK 16389
70    FOR I=0 TO 55
80    POKE K+I, PEEK (J+I)
90    NEXT I
100   PRINT "POSITION CASSETTE BEFORE CHAIN1"
110   PRINT "AND THEN HIT A KEY"
120   PAUSE 32768
130   LOAD "CHAIN1"
```

The second program is called CHAIN1 and must be SAVEd on cassette before the first is loaded into the ZX81.

```
10    DIM B(10)
20    LET J=PEEK 16400 + 256*PEEK 16401
30    LET K=PEEK 16388 + 256*PEEK 16389
40    FOR I=0 TO 55
50    POKE J+I, PEEK (K+I)
60    NEXT I
70    FOR I=1 TO 10
80    PRINT B(I)
90    NEXT I
```

Both programs put the value of VARS into J and RAMTOP into K and copy 56 bytes to or from the variables area in a loop because a quick calculation using the table in the chapter on Saving Space (page 8) shows that a one dimensional array of length ten occupies 56 bytes in the variables area.

Notice that the array is declared in both programs before all other variables to ensure that it lies at the bottom of the variables area. The first program puts the numbers 1 to 10 into B at lines 20 to 40. CHAIN1 does not give B any values and yet at lines 70 to 90 the values loaded by the first program are PRINTed. This is the proof that the data has been successfully transferred.

Here is a checklist to use when designing programs to exchange data :

1) Move RAMTOP and enter NEW before loading any programs either from the keyboard or from cassette.
2) Decide what variables you wish to exchange and calculate the number of bytes, N, that they require in the program area.
3) Declare the variables at the beginning of every program.
4) Copy N bytes to and/or from the variables area.

CALLING SUBROUTINES FROM CASSETTE

The program CHAIN1 above must be RUN by the user after it is LOADed. If the following line is added to the program :

```
5     SAVE "CHAIN1"
```

and a new copy is SAVEd on cassette by entering RUN the new copy will carry on from line 10 as soon as it is reLOADed. Thus the first program can pass control to CHAIN1 without any user intervention except to rewind the cassette if required. The same technique can be used to pass control back again as illustrated by the following pair of programs :

```
10    SAVE "BORG"
20    PRINT "THIS IS BORG"
30    PRINT "POSITION CASSETTE BEFORE MCENROE"
40    PRINT "AND HIT A KEY"
50    PAUSE 32768
60    LOAD "MCENROE"

10    SAVE "MCENROE"
20    PRINT "THIS IS MCENROE"
30    PRINT "POSITION CASSETTE BEFORE BORG"
40    PRINT "AND HIT A KEY"
50    PAUSE 32768
60    LOAD "BORG"
```

These programs will pass control from one to the other as long as the user is prepared to manipulate the cassette. Note that copies must be SAVEd on cassette by the programs themselves. The technique will not work if the programs are SAVEd manually by entering SAVE from the keyboard.

The LOAD command can also be used with character variables and so

```
60    LET A$="BORG"
70    LOAD A$
```

is legal. Hence if a program passes its own name (plus other data if required) to another the second program can recall the first by name. This effectively makes the second a subroutine of the first although care must be taken to see that re-entry is made at the correct point.

ESTABLISHING DATA FILES

The same technique can be used to make separate data files on cassette if each program which requires access to the data includes the following routine :


```
1000  SAVE "DATAFILE"
1001  LET J= PEEK 16400 + 256*PEEK 16401
1002  LET K=PEEK 16388 + 256*PEEK 16389
1003  FOR I=0 TO M
1004  POKE K+I, PEEK (J+I)
1005  NEXT I
1006  RETURN
```

where M is set to the number of bytes in the file. To SAVE the current data on cassette in DATAFILE the program GOSUBs to this routine. To recover the data at a later date LOAD "DATAFILE" and the program automatically picks up the current values. More importantly the values are already transferred above RAMTOP so that another program can have access to them.

MACHINE CODE PROGRAMS

A full examination of the machine code of the Z80A microprocessor is beyond the scope of this book but this chapter is designed to introduce some basic concepts. The machine code instruction set of the Z80A is very extensive and so no attempt is made to describe every instruction in great detail rather the various classes of instructions are described together with an outline of their use. The chapter closes with a description of several programs which illustrate the use of machine code.

The chapter is best read after reading chapters 24, 25 and 26 of the ZX81 Operating Manual.

BITS, BYTES, ADDRESSES AND HEXADECIMAL

A bit is fundamental unit of computer memory and the Z80A is an 8 bit processor which means that it can operate on 8 bits at the same time. This compares with the 16, 32 and 36 bit processors typically used by large commercial computers. A bit can only be in one of two states and these states can be thought of as representing On or Off; True or False; Yes or No; Up or Down or any other pair of logically opposite conditions. The usual notation is to think of one state as the Zero state and the other as the One state. A bit is considered to be 'set' when it is in the state representing One and to be 'reset' otherwise. This notation allows us to speak of a given bit pattern in terms of its binary equivalent and by converting the binary equivalent to a decimal number we can give each bit pattern a unique positive integer decimal number.

For example consider 8 bits of which the rightmost

four are set and the four leftmost are reset. Such a bit pattern is written as

00001111

This binary number can be converted to decimal if it is remembered that, in a binary number, the rightmost column is the units column, the next rightmost is the twos column, the next rightmost is the fours column and so on doubling at each move to the left. The number above is

1*1+1*2+1*4+1*8+0*16+0*32+0*64+0*128 = 15

It is obviously inconvenient to refer to bits as 'the rightmost' or as 'the second from the left' and so they are numbered from the right starting at zero. In the above example bits 0 to 3 are set and bits 4 to 7 are reset.

A group of 8 bits is called a byte and because the Z80A usually operates on one byte at a time the memory elements in ROM and RAM are also grouped into bytes. The 1K of RAM available in the basic ZX81 consists of 1024 bytes of memory and the 16K available on the expanded ZX81 contains 16*1024 = 16384 bytes. 1024 and 16384 seem, to the uninitiated, to be strange numbers to use but the reason is quite straightforward and is explained below.

A computer does not alter the contents of memory directly when it is executing a program rather it copies the contents of a byte in memory into a register, alters the contents of the register according to its instructions (in computer parlance it 'operates' on the register contents), and then it copies the contents of the register back to memory. Naturally the computer must be able to identify each location in memory so that it can copy to and from the right locations. Thus each location in memory is given a unique address just as houses are

given unique addresses so that the postman knows which letters go where. Many Z80A instructions are of the form 'copy the contents of the following address into such-and-such a register'. The address following the instruction occupies two bytes and so the number of addresses which the processor can address uniquely is limited to the number of different addresses which can be held in two bytes. This number is the same as the number of different bit patterns which can be adopted by the 16 bits which make up the two address bytes. Remember that :

one bit can be in either one of 2^1 = 2 states
ie 1 or 0

two bits can be in any of 2^2 = 4 states
ie 00, 01, 10 or 11

three bits can be in any of 2^3 = 8 states
ie 000, 001, 010, 011, 100, 101, 110 or 111

It would be pointless to enumerate all the possible arrangements of 16 bits but by analogy with the examples above it can be seen that there are 2^{16} = 65536 such arrangements.

In the ZX81 only 15 of the 16 bits are used giving 2^{15} = 32768 addresses (0 to 32767 inclusive) and of these numbers 0 to 16383 are reserved for the 8K ROM plus any future expansion requirements. The remainder from 16384 to 32767 are available for user RAM.

The electronic circuitry uses the bit pattern for a particular address to set a series of switches thus selecting the right address. Assuming the address is in RAM then ten switches are required to select the right address out of 1024 possibilities because 2^{10} = 1024. Even if only one further byte of memory was available then one more switch would be required. Thus the manufacturers have provided the maximum amount of

memory consistent with the use of ten switches (in the unexpanded machine). The 16K add-on RAM pack contains 16384 bytes of memory and requires 14 switches to 'decode' its addresses because 2^{14} = 16384. 1024 is close enough to a thousand for it to be referred to as 1K for short where K stands for kilo meaning one thousand as in kilometre or kilogram. Multiples of 1024 are referred to as 2K, 3K etc.

A single byte can be in any one of 2^8 = 256 states ranging from 00000000 to 11111111. The binary form is inconveniently longwinded and whilst a decimal notation is used by the PEEK and POKE commands the general practice when using machine code is to use a two digit hexadecimal notation 00 to FF. The HEX LOADER/PRINTER (page 59) converts pairs of hexadecimal code characters to decimal (line 180) and vice versa (line 100). It should be noted that the ZX81 character code which interprets decimal 28 as 0 for example bears no relationship to the binary interpretation of bit patterns. Thus 00011100 would be read as 1*16+1*8+1*4 = 28 by the PEEK command but the same byte would be read as 0 by the CHR$ command. Its hexadecimal form is 1C.

THE Z80A REGISTERS

As was explained above a computer works by copying the contents of a memory location into one of its registers, operating on the register and copying the result back to memory. The Z80A is a powerful microprocessor because it has several registers and so it can hold several numbers at once thereby reducing the need to make time-consuming transfers between registers and memory. Most of the registers have one or more special features and before learning Z80A machine code it is necessary to become acquainted

with these features.

The most important register is the A register, sometimes known as the accumulator. Most of the arithmetic and logical instructions in the instruction set operate on the contents of A with some of these instructions requiring the use of a second register or memory address as a source of data. An example of an arithmetic instruction is add-n-to-A where n is a number between 0 and 255 (mnemonic ADD A,n). The instruction causes the contents of A to be increased by n. An example of a logical instruction is complement-A (mnemonic CPL) which causes every set bit in A to be reset and every reset bit to be set ie. those bits which were One become Zero and vice versa.

Most of the Z80A registers occur in pairs in the sense that some instructions operate on pairs of registers together. The F or flag register is paired with the A register in that sense. The F register is rather different from all the other registers because each of the 8 bits in it is used as a so-called 'flag'. A 'flag' indicates that one of two logically opposite events has occurred in much the same way that a red traffic-light means 'stop' and a green traffic-light means 'proceed'. The more important flags in the register and their meaning is as follows :

1) The sign flag is set when the result of the last operation was negative. (By convention the leftmost bit of a number is used to indicate its sign, the number being negative when the leftmost bit is set.)
2) The zero flag is set when the result of the last operation was zero.
3) The carry flag is set when the result of, for example, an addition is too large to be held in 8 bits.
4) The parity flag is set when a result is even and reset when it is odd.
5) The overflow flag is set when an operation affects the sign bit as a consequence of an overflow from other bits.

The effect of some instructions depends on the current setting of particular flags. Thus the flag register is used to control the program during execution. For example the relative-jump-on-zero-displacement-d (mnemonic JR Z,d) causes the Z80A to jump forward (or backward if d is negative) d instructions if the Zero flag is set. In other words JR Z,d functions in a similar fashion to the BASIC line

```
IF A = 0 THEN GOTO 99
```

The B register and to some extent the C register with which it is paired is available as a counter through the use of the decrement-jump-on-not-zero-displacement-d (mnemonic DJNZ,d) and the compare-decrement-repeat (mnemonic CPDR) instructions. The former causes the B register to be decreased by one and if the result is not zero the Z80A jumps d instructions. If the result is zero then the next instruction in the sequence is executed. If the programmer uses a negative value for d the jump goes back to earlier in the program and, assuming there are no other branches the processor will eventually execute the DJNZ,d instruction again. Thus the B register and the DJNZ,d instruction together provide a machine code facility which is similar to the FOR-NEXT loop in BASIC. For example if the B register is first set to ten the program can be made to loop through a set of instructions ten times by making the final instruction DJNZ,d with an appropriate value for d.

The compare-decrement-repeat instruction mentioned above uses the 16 bits of the BC register pair as a counter and at each cycle it

1) decrements BC by one,
2) compares the contents of A with an address in memory,
3) decrements the address in memory to be inspected next.
The instruction repeats these actions until either the contents of the memory address match those of A or BC=0. Thus this instruction can be used to search through memory for an address holding a particular number.

The D and E registers do not have any individual functions and are mostly used as temporary, rapidly accessible memory. The H and L registers are usually used as a pair to hold an address in memory (the compare-decrement-repeat instruction already described uses HL for this purpose). H stands for high address and L stands for low address and the address is held in the form

address = 256 * H + L

giving a maximum of 65536 unique addresses.

The IX and IY registers can only be used as 16 bit registers unlike B,C,D,E,H,L which can either be used individually as 8 bit registers or, with their pair as 16 bit registers. IX and IY individually can mostly be used in a similar fashion to the HL register pair although the instructions which drive them take one more byte of storage than the equivalent HL instruction. For example add-the contents-of BC-to the contents-of-HL (mnemonic ADD HL,BC) is a one byte instruction whereas the same instruction using IX instead of HL is held in two bytes. IX and IY have one further property which is not available with HL and that is that they can be used with a displacement,d. This means that an instruction which references (IX+d) uses not the memory location held in IX but rather the memory location d units further along.

The Z80A possesses two other accessible 16 bit registers - SP, the stack pointer, and PC, the program counter. The stack pointer holds, as its name suggests, the address in memory of the bottom of the stack. The stack is an area at or near the top of RAM which is available for the temporary storage of the contents of pairs of registers. It is designed to grow down the RAM as it is filled and to shrink back up the RAM as it is emptied. Transfers to and from the stack are made by means of the PUSH and POP instructions. For example the PUSH HL instruction causes the Z80Ato copy the current contents of H to the byte at the bottom of the stack decrement SP so that it points to the next lowest address, copy L to the stack and then decrement again. POP HL is the reverse of HL.

The program counter, PC, is a very important 16 bit register because it holds the address in memory of the next instruction to be executed. The normal flow of events when an instruction is executed is as follows :
1) Copy the contents of the address held in PC into yet another, hidden register.
2) If the instruction is held in several bytes increment PC and copy the contents of the new address pointed to into another hidden register. Repeat 2) until the full instruction has been copied into the processor.
3) Increment PC so that it points to the next instruction to be executed.
4) Execute the instruction which has just been copied in.
A jump instruction such as DJNZ,d or JR Z,d described above upsets the normal flow of events by altering PC at step 4). For each of these instructions the program counter is increased by the displacement d (or decreased if d is negative) so that the pointer indicates a new address from which to take the next instruction.

Note that this alteration occurs after PC has been incremented so that displacements should always be calculated relative to the instruction following the one containing the displacement.

Finally, the Z80A possesses duplicates of each of the A,B,C,D,E,H and L registers. The duplicates are referred to by using a prime for example A' is the duplicate A register. No instructions operate on these duplicates directly but exchange instructions are available to swop registers out of use and swop the duplicates into use. These exchange instructions are executed very rapidly because the contents are not physically copied from one register to the other but rather the prime register is used by subsequent instructions whilst the original register becomes dormant. For example the exchange-A-and-F-with-A'-and-F' instruction (mnemonic EX AF,AF') causes the A and F registers to become dormant and the A' and F' registers to come into use as the new A and F registers. Note that the ZX81 ROM software uses A' and F' whilst in the compute and display mode and so EX AF,AF' should only be used when the FAST command is operating (see page 167 of the Operating Manual).

ABOUT THE INSTRUCTION SET

There are more than 600 elements in the Z80A instruction set and most but not all of them are listed in Appendix A of the Operating Manual. As there are only 256 different arrangements of 8 bits (because $2^8 = 256$) less than half the instructions can be held in one byte. The remaining instructions are held in two or even three bytes. Some instructions are followed by a one byte displacement, d, or a number, n, or a two byte address, nn, to which the instruction refers. In this way a single

instruction can require as many as four bytes of storage in total. For example the JR NZ,d instruction which we have already met requires one byte to hold the code for it which is 20 hexadecimal (32 decimal) and a second byte to hold the displacement, d.

The JR NZ,d instruction appears close to the bottom of the first page of Appendix A of the Operating Manual and to the right of it, in the column headed 'after CBh' appears the mnemonic 'SLA B' (note that upper case is used here rather than lower case as in the Operating Manual). This means that if a byte containing 20 hexadecimal is preceded by a byte containing CB hexadecimal (203 decimal) the pair of bytes is interpreted as a shift-left-arithmetic on register B. (SLA B is a leftward shift of one step of the bits in the register and is described in more detail in a later section.) Thus JR NZ,d is a one byte instruction (plus one for the displacement) and SLA B is a two byte instruction.

The first byte of a two byte instruction is either CB, ED, DD or FD and the first two bytes of a three byte instruction are either DD CB or FD CB. Appendix A of the Operating Manual gives a complete list of all one byte instructions and of those two byte instructions which start with either CB or ED. The table on the next page lists the remaining two byte and all the three byte instructions.

In the remainder of this chapter instructions will be referred to by their mnemonic Op Code and by their two-digit-per-byte hexadecimal representation which reflects directly the bit pattern of the byte. A mnemonic Op Code is a shorthand way of describing an instruction and is for human convenience only. The ZX81 will not

TWO BYTE INSTRUCTIONS

Each of the following codes is preceded by DD (decimal 221) when referring to the IX register or FD (decimal 253) when referring to the IY register. For clarity IX only is used below.

DEC	HEX	OP CODE
9	09	ADD IX,BC
25	19	ADD IX,DE
33	21nn	LD IX,nn
34	22nn	LD (nn),IX
35	23	INC IX
41	29	ADD IX,IX
42	2Ann	LD IX,(nn)
43	2B	DEC IX
52	34d	INC (IX+d)
53	35d	DEC (IX+d)
54	36dn	LD (IX+d),n
57	39	ADD IX,SP
70	46d	LD B,(IX+d)
78	4Ed	LD C,(IX+d)
86	56d	LD D,(IX+d)
94	5Ed	LD E,(IX+d)
102	66d	LD H,(IX+d)
110	6Ed	LD L,(IX+d)
112	70d	LD (IX+d),B
113	71d	LD (IX+d),C
114	72d	LD (IX+d),D
115	73d	LD (IX+d),E
116	74d	LD (IX+d),H
117	75d	LD (IX+d),L
119	77d	LD (IX+d),A
126	7Ed	LD A,(IX+d)
134	86d	ADD A,(IX+d)
142	8Ed	ADC A,(IX+d)
150	96d	SUB (IX+d)
158	9Ed	SBC A,(IX+d)
166	A6d	AND (IX+d)
174	AEd	XOR (IX+d)
182	B6d	OR (IX+d)
190	BEd	CP (IX+d)
225	E1	POP IX
227	E3	EX (SP),IX
229	E5	PUSH IX
233	E9	JP (IX)
249	F9	LD SP,IX

THREE BYTE INSTRUCTIONS

Each of the following codes is preceded by DD CB (decimal 221 203 when referring to the IX register or FD CB (decimal 253 203 when referring to the IY register. For clarity IX only is used below.

DEC	HEX	OP CODE
6	d06	RLC (IX+d)
14	d0E	RRC (IX+d)
22	d16	RL (IX+d)
30	d1E	RR (IX+d)
38	d26	SLA (IX+d)
46	d2E	SRA (IX+d)
62	d3E	SRLR (IX+d)
70	d46	BIT 0,(IX+d)
78	d4E	BIT 1,(IX+d)
86	d56	BIT 2,(IX+d)
94	d5E	BIT 3,(IX+d)
102	d66	BIT 4,(IX+d)
110	d6E	BIT 5,(IX+d)
118	d76	BIT 6,(IX+d)
126	d7E	BIT 7,(IX+d)
134	d86	RES 0,(IX+d)
142	d8E	RES 1,(IX+d)
150	d96	RES 2,(IX+d)
158	d9E	RES 3,(IX+d)
166	dA6	RES 4,(IX+d)
174	dAE	RES 5,(IX+d)
182	dB6	RES 6,(IX+d)
190	dBE	RES 7,(IX+d)
198	dC6	SET 0,(IX+d)
206	dCE	SET 1,(IX+d)
214	dD6	SET 2,(IX+d)
222	dDE	SET 3,(IX+d)
230	dE6	SET 4,(IX+d)
238	dEE	SET 5,(IX+d)
246	dF6	SET 6,(IX+d)
254	dFE	SET 7,(IX+d)

recognise Op Codes except through the medium of an assembler program. (The function of an assembler program, amongst other things, is to translate Op Codes into the correct bit pattern.)

The Op Codes used in this text follow certain conventions as follows :

1) Single registers are referred to by their letter eg B, register pairs are named in alphabetical order eg BC.

2) A displacement,d is taken to be positive if it lies in the range 0 to 127 and negative if it lies between 128 and 255. Larger or smaller numbers are not allowed. The negative value is calculated by subtracting d from 256. For example the relative-jump instruction JR,d causes a jump forward of 8 bytes if d=8 and a jump backward of 8 bytes if d=248 (=256-8). Remember when using a displacement that the jump is made from the address of the first byte of the instruction immediately following the JR instruction.

Question : If d = FA in hexadecimal what displacement will it cause ?

Answer : FA in hexadecimal = 250 in decimal. 256-250=6 therefore the displacement is six bytes backward.

3) A single byte number is represented by n and lies in the range 0 to 255.

Question : If n = E0 in hexadecimal what decimal value does it represent ?

Answer : E0 in hexadecimal = 224 in decimal.

4) A two byte number or an address is represented by nn and lies in the range 0 to 65535. The value is calculated by adding the first n to 256 times the second.

Question : What decimal address is OC,40 in hexa-decimal ?

Answer : OC in hexadecimal = 12 in decimal; 40 in hexadecimal = 64 in decimal therefore the address is

12 + 256 * 64 = 12 + 16384 = 16396. Addresses 16396 and 16397 are in the system variables area and their contents together point to another address - that of the beginning of the display file.

5) nn in brackets viz (nn) means ' the contents of the address nn ' whereas nn means 'the number nn'. Similarly (HL) means 'the contents of the address held in HL' whereas HL means 'the number in HL'.

Question : If INC means 'increment' or 'add one to' what does i) INC HL mean, ii) INC (HL) mean ?

Answer : i) Increment (or add one to) the value in HL, ii) Increment (or add one to) the value held in the address pointed to by HL.

6) The destination of the result of an instruction is always given first eg ADD A,B means 'add the contents of B to the contents of A and leave the result in A'.

Question : If LD means 'load' or 'copy into', what does LD A,B mean ?

Answer : Load (or copy) the value in B into A.

A GLOSSARY OF MACHINE CODE INSTRUCTIONS

In this section all numbers are two digit hexadecimals except where otherwise stated.

No-operation (NOP)

This is the simplest instruction and as its name implies the processor does nothing when it encounters it. Hence it can be very useful when debugging a program because it can be substituted for a suspect instruction temporarily and it will not upset the functioning of the remainder of the program. It can be used to plug any gaps when making small alterations to existing programs and it can also be used to introduce a delay during the execution by incorporating it into a suitable loop. The hexadecimal code is 00.

Load (LD)

Load copies the contents of one register or address to another register or address. There are more than a hundred load instructions and they fall naturally into eight classes.

a) 8 bit register to register.

The contents of any of the registers A,B,C,D,E,H, L can be copied to one anther.

b) 8 bit memory to register.

(HL), (IX+d) or (IY+d) can be copied to any of the registers A,B,C,D,E,H,L. (BC), (DE) or (nn) can be copied to A.

c) 8 bit register to memory.

A,B,C,D,E,H,L may be copied to (HL), (IX+d) or (IY+d). A can be copied to (BC), (DE) or (nn).

d) 8 bit register or memory immediate.

An immediate is a number read from the program itself rather than from a register or from an arbitary address in memory. A number n can be loaded into A, B,C,D,E,H,L, (HL), (IX+d), (IY+d).

e) 16 bit register to register.

The contents of HL, IX or IY can be copied to SP.

f) 16 bit memory to register.

(nn) can be copied to BC, DE, HL, IX, IY or SP.

g) 16 bit register to memory.

BC, DE, HL, IX, IY or SP can be copied to (nn).

h) 16 bit register immediate.

nn can be loaded into BC, DE, HL, IX, IY or SP.

Push and Pop (PUSH, POP)

A PUSH instruction copies the contents of a named 16 bit register to the bottom of the stack and decrements the stack pointer twice. A POP instruction does the reverse and so the two instructions in concert can be used to save register values temporarily. AF, BC, DE, HL, IX and IY can be PUSHed and POPped.

Exchange. (EX)

Exchanges can be made between HL and DE, HL and (SP), (SP) and IX, (SP) and IY, AF and AF' and between BCDEHL and BCDEHL' (a single instruction swops all six 8 bit registers).

8 bit add and subtract. (ADD, SUB, etc)

A,B,C,D,E,H,L,(HL),n, (IX+d) and (IY+d) can be added or subtracted to or from the A register with or without the carry flag. Carry instructions end in C.

8 bit AND, OR and XOR. (AND, etc)

A,B,C,D,E,H,L,(HL),n,(IX+d) and (IY+d) can be combined with the A register using any of the three logical operators above. AND sets each bit in the result which was set in both the sources; OR sets each bit which was set in either or both of the sources and XOR sets each bit which was set in one or other source but not those which were set in both.

Compare. (CP)

Compare is like subtract except that the flags only and not the contents of A are affected. A,B,C,D, E,H,L,(HL),n, (IX+d) and IY+d) can be compared with the accumulator.

8 bit increment and decrement. (INC, DEC)

A,B,C,D,E,H,L,(HL), (IX+d) and (IY+d) can be incremented or decremented.

16 increment and decrement. (INC, DEC)

BC,DE,HL,IX,IY and SP may be incremented or decremented.

16 bit add and subtract. (ADD, SUB, etc)

BC,DE,HL,IX can be added with or without carry

or subtracted with carry only to or from HL. BC,DE,SP, IX can be added without carry to IX. BC,DE,SP and IY can be added without carry to IY.

Jump, Call and Return

The flag register, F, contains a carry flag, C, a parity flag, P, which is set if a result is even, a sign flag, S, which is set if a result is negative, an overflow flag, V, which is set on overflow, and a zero flag, Z, which is set on a zero result. These flags can be used to control jumps, subroutine calls and subroutine returns.

a) Jump (JP or JR)

The following jumps to address nn are possible : absolute jump (JP); jump on zero or not zero (JP Z and JP NZ); jump on carry or not carry (JP C and JP NC); jump on positive or negative (JP P and JP M); jump on P/V=1 or P/V=0 (JP PE and JP PO). The following relative jumps to an address d relative to the current position are available where d is interpreted as lying in the range -128 to 127 : absolute relative jump (JR); relative jump on zero or not zero (JR Z or JR NZ); relative jump on carry or not carry (JR C and JR NC). Jumps can also be made to the addresses held in HL,IX or IY (JP (HL), JP (IX), JP (IY)). The DJNZ instruction decrements the B register and jumps to d if B is non zero.

b) Call (CALL)

This instruction serves a similar function to the BASIC GOSUB command. If the call condition is met then the program transfers to the instruction held in address nn. The following calls may be made : absolute call (CALL); call on zero or not zero (CALL Z and CALL NZ); call on carry or not carry (CALL C and CALL NC); call on positive or negative (CALL P and CALL M); call on P/V=1 or P/V=0 (CALL PE and CALL PO).

c) Return (RET)

This instruction serves a similar function to the

BASIC RETURN command. Return conditions are available to match each call condition and returns can also be made from the interrupt and the non-maskable interrupt. (RETI and RETN).

Bit Instructions

The eight bits in each register are numbered from 0 to 7 from right to left. Each of the following operations can be performed on the A,B,C,D,E,H,L registers and on (HL), (IX+d) and (IY+d).

a) Bit test (BIT)

The bit test instruction sets the zero flag to the opposite of the setting of the named bit. Any bit can be tested.

b) Bit set (SET)

Any bit can be set.

c) Bit reset (RES)

Any bit can be reset.

d) Rotate left (RL)

Bit 7 is copied to the carry, the carry is copied to bit 0 and all other bits are copied one place to the left.

e) Rotate right (RR)

Bit 0 is copied to the carry, the carry is copied to bit 7 and all other bits are copied one place to the right.

f) Rotate left circular (RLC)

Bit 7 is copied to the carry and to bit 7. All other bits are copied one place to the left.

g) Rotate right circular (RRC)

Bit 0 is copied to the carry and to bit 7. All other bits are copied one place to the right.

h) Shift left arithmetic (SLA)

All bits are copied one place to the left, bit 7 is copied to the carry and bit 0 is reset.

i) Shift right arithmetic (SRA)

All bits are copied one place to the right, bit 0 is copied to the carry and bit 7 is copied to itself.

j) Shift right logical (SRL)

As shift right arithmetic but with bit 7 reset.

Rotate Left Digit (RLD)

Bits 0 to 3 of A are copied to bits 0 to 3 of (HL); bits 0 to 3 of (HL) are copied to bits 4 to 7 of (HL); bits 4 to 7 of (HL) are copied to bits 0 to 3 of A.

Rotate Right Digit (RRD)

Bits 0 to 3 of A are copied to bits 4 to 7 of (HL); bits 4 to 7 of (HL) are copied to bits 0 to 3 of (HL); bits 0 to 3 of (HL) are copied to bits 0 to 3 of A.

Accumulator Operations

a) Complement A (CPL)

Every set bit of A is reset, every reset bit is set.

b) Negate A (NEG)

Complement A and add one.

c) Complement carry (CCF)

Sets the carry flag if it is reset, resets it otherwise.

d) Set carry (SCF)

Sets the carry flag.

e) Decimal adjust (DAA)

Corrects A after BCD addition and subtraction.

Restart (RST)

Save the program counter on the stack and jump to location 8*n where n is held in the byte following.

Block Handling

These are compound instructions designed to move data or to search data in memory.

a) Load and increment (LDI)

Move one byte from (HL) to (DE), increment DE and HL and decrement BC.

b) Load, increment and repeat (LDIR)

As load and increment then repeat until BC=0.

c) Load and decrement (LDD)

Move one byte from (HL) to (DE) and decrement DE,HL and BC.

d) Load, decrement and repeat (LDDR)

As load and decrement then repeat until BC=0.

e) Compare and increment (CPI)

Compare A and (HL), increment HL and decrement BC.

f) Compare, increment and repeat (CPIR)

As compare and increment then repeat until BC=0.

g) Compare and decrement (CPD)

Compare A and (HL) and then decrement HL and BC.

h) Compare, decrement and repeat (CPDR)

As compare and decrement then repeat until BC=0.

LOADING AND RUNNING MACHINE CODE

The first problem to be resolved when writing a machine code program is to decide where to store it in RAM. The Operating Manual suggests on pages 167 and 168 that it can be put a) in a REM statement, b) in a string, and c) above RAMTOP. Each of these options has advantages and disadvantages which can be summarised as follows :

In a REM statement - advantages

1) If the REM is the first line in a BASIC program, the routine will not be moved in memory when lines are added to or removed from the BASIC program.
2) The REM statement can be editted, lengthened or shortened in the normal way using the EDIT key.
3) The REM statement can be SAVEd on cassette and LOADed in the usual way.

In a REM statement - disadvantages

1) Machine code in a REM statement appears nonsensical when the line is listed.
2) Some codes eg decimal 126 can cause strange effects in the listing of subsequent BASIC lines although the operation of the lines is unaffected.

In a string - advantages

1) The code does not appear in program listings.
2) Strings can be equated, lengthened and shortened at will.
3) Strings are SAVEd and LOADed from cassette.

In a string - disadvantages

1) The string and the code which it contains will be deleted from memory if RUN or CLEAR is entered.
2) If the BASIC program is altered or the display file grows or shrinks, then the variables area and the string contained in it moves in memory.

Above RAMTOP - advantages

1) The code does not appear in program listings.
2) Changing or running the BASIC program will not move the code in memory.

Above RAMTOP - disadvantages

1) The total size of the area above RAMTOP must be set before the program is entered and it cannot easily be changed without destroying the entire program.
2) The contents of the area above RAMTOP are not copied to or from cassette by the SAVE and LOAD commands.

For practical purposes use of a string is ruled out because of its susceptibility to accidental deletion and to move around in RAM. It is vital to give the right address using the USR command when calling a machine code routine which is rather difficult if the routine does not stay in one place. The call and the jump machine code instructions (CALL and JP) cause the next instruction to be taken from a particular address in memory and these too will be erroneous if the routine moves from its intended position. The fact that code stored above

RAMTOP cannot be SAVEd or LOADed without special software effectively rules out the use of this area as well. Hence use of a REM statement despite its messy effect on program listings is generally the best solution and it is this technique which will be described in the remainder of this chapter.

Having decided to use a REM statement at line 1 in which to store a machine code routine or program the HEX LOADER/PRINTER program on page 59 can be used to load the code into place. The first address of user RAM is 16509 and therefore allowing two bytes for the line number, two bytes for the line length indicator and one byte to hold decimal 234 (the code for REM) the first address occupied by the machine code is 16514.

The PRINT THE STACK POINTER ROUTINE on page requires 7 bytes of RAM and can be loaded and run using HEX LOADER/PRINTER as it stands. The decimal and hexadecimal codes and their mnemonics all appear in Appendix A of the Operating Manual. Note that the final code is RET (C9 hexadecimal) ie. unconditional return. This causes the ZX81 to return to the BASIC program passing to it the value left in BC. The routine could be incorporated into a longer machine code program simply by omitting C9. ZX80 owners should note that it is the value in BC which is returned by the USR function not the value in HL as on the ZX80.

A machine code routine is incorporated into the CLOCK program on page 70 . This routine waits intervals of 5 seconds by counting the time it takes for 250 frames to be passed to the TV screen. The BASIC program first POKEs an adjusted current value of the less significant byte of the frame counter into location 16514 (line 20).

The adjustment is to allow for the time taken by the BASIC program to perform its calculations and set up the display after the user starts the clock. The machine code routine picks up the value in 16514 (LD A,16514), subtracts 250 (ie 5 seconds x 50 cycles per second - SUB A,250) and copies the new value back into 16514 for use the next time. It then loads the low byte of the frame counter into (LD HL,16436; LD D,(HL)) and compares the value in D with that in A. If the values don't match it jumps back (JR NZ,-4) to LD D,(HL). The routine stays in this loop until the compute and display software in the 8K ROM has decremented 16436 250 times at which time the value loaded into D will match the value held in A. The routine returns control to the BASIC system which displays the current time.

Notice that the JR NZ instruction jumps back 4 bytes counting from 0 at the byte following the instruction ie

byte contains decimal	count
201	0
252	-1
32	-2
186	-3
86	-4

Thus a JR NZ,-2 instruction would jump back to itself - endlessly ! Notice that a jump back of -4 is represented in decimal by 256-4 = 252.

The routine traverses the loop LD D,(HL); CP D; JR NZ,-4 many, many times and it would be quite possible to write a much longer m/c program which spent most of its time doing something else, stopping occasionally to see if 5 seconds had passed and if so updating the time display. In fact the compute and display facility which is used when the ZX81 is running in SLOW

mode operates in more or less this fashion. The difference is that it is a separate part of the electronics which interrupts the ZX81 to tell it that it is time to pass another frame to the screen rather than an integral part of the 8K software. The computer then stops what it was doing, gathers up the material in the display file and passes it, with appropriate delays etc, to the TV. It then returns to its previous task.

The CLEAR NUMBERS routine listed on page 68 can be used to clear the display of digits but not of other characters. The routine occupies only 65 bytes which shows that the instructions for a complicated task can be held in a relatively small amount of space. 65 bytes is equivalent to about two complete lines of characters in the display. Thousands of machine code instructions can be executed in a second and so the routine is extremely fast as can be seen by loading and running the example program. This program fills the screen with digits and characters and when the routine is called the digits disappear in a flash.

The routine first calculates the total length of the current display by loading the system variables D-FILE and VARS into BC and HL respectively and subtracting BC from HL. The result is transferred to BC and then HL is loaded with the value in D-FILE. Next the A register is loaded with the contents of the first byte of the display file (LD A,(HL)). If the value now in A does not lie between 28 and 37 inclusive (ie. the byte does not represent a digit) a jump is made to address 16568. If the value does lie between 28 and 37 the routine sets the byte to zero and then looks at the contents of the following byte (LD (HL),0; INC HL; LD A,(HL)) to see if it contains decimal 42 - the code for E. The routine does this because very large and very small numbers

are displayed in E format eg. 1.2384672E+9 and so the routine must also delete an E (or a +, - or .) if it forms part of a number but not if it forms part of a word. If 42 is found then that byte is also set to zero (LD (HL),0). Now the routine looks back two bytes to the byte before the one which it first looked at by decrementing HL twice and loading the contents into A. If this byte contains 21, 22 or 27 (+, - or .) then it too is set to zero. Finally the value in HL is incremented so that it once again points to the first byte which the routine looked at.

At this point the program counter has reached 16568 which was the address to which the jump would have been made if the byte did not contain a digit code. Hence the two possible streams of actions come together at this instruction. The machine now moves on to consider the next byte by incrementing HL, ignoring the fact that it may have looked at the byte previously. Before looping back to repeat the task for the new byte the machine checks to see if it has reached the end of the display file. It will be remembered that BC was set to the total length of the display file originally. The routine now decrements BC to account for the byte which it has just looked at, loads A with zero and if both B and C are the same as A ie. both B and C contain zero the routine drops through to the RET instruction and hence returns to BASIC. Otherwise the routine jumps back 46 or 49 bytes of instructions, depending on the point from which the jump is made, to the LD A,(HL) instruction so that it can go through the entire procedure again for the next byte.

In the example program the loop is executed about 600 times to clear all the digits from the display. Note that the JP P,16568 and JP M,16568 instructions use absolute addresses for their destinations therefore if this routine is loaded at a starting address other than

16514 these addresses must be changed accordingly. This comment does not apply to JR NZ,-46 and JR NZ,-49 because the destinations for these jumps are calculated relative to the current address.

DISAPPEARING CODE

It has been mentioned that machine code instructions have an ugly effect on program listings when stored in REM statements. The following machine code routine can be used to make REM statements invisible at the expense of making subsequent BASIC statements invisible too. The routine will appeal to those who wish to keep their programs safe from the eyes of others.

The function of the routine is to set the line number of a BASIC line to a value than greater than the maximum legal value - 9999. Line numbers are stored in two bytes with the most significant byte first for example line number 123 is stored as 0, 123 because 0*256 + 123 = 123. The easiest way to change any number in the range 1 to 9999 to another number larger than 9999 is to complement the most significant byte (*ie.* set all zero bits to one bits and vice versa). The original number can be retrieved simply by complementing the byte again. When a line number is altered in this fashion the BASIC system ignores it and all subsequent lines. The line will not appear in program listings nor will they be executed. However they are SAVEd and LOADed to and from cassette.

Machine code

DEC	HEX	OP CODE
33 n n	21 n n	LD HL,nn
126	7E	LD A,(HL)
47	2F	CPL
119	77	LD (HL),A
201	C9	RET

nn is the address (in the appropriate units) of the most significant byte of the first line to disappear.

Example BASIC program

Set nn to 16529 (145 64 decimal; 91 40 hex)

```
10    LET B=USR 16534
20    REM containing 7 byte m/c routine above
30    FOR I=1 TO 80
40    PRINT PEEK (16508+I);"∧";
50    NEXT I
60    LET B=USR 16534
```

When the BASIC program is first RUN it will stop after executing line 10 because the m/c routine will have caused lines 20 onwards to disappear. On entering RUN on second and subsequent occasions the m/c routine will cause lines 20 onwards to reappear. The lines will then be executed, printing the contents of the first 80 bytes of RAM as proof and then the call to the m/c routine at line 60 will cause the lines to disappear again.

Notice that the m/c routine has a byte containing decimal 126. The BASIC system treats such bytes with special respect. That byte and the five following bytes will not appear in LISTings which has an unusual effect on the LISTing of line number 30. It will be remembered that a byte containing decimal 126 is used to separate the character codes for a number from its five byte

numeric form in program listings. In effect whenever the BASIC comes across a byte containing 126 it ignores the next 5 bytes.

ZX81 HEX LOADER/PRINTER

This program loads, prints and runs machine code using the space available in the REM statement at line 10. The example text contains 22 characters including spaces and so at most 22 bytes of code can be loaded with the program as it stands. The loops at lines 90 and 140 are set to cycle 22 times ie. once for each byte. To load a longer machine code routine lengthen the REM statement and increase the number of cycles for the loops correspondingly.

```
10   REM MACHINE CODE GOES HERE
20   GOSUB 240
30   PRINT "INPUT, PRINT, RUN OR STOP
(I,P,R,S) ?"
40   INPUT Z$
50   GOSUB 240
60   IF Z$="I" THEN GOTO 140
70   IF Z$="R" THEN GOTO 210
80   IF Z$="S" THEN STOP
90   FOR I=16514 to 16535
100  PRINT CHR$(28+INT(PEEKI/16));CHR$(
28+PEEK I-16*INT(PEEK I/16));"^^";
110  NEXT I
120  PAUSE 999
130  GOTO 20
140  FOR I=16514 TO 16535
150  INPUT Z$
160  IF Z$="S" THEN GOTO 20
170  PRINT Z$;"  ";
180  POKE I,16*CODE Z$+CODE Z$(2)-476
190  NEXT I
200  GOTO 20
210  PRINT USR 16514
220  PAUSE 999
```

```
230   GOTO 20
240   CLS
250   PRINT "HC HEX LOADER/PRINTER"
260   RETURN
```

On running the program the user is asked if he wishes to "INPUT, PRINT, RUN OR STOP (I,P,R,S) ?". He then enters the letter which indicates his choice. If the input mode is selected the program prompts for each two digit hexadecimal code in turn. The codes are printed and then decoded and POKEd into the next available byte. If S is entered the program returns to the menu.

A print request causes the current contents of the REM to be listed in hexadecimal and a run request causes the machine code to be executed. As an example the following code prints the contents of the stack pointer SP. Note that two hexadecimal codes must always be entered thus 00 must be entered as such and not as 0.

ZX81 PRINT THE STACK POINTER ROUTINE

DECIMAL	HEX	MNEMONIC
33 00 00	21 00 00	LD HL,0
57	39	ADD HL,SP
68	44	LD B,H
77	4D	LD C,L
201	C9	RET

The routine works by clearing the HL register and copying the contents of SP to it. H is then copied to B and L to C. The value held in BC is returned by the USR function and so this value is printed when a machine code routine is called by a PRINT USR ..... instruction.

ZX81 CONTENTS OF RAM

This program prints the contents of the first 32 bytes of the program area (see page 2), the display file (see page 14) or the variables area (see page 8).

```
10    GOSUB 500
20    PRINT "PROGRAM, DISPLAY OR VARIABLES
(P,D,V) ?"
30    INPUT A$
40    DIM B(32)
50    GOSUB 500
60    LET A=16509
70    IF A$="D" THEN LET A=PEEK 16396+256*
PEEK 16397
80    IF A$="V" THEN LET A=PEEK 16400+256*
PEEK 16401
90    FOR I=1 TO 32
100   LET B(I)=PEEK(A+I-1)
110   NEXT I
120   FOR I=1 TO 32
130   PRINT AT (I-5)/8+2,4*(I-1)-32*INT((I-1)/8);
B(I)
140   NEXT I
150   STOP
500   CLS
510   PRINT "HC CONTENTS OF RAM"
520   RETURN
```

The address of the first byte of the area required is retrieved, the contents of each byte are stored temporarily in array B and then the whole of B is printed. It is necessary to store the data temporarily because if each item were printed immediately then the display file would alter and grow and the variables area would move up the RAM so that the next item would have moved too.

ZX81 LINE RENUMBER

This routine will renumber all or part of a program except for the destination line number after GOTO or GOSUB. Lines 10-50 are a short test program and should be deleted before the routine is used in earnest.

```
10    REM HC LINE RENUMBER
20    REM TEST
30    REM
40    REM
50    REM END OF TEST
9000  PRINT "INPUT FIRST LINE NO, FINAL LINE NO,
NEW FIRST LINE NO, STEP SIZE"
9002  INPUT A
9004  INPUT T
9006  INPUT I
9008  INPUT S
9010  LET Z=16509
9012  GOSUB 9050
9014  GOTO 9018
9016  GOSUB 9042
9018  IF A>V THEN GOTO 9016
9020  POKE Z,INT(I/256)
9022  POKE Z+1, I-256*INT(I/256)
9024  LET I=I+S
9026  GOSUB 9042
9028  IF T>=V THEN GOTO 9020
9030  STOP
9042  IF PEEK Z=118 THEN GOTO 9048
9044  LET Z=Z+1
9046  GOTO 9042
9048  LET Z=Z+1
9050 LET V=256*PEEK Z+PEEK(Z+1)
9052  RETURN
```


ZX81 BOUNCER

This program can be left to entertain your dinner guests while you look after the sauce. To increase/decrease the rate of repetition increase/decrease the decimal in line 36.

```
2     PRINT "HC ∧∧∧∧∧ BOUNCER"
4     RAND
6     LET U= RND / RND
8     LET V= RND
10    LET A= RND
12    LET X=43* RND
14    LET Y=28+7* RND
16    LET X=U+X
18    IF X > 0 AND X < 43 THEN GOTO 24
20    LET U=-U
22    LET X=-X+86*(X >=43)
24    LET Y=V+Y
26    IF Y > 0 AND Y < 35 THEN GOTO 32
28    LET V=-V
30    LET Y=-Y+70*(Y >=35)
32    LET V=V-A
34    PLOT X,Y
36    IF RND > .003 THEN GOTO 16
38    CLS
40    GOTO 2
```

ZX81 SHOOT

Shoot down the aircraft. "P" fires your rocket or resets it, "A" moves it to the left, "L" moves it to the right. Good luck!

```
5     RAND
10    LET A=-1
20    LET B=22
30    LET C=15
40    LET D=1
50    LET E=0
60    FOR I=1 TO 900
70    IF INKEY$="P" THEN LET D=-D
80    IF INKEY$="L" THEN LET C=C+2
90    IF INKEY$="A" THEN LET C=C-2
100   IF ABS(C-15)>15 THEN LET C=15
110   LET A=A+1
120   IF A>31 THEN LET A=0
130   IF D>=0 THEN LET B=21
140   CLS
145   PRINT "HC SHOOT"
150   PRINT AT D,A;"■"
160   PRINT AT B,C;"*"
170   IF B=ABS D AND C=A THEN GOTO 240
180   LET B=B-1
190   IF B<0 THEN LET D=ABS D
200   NEXT I
210   PRINT "TOTAL SCORE ";E
230   STOP
240   LET E=E+1
250   PRINT "P O W  ";E;" HITS"
260   LET D=1+INT(15*RND)
270   PAUSE 50
280   NEXT I
```


ZX81 FACTORS

This program will print the prime factors of any positive integer in the range 2 to 4,294,967,295.

```
10    CLS
20    PRINT "HC FACTORS"
30    PRINT " INPUT A POSITIVE INTEGER WHICH
YOU WISH TO FACTORISE"
50    INPUT A
60    IF A=INT A AND A>1 THEN GOTO 90
70    PRINT A;" IS NOT AN INTEGER. TRY AGAIN"
80    GOTO 50
90    PRINT
100   PRINT " THE FACTORS OF ";A:" ARE ";
110   FOR I=2 TO 2147483648
120   IF A/I<1 THEN GOTO 180
130   IF A<>I*INT(A/I) THEN GOTO 170
140   PRINT I;"^^";
150   LET A=A/I
160   GOTO 130
170   IF I<SQR A THEN NEXT I
180   IF A<>1 THEN PRINT A
200   PRINT
210   PRINT "THATS ALL"
220   PAUSE 500
230   GOTO 10
```

ZX81 STATISTICS

This program illustrates several novel techniques which are described in detail in the main text. It can be used to calculate a mean and standard deviation (STATS), a least squares regression (REGN) or a trend line (TREND).

```
 2    LET J= CODE "■"
 4    LET A=J-J
 6    LET B=A
 8    LET C=A
10    LET D=A
12    LET E=A
14    PRINT "HC∧∧∧∧STATISTICS"
16    PRINT AT J+J,A;"REGN, STATS OR TREND
(R,S,T) ?"
18    INPUT Z$
20    LET Z=CODE Z$
22    IF Z < CODE "R" OR Z > CODE "T" THEN
GOTO CODE "("
24    LET I=J
26    CLS
28    PRINT "HC∧∧∧∧STATISTICS"
30    LET P=I
32    GOTO Z-CODE "+"
34    INPUT P
36    LET F=P-A
38    LET B=B+(I-J)*F*F/I
40    LET A=((I-J)*A+P)/I
42    INPUT P
44    LET D=D+(I-J)*(P-C)*(P-C)/I
46    LET E=E+(I-J)*F*(P-C)/I
48    LET C=((I-J)*C+P)/I
50    IF I=J THEN GOTO CODE "Y"
51    PRINT "MEAN","SD"
52    PRINT C, SQR (D/(I-J))
```



```
54    IF Z$="S" THEN GOTO CODE "W"
56    PRINT A, SQR (B/(I-J))
58    PRINT "INTERCEPT","SLOPE",C-E*A/B,E/B
60    PAUSE 999
62    LET I=I+J
64    GOTO CODE ","
```

Example data

1 newline	2 newline
2 newline	3 newline
3 newline	5 newline
4 newline	7 newline

Results

Selecting STATS and entering the data in the right hand column gives :

MEAN	SD
4.25	2.2173558

Selecting REGN and entering all the data in the order 1, 2, 2, 3, 3, 5, 4, 7 gives :

MEAN	SD
4.25	2.2173558
2.5	1.2909945
INTERCEPT	SLOPE
-1.8626452E-9	1.7

Selecting TREND and entering the data in the right hand column gives the same results as for REGN.

ZX81 CLEAR NUMBERS

Overwriting numbers in the display with new values using the PRINT AT facility can lead to confusing results because the ZX81 does not blank out the old value before overwriting it. Thus if 3.333333333 is overwritten by 4.5 the display shows 4.533333333. This routine clears the screen of numbers but leaves the rest of the display intact. The BASIC program can be used to see the effect of the routine.

To enter the routine from the keyboard first load HEX LOADER/PRINTER and use it to load the 65 byte routine into the REM statement at line 1. Then delete all but line 1 and enter your BASIC program or the example program in the usual way.

Machine code routine

DEC	HEX	OP CODE
237 75 12 64	ED 4B 0C 40	LD BC,16396
42 16 40	2A 10 40	LD HL,16400
167	A7	AND A
237 66	ED 42	SBC HL,BC
68	44	LD B,H
77	4D	LD C,L
42 12 64	2A 0C 40	LD HL,16396
126	7E	LD A,(HL)
254 38	FE 26	CP,38
242 184 64	F2 B8 40	JP P,16568
254 28	FE 1C	CP, 28
250 184 64	FA B8 40	JP M,16568
54 0	36 00	LD (HL),0
35	23	INC HL
126	7E	LD A,(HL)
254 42	FE 2A	CP,42
32 2	20 02	JR NZ,2
54 0	36 00	LD (HL),0

43	2B	DEC HL
43	2B	DEC HL
126	7E	LD A,(HL)
254 21	FE 15	CP,21
40 8	28 08	JR Z,8
254 22	FE 16	CP,22
40 4	28 04	JR Z,4
254 27	FE 1B	CP,27
32 2	20 02	JR NZ,2
54 0	36 00	LD (HL),0
35	23	INC HL
35	23	INC HL
11	0B	DEC BC
62 0	3E 00	LD A,0
184	B8	CP B
32 210	20 D2	JR NZ,-46
185	B9	CP C
32 207	20 CF	JR NZ,-49
201	C9	RET

Example BASIC program

```
    1   REM 12345678901234567890123456789012345678901234567890123456789012345
  200   FOR I=0 TO 127
  210   PRINT CHR$ I;
  220   NEXT I
  230   FOR I=-4 TO 4
  240   PRINT RND*9999**I, RND/9999**I
  250   NEXT I
  260   LET I= USR 16514
```

ZX81 CLOCK

To enter this program from the keyboard first load HEX LOADER/PRINTER and use it to load the 17 byte machine code routine into the REM statement at line 1 and then delete all but line 1 and enter the BASIC lines.

Machine code routine

DEC	HEX	OP CODE
0	00	Variable
58 130 64	3A 82 40	LD A,16514
214 250	D6 FA	SUB A,250
50 130 64	32 82 40	LD (16514),A
33 52 64	21 34 40	LD HL,16436
86	56	LD D,(HL)
186	B0	CP D
32 252	20 FC	JR NZ,-4
201	C9	RET

BASIC program

```
1   REM **********
5   GOSUB 80
8   PRINT "SET THE TIME"
10  PRINT "HOUR ?"
12  INPUT H
14  PRINT "MINUTES ?"
16  INPUT M
18  PRINT "SECONDS ?"
19  INPUT S
20  POKE 16514, 80-256*(PEEK 16436 > 175)+
PEEK 16436
22  IF S < 0 OR S > 59 OR H < 1 OR H > 12 OR M
<0 OR M > 59 THEN GOTO 5
24  LET T=S+3600*H+60* M
26  GOSUB 80
```



```
28  PRINT AT 10,11;"^^^^^^^^"
30  PRINT AT 10,11;H;":";M;":";S
32  LET H=USR 16515
40  LET T=T+5-43200*(T >46794)
42  LET H=INT (T/3600)
44  LET M=INT (T/60-60*H)
46  LET S=T-60*M-3600*H
79  GOTO 28
80  CLS
82  PRINT "HC ^^^^^^ CLOCK"
84  PRINT
86  RETURN
```

The effect of the machine code routine is to lock the timing of the CLOCK to the TV frame counter which is in turn dependent on mains frequency. The same effect could be achieved with a BASIC routine but updates to the time could only be made every ten to fifteen seconds because BASIC is very slow. When running the program the user will notice that the CLOCK runs fast or slow depending on the time of day. This effect is due to variations in mains frequency. The error should never exceed about 30 seconds and the net error over 24 hours will be very small.

ZX81 LIFE

This program just squeezes into the 1K machine and shows birth, LIFE and death in two lines of display. The rules are that a cell with 0, 1 or more than 4 neighbours dies, a cell with 2 or 4 neighbours survives to the next generation and a cell with 3 neighbours, shown in inverse video, gives birth to a new cell and survives itself.

If you have extra memory then you can get a larger display by increasing the number of loop cycles in multiples of 32 at line 2 and the same multiple of 33 at lines 8,14,24 and 46.

The variable I should be set before running by entering LET I= PEEK 16396 + 256* PEEK 16397 and the program should be started by entering GOTO 1.

```
2    FOR J=1 TO 64
4    PRINT CHR$ (23*( RND > 0.6));
6    NEXT J
8    FOR J=I TO I+66
10   IF PEEK J=23 THEN GOSUB 40
12   NEXT J
14   FOR J=I TO I+66
16   IF PEEK J=24 THEN POKE J,0
18   IF PEEK J <> 151 THEN GOTO 28
20   POKE J,23
22   LET K=J+33*( INT (3* RND)-1)+
INT (3* RND)-1
24   IF PEEK K=23 OR K< I OR K > I+66 OR
PEEK K=118 THEN GOTO 22
26   POKE K,23
28   NEXT J
```



```
30   GOTO 8
40   LET K=0
42   FOR M=-4 TO 4
44   LET L=J+SGN M*(30*( ABS M > 1))+M
46   IF L > I+66 OR L < I THEN GOTO 50
48   IF PEEK L <> 118 AND PEEK L >=23 THEN
LET K=K+1
50   NEXT M
52   IF K > 5 OR K <=2 THEN POKE L-34,24
54   IF K=4 THEN POKE L-34,151
58   RETURN
```

ZX81 SIMON

In this game the player attempts to memorise a string of a chosen length. The ZX81 awards a score based on the number of correct characters and the time taken. The program just squeezes into the 1K machine and so it is vital not to put more than the number of characters shown in PRINT statements etc. Note that the word "INPUT" at line 32 is not spelt out in full but is held as the character code 238. To load this line enter :
32 INPUT (backspace) PRINT " (forwardspace) etc.

```
2    LET Z$="12345678901234567890"
4    LET Z=-9999999
6    GOSUB 62
8    PRINT "NUMBER OF CHARACTERS ?"
10   INPUT J
12   IF J < 1 OR J > 20 THEN GOTO 10
14   GOSUB 62
16   PRINT "MEMORISE"
18   FOR I=1 TO J
20   LET Z$(I)=CHR$ (INT (36*RND )+28)
22   PRINT Z$(I);
24   NEXT I
26   PAUSE 32768
28   LET T=PEEK 16436/256+PEEK 16437
30   GOSUB 62
32   PRINT "INPUT STRING"
34   INPUT A$
36   PRINT A$
38   PRINT "STRING WAS"
40   PRINT Z$( TO J)
42   FOR I=1 TO J
44   IF I >LEN A$ THEN GOTO 48
46   IF A$(I)=Z$(I) THEN GOTO 50
48   LET J=J-2
```



```
50   NEXT I
52   LET J=J+SGN J*INT (J*J*T)
54   IF J > Z THEN LET Z=J
56   PRINT "SCORE ";J;" BEST SO FAR ";Z
58   PAUSE 999
60   GOTO 6
62   CLS
64   PRINT "HC ^^^^ SIMON"
66   PRINT
68   RETURN
```

ZX81 PLANET LANDER

You are at an altitude of 7350 feet in your spacecraft travelling planetwards at a velocity of 480 feet per second. You have 650 pounds of fuel on board which you must burn wisely if you are to land safely. You can burn fuel in units of 0 to 75 pounds inclusive. On your screen is a visual display of your approach to the planet. The display will expand as you make your approach. Enter an amount for your next unit burn. Good luck!

```
    2    LET S = 7350
    4    LET V = 480
    6    LET F = 650
    8    LET K = 17-S/(7*(S>16)+56*(S>128)
+448*(S>1024)+1)
   10    CLS
   12    PRINT "HC PLANET LANDER"; AT 20,
3;"          "
   14    PRINT "HT=";S;"V=";V;"FUEL=";F
   16    IF F < 0 OR S < 0 THEN GOTO 34
   18    PRINT AT K,6;"▬"; TAB 5;"▀▀▀";
TAB 5;"T""T"
   20    IF S=0 AND ABS V < 0.3 THEN GOTO 38
   22    INPUT H
   24    IF H < 0 OR H > 75 THEN GOTO 22
   26    LET F = F - H
   28    LET S = S - V - (5 - H)/2
   30    LET V = V + 5 - H
   32    GOTO 8
   34    PRINT AT 19,5;"▄▄▄"
   36    PRINT AT 5,0;"YOU CRASHED"
   38    PRINT AT 9,0;"AGAIN (A) ?"
   40    INPUT A$
   42    IF A$ = "A" THEN GOTO 1
```