UNDERSTANDING YOUR SPECTRUM

BASIC AND MACHINE CODE PROGRAMMING

DR. IAN LOGAN



UNDERSTANDING YOUR SPECTRUM

BASIC AND MACHINE CODE PROGRAMMING

by

Dr. Ian Logan

Lincoln, England 1982



Melbourne House Publishers

Published in the United Kingdom by:
Melbourne House (Publishers) Ltd.,
Glebe Cottage, Glebe House,
Station Road, Cheddington,
Leighton Buzzard, Bedforshire, LU7 7NA,
ISBN 0 86161 111 X

Published in Australia by:
Melbourne House (Australia) Pty. Ltd.,
Suite 4, 75 Palmerston Crescent,
South Melbourne, Victoria, 3205,
National Library of Australia Card Number and
ISBN 0 86759 114 5

Published in the United States of America by:
Melbourne House Software Inc.,
347 Reedwood Drive,
Nashville NT 37217.

Copyright (c) 1982 by Dr. Ian Logan

All rights reserved. This book is copyright. No part of this book may be copied or stored by any means whatsoever whether mechanical or electronic, except for private or study use as defined in the Copyright Act. All enquiries should be addressed to the publishers

Printed in Hong Kong by Colorcraft Ltd.

Preface

It is almost impossible to believe that within the space of only 2½ years SINCLAIR RESEARCH of CAMBRIDGE has manufactured and sold about **500,000** microcomputers. It was in the spring of 1980 that the revolutionary **ZX80** was launched. The machine was an instant success as it was the first really cheap microcomputer for the hobbyist. However, within only one year Clive Sinclair and his team were ready with the **ZX81.** This model was a great improvement on the ZX80 and took the development of a microcomputer with a low resolution, black and white display to a stage that is never likely to be attained again.

But now we have the **ZX SPECTRUM.** This machine has been developed directly from the ZX80 and ZX81, and by so doing Sinclair Research has produced a microcomputer with a superb **High Resolution** and **Colour** display.

It is, however, with some regret that the ZX80 and ZX81 have been superseded as they were both beautiful machines. They had a simplicity of operation that made them a pleasure to program. This does not mean the SPECTRUM is a difficult machine to use but I do feel that in order to get the 'best' from the new machine it will now take longer to write 'finished' and 'polished' programs.

This book has been written so that the reader can 'go beyond' the two fine manuals that come with the actual machine, and thereby develop a deeper 'understanding' of both the SPECTRUM and microcomputer systems in general.,

I wish to acknowledge the help given to me by:

Alfred Milgrom
The president of Melbourne House (Publishers) who has such a keen interest in microcomputing and who has done a great deal to advance the 'Sinclair' machines world-wide.

Dr. Frank O'Hara
My co-author on the 'Sinclair ZX81 ROM Disassembly: Part B' from whom I have learnt so much about the 'calculator routines'.

Nigel Searle
The head of the computer division of Sinclair Research who kindly sent me a SPECTRUM in June 1982.

And my wife Liz and my two daughters — Jackie and Carolyn, who have had to endure the writing of this book.

Contents pages

Preface

Chapters

1. The SPECTRUM microcomputer system 7
2. The BASIC commands and functions 22
3. The Z80 microprocessor 47
4. The mathematics of machine code programming 60
5. The Z80 machine code instruction set 71
6. Demonstration machine code programs 110
7. An outline of the 16K monitor program 135
8. Using the monitor program's subroutines 159

Appendices

i. Tables of Z80 machine code instructions 180
ii. DECIMAL-HEXADECIMAL conversion tables 186
iii. Currently available machine code handling programs 188
iv. SPECTRUM 'bugs' 189

1. UNDERSTANDING – The SPECTRUM microcomputer system

1.1 Making a start

It is always fun to dip into a book, opening it here . . ., and there . . .; but computers are the most logical of machines and everyone trying to improve their 'understanding' should start 'here' – at the beginning.

1.2 Three views of the machine

It is possible to describe any microcomputer system by taking three different views of the system.

The first is an overall 'system' view which will encompass the actual microcomputer and all its attendant peripherals. The second view is of the 'physical' parts of the microcomputer itself.

The third view is obtained by looking at the 'logical' workings of the microcomputer system.

The 'system' view is probably already familiar to most readers but it is included as there will be some people who are unfamiliar with the SPECTRUM system.

1.3 The 'system' view

The SPECTRUM microcomputer itself is a black plastic box of width 233 mm., depth 144 mm. and height 30 mm. On the top surface are the forty rubber keys that form the keyboard. Along the rear edge are, from left to right, the output socket that connects to the T.V. aerial input, the input socket that connects to the cassette player output, the output socket that connects to the cassette player input, the expansion port that may be joined to the printer, to the microdrives and other input/output devices, and finally the power socket.

The main printed circuit board with the Z80 microprocessor and the other electronic components, including the single loudspeaker, is found within the black box but separate from the keyboard. The main board and the keyboard are linked by two ribbon cables.

The system can be shown diagrammatically – see diagram 1.1.

Although the system will 'run' without the T.V. display it is not the manufacturer's intention that this be done as all the 'system reports' appear on the T.V. display and cannot easily be made to appear on the printer, or any other peripheral.

1.4 The 'physical' view

The main board of the SPECTRUM can be easily inspected by first removing the five retaining screws on the under surface of the black box and then

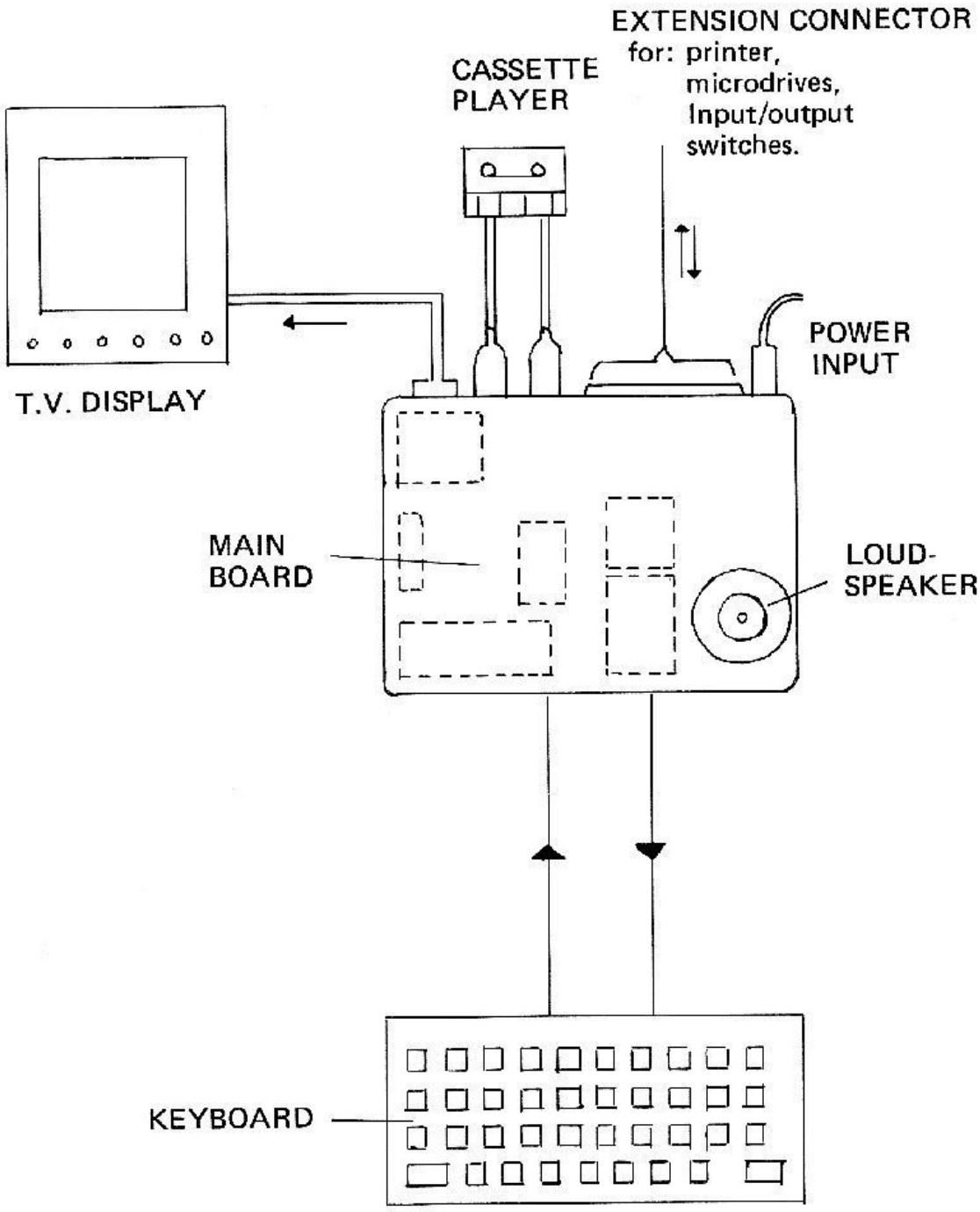


Diagram 1.1 The SPECTRUM microcomputer system (not to scale)

lifting up the upper half of the box. Care must be taken as the upper part of the box contains the keyboard and it is linked to the main board by two rather fragile ribbon cables. These cables may be pulled out of their sockets but it is not the author's advice that this be done, unless necessary, as the cables may be damaged. The use of two equal length pencils as 'stays' can also be helpful.

The major components found on the main board are shown in diagram 1.2.

Each of the major components will now be discussed in turn:

The Z80 microprocessor

This silicon chip is the most important of all the components. It is a 'microprocessor' and as such it is a machine capable of acting as a 'computer' which in a widely accepted way is 'a machine capable of following a stored program'. The program for a Z80 microprocessor will always be in the form of a set of Z80 machine code instructions and any associated data.

In the SPECTRUM the Z80 microprocessor is 'clocked' at 3.5 MHz and at that speed is capable of processing 875,000 of the more simple machine code instructions a second. It is interesting to note that at any time that the correct 'power', 'ground' and 'clock' connections are made, the microprocessor will be working. However the results of its work will be 'nonsense' unless the microprocessor is following a sensible machine code program.

The 16K ROM (= read only memory)

The machine code program that is normally followed by the Z80 microprocessor is supplied by Sinclair Research in a 'read only memory' chip that holds 128K bits, or 16K bytes, of information.

In the '16K monitor program' of the SPECTRUM roughly 7K is allotted to the 'operating system', 8K to the BASIC interpreter and the 1K remaining to the 'character generator'.

The 16K of RAM (=random access memory)

In the standard 16K version of the SPECTRUM there are eight 2K byte, or 16K bit, memory chips, whereas in the 48K version there is an additional 32K of memory.

Three of the eight memory chips form the 'memory mapped display' and would normally only be used for this purpose. The fourth memory chip is devoted to holding the attribute bytes for the 768 character areas of the display and the system variables. A little over 8K of RAM is left free in the 16K version.

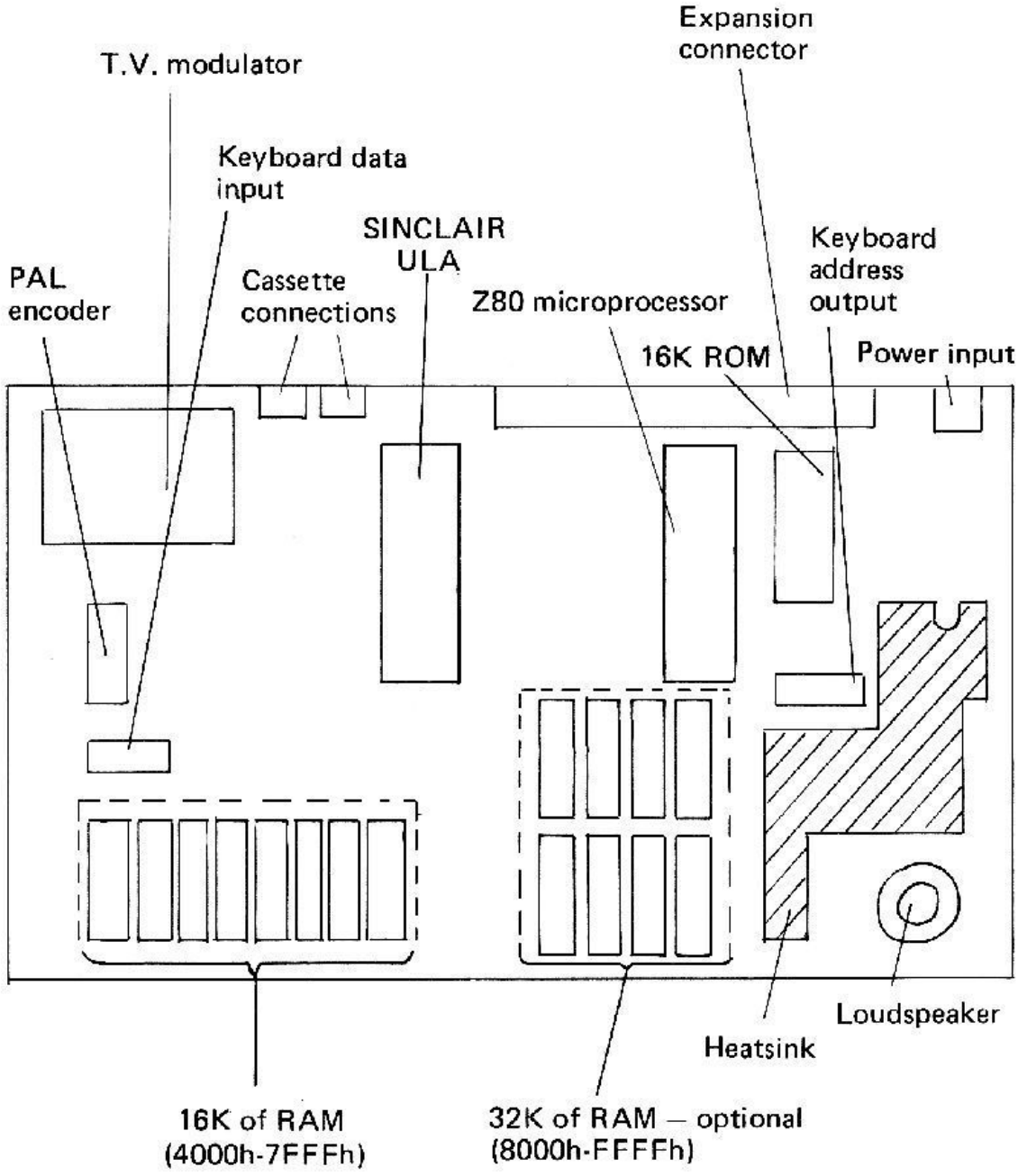


Diagram 1.2 The major components of the SPECTRUM'S main board (Issue Two)

The SINCLAIR ULA (=uncommitted logic array)

This chip can be considered as being a large chip made up of many smaller chips. In the SPECTRUM the ULA is largely concerned with the scanning of the 'memory mapped display area' and the 'attribute area' to produce the T.V. signal.

The PAL encoder

This chip receives the 'colour' information from the ULA and uses it to prepare the required signal for the UHF modulator. The signal produced from the modulator is nominally on channel 36 in the U.K. version of the SPECTRUM.

In addition to these major components there are the loudspeaker, the heat sink, the voltage regulator, the system clock, various address decoders and buffering chips and a modest number of other minor components.

1.5 The logical view

In this view the links between the various components of the microcomputer system are considered. These links do have a real existence — they are tracks on the printed circuit board, or even actual wires — but it is the use to which these links are put that has to be understood.

A Z80 microprocessor can generate an individual address for 65,536 different memory locations (64K). The limit on the amount of memory that can be linked to a Z80 microprocessor, in a straightforward manner, is therefore 64K. In the standard 16K SPECTRUM only the locations with the addresses, in decimal, from 0 to 32,767 are available to be used. Whereas in the 48K SPECTRUM all of the addresses from 0 to 65,535 actually address memory locations.

In the SPECTRUM, addresses are produced in the form of 16 binary signals. The address of location 0 is thereby 0000 0000 0000 0000 and that for location 65,535 is 1111 1111 1111 1111. The addresses are generated by the Z80 microprocessor and are carried around the computer on an ADDRESS BUS. There are 16 lines, or tracks, on the address bus of the SPECTRUM and an address will be specified by considering which of the lines carry a 'high' voltage and which carry a 'low' voltage. Because an address requires 16 binary signals it can also be described in 'two bytes', each of eight bits.

Whereas the address bus has 16 lines, the DATA BUS of the SPECTRUM has only eight lines. Therefore any data, whether it be a machine code instruction or a byte of data proper, can only be considered as being in the decimal range of 0 to 255, or the binary range of 0000 0000 to 1111 1111.

Diagram 1.3 shows a simplified view of how the address bus and the data bus are linked to the other major components of the SPECTRUM. The diagram also shows how it is possible to consider the 'ULA chip' as viewing

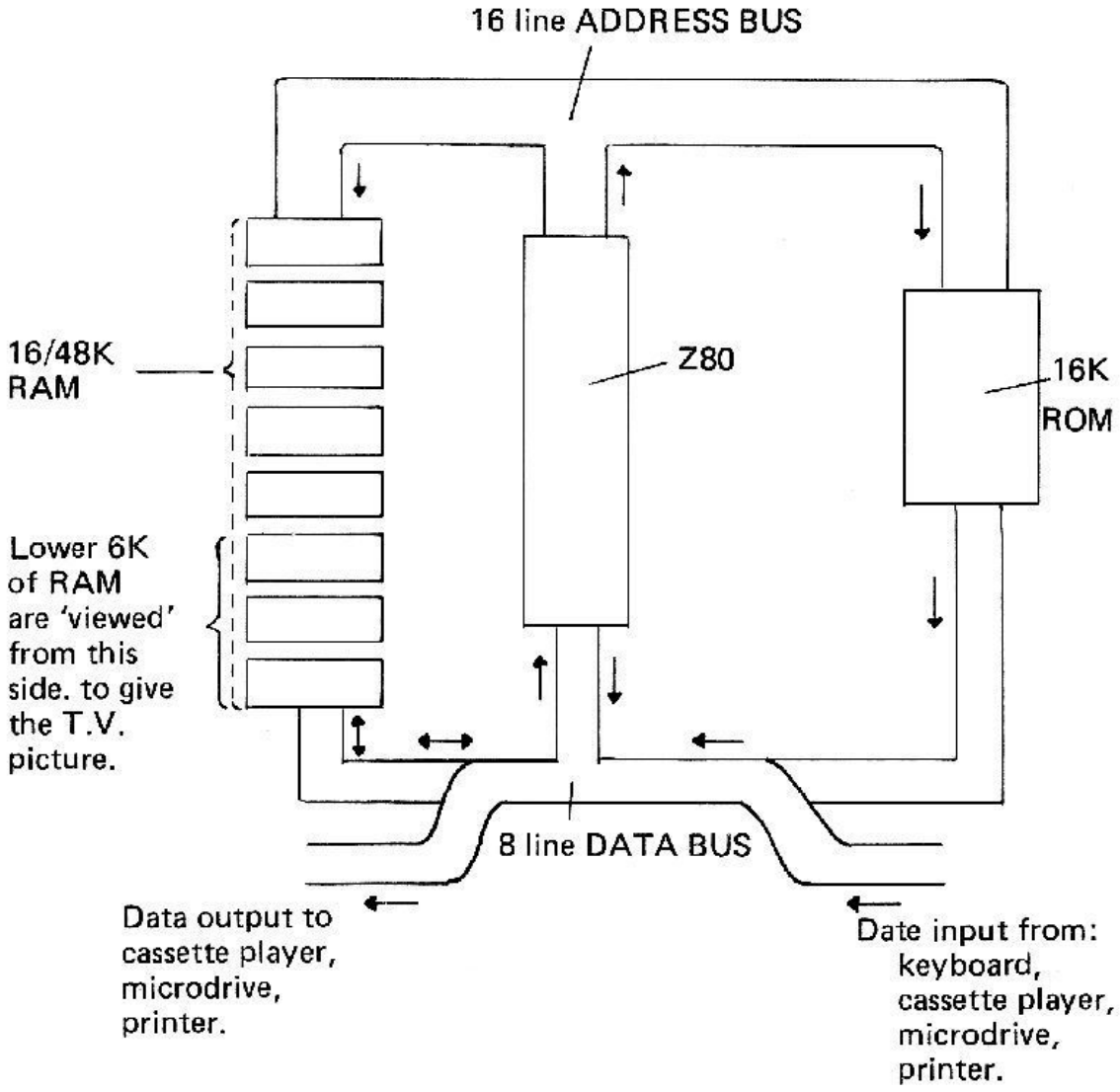


Diagram 1.3 The ADDRESS and DATA buses of the SPECTRUM system.

the 6k of memory reserved for the display 'from the other side' to that viewed by the Z80 microprocessor.

As part of this 'logical view' of the SPECTRUM it is also appropriate to consider the normal mode of operation of the system and discuss the 'memory map'.

The SPECTRUM is supplied by SINCLAIR RESEARCH with a 16K monitor program that provides the user with an operating system and a BASIC interpreter. It is indeed possible to leave this monitor program and have the Z80 microprocessor execute one's own machine code program if desired. In normal use the operating system of the SPECTRUM does not require any action on the part of the user and all the actions of the operating system are said to be 'transparent' to the user. Therefore, it appears that whenever the SPECTRUM is in use, it is the BASIC interpreter part of the monitor program that is being executed. The user is able to enter immediately BASIC program lines or execute BASIC programs. In a way of thinking, the operating system considers the BASIC interpreter as a subroutine that is to be 'run' as required; and the BASIC interpreter considers the line, or lines, of a BASIC program as containing instructions that direct it to 'run' the required subroutines of the interpreter.

Note that in no way does the Z80 microprocessor itself execute a BASIC program but only the monitor program that is in Z80 machine code. The only exception to this occurs when a user-written machine code program is being executed.

The memory map of the standard 16K SPECTRUM is outlined in diagram 1.4 and each of the 'areas' will now be discussed briefly.

The ROM area

The 16K ROM containing the operating system, the BASIC interpreter and the character generator occupies the locations decimal 0 to 16,383, hex. 0000-3FFF. As in any Z80 based microcomputer system, the start of the machine code program is found at location 0.

The memory mapped display area

The 6K of memory from locations decimal 16,384 to 22,527, hex. 4000-57FF, form the 'high resolution' display area. It is important to realise that this 'area' is fixed by the hardware of the SPECTRUM and cannot be altered under software control.

There is a one-to-one relationship between all the bits in this memory area and the pixels of the T.V. display and the following calculation shows that the number of bits in 6K of memory does equal the number of pixels of the display.

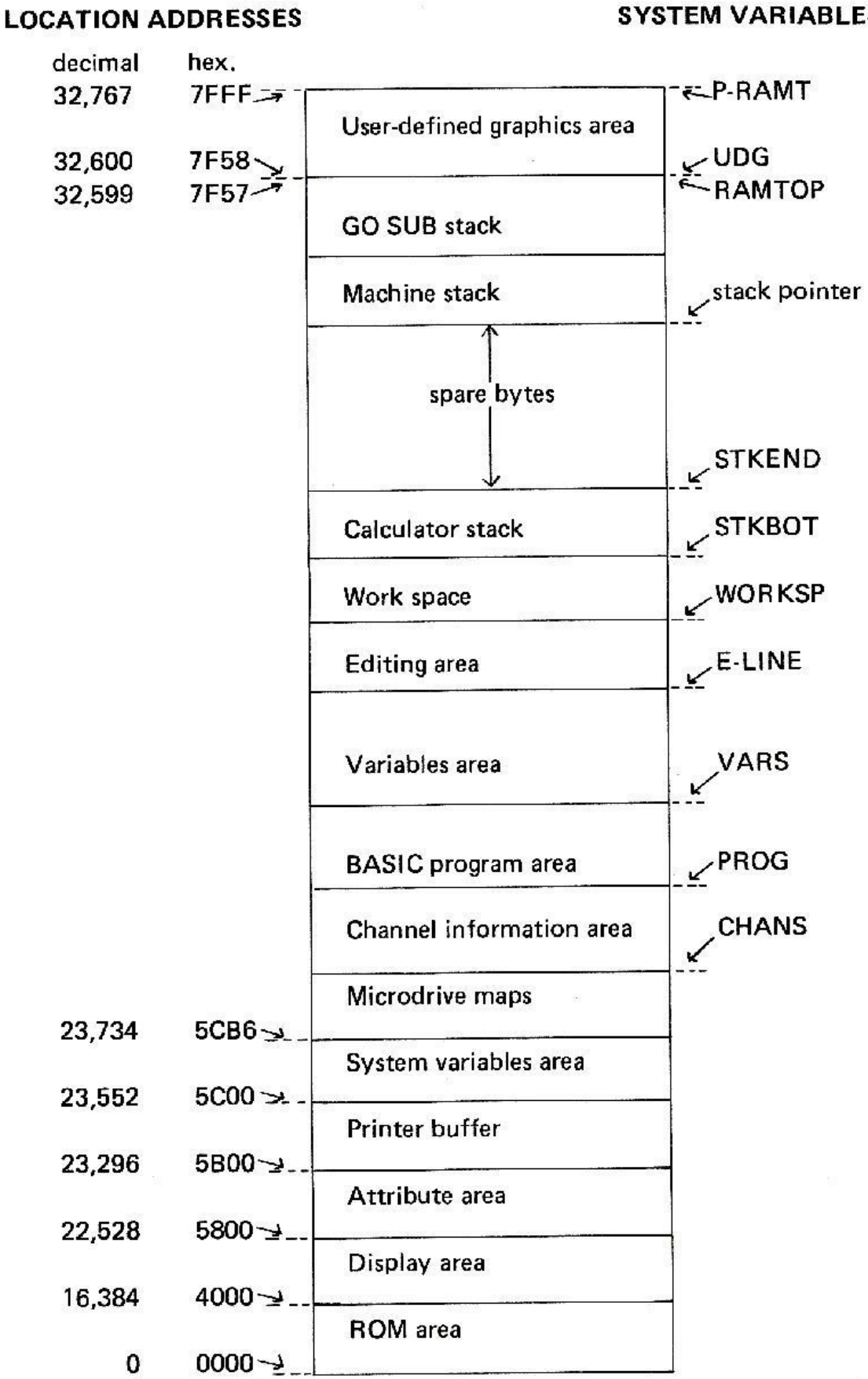


Diagram 1.4 The memory map of a 16K SPECTRUM

No. of bytes in 6K of memory = 1,024 * 6
= 6,144
No. of bits in 6K of memory = 6,144 * 8
= 49,152

No. of pixels in a 32 column by 24 line display with 64 pixels/character = 32 * 24 * 64
= 49,152

The relationship between the memory bytes and the character areas of the T.V. display is very straightforward but does lead to confusion.

Initially, consider the T.V. display in thirds. The top third of the display, lines 0 to 7, is produced by scanning locations decimal 16,384 to 18,431, hex. 4000-47FF. The middle third, lines 8 to 15, by locations 18,432 to 20,479, hex. 4800-4FFF; and the bottom third of the display, lines 16 to 23, by locations 20,480 to 22,527, hex. 5000-57FF.

Next, consider each of these 2K blocks as consisting of eight ¼K areas. The first of these smaller areas in each of the three blocks contains the bits for the top lines of the 256 characters in its third of the display, the second area the bits for the second lines of the characters, and so on for the eight lines of all the characters. This relationship applies to all of the 24 ¼K areas in the display area.

The attribute area

The T.V. display has 768 character areas, each of which can have one of eight PAPER colours, eight INK colours, be FLASHing or steady, and be BRIGHT or normal.

The locations from decimal 22,528 to 23,295 hex. 5800-5AFF, are used to hold the data that determines the current attributes for the display.

The relationship between the character areas and the attribute bytes is uncomplicated as the bytes are scanned so as to give the appropriate value for the characters on the top line of the T.V. display, going from left to right, then the characters on the second line and so on down the screen. In the attribute bytes, bits 0, 1 & 2 determine the INK colour, bits 3, 4 & 5 the PAPER colour, bit 6 is set for BRIGHT and reset for normal, and bit 7 is set for FLASHing and reset for steady.

The printer buffer

The locations between decimal 23,296 and 23,551, hex. 5B00—5BFF, are used as a printer buffer.

These 256 bytes are sufficient to hold 32 characters in their high resolution form with the first 32 bytes holding the bits for the top line of the characters, the next 32 bytes the bits for the second line, and so on.

Note that the printed buffer can be used as a 'work space' if required.

The system variables

The 182 locations from decimal 23,552 to 23,733, hex. 5C00—5CB5, hold the many different system variables of the SPECTRUM system. These system variables will be discussed in detail as the need arises throughout the remainder of this book.

The microdrive maps

This area of the memory begins at location decimal 23,734, hex. 5CB6, and has only a theoretical existence in a standard SPECTRUM, that is to say that the area is not used unless a microdrive has been fitted.

As this book is being written before the appearance of microdrives it is not possible to discuss the microdrive maps any further. However, by making the system variable CHANS point further 'up' the available memory it is possible to reserve any amount of memory (within the limit of available RAM) for the microdrive maps. (The use of the microdrive map area as a place to keep user-written machine code is an interesting point.)

The channel information area

This special area of the memory starts at the location pointed to by the system variable CHANS – itself stored in locations decimal 23,631 & 23,632, hex. 5C4F-5C50. The area is of variable size but ends with a location holding an end-marker of value decimal 128, hex. 80.

In the standard SPECTRUM, that is a machine without any microdrives attached, there are the input and output details for four channels. These channels are:

Channel 'K' – which allows input from the keyboard and output to the lower part of the display.

Channel 'S' – which does not allow any input but will allow output to go to the upper part of the display.

Channel 'R' – which again will not allow any input but will allow output to be passed to the work space. The size of the work space is expanded as required.

Channel 'P' – which again will not allow any input but will allow output to go to the printer.

The channel information consists, for each channel, of five bytes of data. These bytes are the output routine address which takes two bytes, the input

routine address which also takes two bytes and the file name which is a single letter code.

As there are four channels and an end-marker, the channel information area in a standard SPECTRUM occupies the twenty one locations from decimal 23,734 to 23,754, hex. 5CB6-5CCA.

The BASIC program area

This area of the memory holds the current program lines, if any. The size of the area depends on just how many BASIC lines exist.

The start of the program area is always given by the value held in the system variable PROG, which itself occupies the locations decimal 23,635 & 23,636, hex. 5C53-5C54.

Note that in the standard SPECTRUM the system variable PROG will indicate that the BASIC program starts at location decimal 23,755, hex. 5CCB, and this will always be so unless the microdrive map area is being used, or extra locations have been reserved for additional channel information.

In the program area BASIC lines are stored in the following format:

The first two bytes of any line hold the line number with the first byte being the 'high' byte and the second byte being the 'low' byte.

The third and fourth bytes of a line hold the 'remaining length'. This time the 'low' byte comes before the 'high' byte. The 'remaining length' is the number of bytes from the fifth byte to the final ENTER character inclusively.

Now comes the BASIC line itself. Sinclair codes are used for the tokens and some characters and ASCII codes for the standard alphanumeric characters.

The last byte of a line is always an ENTER character.

Within a BASIC line multiple statements are separated from each other by colons — character decimal 58, hex. 3A. There are no further markers for multiple statements. Note that if a decimal number occurs in a BASIC line then it is stored as its ASCII characters and followed by the NUMBER character — decimal 14, hex. OE, and the floating-point form for the number, or the integer form for integers in the range —65,535 to +65,535, which in either case will take a further five bytes. This leads to there being six extra bytes of RAM being used for every decimal number that is included in a BASIC program.

The following demonstration program 'looks at itself' in the program area and shows the above points.

PROGRAM AREA DEMONSTRATION PROGRAM

```
1Ø FOR A=23755 TO 23817: PRINT
A;TAB 9;PEEK A;TAB 15;CHR$ PEEK
A: NEXT A
 RUN
```

The variables area

The starting location of this area that holds the current BASIC variables is always given by the value held in the system variable VARS, which itself occupies the locations decimal 23,627 & 23,628, hex. 5C4B-5C4C.

Note that in the SPECTRUM system the start of the variables will not change during the execution of any given BASIC program. Its size will however change as new variables are defined.

The last location in the variables area always contains the end-marker character decimal 128, hex. 80.

The following program looks at its own variables area which contains only the control variable for the FOR-NEXT loop.

VARIABLES AREA DEMONSTRATION PROGRAM:

```
1Ø FOR A=23804 TO 23823: PRINT
A;TAB 9;PEEK A: NEXT A
RUN
```

The editing area

The starting location of this area that holds the BASIC line being entered, or edited, is always given by the value held in the system variable E-LINE, which itself occupies the locations decimal 23,641 & 23,642, hex. 5C59-5C5A. When the lower part of the T.V. display shows only the flashing cursor the editing area will have three locations allocated to it. The first location, whose address is also held by the systems variable K—CUR, holds an ENTER character, and the second location an end-marker — again a character decimal 128, hex. 80. The lower part of the T.V. display is obtained by copying over the 'edit-line' and displaying it.

Then, as characters are entered from the keyboard the editing area is expanded to hold them.

A similar procedure occurs when the EDIT key is used to fetch a BASIC line to the lower part of the display. First of all the editing area is expanded to the correct extent to allow for the BASIC line. Then the line is copied over from the program area to the editing area and finally the line in the editing area is copied over to the lower part of the display RAM. This last stage does involve the forming of high resolution representations from the character codes.

As the editing area is a dynamic area, that is it changes whenever it is used, it is not practical to give an example in BASIC at this point.

The work space

This area of the memory is used for many different tasks, e.g. INPUT data, the concatenation of strings, etc. The starting location of the area is given by the value held in the system variable WORKSP, which itself occupies the locations decimal 23,649 & 23,650, hex. 5C61-5C62. Whenever space is required in the work space this area of the memory is expanded. After use the work space is emptied, that is 'collapsed to nothing' so as to avoid using more locations than is absolutely necessary.

Once again as this area is dynamic it is not possible to give a simple BASIC example of its use.

The calculator stack

This very important area of the memory starts at the location addressed by the system variable STKBOT, which itself occupies the locations 23,651 & 23,652, hex. 5C63-5C64, and extends to the location before that addressed by the system variable STKEND, which occupies the locations 23,653 & 23,654, hex. 5C65-5C66.

The calculator stack is used to hold floating-point numbers, five byte integers and when dealing with strings, five byte sets of string parameters.

The stack is manipulated on a 'last-in first-out' basis and the value held at the top of the stack can be considered, if one does exist, as a 'last value'.

The spare memory

The area of memory between the calculator stack and the machine stack represents the amount of memory that is available to the user. In a standard 16K SPECTRUM there are nominally 8,839 locations in this area when the system is turned on. However it is interesting to note that the lowest acceptable value for CLEAR is 23,821 which pushes RAMTOP down by 8,878 bytes.

The machine stack

The Z80 microprocessor has to have an area of work space for its own use and this is termed the machine stack. The stack pointer of the Z80 always points to the last location to have been filled.

The machine stack will be considered in much greater detail in the machine code part of this book.

The GO SUB stack:

Whenever there are any active GO SUB loops the looping line numbers are kept on the GO SUB stack.

The stack grows downwards in memory and each GO SUB looping address requires three locations. The highest location holds the number of the state-

ment within the BASIC line to where the return is to be made. The next location holds the 'low' part of the looping line number and the third location the 'high' part.

The following demonstration program shows the GO SUB stack being used to hold the looping line numbers for three nested subroutines.

GO SUB STACK DEMONSTRATION PROGRAM:

```
1Ø GO SUB 2Ø: STOP

2Ø GO SUB 3Ø: RETURN

3Ø GO SUB 4Ø: RETURN

4Ø FOR A=32597 TO 32589 STEP -
1: PRINT A,PEEK A: NEXT A: RETUR
N

 RUN
```

The two locations above the GO SUB stack always contain the values Ø and 62, hex. 00 and 3E, and between them they represent an illegal line number. A BASIC program that contains an extra RETURN command will by attempting to make a jump to the illegal line cause the 'RETURN without GOSUB' error message to be printed. (Note: Sinclair is not totally consistent over the spelling of GO SUB.)

The system variable RAMTOP, which occupies the locations decimal 23,730 & 23,731, hex. 5CB2-5CB3, holds the address of the location that contains the value 62. This location is considered as being the last location in the BASIC system area.

The user-defined graphics area

Unless the BASIC system area has been moved down by the use of a CLEAR command, the top one hundred and sixty eight locations in the memory hold the bit representations of the 21 user-defined graphics.

As part of the initialization procedure of the SPECTRUM the bit representation of the latters A to U are copied to this area. Later on the user is able to change these representations to give up to 21 user-defined graphics.

The topmost location in the memory is always addressed by the system variable P-RAMT, which occupies the locations decimal 23,732 & 23,733, hex. 5CB4-5CB5.

In the standard 16K SPECTRUM the value held in P-RAMT ought to be 32,767 as this shows that all of the 16K of memory is in working order.

It certainly does no harm to occasionally enter the line:

PRINT PEEK 23732+256*PEEK 23733

and see that the result is indeed the value 32767. (In a 48K SPECTRUM the result should be 65535.)

2. UNDERSTANDING – BASIC commands and functions

2.1 Introduction

It is expected that the readers of this book will already have acquired a reasonable knowledge of the SPECTRUM'S BASIC so this chapter discusses the BASIC commands and functions trying to bring out points that are not mentioned in detail in the two handbooks supplied with each SPECTRUM.

The BASIC interpreter of the SPECTRUM recognizes fifty different commands and thirty three functions. Each of these will now be discussed briefly. They will be dealt with in alphabetical order so as to make reference to them a little easier. The control characters are discussed in section 2.4.

2.2 The BASIC commands

BEEP x,y

This command causes a note to be produced by the loudspeaker. x is the duration in seconds and y the pitch of the note away from middle C, within the range decimal –60 to +69.8. Either x or y, or both, can be expressions.

It is interesting to note that a BEEP cannot be interrupted as the program of the BEEP command routine does not check the BREAK key. A BREAK is only possible at the end of the statement containing the BEEP command.

BORDER m

There are eight possible colours that can be given to the border area of the T.V. display. The integer range for m is thereby 0 to 7. However m is rounded and the true acceptable range is $-0.5 \leq m < 7.5$. m can be an expression and is accepted as long as the result lies within the given range.

The effect of a BORDER m command is to send an OUT signal on port 254, and this can be shown by trying the line

OUT 254, m (where m=2 will give a red border).

But OUT and BORDER are different. The colour given to the border area by an OUT command is a 'temporary' colour, whereas a BORDER command gives a 'permanent' colour with the colour being stored in the system variable BORDCR. (location decimal 23,624, hex. 5C48)

This permanence can be shown as follows:

Enter BORDER 2 & ENTER
which will make the border area RED.
Then enter OUT 254,1 & ENTER
and the border will go BLUE but this is just temporary as a further ENTER will return the RED border.

Note also how BORDCR keeps the colour of the paper in the lower part of the T.V. display.

BRIGHT m

This is the first of the 'colour item' commands. All of these commands can be used either as the only command in a BASIC statement, in which case the command is 'permanent', or embedded in a printing statement, in which case the command is 'temporary'.

m can be an expression but only the integer results of 0, 1 & 8 are acceptable.

With m=0 the display will be of normal brilliance but with m=1 any future printing will be done on BRIGHT paper. The use of BRIGHT 1 & CLS is the easiest way of making the whole screen display become BRIGHT.

With m=8 any printing to be done will use the brilliance for a character area that is already assigned.

The following lines show the four different ways in which a colour item, such as BRIGHT can be used.

```
10 BRIGHT 1: PRINT "Bright—";:
   BRIGHT 0: PRINT "Normal"
```

which on two occasions changes the brilliance in a 'permanent' manner.

```
20 PRINT BRIGHT 1; "Bright—";B
   RIGHT 0; "Normal"
```

which changes the brilliance 'temporarily' during the statement.

```
30 PRINT CHR$ 19+CHR$ 1; "Brigh
   t—"; CHR$ 19+CHR$: 0 "Normal"
```

which replaces the command with its CHR$ equivalent.

```
40 LET A$=CHR$ 19+CHR$ 1: LET
   B$=CHR$ 19+CHR$ 0: PRINT A$;"Bri
   ght—"; B$; "Normal"
```

which puts the colour items into string variables.

It is also possible to use the keystrokes 'extended, unshifted 9' for BRIGHT 1, and 'extended, unshifted 8' for BRIGHT 0. These keystrokes may be placed inside a quote area, ie. between the open-quote and the first character, or quite advantageously in string variables to be used as required, e.g. LET A$"": REM Inside the quotes is an extended unshifted 9. Printing A$ will then act as BRIGHT 1.

CAT

For use with microdrives. (No details available yet)

CIRCLE x, y, z

This command draws a circle of radius z, with x & y giving the centre.

z is taken as an absolute integer, whereas x & y are manipulated as floating-point numbers.

The largest circle that can be drawn is of radius 88 units as with the line – CIRCLE 127.5,87.5,88 whereas a 'circle' with radius zero is a single point. Any of the 'colour item' commands may be embedded inside a CIRCLE statement and their effect will always be 'temporary'.

By some standards the CIRCLE command is rather slow and inaccurate but nevertheless it is very useful.

CLEAR and CLEAR n

As the SPECTRUM system has such a large amount of RAM available to the user, the use of a CLEAR command by itself is unlikely to be very helpful. However, the extension of the command to include a facility for moving RAMTOP makes it a powerful command. RAMTOP is the pointer to the top of the BASIC system and the contents of any location below RAMTOP is liable to be overwritten and thereby destroyed, whereas any locations above RAMTOP are safe — even from a NEW command.

The lower limit for n that is possible is 23,821 after which the SPECTRUM will buzz when a key is pressed. This shows that there is insufficient RAM available for the task.

The upper limit for n is, for a 16K system — 32,767, and for 48K system — 65,535. The use of CLEAR n with the appropriate number has the effect of putting the machine and GO SUB stacks in the area used for user-defined graphics. It is instructive to try the following steps as it is possible to see, indirectly, the contents of the stacks changing as they are used.

Enter CLEAR 32767 (or 65535).

Change the cursor to G and enter the letters L to U. Then hold down the SPACE key for several seconds and watch the user-defined graphics changing as each key press leads to the machine stack being used.

CLOSE

For use with microdrives.

CLS

An apparently very simple command which takes less than a tenth of a second to execute but it does involve the microprocessor in a lot of work.

The CLS command clears the display file. In effect it writes zeroes into all the locations from decimal 16,384 to 22,527, hex. 4000-5800, and resets all the attribute bytes in locations 22,528 to 23,295, hex. 5800-5AFF. The command does not set these latter locations to zero but rather copies the system variable ATTR-P into each location. ATTR-P, which is surely the abbreviation of 'permanent attribute' holds the current permanent attributes. By POKEing different values into ATTR-P at location decimal 23,693, hex. 5C8D, and then pressing ENTER an extra time, it will be seen that the screen changes predictably. In ATTR-P, bit 7 controls the FLASH, bit 6 the BRIGHTness, bits 3-5 the PAPER colour and bits 0-2 the INK colour.

CONTINUE

In most of the required instances this command works well. But when dealing with direct commands the user may find that the computer goes into a 'loop' that can be exited only be resorting to the BREAK key.

There are two distinct facets to the CONTINUE command. The first is to allow the user to have STOP statements in a BASIC program which may be stepped over by using the CONT key. The same operation works in respect to using the BREAK key. In this type of operation the user is able to examine variables, set variables and change the BASIC program in any way, except for deleting the STOP statement. The use of a STOP statement and the CONTINUE command can be very helpful when de-bugging programs.

The second facet allows the user to repeat the interpretation of a statement after correcting an error. For example it a program stops with a 'variable not found' error then the variable can be defined using a direct command and the program restarted using a CONTINUE command.

Six system variables — NEWPPC, NSPPC, PPC, SUBPPC, OLDPPC and OSPCC — are involved in some way with the execution of the CONTINUE command and the details are given in Chapter 25 of 'BASIC programming'.

COPY

This is a very straighforward command. Unless the printer is not attached to the SPECTRUM, the top 22 lines of the T.V. display are sent to the printer. The one hundred and seventy six high resolution lines of the T.V. display area are dealt with in turn. The COPY command is one of several commands that switch off the 'real time clock' and hence the clock loses time if COPY is used. This can be shown by examining the system variable FRAMES before and after the use of COPY.

DATA e, e, . . .

This command, which can only be used in a program line, sets up a data list. Although it is mentioned in the manual, it is not made very clear that the items in a DATA statement are dealt with as expressions a feature that does make this command very useful. For further details see READ & RESTORE.

DEF FN a(a, . . .z)=e & DEF FN a$(a, . . .z)=e

The DEF FN command is very powerful in the SPECTRUM.

The user is able to define up to 52 functions — 26 numeric and 26 string. The names used for the functions must always be single characters (+$ for strings) and they can be names that are used elsewhere as simple variables.

There is a slight restriction on the names of the arguments that can be used as these must also be single characters (+$ for strings). Therefore again up to

52 arguments are possible should they be needed. The expression of a defined function can be anything that gives the appropriate numeric or string result. However including the function itself as a defined function in the expression does lead to confusion (to date the only way the author knows to 'crash' the SPECTRUM in BASIC.)

DIM a(e_1,. . .,e_k) & DIM a$($e_1$,. . .,$e_k$)

The DIM command 'reclaims' any existing array with the same name and then sets up a new array as directed. Numeric arrays have zero in every location, whilst in string arrays 'space' characters are used.

In the SPECTRUM system all subscripts start as '1', or more strictly $0.5 \leqslant e < 1.5$. The common error of having a subscript reaching zero gives the 'subscript wrong' message. The use in the SPECTRUM system of fixed-length strings in string arrays, whereas simple string variables are of changing length, does lead to some confusion. But it is most straightforward as the last subscript used in the definition of any string array always fixes the length of the strings. This feature can be most useful when formatting a screen display.

DRAW x,y & DRAW x, y, z

This command draws a line from the current plot position, but not including it, to a point x away from it horizontally and y away from it vertically. If the argument z is specified then an arc is drawn instead of a straight line. z has to be specified in radians with z=PI giving a semi-circle. With positive values of z the arcs are drawn to the right-hand side of where the straight line would have been and with negative values the arcs appear on the lefthand side.

Any of the 'colour item' commands can be embedded in a DRAW statement where their effect will always be 'temporary'.

ERASE

For use with microdrives.

FLASH m

This is the second of the 'colour item' commands. When used alone in a BASIC statement its effect is 'permanent' but when used embedded in a printing statement its effect is 'temporary'.

As with BRIGHT, see above, m can have the values 0, 1 or 8. With m=0 the character areas with not flash; with m=1 they will flash; and with m=8 the former condition will apply.

FOR A=x TO y STEP z

The FOR command is a most interesting command and generally very poorly understood despite its wide usage.

The tasks undertaken by the interpreter when dealing with a FOR command are to a) delete any existing variable which has the same name, or any existing control variables with the same name; b) Add to the existing variables a new control variable. This variable takes up nineteen locations of the memory. The first location holds the variable's naming letter, suitably marked. The next five locations hold the initial value of the FOR loop, stored as a five byte floating-point number. The next five locations hold the 'limit' value as a floating-point number. The 'limit' value is so called as it limits the number of times the FOR-NEXT loop is used. The next five locations hold the 'step' value. If unspecified the 'step' will be set to '1'. The final three bytes of the control variable hold the details of the looping line. The first two of these three bytes will hold the number of the line containing the FOR command and the third byte the statement number within the line, increased by one. The looping is thereby done to the next statement after the FOR command whether it be in the same line or not.

If the value, limit & step are integers between −65,535 and +65,535 then they are stored as 'integral' numbers rather than true floating-point numbers and are handled 20% faster. The following program demonstrates the contents of a FOR control variable. Note that the FOR command in line 1Ø sets up a dummy variable named 'A' that is not used later in the program.

PROGRAM TO DEMONSTRATE A FOR CONTROL VARIABLE.

```
1Ø FOR A=1.6 TO 2.1 STEP Ø.1
2Ø LET V=PEEK 23627+256*PEEK 2
   3628-1
3Ø FOR B=1 TO 19
4Ø PRINT B,PEEK (V+B)
5Ø NEXT B
 RUN
```

adding 25 LET V=V+25 will show the 'B' control variable.

c) The third action undertaken by the interpreter is a very special one. In the

SPECTRUM no error is caused by setting the STEP value in the wrong direction. This means that a line such as:

```
FOR A=2 TO Ø STEP 1
```

is allowable. However, it does lead to the whole of the FOR-NEXT loop being ignored. This can be shown by changing the above demonstration program as follows:

```
1Ø FOR A=1.6 TO 2.1 STEP -Ø.1
6Ø NEXT A
```

and the program will be executed successfully but there will not be any printing.

This ability to 'jump over' whole loops can both be an advantage — it allows limits of zero to be accepted, and a disadvantage as sometimes the expected results just fail to appear at all.

FORMAT f

For use with microdrives.

GO SUB n

This command leads to the execution of the subroutine that starts with the line number n, or the first line after that number.

After the subroutine has been completed control returns to the first statement in the program after that containing the GO SUB command. The following little program shows that in a 16K SPECTRUM the GO SUB stack starts by using locations 32,595 to 32,597.

```
PROGRAM TO DEMONSTRATE THE GO SUB STACK.
  1Ø GO SUB 2Ø: STOP
  2Ø FOR A=1 TO 3: PRINT A,PEEK
     (32594+A): NEXT A
  RUN
```

A powerful feature of the SPECTRUM's BASIC is that it allows for GO SUB and GO TO commands to make computed jumps. This feature is not available on many other machines although the common 'ON x GO SUB' and 'ON x GO TO' does allow a certain amount of computed jumping to be made.

GO TO n

A very straightforward command. The next line to be interpreted is to be line n, or the first line after n. Jumps may be made to REM statements although to do so is perhaps a little untidy.

A GO TO command stores its destination line number in the system variables NEWPPC & NSPPC, locations decimal 23,618 to 23,620, hex. 5C42–5C44.

Jumps to statements in a line that are other than the first are not allowed in normal practice.

IF x THEN s

In the SPECTRUM system the quantity zero is considered to be logically FALSE, and any quantity that is other than zero is considered logically TRUE. An IF x THEN s command will proceed to the 's', and any other statements in the program line, only if 'x' is TRUE.

The IF – THEN command does work well but the following little program has been written to show by the use of 'repeated division' that on occasions a quantity will never become FALSE. The problem arises in this manner from the way the SPECTRUM deals with the quantity 2 ↑ –128.

PROGRAM TO DEMONSTRATE A SPECIAL CASE OF IF – THEN.

```
1Ø LET A=1
2Ø IF NOT A THEN PRINT A: STOP
3Ø PRINT A
4Ø LET A=A/2
5Ø GO TO 2Ø
 RUN
```

And A will be printed out, over five screens, until it reaches a value of 2 ↑ –128 when a loop is created. However if line 4Ø is changed to : LET A=A*0.5, then zero will be reached.

INK n

This is the third of the 'colour item' commands. When used alone in a BASIC statement its effect is 'permanent' but when used embedded in a printing statement its effect is 'temporary'.

The value of n can be between 0 and 9. The eight principal colours used in the SPECTRUM have the codes in the range 0 to 7 inclusive and are clearly shown on the keyboard. The use of INK 8 results in any printing appearing with the ink colour already attributed to that character area. The use of INK 9 is, however, a little more complicated as the ink becomes only black or white depending on the paper colour for the character area being used. When the paper colour is dark, ie. black, blue, red or magenta, then the ink colour is white, but with a light paper colour, i.e. green, cyan, yellow or white, then the ink colour is black. The actual action of a 'permanent' INK

command is; for n=0 to 7, to set bits 0, 1 & 2 of ATTR—P as required; for n=8 to set bits 0, 1 & 2 of MASK—P; and for n=9 to set bit 5 of P—FLAG. If the INK command is 'temporary' then the appropriate bits of the temporary system variables are set instead.

INPUT . . .

This is a very powerful command and allows for a series of INPUT items together with the printing of any 'print items' that may be desired.

The following BASIC program shows the use of multiple INPUT items. Note that INPUT only works for the 'editing-area' of the T.V. display.

PROGRAM TO DEMONSTRATE MULTIPLE INPUT ITEMS.

```
10 INPUT "Name please ",A$;CHR
$ 13;"Age please ";B$
  20 PRINT AT 5,0;"Name";TAB 7;A
$;CHR$ 13;"Age";TAB 7;B$
```

The use of INPUT LINE . . . allows for a whole line to be taken in for a string variable. Once again 'print items' can be used but they must come before the word LINE. The following BASIC line is an alternative to line 10 in the above program.

```
10 INPUT "Name please", LINE A
   $;CHR$ 13;"Age please", LINE B$
```

It is interesting to see that the programmer has chosen 'cursor down' as the exit key for INPUT LINE.

INVERSE n

This is the fourth of the 'colour item' commands. Once again its effects can be 'temporary' or 'permanent'.

If INVERSE 1 is used then subsequent printing will appear with 'paper on ink', whereas with INVERSE 0 the printing is the normal 'ink on paper'.

The actual action of setting the inverse mode is to set bit 5 of P—FLAG when 'permanent' and bit 4 when 'temporary'.

LET v = e

This is the most fundamental of all BASIC commands. A variable is selected by the user and the interpreter then has to determine whether or nor the variable is a simple one, is to be found in a FOR-NEXT control variable, or forms part of an array. A simple variable will have any existing reference to

it 'reclaimed' and new space allotted. A control variable and an array variable will be located but not 'reclaimed'. Next the expression that is to give the value of the variable is evaluated and copied to the correct space.

All numeric variables have five locations allotted for a value. This value is then kept as a floating-point number or an integral. Simple strings have a dynamic length, that is the number of locations allotted to a simple string depends on the number of characters in the string at a particular moment. All array strings are of fixed length and a string that is assigned to an array will be truncated if it is too long, and assumed to be filled-out with space characters if it is too short, for the fixed area allotted to it.

The following demonstration shows a simple way to look at the variable area and allows the user to enter a variety of variables for examination.

PROGRAM TO LOOK AT VARIABLES IN THE VARIABLES AREA.

```
10 REM enter your variable.
20 LET V=PEEK 23627+256*PEEK 2
   3628
30 PRINT V,PEEK V: LET V=V+1:
   GO TO 30
```

Some suggestions for line 10 might be:

```
LET A=0
LET A=9E4
LET A$="AAA"
DIM A(2): LET A(1)=9E4
DIM A$(5): LET A$="1234567"
DIM A$(2,5): LET A$(1)="AAA"
```

LIST n

This command assumes that the value of n is to be zero unless specified otherwise. If line n exists then that line is made the current line and marked with the cursor. It is an interesting quirk of the system that LIST 49172, for example, is the same as LIST 20.

LLIST n

This command sends the listing of the BASIC program to the printer. Again n becomes the current line. For those readers with a sense of humour it is interesting to try LIST #3 & LLIST #2, and include them in programs.

LOAD

The LOAD command has many different facets to it and allows for the loading of BASIC programs, variable arrays and code blocks.

In general the LOAD command is covered very well in 'BASIC programming' and the following discussion will concentrate on the points not covered in the manual.

The information, code or program, stored on a cassette tape is held in two parts. The first part is a seventeen byte 'header' and the second part a 'code block'.

Each of these parts uses the same format of a set bit being represented by a burst of sound that is twice as long as that of a reset bit. In the SPECTRUM there is a leading marker byte – +00 for a 'header' and +FF for a 'code block' – and a trailing parity byte after both the 'header' and the 'code block'.

The 'header' block is split into five parts as follows:

1. A single byte of information which is '0' for a BASIC program, '1' for a numerical array, '2' for a character array and '3' for a block of code.
2. The next ten bytes hold the file name. A name that has more than ten characters is not acceptable.
3. Two bytes that hold the total length of the code block. In the case of a BASIC program only the program area and the variables area are kept on the tape.
4. Two bytes that for a BASIC program hold the starting LINE number, or for a block of code the start of that block.
5. Two bytes that for a BASIC program hold the length of the program area.

The 'code block' is simply loaded into the required area of RAM as directed by the information in the 'header'.

LPRINT

This command causes any 'print items' that follow to be sent to the printer. For further discussion see PRINT below.

MERGE f

This command allows for a BASIC program and its variables to be loaded from cassette tape and merged with the existing program and variables. Where the same line numbers occur in both programs only the new version is kept.

This applies also to variables with the same name and nature. The command routine works by treating the new program as a block of data that is to be loaded into the work space. Thereafter, the program areas are compared, line by line, and the new lines copied from the work space to the main program area. The program area is 'reclaimed' and 'expanded' as needed. Once the BASIC lines have been merged, a similar operation merges the variable areas.

MOVE f_1,f_2
For use with microdrives.

NEW
This command does a total system restart except that the system variables RAMTOP, P-RAMT, RASP, PIP and UDG keep their original values. The existing definitions of the user-defined graphics are also left untouched.

NEXT a
This command should be viewed as the command that does the work in a FOR-NEXT loop once the control variable has been set up. (see FOR above.) The steps involved in the execution of a NEXT command are as follows:

a) The control variable is located in the variables area and the value of the STEP is added to the VALUE, no matter whether the STEP is positive or negative.

b) Next the new VALUE is tested against the LIMIT, but here note does have to be taken of sign of the STEP. If the LIMIT is exceeded then no further looping is possible — the NEXT command is finished. However, if the LIMIT has not been reached then a further pass of the FOR-NEXT loop is made.

In the SPECTRUM the VALUE is always incremented before the limiting test is made, hence the final VALUE on leaving the FOR-NEXT loop, upon completion, will always be greater than the LIMIT for a positive STEP, and less than the LIMIT for a negative step.

OPEN #
For use with microdrives. However, it is possible to OPEN and CLOSE streams on a standard machine as follows:

Try the line PRINT #5;"WORKS?" and it will not work until the stream is opened by using the line OPEN #5,"S". The line CLOSE #5 will close the stream.

The following lines show INPUT being accepted from the keyboard and output being passed to the T.V. screen.

```
10 OPEN  #5,"K"
20 INPUT  #5;A$
30 OPEN  #5,"S"
40 PRINT  #5;A$
```

OUT m,n

This command enables the user to send signals to the output port of the SPECTRUM from BASIC.

In BORDER, see above, it was shown how port 254 controls the colour of the border, a different colour being given for n=0 to n=7. However, port 254 can also be used to control the loudspeaker and the following program shows how this is done. The program interestingly shows how it is possible to get an estimate of the time that the SPECTRUM takes to interpret a statement as a fast statement will give a high note, whilst a slowly interpreted statement gives a series of clicks.

PROGRAM TO SHOW THE LOUDSPEAKER BEING CONTROLLED

```
10 OUT 254,23: ----------: OUT 254,7: GO TO 10
```

In the program the user's statement under test might be, for example: PRINT;:, RANDOMIZE:, LET A=0:, or POKE 0,0:.

In the program OUT 254,23 de-activates the loudspeaker and OUT 254,7 activates it. The operation is repeated very rapidly to produce a sound.

As further input/output devices become available the OUT command will be used more frequently.

OVER n

This is the fifth of the 'colour item' commands. Once again its effects can be 'temporary' or 'permanent'. If OVER 1 is used then subsequent printing will be 'XORed' with that already existing in the character area being handled.

'XORing' a bit flips the state of that bit — a set bit becomes reset and hence a 'unplot' will occur, or a reset bit becomes set and hence a 'plot' will occur.

With OVER 0 all the bits in a given character area that are involved will be 'plotted'.

Note that with OVER 1, if a character or line is printed twice then it will disappear:

i.e. `10 OVER 1: PRINT "A"; CHR$ 8;"A"`

where CHR$ 8 is 'backspace'.

Bit 1 of P-FLAG is the actual controlling flag for OVER.

PAPER n

This is the sixth, and final, 'colour item' command. Once again its effect can be 'temporary' or 'permanent'.

As with INK, the command PAPER can be used with n=0 to n=9. The use of n=0 to n=7 is very straightforward. With n=8 the 'transparent' mode is set and the colour of the paper in the character area currently being handled will remain unchanged.

The use of PAPER 9 is a very useful command for the provision of titles. PAPER 9 determines that the paper colour of an area shall 'contrast' with the ink colour being used. Hence, with light writing — green, cyan, yellow & white — the paper will be black, but with dark writing — black, blue, red & magenta — the paper will be white.

The actual action of a 'permanent' PAPER command is; for n=0 to 7 to set bits 3, 4 & 5 of ATTR-P as required; for n=8 to set bit 3, 4 & 5 of MASK-P; and for n=9 to set bit 7 of P-FLAG.

PAUSE n

This command HALTs the Z80 for the period of n interrupts. The only work that is done during a pause period is therefore to handle the keyboard interrupt routine. The pause period will end after n interrupts, or upon a key stroke, whichever occurs first.

PLOT x,y

In the SPECTRUM's display there are 256 * 176 pixels, each of which may be controlled individually with the PLOT command. In normal operation the PLOT command will set the bit of the RAM corresponding to that pixel. The command also makes that pixel the current object of the system variable COORDS that records the details of the last pixel to be addressed.

Any of the 'colour items' may be used embedded within a PLOT statement and the use of OVER 1 can lead to 'UNPLOT' and INVERSE 1 to 'NOPLOT'. A PLOT command does convey the permanent INK colour to the character area involved but none of the other items unless they are specified 'temporarily'. (i.e. PAPER 8; FLASH 8; BRIGHT 8; is implied.)

POKE m,n

This command enables the user to enter values directly into the memory of the SPECTRUM. The acceptable range for m is from 0 to 65,535, but no error is given when the user tries futilely to POKE values into the 'read only memory'. The acceptable range for n is from −255 to +255. The values 0 to +255 are dealt with directly but those from −255 to −1 have +256 added to poke them.

PRINT . . .

This command allows for a variety of 'printing items' to appear on the T.V. display.

The items are separated from each other by 'separators' that determine whether, or not, space is to be left free in the display. The ';' means no space, the ',' means that tabulation to the next half-line is to be made, the '"' means that printing is to start on the next line. The position of the printing can also be positioned by using TAB and AT.

The items can be 'colour items', whose action will be 'temporary', numeric expressions or string expressions. Note that all the 'colour items', separators, and positioning controllers can be included as their appropriate CHR$ values and this can be useful.

RANDOMIZE n

This command sets the value of SEED, the system variable held in locations decimal 23,670 & 23,671, hex. 5C76-5C77.

If n is unspecified then the value for SEED is taken from the lower two bytes of FRAMES and can be assumed to be fairly random. If n is specified then this number is copied to SEED.

For further details of the pseudo-random number generator see RND below.

READ $v_1,v_2 \ldots$

This command is used in conjunction with a DATA list and can be considered as a multiple LET command. The items in the DATA are expressions and the items following the READ command are variables to which the expressions are assigned. Error messages are given if either the variables are inappropriate or the expressions are of the wrong nature. Note that it is indeed possible for a READ statement to include previously undeclared simple variables.

The system variable DATADD, held in locations decimal 23,639 & 23,640, hex. 5C57-5C58, is used to point along the DATA list. Initially this pointer is set to the location before the program area and is reset to this value by RUN.

REM . . .

This is a useful command as it allows comments to be made. The whole of the BASIC line after a REM command is considered to be the REM statement so any statement put after a REM statement will not be found by the BASIC interpreter.

RESTORE n

This command is used in conjunction with the DATA list set up in one, or more, DATA statements. If n=0 then the pointer DATADD is made to point to the location before the program area. If n is specified then DATADD is made to point to the location before the start of that line, should it exist, or the first line after line number n, if n should not exist. If n exceeds 9999, the

highest legal line number, or is past the final line number, DATADD will point to the last location in the program area.

RETURN

This command leads to the last entry on the GO SUB stack being fetched. If the entry forms a valid statement number then the interpreter will execute that statement next. Invalid entries will give the 'RETURN without GOSUB' error message.

RUN n

This important command allows the user to execute BASIC programs. If n is not specified it will be assumed that the user wishes the interpreter to commence at line 1. When n is specified the interpreter searches in the program area for a BASIC line with that line number, or the first BASIC line after if line n does not exist, and proceeds to interpret the line.

The RUN command initialises the required pointers by performing a RESTORE and a CLEAR before interpreting any lines.

SAVE

This command is covered very well in the manual — 'BASIC programming' and further details are given under LOAD, see above.

STOP

Whenever this command is interpreted the error message 'STOP statement' will be given. The use of CONTINUE as a direct command will then allow the program to be continued from the next statement.

As with any error situation the effect of a STOP 'error' is to transfer the error number to the system variable ERR-NR which is held at location decimal 23,610, hex. 5C3A. The error number is always one less than the error code that appears before the error message.

Once an error has occurred the interpreter stops its program execution routine and jumps to the error handling routine. There, the error number is transferred to ERR-NR and the 'command mode return address' (location decimal 4,867, hex. 1303) is collected. This 'return address' is always present during the time that a program is being interpreted in the two locations below the GO SUB stack. If there are no current GO SUB loops then the GO SUB stack will be empty but as different subroutines are entered the 'return address' will be moved down in memory and then up again as the subroutines are completed. The 'return address' is always pointed to by the system variable ERR-SP, which is held in locations decimal 23,613 & 23,614, hex. 5C3D-5C3E.

Note that the commands RUN and GO TO do not clear the GO SUB stack and hence if a BASIC program halts with open subroutine calls then ERR-SP

will gradually point lower in memory by three bytes for each subroutine. However the CLEAR command does remove any unwanted statement numbers on the GO SUB stack as part of its duties and thereby resets ERR-SP.

VERIFY

The presence of this command can be most reassuring to the user of a SPECTRUM. VERIFY allows for any 'program' that has been passed to the cassette player by SAVE to be checked against the original. The verification process does check both parts of the program, that is the 'header' and the 'data block'. Error R – tape loading error – is signalled if the recording on the tape does not match the original exactly.

2.3 The BASIC functions

ABS

All negative numbers are made positive, whilst positive numbers are left unaltered.

ACS

The argument x is taken as a cosine and the function returns the value in radians of the angle concerned.

AND

This is a binary operation and thereby requires two operands. If both operands are logically 'true' then the operation is 'true' overall. However, if one or both of the operands are logically 'false' then the operation is 'false' overall. A numeric expression that is 'false' will have the value zero and a string expression that is 'false' will have zero length, i.e. a null string. When the result is 'true' overall it is the first operand that is returned to the user.

ASN

The argument x is taken as a sine and the function returns the value in radians of the angle concerned.

ATN

The argument x is taken as a tangent and the function returns the value in radians of the angle concerned. ATN is one of the four functions evaluated in the SPECTRUM by using Chebyshev polynomials.

ATTR

This functions has the form ATTR (Line,Col). It returns to the user the value held in the specified attribute byte. The function is equivalent to:

```
PEEK (22528+Line*32+Col)
```

The attribute value can be considered as:

INK + PAPER*8 + BRIGHT*64 + FLASH*128

BIN

This is an interesting function as it allows any integer in the range 0 to 65,535 to be entered in its binary form. Any number of binary digits up to a limit of sixteen is permitted.

The following BASIC line is therefore quite legal and will produce the numbers 1 to 10.

```
10 FOR A=BIN 1 TO BIN 1010 STE
   P 00000001: PRINT A: NEXT A
```

Although the SPECTRUM allows for numbers to be entered in their binary form they cannot be printed in that form without using a BASIC routine.

CHR$

This function returns to the user the character, as a string, for the given code x.

The following BASIC line shows the character set being printed. The CONTINUE & ENTER keys will have to be used six times to step over the colour control codes.

```
10 FOR A=0 TO 255: PRINT CHR$
   A;: NEXT A
```

CODE

This function returns to the user the code for the first character in the argument x$. The code will be zero if x$ is a null string.

COS

The argument x is considered to be in radians and the function returns the appropriate cosine for that angle.

EXP

For a given argument x the function returns the value 'e to the x', where e is 2.7182818 . . .

EXP is another of the functions evaluated in the SPECTRUM by using Chebyshev polynomials.

FN

In the SPECTRUM there may be up to twenty six numeric and twenty six string user-defined functions. The function token FN must be followed by a single letter, or a single letter and the '$' character, and then the arguments required by that function enclosed in a pair of brackets.

IN

The argument x is used as a port address and the resultant value read in will be in the range 0-255. If the port is not being used the value will be 255.

INKEY$

This function allows for the keyboard to be scanned without resort to the INPUT command. If a single key is being pressed then the in-key-string will be a string of length one character, with that character being the appropriate one. INKEY$ does distinguish both lower and upper case characters, and the symbol shift tokens. Various of the other keys give spaces whilst others give question marks. If there are no keys being pressed, or more than one letter/digit key being pressed, then the in-key-string is a null string.

INT

For a given value x this function returns only the integer part. Negative values are first of all truncated, then '1' is subtracted so as to 'round down' the result.

e.g. +4.66 will yield +4, whereas −5.66 would be truncated to −5 and then reduced to −6.

LEN

This function finds the length of a given string. A null string will give the result zero.

However, note that 'colour items' obtainable from keys may be included in strings and these key items do have a length of two characters.

LN

For a given argument x the function returns the logarithm, to the base e, for that value of x.

LN is another of the functions evaluated in the SPECTRUM by using Chebyshev polynomials.

NOT

This is an interesting function as it is the only function that gives the 'logical' state of a value in the SPECTRUM system. However, it is important to realise that the result is inverted and that the true logical result is given by the rather awkward looking 'NOT NOT x'.

i.e. for x = 2 which is a 'true' value.
NOT x is 0 which is 'false',
& NOT NOT x is 1 which is the correct 'true' answer.
for x = 0 which is a 'false' value.
NOT x = 1 which is 'true',
& NOT NOT x = 0 which is the correct 'false' answer.

OR

This is a binary operation and requires two numeric operands. If either of the operands are logically 'true' then the operation is 'true' overall. However, if neither of the operands are 'true' then the operation is 'false' overall. A 'true' result will have the value one if the second operand is other than zero and the 'value of the first operand' if the second operand is zero.

PEEK

This useful function returns the value found in the location x in memory. The result will always be an integer in the decimal range 0 to 255 inclusive.

PI

A most straightforward function. The floating-point representation of PI/2 is held as a constant in the 'table of constants' in the read only memory. This value is collected, doubled and passed on for the user.

The value of PI used is approximately 3.141592653 . . . .

POINT

This function has the form POINT (x-plot,y-plot), and returns the value '1' if the pixel at that position is 'set' and the value '0' if reset.

RND

In the SPECTRUM the random numbers are generated by a pseudo-random number generator. The system variable SEED, located at decimal 23,670 & 23,671, hex. 5C76-5C77, is collected, modified and restored with each call to the RND function. The random number returned to the user is the new value of SEED divided by 65,536.

The value of SEED is set to zero upon initialization of the SPECTRUM. Thereafter it follows a sequence through the set of integers 0 to 65,535. No value will be repeated until all 65,536 numbers have been used.

The following two BASIC examples show these features.

a) THE CIRCULAR NATURE OF RND.

```
10 PRINT RND: FOR A=1 TO 65535
   : POKE 0, RND: NEXT A: PRINT RND
```

This program takes over twenty minutes to run, but it does show that the same RND number is produced every 65,536 calls. (POKE 0,RND is a dummy statement.)

b) THE MODIFICATION OF SEED.

```
10 POKE 23670,0: POKE 23671,0

20 LET Seed=0
```

```
3Ø LET Seed=Seed+1
4Ø LET Seed=Seed*75
5Ø LET Seed=Seed-INT (Seed/655
   37)*65537
6Ø LET Seed=Seed-1
7Ø PRINT Seed/65536,RND
```

Line 1Ø resets the system variable SEED to zero. The program shows that the same results are produced by the modifying steps – lines 30 to 60 – as by the call to RND.

SCREEN$

This is a most interesting function and very poorly covered in the manuals.

The form for this function is SCREEN$ (line,column) and the function returns a string containing the character at the given position. The function recognises the characters with codes in the range decimal 32–127, hex. 20–7F, whether they are normal or inverted.

The function works by comparing the contents of the 8 bytes for the given character area against the 8 byte representations held in the character generator. The function returns a null string if the character is not one of the ninety six characters in the 'generator'.

It is unfortunate that the function does not allow for comparisons to be made against the user-defined characters as this would indeed appear to be quite possible. It would, however, be fairly difficult to test for the normal graphic characters as these are not stored in 8 byte forms but are constructed when required.

Despite the amount of work involved in evaluating this function it is surprisingly fast. This can be shown quite simply by:

```
1Ø PRINT " @ ";SCREEN$ (Ø,Ø)
```

which appears almost instantaneous.

(See also appendix iv. The 'SCREEN$' error).

SGN

All positive numbers return the value '+1'. All negative numbers return the value '–1', whilst zero gives zero.

SIN

The argument x is considered to be in radians and the function returns the appropriate sine for that angle. This function is the fourth, and last, of the functions that are evaluated by Chebyshev polynomials.

SQR

This function returns to the user the square root of the argument x. An alternative to SQR is to use ' ↑ .5' and this is indeed how the SPECTRUM considered SQR in its calculator.

Square roots of negative numbers are not evaluated and give the 'Invalid argument' error message.

STR$

The argument for this function is considered as a numerical expression. The operation of STR$ involves evaluating the expression, printing it in the work space as it would be expected to appear on the T.V. screen if requested, and then returning to the user the parameters of the string.

TAN

The argument x is considered to be in radians and the function returns the appropriate TAN for that angle. The tangent is found by finding both the sine and the cosine of the angle, separately, and then dividing the one by the other. The evaluation of a tangent therefore takes twice as long as that of a sine or a cosine.

USR

In the SPECTRUM the token USR followed by a numeric expression is quite distinct in its usage from that indicated by USR followed by a string expression.

USR – number

This most important function can in essence be considered as a command. The argument x, which is a numerical expression, is taken to be the address of a user-written machine code program. When 'USR – number' is called the Z80 stops its execution of the monitor program and, instead, executes the instructions stored at location x and onwards. A machine code RETurn instruction will cause a return to the monitor program.

Note that whilst in a machine code program written by the user it is quite in order to modify the IY register as it is reset upon return by the monitor program. (But the maskable interrupt must be disabled whilst the IY register pair holds any value other than +5C3A.) However, the value in the alternate HL register pair must be saved and restored correctly if a successful return to BASIC is required.

The making of 'USR – number' a function rather than a command does have the advantage that a value can be returned. In the SPECTRUM the value is numeric and corresponds to the contents of the BC register pair.

USR – string

This function returns to the user the address in memory of the required user-defined graphic.

The argument of 'USR – string' is required to yeild a string of one character and that character must be in the range 'A to U', or 'a to u'.

In the standard 16K SPECTRUM the user-defined graphic area will normally start at location 32,600, hex. 7F58, but as it can be moved it is better to always consider the area as starting at the location addressed by the system variable UDG, which is held in locations 23,675 & 23,676, hex. 5C7B–5C7C.

If the user-defined graphic area does start, for the purposes of this discussion, at location 32,600 then this location is given by USR "A". The eight locations 32,600 to 32,607 will thereby hold the representation of the graphic 'A' that will be 'A' or as the user defines it. USR "B" will return the address 32,608, USR "C" gives 32,616, and so on for the twenty one graphic characters.

There would not appear to be a particularly easy method of defining characters but the use of multiple data statements and BIN does at least give a fair idea of the finished design. The following program illustrates this point and produces a fair attempt at a 'tick'.

PROGRAM TO DEFINE A GRAPHIC CHARACTER

```
10 DATA BIN ØØØØØØØ1
11 DATA BIN ØØØØØØ1Ø
12 DATA BIN ØØØØØ1ØØ
13 DATA BIN ØØØØØ1ØØ
14 DATA BIN ØØØØ1ØØØ
15 DATA BIN Ø1ØØ1ØØØ
16 DATA BIN ØØ1Ø1ØØØ
17 DATA BIN ØØØ1ØØØØ
18 FOR A=Ø TO 7: READ B: POKE
   USR "A"+A,B: NEXT A
19 PRINT CHRØ 144: REM = "A"
```

VAL

This function requires a string as its argument. The string is then treated as a numeric expression, evaluated and returned to the user. This function is really quite useful as it does allow for the evaluation of expressions that are held in strings.

VAL$

Whereas VAL is a useful function it would appear that this function can always be circumvented by programming the statement in another way.

VAL$ requires a string as its argument and as this string must be enclosed in quotes even a simple statement containing VAL$ may thereby have treble quotes, twice. The function then returns to the user a string result.

Both VAL and VAL$ might indeed prove a little more useful if they did not give the error message 'Nonsense in BASIC' when the functions fail but gave a null result.

2.4 The control characters

In the SPECTRUM there are eleven control characters that may be used.

The control character procedure is, perhaps, not very useful to the BASIC programmer but it can be most useful when called from machine code.

CHR$ 6 — print comma.

The print position is modified to be at the 'next half screen' position. Within a PRINT statement the use of ';CHR$ 6;' and ',' are interchangeable.

CHR$ 8 — backspace.

The print position is modified by backspacing one character area. This operation works even when backspacing to a previous line is involved. (But see appendix iv. The 'CHR$ 8' error.)

A particular use of CHR$ 8 is to allow for 'underlining' as in the following line.

```
10 PRINT "*";CHR$ 8; OVER 1;"_"; OVER 0; etc.
```

CHR$ 9 — rightspace

See appendix iv. The 'CHR$ 9' error.

CHR$ 13 — next line.

The print position is modified to be at the start of the next line. Within a PRINT statement the use of ';CHR$ 13;' and "'" are interchangeable but the use of CHR$ 13 is somewhat clearer in a printed listing.

CHR$ 16 to CHR$ 21 — colour items.

These control characters produce 'temporary colours'. They are, taken in order, equivalent to: INK, PAPER, FLASH, BRIGHT, INVERSE & OVER.

Whenever one of these 'colour item' control characters is used it must be followed by an appropriate 'colour number'.

The following line shows the INK colour being set to RED.

```
10 PRINT "xxxx";CHR$ 16;CHR$ 2;"yyyy"
```

CHR$ 22 – at.

The print position is modified to be that given by the following two characters. The first character gives the line number and the second character the column number.

The following line shows how a statement equivalent to:

```
10 PRINT AT 12,6; "Here"
```

would be written.

```
10 PRINT CHR$ 22;CHR$ 12;CHR$ 6; "Here"
```

CHR$ 23 – tab.

The print position is modified to be at the column given by the next two characters. If the present column number already equals or exceeds the new column number then the print position will be on the next line of the display.

A particular point to be made about TAB either used as a token, or as its control character form, is that 'space' characters are written from the old print position to the new position. This feature can on occasions be quite annoying but at other times quite useful.

The following lines show this feature.

```
10 FOR A=0 TO 20
20 PRINT PAPER 7;TAB RND*7; PA
   PER RND *6.5;TAB RND*20+9;A
30 NEXT A
```

Line 20 could also be written as:

```
20 PRINT PAPER 7;CHR$ 23; CHR$
   (RND*7);CHR$ 0; PAPER RND*6.5;CH
   R$ 23; CHR$ (RND*20+9);CHR$ 0;A
```

3. UNDERSTANDING — The Z80 microprocessor

3.1 Introduction

The Z80 microprocessor is the most important silicon chip in the SPECTRUM microcomputer system. The Z80 microprocessor was developed by Zilog Inc. of California, U.S.A. and has proved to be one of the most successful microprocessors ever designed. In the SPECTRUM it is actually a Z80A produced under a licence from Zilog Inc. that is used.

Once again it is possible to split the discussion on the Z80 microprocessor into two parts. The first will give a 'physical' view of the chip and the second a 'logical' view. Following on from the 'logical' view the structure of a machine code program will be discussed.

3.2 A 'physical' view of the Z80

The Z80 is a silicon chip with forty pins numbered from 1 to 40. Diagram 3.1 shows the pin arrangement of the Z80 and the 'names' attributed to the pins. The functions of the pins, or lines, will now be discussed.

Pin 11 — the power line. A +5v. supply is required by the Z80 microprocessor.

Pin 29 — the ground line.

Pin 6 — the clock input. In the SPECTRUM the clock rate is 3.5 mhz. i.e. a clock pulse every 0.000000286 of a second.

Pins 7-10, 12-15 — These eight lines form the data bus that carries 'data bytes' to and from the microprocessor.

Pins 1-5, 30-40 — These sixteen lines form the address bus that carries 'addresses' from the microprocessor to the memory.

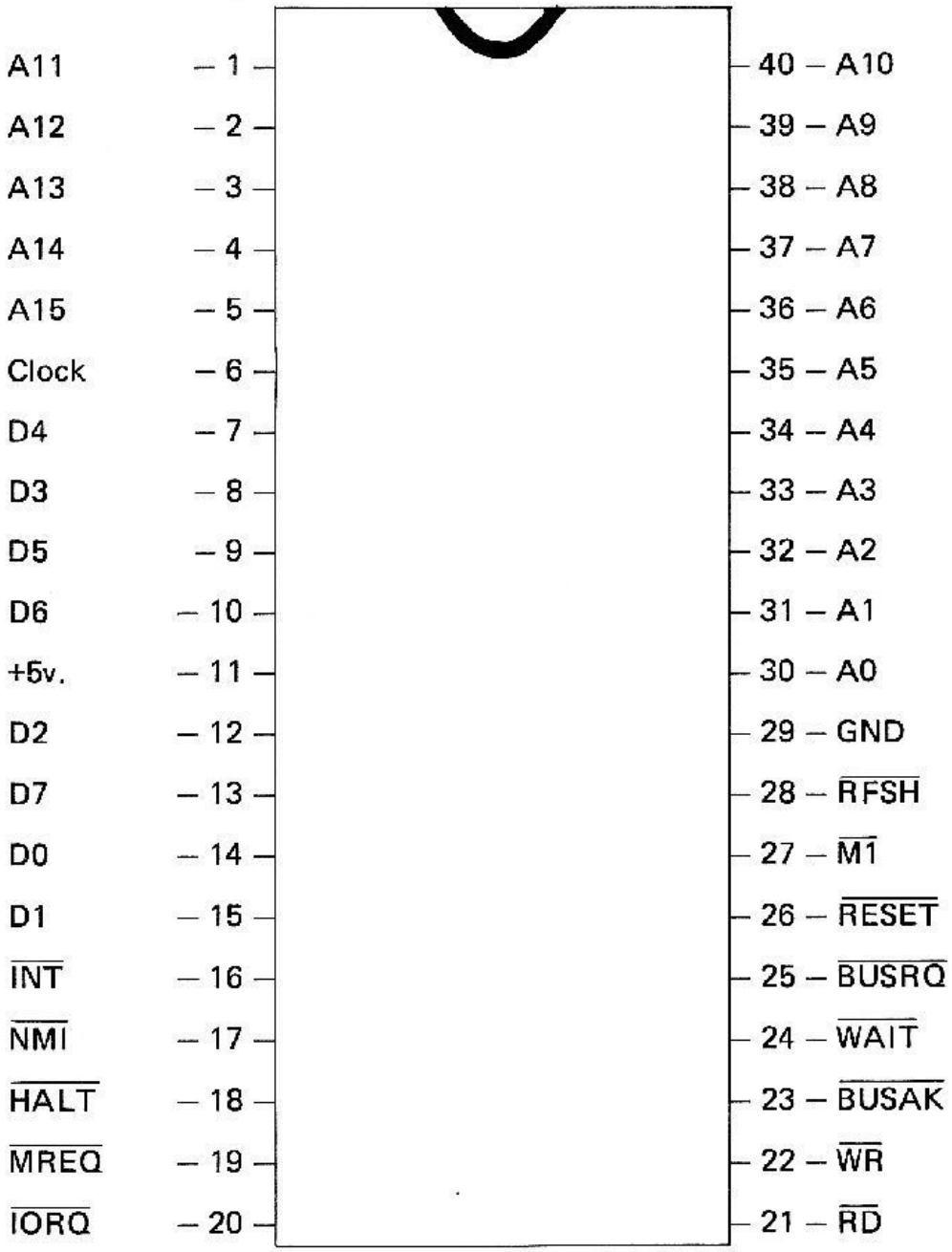
The remaining thirteen pins are attached to lines that carry control signals.

Pin 21 — The 'read' line, $\overline{RD}$. This line will be active whenever a byte of data is to be read from memory.

Pin 22 — The 'write' line, $\overline{WR}$. This line will be active whenever a byte of data is being passed from the microprocessor to the memory.

Pin 19 — The 'memory request' line, $\overline{MREQ}$. This line is activated whenever a byte of data is being passed to, or from, the microprocessor.

A byte of data is fetched from memory by first having the correct address placed on the address bus. Then in response to the signals $\overline{RD}$ & $\overline{MREQ}$ the appropriate memory chip will place the correct data byte on the data bus from where it is read into the microprocessor. A byte of data is written to memory by the microprocessor placing the required address of the memory location destined to receive the data byte on the address bus. The lines $\overline{MREQ}$ & $\overline{WR}$ are then activated and the data byte placed on the data bus. Thereafter, if a memory chip is indeed being addressed, the data byte will be copied into a memory location.



A0 – A15 are the 16 pins of the address bus and D0 – D7 are the 8 pins of the data bus.

Diagram 3.1 The 40 pins of the Z80

Pin 28 — The 'refresh' line, $\overline{\text{RFSH}}$. This line is used to refresh dynamic memories. In the SPECTRUM this line is partly involved in the generation of the T.V. scanning signals.

Pin 27 — The 'memory fetch' line, $\overline{\text{M1}}$. This is a most important line that is active whenever a machine code instruction, or an associated byte of data, is being fetched from memory.

The fetching of an instruction, or an associated byte of data, will require the three lines $\overline{\text{M1}}$, $\overline{\text{MREQ}}$ & $\overline{\text{RD}}$ all to be active, whereas the fetching of a byte of data from a location that is in another part of the memory will only require the lines $\overline{\text{MREQ}}$ & $\overline{\text{RD}}$ to be active. The time taken in the SPECTRUM for an instruction to be fetched is 1.14 micro-seconds which is four clock cycles, or T states.

Pin 20 — The 'input/output' line, $\overline{\text{IORQ}}$. This line is active whenever the specialised IN and OUT instructions are being executed.

Pin 18 — The 'HALT' line, $\overline{\text{HALT}}$. This line is active only when the HALT machine code instruction is being executed.

Pin 25 — The 'bus request' line, $\overline{\text{BUSRQ}}$. The Z80 allows for external devices to use the address bus and data bus in a 'cycle stealing' operation. The request to the microprocessor is 'steal the next cycle' is made by the external device by activating this line.

Pin 23 — The 'bus acknowledge' line, $\overline{\text{BUSAK}}$. The microprocessor acknowledges the 'request' by stopping the execution of further instructions and activating this line.

The remaining four pins are all under the control of the user.

Pin 26 — The 'reset' line. $\overline{\text{RESET}}$. This line is used to initialise the microprocessor. It is therefore activated when the power is first connected to the SPECTRUM. A 'reset' button can be provided for the SPECTRUM by joining $\overline{\text{RESET}}$ and GND in a suitable manner.

Pin 24 — The 'wait' line, $\overline{\text{WAIT}}$. A 'slow' memory may require extra time in 'read cycles' or 'write cycles' and this is signalled to the microprocessor by making $\overline{\text{WAIT}}$ active.

Pin 17 — The 'non-maskable interrupt' line, $\overline{\text{NMI}}$. This line, when active, leads to the microprocessor stopping the execution of the current machine code program. Instead the microprocessor executes an 'interrupt handling' routine written specially for this purpose. In the SPECTRUM system the non-maskable interrupt handler forms a 'system reset' that is dependent on the contents of location 23,728 being zero.

Pin 16 — The 'maskable interrupt' line, $\overline{\text{INT}}$. In the SPECTRUM system the scanning of the keyboard and the updating of the real-time clock are said to be 'interrupt driven'. By this is meant that the hardware of the system contains a clock that every 1/50th. of a second activates $\overline{\text{INT}}$ causing the

microprocessor to stop the execution of the main machine code program and, instead, execute the 'keyboard scanning and real-time clock' subroutine. The susceptibility of the Z80 to respond to the line $\overline{INT}$ can be controlled by the programmer with special machine code instructions.

3.3 A 'logical' view of the Z80

The internal structure of the Z80 microprocessor is amazingly complicated but fortunately it can be divided into five functional parts. These parts are the Control Unit, the Instruction Register, the Program Counter, the 24 User-registers and the Arithmetic-logic Unit.

Diagram 3.2 shows this simplified view of the Z80 microprocessor.

Each of the five parts will now be discussed in turn.

The Control Unit

The Control Unit of the Z80 can be likened, in a simplistic manner, to the 'manager of a production line'. It is the responsibility of the Control Unit, the manager, to arrange that materials (data bytes) are brought into the Z80, that finished products (also data bytes) are sent out to the correct destinations and to ensure that the 'production' is timed successfully.

In the Z80 the Control Unit produces a vast number of internal control signals that go to the other parts of the internal structure as well as the control signals that go out on the control lines $\overline{RD}$, $\overline{WR}$, $\overline{MREQ}$, etc.

It is important to appreciate that the Control Unit, like the production manager, is in no way responsible for deciding which work is to be done, but only for doing the actual work. The Z80 has to follow the program as written by the programmer and the production manager has to follow the 'program' as set out by his company directors.

The Instruction Register

The term 'register' is used to describe a single 'location' within the Z80 itself. It therefore is an actual place where the eight bits of a byte of data can be held together. In the Z80 microprocessor there is a large block of registers and the moving of bytes of data 'into and out of' the registers is the single most important feature of machine code programming.

The Instruction Register is a special register within the microprocessor that holds a copy of the machine code instruction currently being executed. One feature of the Z80 machine code instruction set is that certain instructions are held in two bytes of data. In these cases the Instruction Register can be considered as holding each of the bytes in turn.

When a machine code program is being executed the Instruction Register will hold each instruction in turn.

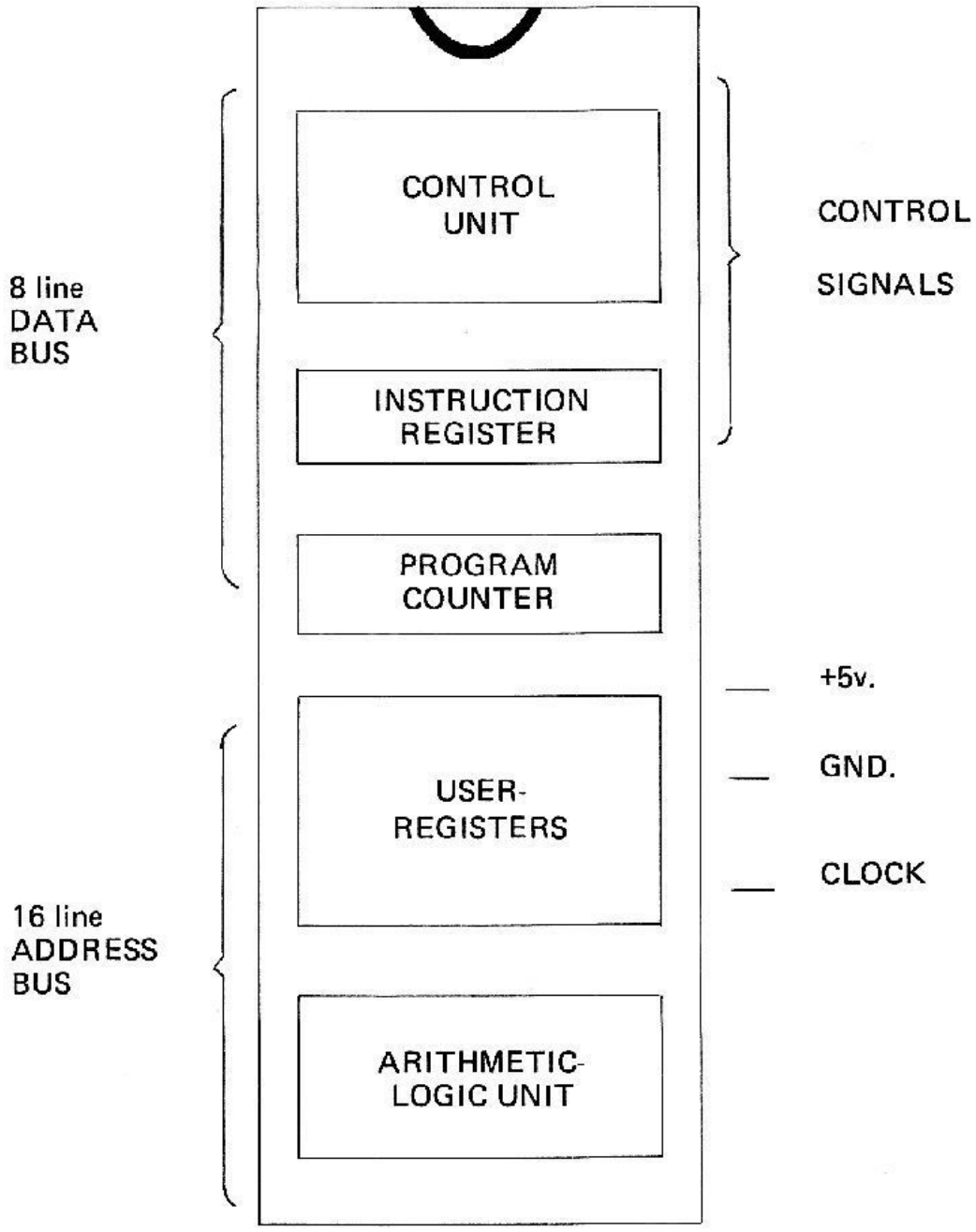


Diagram 3.2 A simplified view of the Z80 microprocessor

The Program Counter

The Program Counter is not a single register but a pair of registers used together. The Program Counter can thereby hold 16 bit numbers.

The Program Counter has the specific purpose of holding the address of the location in memory either of the current instruction being 'executed', or of the next instruction to be 'fetched', depending on whether the Program Counter has been advanced, or not.

When an instruction is to be 'fetched', the Control Unit uses the current address in the Program Counter as the address of the location in memory that holds the 'instruction' that is the next one to be executed. This instruction is copied into the Instruction Register. It is then that the Control Unit 'advances' the Program Counter.

The actions of the Program Counter are very similar to those of the BASIC interpreter's system variable PPC which holds the line number of the current BASIC line and is also 'advanced' once a line has been completed.

The User-Registers (Main Registers)

There are twenty four User-registers within the Z80 microprocessor. They are termed 'user' registers because they can be filled with bytes of data specified by the user, or programmer.

The names given to these twenty four registers are not, at a first glance, logically arranged. The reason for this being that the Z80 microprocessor has evolved from earlier, and less complicated, models. Certain of the names hark back to the earliest microprocessors to be built whilst later names have been added in an 'ad hoc' fashion, some names proving to be more appropriate and informative than others.

All of the registers are single byte registers but they are commonly used as register pairs.

Diagram 3.3 shows the twenty four User-registers of the Z80 microprocessor displayed as twelve register pairs. The 'bit' numbers are also shown.

Each of these registers will now be discussed briefly:

The A register

This register is the most important register of the Z80. It is often called the 'accumulator', a name that goes back to those models in which there was only a single register that could be used to 'accumulate' a result.

In the Z80 the A register is used extensively for arithmetic and logical operations, and indeed there are many operations that can be performed only using the A register.

There is a great number of different ways in which a byte of data can be entered into the A register by the programmer and hence there are many machine code instructions that involve the A register.

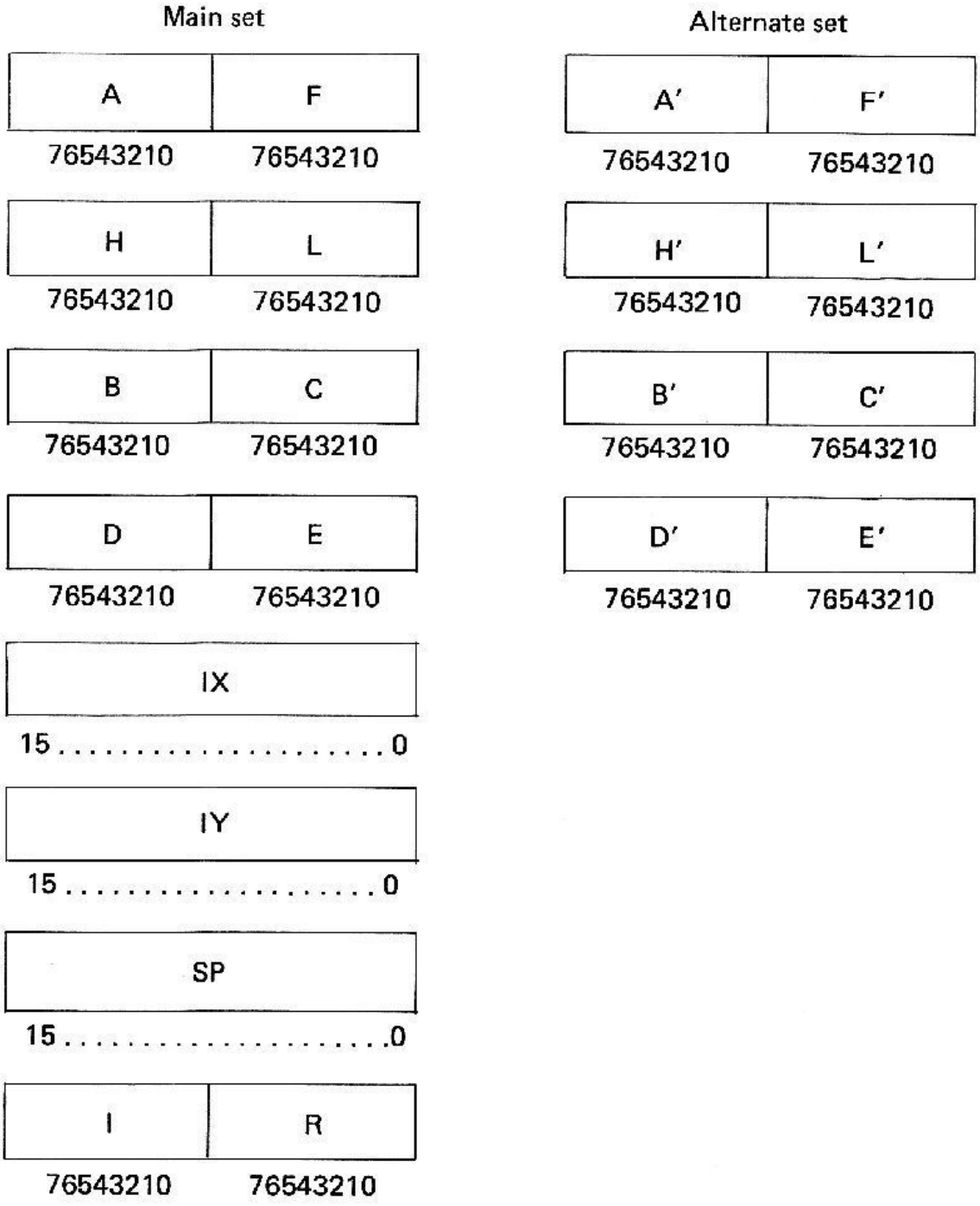


Diagram 3.3 The 24 User-Registers of the Z80

The F register

This is the 'flag register' and it is often considered as a collection of eight flag bits held together rather than as a true register.

The concept of flags will be dealt with in chapter 5, but simply a flag bit can be 'set', holding value '1' or 'reset', holding value '0'.

The flag register does hold eight bits but the programmer is normally only concerned with the 'four major flags'. These are the Zero flag, the Sign flag, the Carry flag and the Parity/Overflow flag.

The 'minor flags' are used by the Control Unit and cannot be used by the programmer in a direct manner.

The HL register pair

In early microprocessors there was a single 'addressing register' that could address 256 locations in memory. However, when 2-byte addressing was introduced it became possible to address individually 65,536 locations in memory in a straightforward manner. In a 2-byte address register microprocessor one of the registers is a High address register and the other a Low address register. The 'H' and 'L' names of the Z80 are thereby derived from the words 'high' and 'low'. It is interesting to note that the 'high' register, by being a later development, has led to the situation where an address is normally given with the 'low' part preceding the 'high' part.

A memory of size 65,536 locations can be considered as being divided into 256 pages of 256 locations and in such a case the value held in the 'high' address byte can be described as indicating which 'page' of memory is being used.

In the Z80 microprocessor the HL register pair is just one of three register pairs that are commonly used as addressing registers. However the HL register pair is the most important. The HL register pair can also be used to hold 16 bit numbers, rather than addresses, and there are a certain number of arithmetic operations that can be performed on these numbers. The H register and the L register can also be used as single registers although there are only a limited number of operations that can be performed.

The BC and DE register pairs

These register pairs are used mainly as addressing registers. It would appear that their names have come about solely because of the existence of an A register. Although 'DE' is clearly a suitable abbreviation for 'destination'.

Once again the individual single registers can be used by the programmer and it is especially common for the B register to be used as a loop counter.

The alternate register set

The Z80 is an interesting microprocessor as it has an alternate set of registers for the A, F, H, L, B, C, D & E registers. These alternate registers are designated A', F', H', L', B', C', D' & E', and spoken of as the A-prime register etc.

There are two special machine code instructions that allow for the contents of the alternate set of registers to be exchanged with the contents of the current set of registers. Once the registers have been 'exchanged' the Z80 will work with the 'former alternate set' believing it to be the 'main set'. The 'former main set' will now be treated as an 'alternate set'.

The programmer may exchange the register sets, totally or in part, as often as is wished in a particular program.

The concept of there being alternate registers may sound very simple but in practice the use of the alternate set of registers can be most confusing. The biggest problem is that it is for the programmer to remember which set of registers is being used as there are not any machine code instructions that only work on one set of registers and not the other.

The alternate set of registers is often used to 'save the environment' when starting an unrelated task. A particular example from the 16K monitor program of the SPECTRUM is that when the floating-point calculator is being used the H' & L' registers hold the 'return address'. Therefore if these registers are corrupted then a return to BASIC will not be possible.

The IX and IY register pairs

These two register pairs are used to perform operations that involve 'indexing'. This is a facility which allows for entries in a list or table to be manipulated. The base address of the list or table must first be placed in the appropriate IX or IY register pair.

In the 16K monitor program of the SPECTRUM the IY register pair is normally set to hold decimal 23,610, hex. 5C3A, which makes this address the base address for the table of system variables. The IX register pair is used extensively as a pointer in the LOAD, SAVE, VERIFY & MERGE command routines.

The Stack Pointer

This register is yet another addressing register. It is used to point to locations in the machine stack area of the memory and is always considered as a single 2-byte register.

The Z80 microprocessor uses a stack that 'grows downwards' in memory, so an analogy might be a high block of appartments in which the first tenant moves into the top apartment, the next tenant into the one below and so on downwards. The stack is used on a 'last-in first-out' principle, so the first tenant to move out will always be the latest tenant to have moved in.

The Stack Pointer is used to point to the different locations in the stack area in a special manner. The Stack Pointer always holds the address of the

last location to have been filled. Therefore when a new entry is to be made the Control Unit reduces the value held in the Stack Pointer before the entry is made. In the Z80 system each transfer to or from the stack involves two bytes of data and therefore the Stack Pointer has to be reduced twice when data is passed to the stack and increased twice when data is copied from the stack.

The machine stack is normally used by the microprocessor as an area in which to hold the return addresses for subroutines but the programmer is quite at liberty to place numeric data on the stack and thereby use it as a work space. However it is a common programming error to then try to use this data as a return address without first ensuring that the Stack Pointer does indeed point to the required location.

The I register

This is the Interrupt Vector register. In Z80 based systems other than the SPECTRUM this register would normally be used to hold the base address of a table of addresses for handling different input/output devices. However in the SPECTRUM this facility is not used and the I register is involved with the generation of the T.V. frame signals.

The R register

This is the Memory Refresh register. This register is a simple counter that is incremented every time a 'fetch cycle' occurs. The value in the register cycles over and over from 0 to 255.

The R register is used to generate part of the address required to address dynamic memory so that it can be 'refreshed' (recharged).

The Arithmetic-logic Unit (ALU)

This part of the Z80 is concerned, as its name implies, with arithmetic and logical operations.

It is important to realise that the operations that can be performed by the ALU are very limited. Simple binary addition and subtraction are possible but multiplication and division are not. Incrementation (adding 1) and decrementation (subtracting 1) are also readily handled. The unit is also able to perform a large number of 'bit' operations and thereby set 'flags' to show the result.

3.4 The structure of a machine code program

As has been stated above the Z80 microprocessor works as a computer as it is a machine capable of following a stored program. This program must exist as a set of machine code instructions, and any associated data, held in consecutive locations in memory. In a Z80 based microcomputer system these

locations in memory hold 8 bits, or 1 byte of data. A machine code program therefore consists of a set of data that appears as a series of 8-bit numbers.

The most elementary description of a machine code program shows the actual bit representation. The following example shows this for the first eight locations of the 16K monitor program of the SPECTRUM.

decimal		binary
Location	0 —	1111 0011
Location	1 —	1010 1111
Location	2 —	0001 0001
Location	3 —	1111 1111
Location	4 —	1111 1111
Location	5 —	1100 0011
Location	6 —	1100 1011
Location	7 —	0001 0001

The above representation is a perfectly valid manner of showing a machine code program but it is a very laborious form and prone to error. It is also undocumented and therefore not too informative.

The next sample shows the same piece of the monitor program expressed in decimal and hexadecimal notation. But once again it is undocumented and unhelpful.

decimal	hex.		decimal	hex.
Location 0	0000	—	243	F3
Location 1	0001	—	175	AF
Location 2	0002	—	17	11
Location 3	0003	—	255	FF
Location 4	0004	—	255	FF
Location 5	0005	—	195	C3
Location 6	0006	—	203	CB
Location 7	0007	—	17	11

The decimal results can he shown by the following BASIC program that PEEKs locations 0 to 7.

```
10 FOR A=0 TO 7
20 PRINT "LOCATION";TAB 10;A;T
   AB 15;PEEK A
30 NEXT A
```

The BASIC program to show the hexadecimal results is a little more complicated.

```
10 FOR A=0 TO 7
20 LET H=INT (PEEK A/16)
30 LET L=PEEK A-H*16
40 PRINT "LOCATION": TAB 10;A;T
   AB 15;CHR$ (48+H+7*(H  >9));CHR$ (
   48+L+7*(L  >9))
50 NEXT A
```

These two programs have been included here as they show the usual manner of producing a decimal or a hexadecimal listing of a machine code program.

Next comes the stage of documenting the listing that has been given in the example.

By reference to a table of Z80 machine code instructions (see appendix i of this book, or appendix A of 'BASIC programming') it can be found that the machine code instructions contained in the first eight locations of the 16K ROM are:

		mnemonic	**comment**
Location 0	—	DI	Disable maskable interrupt.
Location 1	—	XOR A	Exclusive OR the A register.
Location 2-4	—	LD DE,+FFFF	Load the DE register pair with a constant.
Location 5-7	—	JP +11CB	Make an absolute jump.

In the above description a 'mnemonic' has been given to each of the machine code instructions. A 'mnemonic' is a stylised way of representing an instruction in a helpful manner. All the machine code instructions of the Z80 instruction set have their own mnemonics and a machine code program is normally described using these mnemonics rather than the binary, decimal or hexadecimal numbers.

Note that in the above description two of the 'instruction lines' use a single location each, whilst the other two 'instruction lines' each take three locations. In the latter cases the first location of the three holds the actual instruction code proper and the remaining two locations the data associated with the instruction.

The usual form for showing a machine code program can now be given for this example.

address	**machine code**	**mnemonic**		**comment**
0000	F3	DI		Disable the interrupt.
0001	AF	XOR	A	Exclusive OR.
0002	11 FF FF	LD	DE,+FFFF	Top of memory address.
0005	C3 CB 11	JP	+11CB	Jump forward.

This form is called the 'assembly format' and normally has the addresses of the locations holding the first byte of the instruction line given in hexadecimal, the instruction codes and their associated data also given in hexadecimal, the mnemonics of the instructions and finally a 'comment field' where the programmer can write additional details.

The above examples show how the 'assembly format' for a given block of Z80 machine code can be derived — an operation that is normally termed 'disassembly' and a computer program that would perform the task is termed a 'disassembler'. (see appendix iii for further details)

The operations detailed above would be undertaken in reverse order when a machine code routine is being written.

First, the programmer would write the program using mnemonics. Then the actual code in binary, decimal or hexadecimal would be found. A computer program that accepts statements containing mnemonics and produces the required machine code — the object code — is called an 'assembler'. (again see appendix iii for further details)

Note that the 'assembly format' given above is often extended to include 'labels' and 'variables'. The example used above might then be written as:

```
                                        comment
        START       equ.    0000        Naturally location zero.
        START/NEW   equ.    11CB        Continue here.
        TOP-MEM     equ.    FFFF        Decimal 65,535.

address label       mnemonic            comment
0000    START       DI                  Disable the interrupt.
0001                XOR  A              Exclusive OR.
0002                LD   DE,+TOP-MEM    Top of memory address.
0005                JP   START/NEW      Jump forward.
```

When discussing 'assembly format' it must be appreciated that it is not a fully defined format and that different 'assembler' programs will have slightly different requirements and limitations.

4. UNDERSTANDING — The mathematics of machine code programming

4.1 Introduction

In a Z80 based microcomputer system such as the SPECTRUM all the data transfers involve the moving of 8-bit data bytes. The most accurate representation of these data bytes is made by using 8-bit binary numbers. But numbers in this particular form are difficult to handle and hence machine code programmers normally use a hexadecimal representation.

In this chapter the different sections will in turn deal with hexadecimal coding, absolute binary arithmetic, 2's complement arithmetic, integral representation and floating-point representation. The first three topics apply to any 8-bit microcomputer system, whereas the latter two topics are somewhat different in the SPECTRUM system.

4.2 Hexadecimal coding

The principle behind hexadecimal coding is to describe numbers 'to the base sixteen' rather than 'to the base two' as in binary coding or 'to the base ten' as in the decimal system.

In hexadecimal coding the first nine characters are the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 & 9 and the additional six characters are the letters A, B, C, D, E & F.

The following table shows the binary, decimal and hexadecimal representations for the numbers one to fifteen.

binary	decimal	hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

As can be seen in the previous table a single hexadecimal character forms a representation for a 4-bit binary number. It therefore follows that an 8-bit binary number is represented by a pair of hexadecimal characters and a 16-bit binary number by four hexadecimal characters.

The examples below illustrate these points:

0000 0000 (binary) = 00 (hex)
0100 1111 (binary) = 4F (hex)
0000 0000 0000 0000 (binary) = 0000 (hex)
0100 1100 1010 1111 (binary) = 4CAF (hex)

The conversion of a number in one number system to another is difficult for most people to do even after some months or years of practice but it is a very useful skill. It is however relatively simple to write computer programs to do the required tasks.

The following BASIC program is a DECIMAL–TO–HEX. conversion program.

DECIMAL–TO–HEX. PROGRAM

```
10 INPUT "Decimal number",D
20 IF D>65535 THEN GO TO 170
30 PRINT "Decimal",D
40 DIM H(4)
50 DIM HØ(4)
60 LET H(1)=INT (D/4096)
70 LET D=D-H(1)*4096
80 LET H(2)=INT (D/256)
90 LET D=D-H(2)*256
100 LET H(3)=INT (D/16)
110 LET D=D-H(3)*16
120 LET H(4)=D
```

```
13Ø FOR A=1 TO 4
14Ø LET H$(A)=CHR$ (H(A)+48+7*
H(A)>9))
15Ø NEXT A
16Ø PRINT "Hexadecimal",H$
17Ø PRINT
18Ø GO TO 1Ø
```

In the program the decimal number is successively reduced to find the 'hexadecimal values'. In lines 13Ø–15Ø these values are converted to ASCII characters.

The next BASIC program shows HEX–TO–DECIMAL conversion.

HEX–TO–DECIMAL PROGRAM

```
1Ø DIM H$(4)
2Ø INPUT "Hex. characters",H$
3Ø IF CODE H$=32 THEN GO TO 2Ø
4Ø LET D=Ø
5Ø FOR A=1 TO 4
6Ø IF H$(A)=CHR$ 32 THEN GO TO
1ØØ
7Ø IF H$(A)<"Ø" OR H$(A)>"9" A
ND H$(A)<"A" OR H$(A)>"F" THEN G
O TO 12Ø
8Ø LET D=D+16↑(4-A)*(CODE H$(A
```

```
)-48-7*(CODE H$(A)>57))
  9Ø NEXT A
 1ØØ PRINT "Hexadecimal",H$
 11Ø PRINT "Decimal",D
 12Ø PRINT
 13Ø GO TO 2Ø
```

In the program the hexadecimal character string, H$, always has four characters. If characters are unspecified by the user the hexadecimal string will appear left-justified.

The last example in this section shows how a hexadecimal number can be converted to decimal be 'longhand'.

Hexadecimal number = 789A

Decimal equivalent =		7	*4,096	=	28,672
		8	* 256	=	2,048
		9	* 16	=	144
	+	A	* 1	=	10
		789A		=	30,874

Or if taken in pairs:

Decimal equivalent =	78	*	256	=	30,720
	+ 9A	*	1	=	154
	789A			=	30,874

Appendix ii contains DECIMAL-HEXADECIMAL conversion tables.

4.3 Absolute binary arithmetic

A single memory location, or a single register within the Z80 itself, holds an 8-bit binary number. This number can be considered as having the binary range 0000 0000 – 1111 1111, the decimal range 0–255 or the hexadecimal range of 00–FF. In none of the cases can the number be taken as being negative, or fractional, and this forms the fundamental point of 'absolute binary arithmetic' – the value in a memory location or single register, is always POSITIVE and is an INTEGER.

It is also important to realise that the value held in a memory location, or a single register, behaves in a 'circular' manner whenever 0 or 255 is reached. That means if an additional operation takes the value past 255 the final value is decreased by 256, whilst for a subtraction operation taking the value below zero the final value is increased by 256.

The following examples show this point.

Dec. 252 + 44 will give 40

or

Hex. FC + 2C will give 28

Dec. 87 − 200 will give 143

or

Hex. 57 − C8 will give 8F

The Carry flag is affected by most operations that require a 'carry'. For further details see chapter 5.

In a Z80 based system all of the numbers are in 'absolute binary arithmetic' but the programmer will often need to place a different interpretation upon the numbers being used so as to view them as 'positive or negative' and 'integer or fraction'. The next three sections in the chapter show the different interpretations used in the SPECTRUM system.

4.4 2's complement arithmetic

The concept behind 2's complement arithmetic is very simple. But, when it is being used in a machine code program the results can be most confusing.

The method allows for the programmer to consider the numbers in the binary range 0000 0000 – 0111 1111 as the equivalent of decimal 0–127, and the binary range 1000 0000 – 1111 1111 as the equivalent of decimal −128 to −1.

A result of this interpretation is to make 'bit 7' (the lefthand bit of an 8–bit number) act as a 'sign bit'. This bit will be reset, 0, for positive numbers and set, 1, for negative numbers.

Diagram 4.1 illustrates 2's complement arithmetic as applied to a single 8-bit number.

Note that it is quite possible to extend the principle of signed 2's complement arithmetic to 16-bit numbers when the range obtained will be −32,768 to +32,767 decimal.

The conversion of a negative decimal number to its 2's complement binary or hexadecimal forms is straightforward but it is often easier to refer to a suitable table. (see appendix ii. for such a table)

The steps involved are:

1. Find the binary form for the absolute decimal value.
   e.g. –54 will give 0011 0110
2. 1's complement the binary number – change the zeros to ones and vice versa.
   e.g. 0011 0110 will give 1100 1001
3. Add one in absolute binary arithmetic.
   e.g. 1100 1001 + '1' will give 1100 1010
4. Use this binary form or consider it in hexadecimal.
   e.g. –54 in 2's complement is 1100 1010 or CA.

The operations may be taken in the reverse order when converting 2's complement numbers to decimal numbers.

4.5 Integral representation

The BASIC interpreter of the SPECTRUM system uses five bytes to represent numbers. Integral numbers, i.e. any integer between –65,535 and +65,535 inclusive are normally held in an 'integral form' whereas any fractional numbers or integers outside the integral range are held as five byte floating-point numbers.

In the integral form the first byte is always zero. The second byte holds zero if the integer is positive and decimal 255, hex. FF, if the integer is negative. The third and fourth bytes hold the actual integer value as an unsigned 16-bit 2's complement number but note that the third byte is always the low byte and the fourth byte the high byte. The fifth byte is unused but always holds zero.

The following demonstration program shows the integral form, in decimal, for any suitable integer entered by the user. Line 20 of the program ensures that an integral form is given.

```
INTEGRAL FORM PROGRAM

  10 INPUT N

  20 IF N<>INT N OR N<-65535 OR
N>65535 THEN GO TO 10

  30 PRINT "Number chosen =",N

  40 LET V=PEEK 23627+256*PEEK 2
3628
```

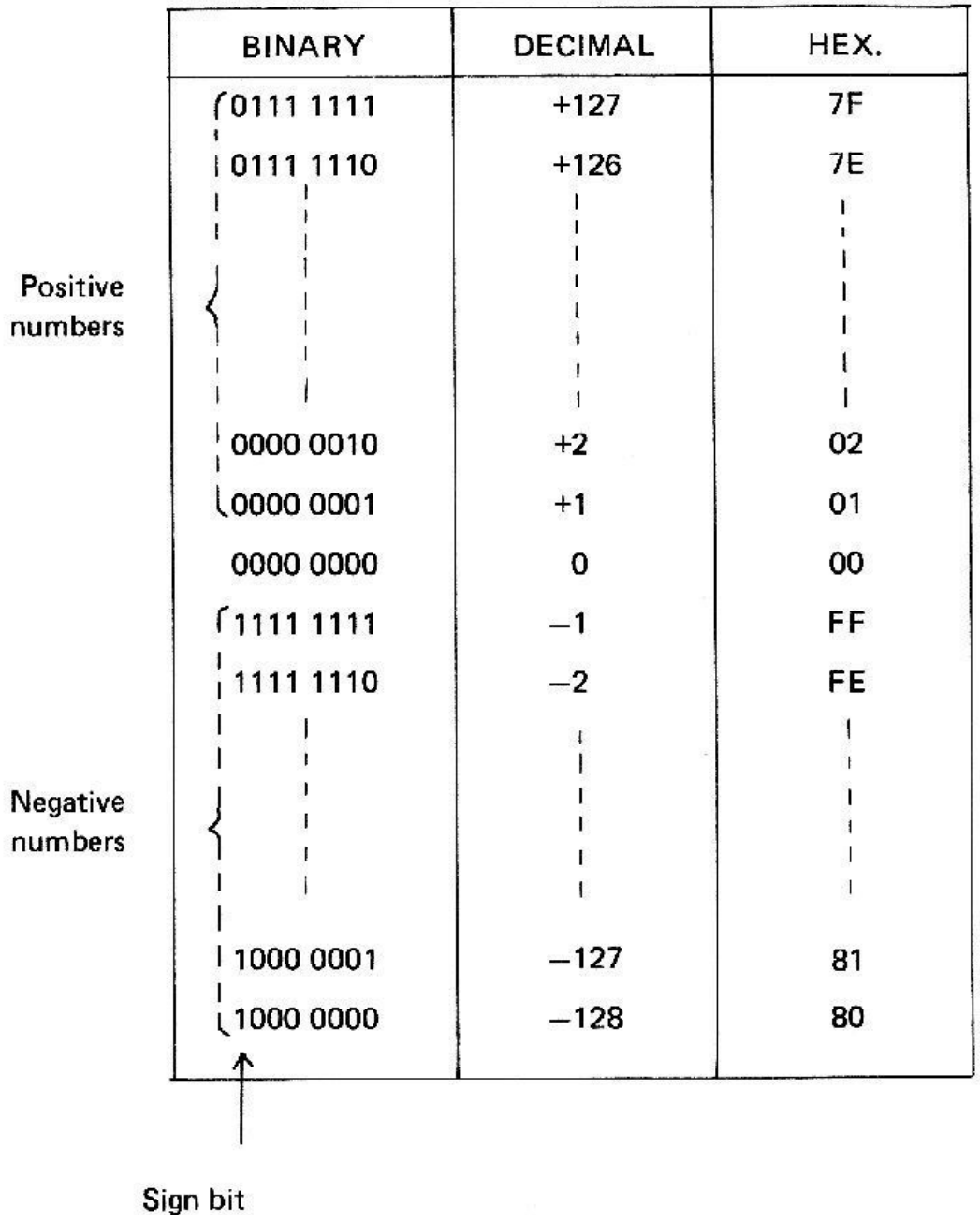


Diagram 4.1 2's complement arithmetic (single byte)

```
5Ø FOR A=1 TO 5
6Ø PRINT A;".";TAB 5;PEEK (A+V)
7Ø NEXT A
8Ø GO TO 1Ø
```

In the above program the pointer V points to the start of the variables area and locations 'V+1' to 'V+5' will hold the five bytes of the input number N.

The program gives results in the form:

Number chosen Ø
1. Ø
2. Ø
3. Ø
4. Ø
5. Ø

so zero is positive and has the value 0*1 + 0*256

Number chosen = 1516
1. Ø
2. Ø
3. 236
4. 5
5. Ø

so 1516 is positive and has the value 236*1 + 5*256.

Number chosen = −1
1. Ø
2. 255
3. 255
4. 255
5. Ø

so −1 is negative and has the 2's complement form of hex. FFFF.

Note: There is a programming bug concerning the handling of −65,536. This number does not get given the integral form: Ø, 255, Ø, Ø, Ø
(see appendix iv. for further details)

4.6 Floating-point representation

The five byte floating-point representation as used in the SPECTRUM system allows for all numbers in the range .29E−38 to 1.7E38 (roughly!).

The value zero is always stored as five bytes all set to zero. All other values are stored with an exponent part in the first byte and a mantissa part in the other four bytes.

The theory of describing numbers in terms of their exponents and mantissas will first be dealt with in decimal notation as that form is easier to manage initially.

Finding the exponent and mantissa for a decimal number is in essence the conversion of the decimal number into its E-format.

Hence, consider the number 1234.5 which can be expressed in E-format as .12345 E+4. To obtain the mantissa part the decimal point is moved leftwards until it comes to be in front of the most significant figure. The exponent is the number of moves required and the mantissa is the fractional decimal number.

Exp. = +4 ; Mantissa = .12345

As the SPECTRUM system deals with binary numbers rather than decimal numbers the same operations will now be considered for a simple binary number.

Hence, consider the binary number 0001 1111 which is equal to +31.

The binary point is taken to lie to the right of the 8-bits and it takes +5 moves to put it ahead of the most significant bit.

At this stage the exponent and mantissa are:

Exp. = +5 ; Mantiss = .1111 1000 . . . . .

In the SPECTRUM these parts are manipulated a little further. So for the exponent:

The true exponent, +5 above, is always increased by decimal 128, hex.80, to give the augmented exponent. In the example case +5 + 128 = 133.

And for the mantissa.

The first bit of the mantissa will always be a set bit so it is taken over for use as a sign bit; i.e.

when dealing with a positive number this bit is reset,

whilst for a negative number it is left set.

In the example case the mantissa becomes .0111 1000 . . . . . It now remains to express the number in five bytes. The unused bytes of the mantissa being set to zero.

In decimal the floating-point form of +31 will be:

133, 120, 0, 0, 0

and in hexadecimal:

85, 78, 00, 00, 00

or if wanted in binary:

1000 0101 0111 1000 0000 0000 0000 0000 0000 0000

The same operations apply to negative and fractional numbers although the steps are not as easy to follow as with a simple integer as above.

The following demonstration program shows the decimal form of floating-point numbers. The inclusion of line 30 ensures that the integral form is not given in response to suitable integers.

FLOATING-POINT FORM PROGRAM

```
 10 INPUT N
 20 IF N=0 THEN GO TO 40
 30 LET N=N+.2E-38
 40 PRINT "Number chosen =";N
 50 PRINT
 60 PRINT "Exp.";TAB 9;"Mantiss
a"
 70 LET V=PEEK 23627+256*PEEK 2
3628
 80 PRINT PEEK (V+1);TAB 9;
 90 FOR A=2 TO 5
100 PRINT PEEK (V+A);CHR$ 32;
110 NEXT A
120 PRINT '''
130 GO TO 10
```

The above program shows that for the following numbers the floating-point forms are:

		Exp.	Mantissa
1	=	129	Ø Ø Ø Ø
2	=	13Ø	Ø Ø Ø Ø
35456	=	144	1Ø 128 Ø Ø
–1	=	129	128 Ø Ø Ø
–35456	=	144	138 128 Ø Ø
6.333	=	131	74 167 239 158

It is interesting to note that .5 is different to 1/2 (again see appendix iv. for further details).

.5	=	127	127 255 255 255
1/2	=	128	Ø Ø Ø Ø

5. UNDERSTANDING — The Z80 Machine Code Instruction Set

5.1 Instructions and data

The stage has now been reached when the instructions of the Z80 machine code language can be discussed in turn.

In this book the instructions are divided into 18 groups, with each group containing those instructions that have a strong resemblance to each other. However, before discussing the groups of instructions mention must be made of the six classes of data that may follow the instruction proper in a line of Z80 machine code.

The classes of data are:

1. A single byte constant. (+dd)
   i.e. A number in the range hex. 00—FF, dec. 0—255. Those instructions that are required to be followed by a single byte constant have this indicated in their mnemonics by there being a '+dd'.
   e.g. The instruction menmonic — 'LD A,+dd'
2. A 2—byte constant. (+dddd)
   i.e. A number in the range hex. 0000—FFFF, dec. 0—65,535. Those instructions that are required to be followed by a 2-byte constant have this indicated in their mnemonics by there being a '+dddd'.
   e.g. The instruction mnemonic — 'LD HL,+dddd'
3. A 2—byte address. (addr)
   i.e. A number in the range hex. 0000—FFFF, dec. 0—65,535, that will be used as an address of a location in memory. Those instructions that are required to be followed by a 2—byte address have this indicated in their mnemonics by there being an 'addr'.
   e.g. The instruction mnemonic — 'JP addr'
4. A single byte displacement constant. (e)
   i.e. A number in the range hex. 00—FF, dec. —128 to +127. The number is always considered to be in 2's complement arithmetic.
   Those instructions that are required to be followed by a single byte displacement constant have this indicated in their mnemonics by there being an 'e'.
   e.g. The instruction mnemonic — 'JP e'
5. A single byte indexing displacement constant. (+d)
   i.e. A number in the range hex. 00—FF, dec. —128 to +127. The number is always considered to be in 2's complement arithmetic.
   Those instructions that are required to be followed by a single byte indexing displacement constant have this indicated in their mnemonics by there being a '+d'.
   e.g. The instruction mnemonic — 'LD A, (IX+d)'

6. A single byte indexing displacement constant AND a single byte constant. (+d,+dd)
   i.e. Two numbers in the range hex. 00—FF, with the first number being considered as dec. —128 to +127 and the second number as dec. 0-255.

   Instructions that are required to be followed by two bytes of data for this purpose have this indicated in their mnemonics by there being a '+d' and a '+dd'.

   e.g. The instruction mnemonic — 'LD (IX+d),+dd'

5.2 The instruction groups

There are many ways in which the hundreds of different machine code instructions may be split into groups. However, the method chosen here is to split the instructions into 18 functional groups.

After studying the instructions in a single group, the reader is advised to RUN the BASIC programs from the following chapter that illustrate those instructions.

Group 1. The NO OPERATION instruction.

mnemonic	**instruction** hex.
NOP	00

The NO OPERATION instruction, when executed by the microprocessor, results in the Z80 marking time for 1.14 microseconds. None of the registers or flags are affected.

The NO OPERATION instruction can be used by the programmer as part of a timed loop or, as is more often the case, to 'delete' unwanted instructions from a machine code program.

A BASIC program that demonstrated this instruction is to be found on page 111

Group 2. The instructions for loading registers with constants.

The following instructions are involved in loading registers with single byte constants.

mnemonic	**instruction** hex.
LD A,+dd	3E dd
LD H,+dd	26 dd
LD L,+dd	2E dd
LD B,+dd	06 dd
LD C,+dd	0E dd
LD D,+dd	16 dd
LD E,+dd	1E dd

An instruction line with one of these instructions will require two locations in memory. One for the instruction proper and a second for the constant. The instructions above can be viewed as 'setting' the contents of a particular register to a given value. This value will then remain in the register until overwritten.

The following instructions are involved in loading register pairs with 2–byte constants.

mnemonic	**instruction** hex.
LD HL,+dddd	21 dd dd
LD BC,+dddd	01 dd dd
LD DE,+dddd	11 dd dd
LD IX,+dddd	DD 21 dd dd
LD IY,+dddd	FD 21 dd dd
LD SP,+dddd	31 dd dd

An instruction line with one of these instructions will require three or four locations in memory. The instruction proper needing one or two locations and the constant a further two locations.

The first byte of the constant goes always to the 'low' register of the register pair, i.e. L,C,E,X,Y or P, and the second byte to the 'high' register, i.e. H,B,D,I or S.

These instructions 'set' the contents of the register pairs to given values. These values are often viewed by the programmer as 2–byte addresses but they may also be 2–byte numerical values or two separate 1–byte numerical values.

The instructions in this group do not affect the flags. BASIC programs that demonstrate the instructions in this group are to be found on page 112 .

Group 3. Register copying and exchanging instructions.

In the Z80 instruction set there are 59 instructions that are involved in the copying of the contents of a register, or register pair. These instructions are best divided into four subgroups.

Subgroup a. Single register-to-register copying instructions.

The following table gives the instruction codes for the instructions that lead to the copying of the contents of a single register, r, to another specified register.

r register	LD A,r	LD H,r	LD L,r	LD B,r	LD C,r	LD D,r	LD E,r
A	7F	67	6F	47	4F	57	5F
H	7C	64	6C	44	4C	54	5C
L	7D	65	6D	45	4D	55	5D
B	78	60	68	40	48	50	58
C	79	61	69	41	49	51	59
D	7A	62	6A	42	4A	52	5A
E	7B	63	7B	43	4B	53	5B

None of the instructions in the above table affect the flags. There are also four instructions that involve the I & the R registers.

mnemonic	**instruction** hex.
LD A,I	ED 57
LD A,R	ED 5F
LD I,A	ED 47
LD R,A	ED 4F

These last four instructions do affect the overflow/parity flag.

Subgroup b. Register pair-to-register pair copying instructions.

There are only three instructions in this subgroup and they all involve copying values to the stack pointer.

mnemonic	**instruction** hex.
LD SP,HL	F9
LD SP,IX	DD F9
LD SP,IY	FD F9

These instructions do not affect the flags.

Note that if the contents of a register pair is to be copied to another register pair and the above instructions are inappropriate then the operation is normally performed by two single register-to-register copying instructions.

e.g. There is no instruction 'LD HL,DE'
and this is normally dealt with by using 'LD H,D' and 'LD L,E'.

Alternatively the contents of the first register pair can be saved on the machine stack and subsequently copied to the second register pair. (See "The 'stack' instructions" on page 96.)

Subgroup c. The 'EX DE,HL' instruction.

In the Z80 instruction set there is only one instruction that allows for the contents of registers to be exchanged — within the 'main' set of registers.

mnemonic	instruction hex.
EX DE,HL	EB

This very useful instruction allows the programmer to exchange the value held in the DE register pair with that held in the HL register pair. No flags are affected.

This instruction is normally used when an address or 2—byte numerical constant has to be moved from the DE register pair to the HL register pair but the original value in the HL register is not to be lost.

Subgroup d. The 'alternate' register set instructions.

There are two instructions in this subgroup.

mnemonic	instruction hex.
EXX	D9
EX AF,A'F'	08

The 'EXX' instruction causes a switching over of the H,L,B,C,D & E registers with the H',L',B',C',D' & E' registers.

The 'EX AF,A'F' ' instruction does as its mnemonic suggests and switches the A & F registers with the A' & F' registers.

The alternate registers are often used to store addresses and numbers. By storing these values in the alternate registers they are safe from corruption and can be retrieved very easily and quickly.

The BASIC programs that demonstrate the instructions from this group are to be found on page 113.

Group 4. Instructions for the loading of registers with data copied from a memory location.

The Z80 instruction set contains many instructions that 'fetch' data from locations in the memory and then 'load' that data into a main register. All of the instructions require that the programmer specifies the address of the location, or a pair of locations, from which the data is to be copied and the register, or register pair, that is to receive the data.

The instructions in this group are best considered in three subgroups as there are three separate addressing techniques available to the programmer.

These addressing techniques are:

a) 'Absolute addressing' — the actual 2—byte address is specified following the instruction proper.

b) 'Indirect addressing' — the 2—byte address is already available in an 'addressing register pair'.

c) 'Indexed addressing' — the address of the location is to be computed by adding the displacement value, d, to the base address already held in the IX, or IY, register pair.

Subgroup a. Instructions using 'absolute addressing'.

The instructions in this subgroup are:

mnemonic	**instruction** hex.	
LD A, (addr)	3A addr	
LD HL,(addr)	2A addr	(usual form)
	ED 6B addr	(unusual form)
LD BC,(addr)	ED 4B addr	
LD DE,(addr)	ED 5B addr	
LD IX,(addr)	DD 2A addr	
LD IY,(addr)	FD 2A addr	
LD SP,(addr)	ED 7B addr	

The 'LD A,(addr)' instruction is the only instruction in the Z80 instruction set that allows for the contents of a location speficied as an absolute address to be loaded into a single register.

An important point to make about the remaining six instructions in this subgroup is that they are really double instructions.

e.g. The instruction 'LD BC,(addr)' should be considered as: 'LD C, (addr)' followed by 'LD B,(addr+1)'.

In every case the contents of the addressed location is copied to the 'low' register and the contents of the following location is copied to the 'high' register.

Subgroup b. Instructions using 'indirect addressing'.

The instructions in this subgroup are:

mnemonic	**instruction** hex.
LD A,(HL)	7E
LD A,(BC)	0A
LD A,(DE)	1A
LD H,(HL)	66
LD L,(HL)	6E
LD B,(HL)	46
LD C,(HL)	4E
LD D,(HL)	56
LD E,(HL)	5E

In each case the address of the location from which the single byte of data is to be copied is to be already present in the HL, DE or BC register pairs.

If the programmer should wish for an operation that is not supported by an instruction in the above list then a 'construction' will have to be used instead.

e.g. The operation 'LD D,(BC)' is not allowed and a possible 'construction' is;

is 'LD A,(BC)' followed by 'LD D,A' which would alter the contents of

the A register, or;

'LD H,B' 'LD L,C' followed by 'LD D,(HL)' which would alter the contents of the HL register pair.

Subgroup c. Instructions using 'indexed addressing'.

The instructions in this subgroup allow the programmer to load single registers with bytes of data held as a table, list or just a block of data. The base address is held in the appropriate indexing register pair.

Diagram 5.1 illustrates this being done.

The instructions in the subgroup are:

mnemonic	**instruction** hex.
LD A,(IX+d)	DD 73 d
LD H,(IX+d)	DD 66 d
LD L,(IX+d)	DD 6E d
LD B,(IX+d)	DD 46 d
LD C,(IX+d)	DD 4E d
LD D,(IX+d)	DD 56 d
LD E,(IX+d)	DD 5E d

For instructions involving the IY register pair change IX to IY and DD to FD.

It is interesting to consider the time taken by the Z80 microprocessor in executing the instructions in this group. The fastest instructions are those that form subgroup b. These instructions require the Z80 to fetch a single byte instruction code and then the actual byte of data. The instruction 'LD A,(HL)', for example, taken only 7 clock cycles.

The instructions in subgroup a. are much more complicated and take, depending on the instruction, 16-20 clock cycles. The instructions in subgroup c. also take a long time — 19 clock cycles — as indexed addressing involves the Z80 in a considerable amount of computation.

None of the instructions in this group affect the flags.

The BASIC programs that demonstrate the instructions from this group are to be found on page 114 .

Group 5. Instructions for loading locations in memory with data copied from registers, or with constants.

In general the instructions in this group perform operations that are opposite in action to those in group 4.

The instructions allow for the contents of the user-registers to be copied to specified memory locations, or for constants to be loaded into those locations.

Once again the instructions are best considered in three sub-groups.

Subgroup a. Instructions using 'absolute addressing'.

The instructions in this subgroup are:

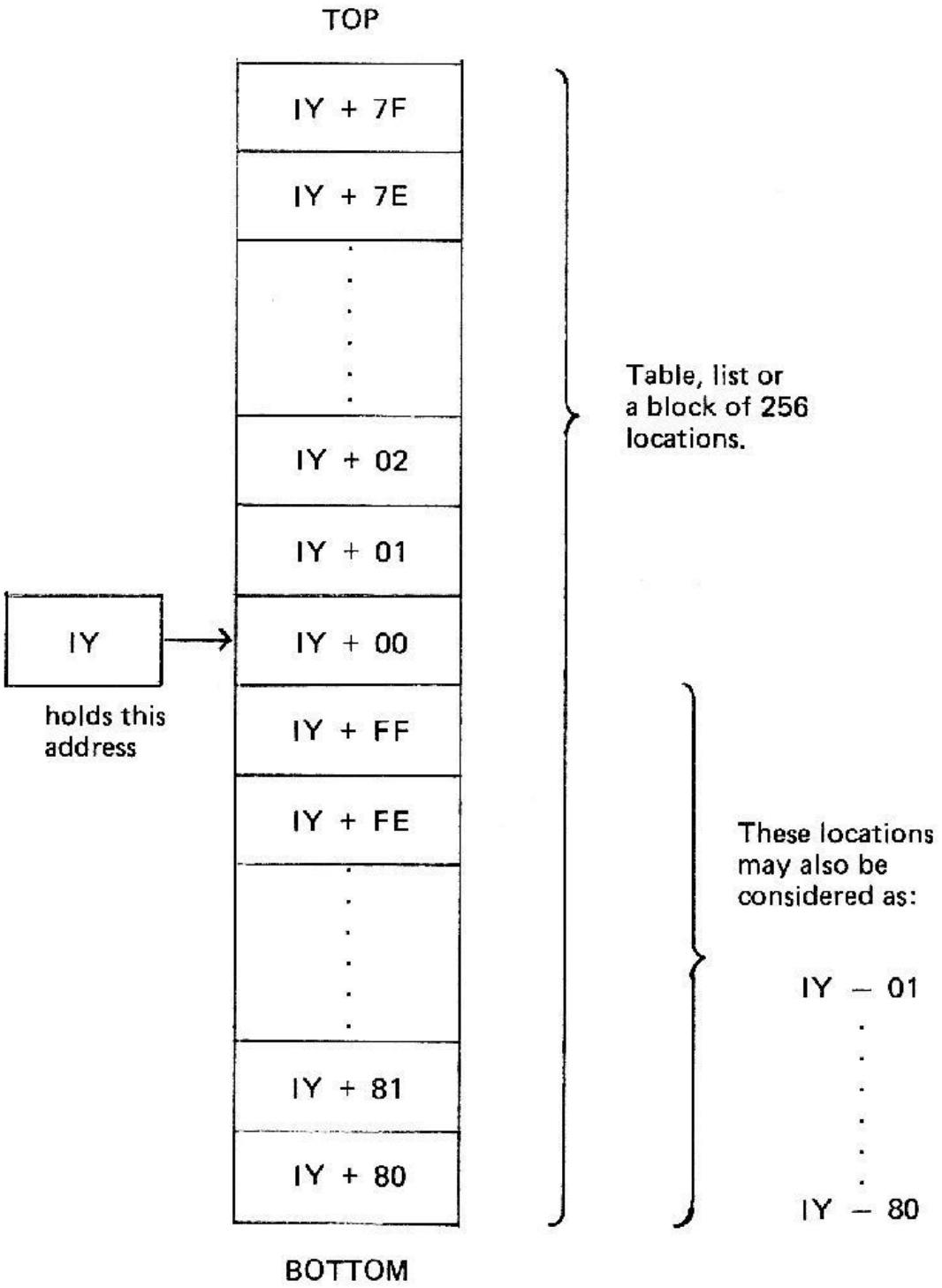


Diagram 5.1 The IY register pair addressing an area of the memory -- indexed addressing.

mnemonic	**instruction** hex.	
LD (addr),A	32 addr	
LD (addr),HL	22 addr	(usual form)
	ED 63 addr	(unusual form)
LD (addr),BC	ED 43 addr	
LD (addr),DE	ED 53 addr	
LD (addr),IX	DD 22 addr	
LD (addr),IY	FD 22 addr	
LD (addr),SP	ED 73 addr	

The above instructions are the only ones to use absolute addressing and it is important to note that there is not an instruction for loading a specified location with a constant. If this operation is required then the constant has first to be loaded into the A register, or dealt with in a different manner.

Again, an instruction such as 'LD (addr),HL' is in reality a double instruction for 'LD (addr),L' & 'LD (addr+1), H'.

The instructions in this subgroup are often used to store addresses and numbers in memory locations when these values are being considered as 'variables'. For example it is common practice to write, for example:

'LD (RAMTOP),HL'

where RAMTOP is the label given to a pair of consecutive memory locations used to hold the current value of the top of memory.

The fetching of the current value of the variable would later be performed by using a group 4 instruction, for example:

'LD HL,(RAMTOP)'

Subgroup b. Instructions using 'indirect addressing'.

The instructions in this subgroup allow the programmer to copy the contents of a single register to a location in memory whose address is held in the HL, BC or DE register pair. There is also an instruction for loading a single byte constant into the location addressed by the HL register pair.

mnemonic	**instruction** hex.
LD (HL),A	77
LD (BC),A	02
LD (DE),A	12
LD (HL),H	74
LD (HL),L	75
LD (HL),B	70
LD (HL),C	71
LD (HL),D	72
LD (HL),E	73
LD (HL),+dd	36 dd

Subgroup c. Instructions using 'indexed addressing'.

The instructions in this subgroup are:

mnemonic	**instruction** hex.
LD (IX+d),A	DD 77 d
LD (IX+d),H	DD 74 d
LD (IX+d),L	DD 75 d
LD (IX+d),B	DD 70 d
LD (IX+d),C	DD 71 d
LD (IX+d),D	DD 72 d
LD (IX+d),E	DD 73 d
LD (IX+d),+dd	DD 36 d dd

For instructions involving the IY register pair change IX to IY and DD to FD.

The BASIC programs that demonstrate the instructions from this group are to be found on page 116.

Group 6. The Addition instructions.

This group of instructions forms the first of four groups in the Z80 instruction set that involve arithmetical or logical operations.

The Additional instructions allow the programmer to add, in absolute binary arithmetic, a specified number to the contents of a single register, a register pair or an indexed addressed location in memory.

The instructions in this group can be divided into three subgroups with each subgroup having its own memonic type.

The three subgroups are:

a) The ADD instructions.

b) The INC instructions. The special case of addition when '1' is added to an existing number.

c) The ADC instructions. The value of the Carry flag is added to the result. The Carry flag is one of the bits of the FLAG register and is used to signify if the 'last' arithmetic operation led to binary overflow of a register or location. It will always therefore have the value '0', and be considered 'reset', or have the value '1', and be considered 'set'. ADD & ADC instructions do in their own turn affect the carry flag but INC instructions do not, a fact that has distinct advantages on occasions.

Subgroup a. The ADD instructions.

The instructions in this subgroup are:

mnemonic	**instruction** hex.	**mnemonic**	**instruction** hex.
ADD A,+dd	C6 dd	ADD HL,HL	29
ADD A,A	87	ADD HL,BC	09
ADD A,H	84	ADD HL,DE	19

ADD A,L	85	ADD HL,SP	39
ADD A,B	80	ADD IX,IX	DD 29
ADD A,C	81	ADD IX,BC	DD 09
ADD A,D	82	ADD IX,DE	DD 19
ADD A,E	83	ADD IX,SP	DD 39
ADD A,(HL)	86		
ADD A,(IX+d)	DD 86 d		

For instructions involving the IY register pair change IX to IY and DD to FD.

The ADD instructions given above are all very straightforward. However it must be realised that the register or location providing the 'addend' is only copied and therefore remains unaltered. Also that the addition sets or resets the carry flag in every case, depending on whether or not there has been binary overflow to the 'left' of the register or register pair involved.

The following examples illustrate these points.

i. With register A holding hex. 60
and register B holding hex. 90
and 'ADD A,B' instruction will give:
register A holding hex. F0
register B holding hex. 90
and the carry flag reset.

ii. With register A holding hex. A8
and register B holding hex. 7E
an 'ADD A,B' instruction will give:
register A holding hex. 26
register B holding hex. 7E
and the carry flag set.

Subgroup b. The INC instructions.

The instructions in this subgroup allow for '1' to be added to the contents of an 8-bit register, a location in memory or a 16-bit register pair. In all cases the carry flag is ignored – that is it remains totally unaffected.

The instructions are:

mnemonic	**instruction** hex.	**mnemonic**	**instruction** hex.
INC A	3C	INC HL	23
INC H	24	INC BC	03
INC L	2C	INC DE	13
INC B	04	INC SP	33
INC C	0C	INC IX	DD 23
INC D	14	INC IY	FD 23

INC E	1C
INC (HL)	34
INC (IX+d)	DD 34 d
INC (IY+d)	FD 34 d

Subgroup c. The ADC instructions

The instructions in this subgroup are:

mnemonic	instruction hex.	mnemonic	instruction hex.
ADC A,+dd	CE dd	ADC HL,HL	ED 6A
ADC A,A	8F	ADC HL,BC	ED 4A
ADC A,H	8C	ADC HL,DE	ED 5A
ADC A,L	8D	ADC HL,SP	ED 7A
ADC A,B	88		
ADC A,C	89		
ADC A,D	8A		
ADC A,E	8B		
ADC A,(HL)	8E		
ADC A,(IX+d)	DD 8E d		
ADC A,(IY+d)	FD 8E d		

The instructions in this subgroup allow the programmer to add two numbers, together with the current value of the carry flag. All of the instructions in this subgroup affect the carry flag. It is reset if the ADC operation does not give binary overflow and set if it does.

The following examples illustrate these points:

i. With register A holding hex. 60
register B holding hex. 90
and the carry flag set.
an 'ADC A,B' instruction will give:
register A holding hex. F1
register B holding hex. 90
and the carry flag reset.

ii. With register A holding hex. A8
register B holding hex. 7E
and the carry flag set.
an ADC A,B' instruction will give:
register A holding hex. 27
register B holding hex. 7E
and the carry flag set.

The BASIC programs that demonstrate the instructions from this group are to be found on page 118.

Group 7. The Subtraction instructions.

The Subtraction instructions allow the programmer to subtract, in absolute

binary arithmetic, a specified number from the contents of a single register, a register pair or an indexed addressed location in memory.

Again the instructions in this group can be divided into three subgroups with each subgroup having its own mnemonic type.

The three subgroups are:

a) The SUB instructions.

b) The DEC instructions. The special case of subtraction when '1' is subtracted from an existing number.

c) The SBC instructions. The value of the carry flag is subtracted from the result.

All SUB and SBC instructions affect the carry flag depending on whether or not a binary 'borrow' has been required. The DEC instructions leave the carry flag unaffected.

Subgroup a. The SUB instructions.

The instructions in this subgroup are:

mnemonic	**instruction** hex.
SUB +dd	D6 dd
SUB A	97
SUB H	94
SUB L	95
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB (HL)	96
SUB (IX+d)	DD 96 d
SUB (IY+d)	FD 96 d

Note: The mnemonics for the SUB instructions are normally written as above. That is 'SUB L' is preferred to 'SUB A,L', etc. as all SUB instructions involve the A register.

In the Z80 the SUB instruction give 'true' absolute binary subtraction as shown in the following examples. The carry flag is reset if the original value in A is 'greater than' or 'equal to' the subtrahend (the second number in the subtraction) but set if it is 'less than'.

i. With the A register holding hex. DC
and the B register holding hex. AA
a 'SUB B' instruction will give:
the A register holding hex. 32
and the carry flag reset. There is 'no borrow'.

ii. With the A register holding hex. AA
and the B register holding hex. DC
a 'SUB B' instruction will give:
the A register holding hex. CE
the B register holding hex. DC
and the carry flag set as there has been 'borrow'.

Subgroup b. The DEC instructions.

The instructions in this subgroup allow for '1' to be subtracted from the contents of an 8–bit register, a location in memory or a 16–bit register pair. In all cases the carry flag is unaffected.

The instructions in this subgroup are:

mnemonic	**instruction** hex.	**mnemonic**	**instruction** hex.
DEC A	3D	DEC HL	2B
DEC H	25	DEC BC	0B
DEC L	2D	DEC DE	1B
DEC B	05	DEC SP	3B
DEC C	0D	DEC IX	DD 2B
DEC D	15	DEC IY	FD 2B
DEC E	1D		
DEC (HL)	35		
DEC (IX+d)	DD 35 d		
DEC (IY+d)	FD 35 d		

Subgroup c. The SBC instructions.

The instructions in this subgroup are:

mnemonic	**instruction** hex.	**mnemonic**	**instruction** hex.
SBC A,+dd	DE dd	SBC HL,HL	ED 62
SBC A,A	9F	SBC HL,BC	ED 42
SBC A,H	9C	SBC HL,DE	ED 52
SBC A,L	9D	SBC HL,SP	ED 72
SBC A,B	98		
SBC A,C	99		
SBC A,D	9A		
SBC A,E	9B		
SBC A;(HL)	9E		
SBC A,(IX+d)	DD 9E d		
SBC A,(IY+d)	FD 9E d		

An SBC operation will give a 'true' subtraction if the carry flag is reset but will 'subtract with borrow' if the carry flag is set. This can be very useful when handling multiple precision numbers as any borrow generated can be taken along the number.

e.g. A four byte number held in the registers H', L', H & L can have another number, held in the registers D', E', D & E, subtracted from it as follows:

AND A	— resets the carry flag.
SBC HL,DE	— subtract the 'low' pair.
EXX	— switch to alternate set.
SBC HL,DE	— subtract the 'high' pair.
EXX	— switch to main set.

and the appropriate 'borrow' is carried from the 'low' subtraction to the 'high' one.

The BASIC programs that demonstrate the instructions from this group are to be found on page 120.

Group 8. The Compare instructions.

The instructions in this group are used very frequently in all machine code programs. They allow the programmer to compare the value held in the A register against a constant, a value held in a register or an addressed location in memory.

A compare instruction performs a subtraction operation, without carry, but discards the answer after using it to set the flags of the F register. The original value in the A register is unchanged.

The carry flag is affected in the same manner as with a subtraction operation. A comparison that is 'greater than' or 'equal' resets the carry flag, and a 'less than' sets the carry flag.

The instructions in this group are 'single comparison' instructions and 'block comparison' instructions are considered on page 102.

The instructions in this group are:

mnemonic	**instruction** hex.
CP +dd	FE dd
CP A	BF
CP H	BC
CP L	BD
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP (HL)	BE
CP(IX+d)	DD BE d
CP (IY+d)	FD BE d

The following examples show the use of 'CP B'.

i. With the A register holding hex. 31
and the B register holding hex. 30
a 'CP B' instruction will leave the registers unchanged and make the carry flag reset.
The result is — '31' is greater than '30'.

ii. With the A register holding hex. 30
and the B register holding hex. 30
a 'CP B' instruction will leave the registers unchanged and make the carry flag reset.
The result is — '30' is equal to '30'.

iii. With the A register holding hex. 01
and the B register holding hex. 30
a 'CP B' instruction will leave the registers unchanged and make the carry flag set.
The result is — '01' is less than '30'.

The BASIC program that demonstrates the instructions from this group is to be found on page 122 .

Group 9. The Logical instructions.

In the Z80 instruction set there are instructions to AND, OR & XOR the contents of the A register with the contents of another specified location. The operations are performed in a 'bitwise' manner and the 8-bit result returned in the A register.

The three types of logical operation will now be discussed in turn.

Subgroup a. The AND instructions.

The logical operation of AND performed with two binary digits will give a 'set' bit as a result only if the two binary digits under test are both set. Otherwise the resultant bit is 'reset'.

The following example shows how an AND instruction performs eight separate 'bit' operations.

	10101010		AA
in	**AND**	in	**AND**
binary.	11000000	hex.	C0
	will result in		results in
	10000000		80

The instructions in this subgroup are:

mnemonic	**instruction** hex.
AND +dd	E6 dd
AND A	A7

AND H	A4
AND L	A5
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND (HL)	A6
AND (IX+d)	DD A6 d
AND (IY+d)	FD A6 d

In use an AND instruction will reset from 0 to 8 bits of the A register. This process is called 'marking-off' and it allows the programmer to control certain bits of a byte of data. The following example shows how this is done.

In the SPECTRUM system the bits 2, 1 & 0 of the system variable ATTR—P hold the 'permanent' ink colour. Then, when the ink colour is changed the 'old' colour is removed by using an AND instruction and the 'new' colour entered by using an ADD instruction.

```
i.e.  LD A,(ATTR—P)      : Fetch system variable.
      AND +F1            : Mask-off 'old' colour.
      ADD +NEW COLOUR    : Add 'new' colour.
      LD (ATTR—P),A      : Return system variable.
```

Subgroup b. The OR instructions.

The logical operation of OR performed with two binary digits will give a 'set' bit as a result if either, or both, of the two binary digits under test are set. Otherwise the resultant bit is 'reset'.

The following example shows how an OR instruction performs eight separate 'bit' operation.

```
        10101010                    AA
in        OR                in      OR
binary. 11000000            hex.    C0
        will result in              results in
        11101010                    EA
```

The instructions in this subgroup are:

mnemonic	**instruction** hex.
OR +dd	F6 dd
OR A	B7
OR H	B4
OR L	B5
OR B	B0
OR C	B1
OR D	B2
OR E	B3

OR (HL)	B6
OR (IX+d)	DD B6 d
OR (IY+d)	FD B6 d

In use an OR instruction will set, or rather ensure they stay set, from 0 to 8 bits of the A register.

The following example shows one use of an OR instruction.

In the SPECTRUM system the bits 5, 4 & 3 of the system variable ATTR–P hold the 'permanent' paper colour. It is therefore possible to make the paper colour 'white' by using an 'OR +dd' instruction.

```
i.e.:  LD A(ATTR–P)        : Fetch system variable.
       OR +38              : Sets bits 5,4 & 3.
       LD (ATTR–P),A       : Return system variable.
```

Subgroup c. The XOR instructions.

The logical operation of XOR performed with two binary digits will give a 'set' bit as a result if either, **but not both,** of the two binary digits under test are set. Otherwise the resultant bit is 'reset'.

The following example shows how a XOR instruction performs eight separate 'bit' operations.

```
        10101010                    AA
in        XOR               in      XOR
binary. 11000000            hex;    C0
        will result in              results in
        01101010                    6A
```

The instructions in this subgroup are:

mnemonic	**instruction** hex.
XOR +dd	EE dd
XOR A	AF
XOR H	AC
XOR L	AD
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR (HL)	AE
XOR (IX+d)	DD AE d
XOR (IY+d)	FD AE d

In use a XOR instruction will 'change' or 'flip' from 0 to 8 bits of the A register. This is perhaps difficult to understand initially but consider the example given above once again. It was:

In hex. AA **XOR** C0 gives 6A

In this example the second operand is the byte 'C0' which is a byte with only bits 6 & 7 set. Therefore the effect of the XOR operation is to 'flip' bits 6 & 7 of the first operand and change 'AA' into '6A'.

The use of XOR instructions in machine code programs is often complicated but the instruction 'XOR A' is, however, frequently used as an alternative to 'LD A,+00'. Both of these instruction lines 'clear' the A register but 'XOR A' uses only one location wereas 'LD A,+00' uses two locations.

All AND, OR & XOR instructions reset the carry flag whenever they are used.

The BASIC program that demonstrates the instructions from this group is to be found on page 122 .

Group 10. The Jump instructions.

In the Z80 instructions set there are seventeen instructions that allow the programmer to make 'jumps' within a program. A machine code 'jump' can be equated with a BASIC 'GO TO' but the comparison must not be taken too far.

The instructions in this group are best considered in eight subgroups. Four of these subgroups contain instructions that are conditional on the state of one of major flags and the flags will be discussed in detail as required.

Subgroup a. The absolute jump instruction.

mnemonic	**instruction** hex.
JP addr	C3 addr

This is the classic jump instruction. When executed a 'JP addr' instruction leads to the 'addr' being loaded into the Program Counter and execution of the machine code program will continue from that location.

Subgroup b. Jump instructions that use indirect addressing.

The instructions in this subgroup are:

mnemonic	**instruction** hex.
JP (HL)	E9
JP (IX)	DD E9
JP (IY)	FD E9

These three instructions lead to the 16-bit value currently held in the appropriate register pair being loaded into the Program Counter. The instructions in this subgroup are often used when a jump is to be made to a location whose address is specified in a table of addresses.

Subgroup c. The relative jump instruction.

mnemonic	**instruction** hex.
JR e	18 e

This instruction allows the programmer to make jumps to locations that are within 127 locations forward and 128 locations backwards from the current location. Note that the current location is in fact the location after the 'e' as the Program Counter is already incremented.

Diagram 5.2 shows how 'e' varies for the different jumps that are possible with this instruction.

'e' is always considered in 2's complement arithmetic and a positive 'e' gives the number of locations that have to be 'jumped over' whilst a negative 'e' shows by how much the Program Counter is to be reduced.

Subgroup d. Jump instructions conditional on the carry flag.

In the Z80 instruction set there are four instructions that allow for a 'jump' to be made **only** if the carry flag is as required by the instruction.

The carry flag will now be discussed further.

The Carry Flag.

This flag is bit 0 of the F register and it is essentially a flag that shows whether or not binary overflow has occurred. However, the manufacturers of the Z80 have also arranged fot it to be set in certain instances and reset in others. There are also many instances when the carry flag is unaffected by the execution of instructions.

In summary the following points can be made:

i. All ADD & ADC instructions affect the carry flag. If there is 'no overflow' the flag is reset but if there is 'overflow' the flag is set.

ii. All SUB, SBC & CP instructions affect the carry flag. If there was a binary 'borrow' then the flag is set but otherwise it is reset.

iii. All AND, OR & XOR instructions reset the carry flag.

iv. The rotation instructions affect the carry flag (see page 99).

The jump instructions that are conditional on the state of the carry flag are:

mnemonic	**instruction** hex.	**comment**
JP NC,addr	D2 addr	Jump only when
JR NC,e	30 e	carry flag reset.
JP C,addr	DA addr	Jump only when
JR C,e	38 e	carry flag set.

As an example of these instructions consider the following routine that distinguishes valid ASCII digits.

```
.....                 : Enter with A register
                      : holding the ASCII code.
CP +30                : The code for digit '0'.
JP C,Error            : Jump if out of range.
CP +3A                : The code for ':'.
```

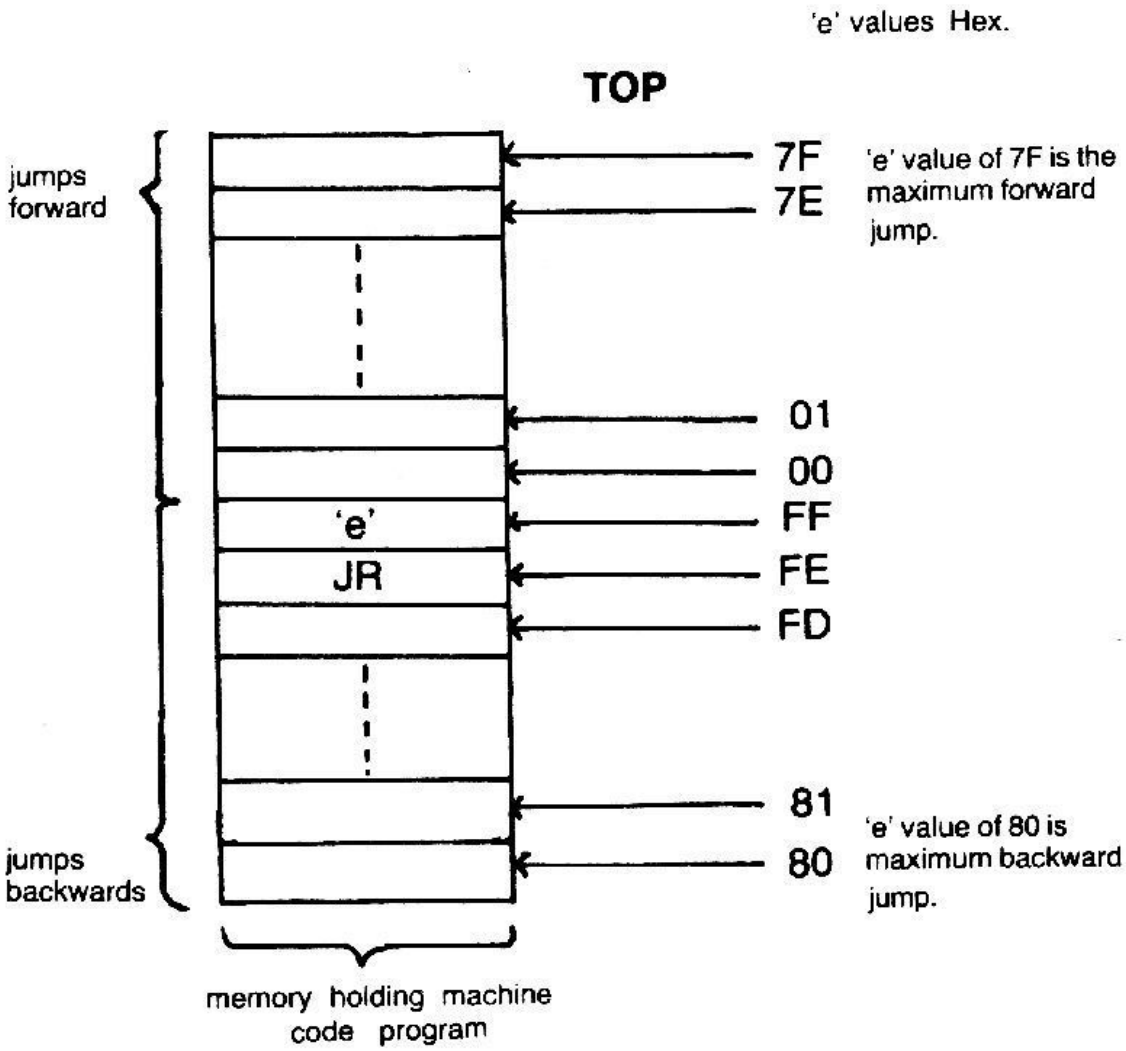


Diagram 5.2 To show how different values of 'e' cause jumps to different locations.

```
JR NC,Error          : Again jump if out of range.
.....                : Continue with ASCII digits only,
                       i.e. range 30 — 39. for the digits
                       '0'—'9'.
```

Subgroup e. Jump instructions conditional on the zero flag.

Again there are four instructions that allow for a 'jump' to be made **only** if the zero flag is as required by the instruction.

The zero flag will now be discussed.

The Zero Flag

This flag is bit 6 of the F register and in most instances of its usage it becomes set if the result of an operation is zero, otherwise it is reset.

For example:

6C ADD 5A gives C6 and zero flag reset.

but 6C ADD 94 gives 00 and zero flag set.

In summary the following points can be made:

i. All ADD, INC, ADC, SUB, DEC, SBC, CP, AND, OR & XOR instructions using single registers, and ADC & SBC instructions using register pairs will give the zero flag set if the result is zero.

ii. Rotation instructions (see page 99), Bit testing instructions(see page 102) and Block Searching instructions (see page 102) affect the zero flag.

iii. LD instructions, with the exception of 'LD A,I' & 'LD A,R', do not affect the zero flag.

The jump instructions that are conditional on the state of the zero flag are:

mnemonic	**instruction** hex.	**comment**
JP NZ,addr	C2 addr	Jump only when zero flag reset.
JR NZ,e	20 e	
JP Z,addr	CA addr	Jump only when zero flag set.
JR Z,e	28 e	

The instructions of this subgroup are very commonly used and the following example shows ASCII characters being separated.

```
.....                : Enter with the A register holding
                       a character code.
CP +3B               : Is the character a ';'?
JR Z,S—colon         : Jump if it is so.
CP +2C               : Is it a ','?
JR NZ,Else           : Jump with everything else.
.....                : Continue with the ',' handling
                       instructions.
```

Subgroup f. Jump instructions conditional on the sign flag.

These are two instructions that allow for a 'jump' to be made **only** if the sign flag is as required by the instruction.

The sign flag will now be discussed.

The Sign Flag

This flag is bit 7 of the F register and in most instances of its usage it is a copy of the lefthandmost bit of a 'result'.

Whenever a 8–bit, or 16–bit, binary number is considered in 2's complement arithmetic then the lefthandmost bit (bit 7 or bit 15) is taken as a sign bit. It is 'reset' for positive numbers and 'set' for negative ones.

Therefore the sign flag can be taken as being 'reset' with positive results and 'set' with negative ones.

In summary the following points can be made:

i. All ADD, INC, ADC, SUB, DEC, SBC, CP, AND, OR & XOR instructions using single registers, and ADC & SBC instructions using register pairs will affect the sign flag as indicated above.

ii. Block searching instructions (see page 102) and most Rotation instructions (see page 99) also affect the sign flag.

iii. LD instructions, with the exception of 'LD A,I' & 'LD A,R', do not affect the sign flag.

The jump instructions that are conditional on the state of the sign flag are:

mnemonic	**instruction** hex.	**comment**
JP P,addr	F2 addr	Jump if result positive.
JP M,addr	FA addr	Jump if result negative.

The instructions in this subgroup are not commonly used partly because they require an absolute address and partly because a sign bit can be read in several other ways.

The following example demonstrates how an instruction from this subgroup might be used.

```
.....          : Enter with the A register holding
                 a character code.
AND A          : This will affect the sign flag.
JP P,Else      : Codes 00 – 7F are dealt with
                 separately.
.....          : Continue with graphic characters
                 only.
```

Subgroup g. Jump instructions conditional on the overflow/parity flag.

Again there are two instructions that allow for a 'jump' to be made **only** if the overflow/parity flag is as required by the instruction.

The overflow/parity flag will now be discussed.

The Overflow/Parity flag

This flag is bit 2 of the F register and is a dual purpose flag. Certain instructions use the flag to indicate 'overflow' whilst other instructions use it to store the result of a 'parity' test.

The concept of 'overflow' does not apply to binary overflow but rather to 2's complement arithmetic overflow as illustrated in the following example.

Consider: in hex. 0A ADD 5C gives 66
in dec. 10 ADD 92 gives 102
which is correct — no overflow.

in hex. 6A ADD 32 gives 9C
in dec. 106 ADD 50 gives −100
which is incorrect — i.e. overflow.

Overflow can also occur with subtractions, viz:
in hex. 83 SUB 14 gives 6F
in dec. −125 SUB 20 gives 111
which is incorrect — i.e. overflow.

The overlow/parity flag is 'set' when overflow occurs.

The concept of 'parity' concerns the number of set bits in a given byte. Parity is said to exist when the number of set bits is even.

The following example shows this.

The byte 0 1 0 1 0 1 0 1 has even parity and the flag is set.

but the byte 0 0 0 0 0 0 0 1 has odd parity and the flag is reset.

In summary the following points can be made:

i. All ADD, ADC, SBC, SBC & CP instructions using single registers, and ADC & SBC instructions using register pairs have their results tested for overflow.

ii. All AND, OR, XOR & most Rotation instructions (see page 99) have their results tested for parity.

iii. An INC instruction will set the flag if the result is hex. 80, and a DEC instruction if the result is hex. 7F.

iv. Various other instructions, viz. 'LD A,I', 'LD A,R' and many of the block handling instructions (see page 102), also affect the overflow/parity flag.

The jump instructions that are conditional on the state of the overflow/parity flag are:

mnemonic	**instruction** hex.	**comment**
JP PO,addr	E2 addr	Jump if parity odd, or no overflow.

JP PE,addr	EA addr	Jump if parity even, or overflow.

The instructions in this subgroup are only used on rare occasions and even then their use can be avoided by using other instructions.

The BASIC programs that demonstrate the instructions from this group are to be found on page 124.

Group 11. The 'DJNZ,e ' instruction.

The single instruction in this group is one of the most useful and most commonly used instructions in the whole of the Z80 instruction set.

The instruction is:

mnemonic	**instruction** hex.
DJNZ,e	10 e

The mnemonic stands for 'decrement the B register and jump relative if the zero flag is reset'.

In use this instruction can be likened to a BASIC FOR-NEXT loop of the form:

FOR B = x TO Ø STEP −1: NEXT B

In this loop the variable B is initialised to a value x. Then with each passage of the loop B is decremented until it reaches zero.

The instruction 'DJNZ,e' is used in a similar manner. Firstly, the programmer has to specify the size of the loop variable and enter it into the B register. Then the substance of the loop is given. Finally the 'DJNZ,e' instruction is used with care being taken to make sure the value of 'e' is appropriate.

The following example shows this instruction as it might be used to print the alphabet.

```
        LD   B,+1A        : 26 letters in the
Loop    LD   A,+5B        : alphabet. 'A' is first;
        SUB  B            : i.e. hex. 5B−1A = 41.
        RST  0010         : print the character, (see later)
        DJNZ,LOOP         : Move on to 'B', 'C' etc.
        ..................: In effect NEXT B.
```

The hex. code for this example is:

06,1A,3E,5B,90,D7,10,FA

where the byte '3E' has been given the label 'loop'. The author finds that the best way to calculate the correct value for 'e' in a short machine code example, such as the above, is to give the 'e' byte the value 'FF' and then count backwards in hex. until the 'loop' byte is reached. In the above example there are five steps backwards and the appropriate value for 'e' is hex. FA.

The BASIC program that demonstrates the instruction in this group is to be found on page 126.

Group 12. The Stack instructions.

In most machine code programs extensive use is made of the machine stack both by the programmer as a place to save data, and the microprocessor to save 'return' addresses. The instructions that form this group can be divided into two user-subgroups and three microprocessor-subgroups.

Subgroup a. The PUSH and POP instructions.

These instructions allow the programmer to PUSH, i.e. save, two bytes of data on the machine stack and later POP, i.e. copy, two bytes from the machine stack.

The pairs of data bytes must be copied from, and to, specified register pairs but it is important to realise that no record is kept that shows which data bytes 'belong' to which register pairs.

The instructions in this subgroup are:

mnemonic	**instruction** hex.	**mnemonic**	**instruction** hex.
PUSH AF	F5	POP AF	F1
PUSH HL	E5	POP HL	E1
PUSH BC	C5	POP BC	C1
PUSH DE	D5	POP DE	D1
PUSH IX	DD E5	POP IX	DD E1
PUSH IY	FD E5	POP IY	FD E1

It is important to understand the operation of these instructions if a machine code program is going to make use of the machine stack in other than a straightforward manner.

When a PUSH instruction is executed the Stack Pointer is first decremented so as to make it point to a free location. A copy of the high register of the register pair is then copied into this location. Then the Stack Pointer is decremented a second time and the value in the low register of the register pair is copied over.

The opposite actions are followed during the execution of a POP instruction. It is important to appreciate that the Stack Pointer will after the execution of one of these instructions always point to the 'last-used' location on the stack.

The instructions in this subgroup are often used in 'pairs' and the following example shows this.

```
PUSH AF          : Save copy of AF.
PUSH BC          : Save copy of BC.
.....            : Perform 'other work'.
POP  BC          : Retrieve the 'old' BC.
POP  AF          : Retrieve the 'old' AF.
.....
```

In the example the 'other work' can make use of the A, F, B & C registers

with impunity but it must leave the Stack Pointer with its former value if the whole routine is to run as desired.

Subgroup b. The Stack Exchanging instructions.

The instructions in this subgroup are not commonly used but they can on occasions be of great use.

The instructions are:

mnemonic	**instruction** hex.
EX (SP),HL	E3
EX (SP),IY	DD E3
EX (SP),IX	FD E3

These instructions allow the programmer to exchange the current value held in a specified register pair with the last entry on the machine stack. The Stack Pointer is unaffected.

The use of these instructions can be confusing and they are best considered as being alternative instructions to the PUSH & POP instructions in special cases.

Consider, for example, the following situation.

A value 'First' is on the machine stack and a value 'Second' is in the HL register pair.

It is then the wish of the programmer to exchange these values with each other.

There are two ways:

i. Use a 'EX (SP),HL' instruction.

ii. Use another register pair as a temporary store for 'First'.

```
i.e. POP BC        : Save 'First' in BC.
     PUSH HL       : 'Second' to stack.
     PUSH BC       : Move 'First' to HL
     POP HL        : in one way or another.
     ..............
```

The instructions in this subgroup can also be used to manipulate 'return' addresses (see below).

Subgroup c. The CALL instructions.

The machine code CALL instructions are directly equivalent to the BASIC 'GO SUB' command. The instructions are included in this group as the microprocessor uses the machine stack as the area in which the 'return' addresses are saved.

There are nine instructions in this subgroup and they allow for a subroutine to be 'called' unconditionally or conditionally on the state of the four major flags.

The instructions in this subgroup are:

mnemonic	instruction hex.	comment
CALL addr	CD addr	Unconditional.
CALL C,addr	DC addr	Carry flag set.
CALL NC,addr	D4 addr	Carry flag reset.
CALL Z,addr	CC addr	Zero flag set.
CALL NZ,addr	C4 addr	Zero flag reset.
CALL M,addr	FC addr	Sign flag set.
CALL P,addr	F4 addr	Sign flag reset.
CALL PE,addr	EC addr	Overflow/parity flag set.
CALL PO,addr	E4 addr	Overflow/parity flag reset.

The actions of a CALL instruction are as follows:

i. The current value of the Program Counter, i.e. the address of the first location after the 'addr' of the CALL instruction, is saved on the machine stack. The Stack Pointer is manipulated as for a PUSH instruction. The high byte of the Program Counter going to the location above the low byte.

ii. The 'addr' is then copied into the Program Counter and the execution of the program procedes.

Subgroup d. The RET instructions.

The machine code RET instructions are directly equivalent to the BASIC 'RETURN' command. There are nine instructions in this subgroup and they allow for an exit from a subroutine either unconditionally or conditionally on the state of the four major flags.

The instructions in this subgroup are:

mnemonic	instruction hex.	comment
RET	C9	Unconditional.
RET C	D8	Carry flag set.
RET NC	D0	Carry flag reset.
RET Z	C8	Zero flag set.
RET NZ	C0	Zero flag reset.
RET M	F8	Sign flag set.
RET P	F0	Sign flag reset.
RET PE	E8	Overflow/parity flag set.
RET PO	E0	Overflow/parity flag reset.

The action of a RET instruction is to copy the last entry on the machine stack to the Program Counter. The Stack Pointer is thereby incremented twice.

It is important to appreciate that the address taken off the machine stack does not necessarily have to have been placed there originally by a CALL instruction.

It is not at all common in BASIC programming to directly manipulate the 'GO SUB stack' but in machine code programming it is a fairly common occurrence.

The following example shows this being done. An address of a routine is to be collected from a table of addresses and a 'jump' made to that routine.

```
.....                  : Start with the A register holding
                         the entry number.
LD   D,+00             : Clear the D register.
ADD  A                 : Double the entry number
LD   E,A               : and transfer to E.
LD   HL,+Table—base    : Point HL to the table.
ADD  HL,DE             : Point to the entry.
LD   D,(HL)            : Pick up the high part of the
INC  HL                : routine address and then
LD   E,(HL)            : the low byte.
PUSH DE                : Save it on the machine stack.
.....                  : Other work may use DE.
RET                    : Will 'jump' appropriately.
```

Subgroup e. The RST instructions.

The last subgroup of instructions in this group contains the special RST, or 'restart', instructions. These instructions are in effect CALL instructions that do not require as 'addr' to be specified.

The instructions in this subgroup are:

mnemonic	**instruction** hex.	**comment**
RST 0000	C7	CALL 0000
RST 0008	CF	CALL 0008
RST 0010	D7	CALL 0010
RST 0018	DF	CALL 0018
RST 0020	E7	CALL 0020
RST 0028	EF	CALL 0028
RST 0030	F7	CALL 0030
RST 0038	FF	CALL 0038

In the SPECTRUM system the eight 'restart' addresses are monitor subroutine entry points and they will be discussed in chapter 7.

The BASIC programs that demonstrate the instructions in this group are to be found on page 127.

Group 13 The Rotation instructions.

In the Z80 instruction set there is a large number of instructions for rotating

the bits of a specified byte. These instructions are often very useful. All the more so as they all pass bits to the carry flag which can then be tested.

Shifting a byte to the left has the effect of doubling the value of that byte as long as the most significant bits are not lost. Whilst shifting a byte to the right halves the value.

Diagram 5.3 shows the variety of rotations (& shifts) that are possible.

The following table shows the instructions in this group.

	RLC	RL	SLA	RRC	RR	SRA	SRL
A	CB 07	CB 17	CB 27	CB 0F	CB 1F	CB 2F	CB 3F
H	CB 04	CB 14	CB 24	CB 0C	CB 1C	CB 2C	CB 3C
L	CB 05	CB 15	CB 25	CB 0D	CB 1D	CB 2D	CB 3D
B	CB 00	CB 10	CB 20	CB 08	CB 18	CB 28	CB 38
C	CB 01	CB 11	CB 21	CB 09	CB 19	CB 29	CB 39
D	CB 02	CB 12	CB 22	CB 0A	CB 1A	CB 2A	CB 3A
E	CB 03	CB 13	CB 23	CB 0B	CB 1B	CB 2B	CB 3B
(HL)	CB 06	CB 16	CB 26	CB 0E	CB 1E	CB 2E	CB 3E
(IX+d)	DD CB d 06	DD CB d 16	DD CB d 26	DD CB d 0E	DD CB d 1E	DD CB d 2E	DD CB d 3E
(IY+d)	FD CB d 06	FD CB d 16	FD CB d 26	FD CB d 0E	FD CB d 1E	FD CB d 2E	FD CB d 3E

There are also four single byte instructions for rotating the A register and two 'nibble' handling instructions.

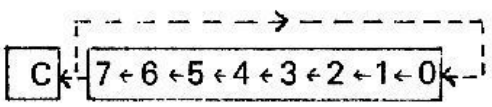
mnemonic	**instruction** hex.
RLCA	07
RLA	17
RRCA	0F
RRA	1F
RRD	ED 67
RLD	ED 6F

As a summary of the effect of the instructions from this group upon the major flags the following points can be made:

i. All the instructions, except for 'RLD' & 'RRD' affect the carry flag. (See diagram 5.3.)

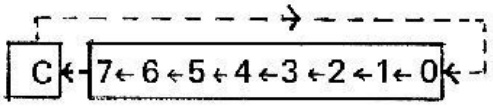
ii. All the instructions, with the exception of the four single byte instructions, affect the zero, sign and overflow/parity flags.

RLC & RLCA
Rotate left with carry.



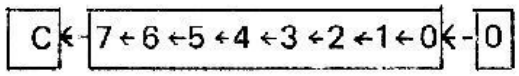
Bit_7 goes to carry.
Bit_7 goes to bit_0.

RL & RLA
Rotate left.



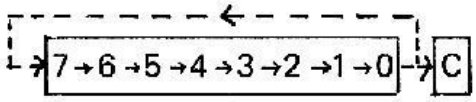
Bit_7 goes to carry.
Carry goes to bit_0.

SLA
Shift left.



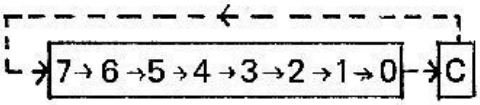
Bit_0 is reset.
Bit_7 goes to carry.

RRC & RRCA
Rotate right with Carry.



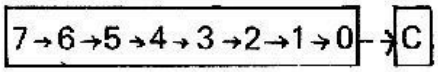
Bit_0 goes to carry.
Bit_0 goes to bit_7.

RR & RRA
Rotate right.



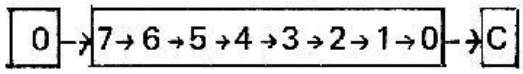
Bit_0 goes to carry.
Carry goes to bit_7.

SRA
Shift right.



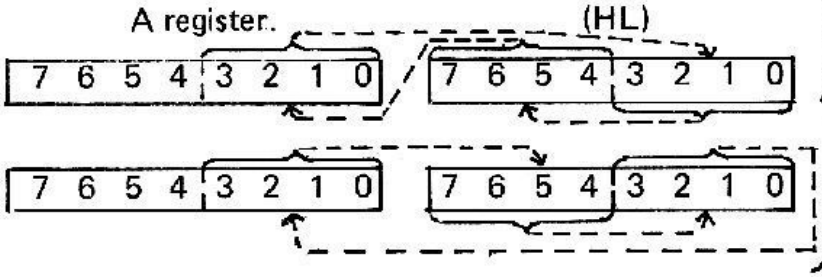
Bit_0 goes to carry.
Bit_7 is unchanged.

SRL
Logical shift right.



Bit_0 goes to carry.
Bit_7 is reset.

RLD
RRD



Two special 'nibble' handling instructions.

Diagram 5.3 The different types of 'rotation'

The BASIC program that demonstrates the instructions from this group is to be found on page 129.

Group 14. The 'Bit handling' instructions.

In the Z80 instruction set there are instructions that allow the programmer to test, set or reset a specified bit within a byte held in a register or addressed location. The three types of instructions will be discussed in turn.

Subgroup a. The BIT instructions.

The BIT instructions allow the programmer to determine the status of a specified bit. Following the use of one of these instructions it is usual to act upon the finding, for example, by using a 'JP Z' instruction.

A BIT instruction will give the zero flag **set** if the bit under test is **reset,** and vice versa.

Subgroup b. The SET instructions.

The SET instructions allow the programmer to set a specified bit. No flags are affected.

In use the instructions are used to set bits as required but on many occasions their use is better described as 'ensuring' a particular bit is set. A SET instruction acting on a bit that is already set will have no demonstrable effect.

Subgroup c. The RES instructions.

The RES instructions allow the programmer to reset a specified bit. Once again no flags are affected.

The instructions of these three subgroups are all given in the following table. *(see page 103)*

Note: For instructions using the indexing registers see appendix i.

The BASIC program that demonstrates the instructions in this group is to be found on page 131.

Group 15. The Block Handling instructions.

There are eight block handling instructions in the Z80 instruction set. These instructions are very interesting and useful. They allow the programmer to move a block of date from one area of the memory to another, or to search an area of memory.

In order to move a block of data the base address must be present in the HL register pair, the destination address in the DE register pair and the number of bytes in the block in the BC register pair.

In order to search an area of memory for the first occurrence of a particular value the base address must once again be in the HL register pair, the number of bytes in the search area in the BC register pair and the A register must hold a copy of the 'particular' value.

		bit 0	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7
A	BIT	47	4F	57	5F	67	6F	77	7F
register	RES	87	8F	97	9F	A7	AF	B7	BF
CB **	SET	C7	CF	D7	DF	E7	EF	F7	FF
H	BIT	44	4C	54	5C	64	6C	74	7C
register	RES	84	8C	94	9C	A4	AC	B4	BC
CB **	SET	C4	CC	D4	DC	E4	EC	F4	FC
L	BIT	45	4D	55	5D	65	6D	75	7D
register	RES	85	8D	95	9D	A5	AD	B5	BD
CB **	SET	C5	CD	D5	DD	E5	ED	F5	FD
B	BIT	40	48	50	58	60	68	70	78
register	RES	80	88	90	98	A0	A8	B0	B8
CB **	SET	C0	C8	D0	D8	E0	E8	F0	F8
C	BIT	41	49	51	59	61	69	71	79
register	RES	81	89	91	99	A1	A9	B1	B9
CB **	SET	C1	C9	D1	D9	E1	E9	F1	F9
D	BIT	42	4A	52	5A	62	6A	72	7A
register	RES	82	8A	92	9A	A2	AA	B2	BA
CB **	SET	C2	CA	D2	DA	E2	EA	F2	FA
E	BIT	43	4B	53	5B	63	6B	73	7B
register	RES	83	8B	93	9B	A3	AB	B3	BB
CB **	SET	C3	CB	D3	DB	E3	EB	F3	FB
(HL)	BIT	46	4E	56	5E	66	6E	76	7E
CB **	RES	86	8E	96	9E	A6	AE	B6	BE
	SET	C6	CE	D6	DE	E6	EE	F6	FE

The instructions in this group are:

AUTOMATIC

mnemonic	**instruction** hex.	**comment**
LDIR	ED B0	Block moving — incrementing
LDDR	ED B8	Block moving — decrementing

CPIR	ED B1	Block searching — incrementing
CPDR	ED B9	Block searching — decrementing

NON-AUTOMATIC

mnemonic	**instruction** hex.	**comment**
LDI	ED A0	Byte moving — incrementing
LDD	ED A8	Byte moving — decrementing
CPI	ED A1	Byte compare — incrementing
CPD	ED A9	Byte compare — decrementing

As can be seen in the above table there are both automatic and non-automatic instructions. The automatic instructions are generally the more common and the more useful.

An automatic instruction, upon execution, will complete its task without any further instruction from the programmer. i.e. When a 'LDIR' instruction is executed by the microprocessor then that instruction will result in a block of data being moved. It therefore follows that the execution of an automatic instruction can take a variable length of time. This time being dependent on the number of bytes to be moved.

A non-automatic instruction however handles only one byte at a time and requires the programmer to use the instruction repeatedly if a block of data is to be moved or searched. The instruction time of a non-automatic instruction is therefore fixed. A non-automatic instruction is a 'block handling' instruction as upon execution the 'source', 'destination' and 'counter' values are incremented, or decremented, by the microprocessor.

Each of the instructions in this group will now be discussed in turn.

LDIR:

This is the most commonly used instruction of the eight instructions in this group.

A 'LDIR' instruction will move a block of data whose 'source' address is held in the HL register pair to the area of memory with the 'destination' address held in the DE register pair with the number of bytes specified by the BC register pair.

In operation a single byte is moved from (HL) — read this as 'where HL points' — to (DE). The value in the BC register pair is then decremented and the values in the HL & DE register pairs incremented. If the 'count' in the BC register has not yet reached zero then another byte of data will be moved. The moving of bytes continues until the 'counter' reaches zero but note that at that moment the register pairs HL & DE point to the locations after the blocks.

The overflow/parity flag is reset by this instruction.

LDDR:
This instruction is the same as the 'LDIR' instruction except that after each byte is moved the values in the HL & DE register pairs are decremented. The instruction therefore requires that the 'base address' of a block refers to the last location of the block. (The address is still referred to as the 'base address'.) The 'destination' address also has to refer to the last location of the destination area.

CPIR:
This instruction searches a specified area of memory for the first occurrence of a 'particular' value. The HL register pair must hold the 'base address', the BC register pair the number of the bytes to be examined and the A register the 'particular' value.

In operation the byte from (HL) is compared to that held in the A register. If there is no match then the instruction decrements the counter and increments the address in the HL register pair and proceeds with the next comparison.

The operation continues until either a match is found or the value held in the BC register pair reaches zero. If a match is found then the zero flag is set, the sign flag is reset and the HL register pair points to the location after the matching byte. The value in the BC register pair will indicate, also, how far through the block the match was found.

If no match is found then the BC register pair will hold zero and the sign, zero and overflow/parity flags are all reset.

CPDR:
The operation of this instruction is similar to the 'CPIR' instruction but the block of data is searched from the top downwards.

Now the non-automatic instructions will be discussed.

LDI:
The execution of this instruction will result in the single byte of data at (HL) being moved to (DE). The value in the BC register pair is decremented. The overflow/parity flag will be set unless the value in the BC register pair has become zero. The values in the HL & DE register pairs are incremented.

It is then for the programmer to decide whether or not to move another byte of data. In normal use the programmer would probably make a test on the byte that is to be moved next and then act accordingly.

LDD:
This instruction is similar to 'LDI' except that the values in the HL & DE register pairs are decremented by the instruction.

CPI:
The execution of this instruction will lead to the byte addressed by the HL

register pair being copied into the microprocessor and saved there whilst the value in the HL register pair is incremented and the value in the BC register pair decremented. The 'saved' value is then compared to the value held in the A register. The zero flag is set if there is a match but otherwise reset. The sign flag is reset and the overflow/parity flag is reset unless the value in the BC register pair has reached zero in which case it is set.

CPD:
This instruction is similar to 'CPI' except that the value in the HL register pair is decremented.

The BASIC programs that demonstrate the instructions from this group are to be found on page 133.

Group 16. The Input and Output instructions

In the Z80 instruction set there is a most comprehensive set of instructions that allow the programmer to collect data from an outside source (IN) or to send data to a peripheral device (OUT).

There are simple, non-automatic and automatic instructions in this group although only the simple instructions are used in the standard SPECTRUM system.

In all cases the data that is handled by an IN or an OUT is in the form of an 8—bit parallel byte of data.

When executing an IN instruction the microprocessor takes the 'byte of data' off the data bus and copies it into a specified register. The control line $\overline{IORQ}$ is active as well as $\overline{RD}$ during the execution of an IN instruction.

When executing an OUT instruction the microprocessor places a copy of the value held in a specified register onto the data bus from where it may be collected by a peripheral device. The lines $\overline{IORQ}$ & $\overline{WR}$ will both be active during the execution of an OUT instruction.

In addition to the state of the $\overline{RD}$, $\overline{WR}$ & $\overline{IORQ}$ lines a peripheral device will be activated by the use of an appropriate address placed on the address bus during the execution of either an IN or an OUT instruction as required. This address is termed a 'port address' and in a Z80 system it a 16—bit address.

In a small microcomputer system such as the SPECTRUM it is common practice to only use only a few of the 65,536 possible port addresses by allowing the peripheral device to be identified by the state of a particular address line. i.e. The ZX printer is selected by the line A2 being active.

The instructions in this group are:

mnemonic	instruction hex.	I/O register	High P.A.	Low P.A.
IN A,(+dd)	DB dd	A	A	dd
IN A,(C)	ED 78	A	B	C

IN	H,(C)	ED 60	H	B	C
IN	L,(C)	ED 68	L	B	C
IN	B,(C)	ED 40	B	B	C
IN	C,(C)	ED 48	C	B	C
IN	D,(C)	ED 50	D	B	C
IN	E,(C)	ED 58	E	B	C
OUT	(+dd),A	D3 dd	A	A	dd
OUT	(C),A	ED 79	A	B	C
OUT	(C),H	ED 61	H	B	C
OUT	(C),L	ED 69	L	B	C
OUT	(C),B	ED 41	B	B	C
OUT	(C),C	ED 49	C	B	C
OUT	(C),D	ED 51	D	B	C
OUT	(C),E	ED 59	E	B	C

The non-automatic and automatic instructions are:

mnemonic	**instruction hex.**	**comment**
INI	ED A2	Non-automatic. Incrementing.
INIR	ED B2	Automatic. Incrementing.
IND	ED AA	Non-automatic. Decrementing.
INDR	ED BA	Automatic. Decrementing.
OUTI	ED A3	Non-automatic. Incrementing.
OTIR	ED B3	Automatic. Incrementing.
OUTD	ED AB	Non-automatic. Decrementing.
OTDR	ED BB	Automatic. Decrementing.

Group 17. The Interrupt instructions.

There are seven instructions in the Z80 instruction set that allow the programmer to manipulate the interrupt system of the Z80 microprocessor.

The instructions in this group are:

mnemonic	**instruction hex.**
EI	FB
DI	F3
IM 0	ED 46
IM 1	ED 56
IM 2	ED 5E
RETI	ED 4D
RETN	ED 45

Each of these instructions will now be discussed in turn.

E1:

When power is first applied to the Z80 the maskable interrupt system is unable to interrupt the Z80. This situation will exist until the system is 'enabled'

by the programmer including an 'EI' instruction in the program being followed by the microprocessor.

In the SPECTRUM system the maskable interrupt system is used for the real-time clock and keyboard scanning and the interrupts are generated by a 50 c.p.s. clock pulse.

DI:

At any point in a machine code program the programmer may decide to turn off (disable) the maskable interrupt system by the use of a 'DI' instruction which renders the microprocessor insensitive to the signals on the $\overline{INT}$ line. In the SPECTRUM system the maskable interrupt is disabled for the duration of the load, SAVE, VERIFY & MERGE operations so as to give a 'un-interrupted' period.

IM 0:

There are three interrupt modes. Interrupt mode 0 is selected automatically by the microprocessor when power is first applied, or by the execution of a 'IM 0' instruction. Interrupt mode 0 allows for peripheral devices to 'tell' the microprocessor which 'restart' routine is to be followed upon receipt of a maskable interrupt on the $\overline{INT}$ line. This mode is not used in the SPECTRUM system.

IM 1:

Interrupt mode 1 is selected by the execution of an 'IM 1' instruction and it is the mode used in the SPECTRUM system. The programmer of the 16K monitor program has included this instruction as part of the initialisation routine.

In this mode 'restart 0038' will always be selected upon receipt of a signal on the $\overline{INT}$ line, that is as long as the maskable interrupt system has been enabled. In the SPECTRUM system the machine code routine at 0038 updates the real-time clock and scans the keyboard.

IM 2:

Interrupt mode 2 is not used in the SPECTRUM system but it is the most powerful of the three interrupt modes. In this mode a peripheral device can indicate to the microprocessor which of 128 different subroutines is to be followed upon receipt of a maskable interrupt. The contents of the I register and a byte supplied by the peripheral device are used together to form a 16—bit address which is then used to address a 'vector table' which must previously have been prepared in memory.

RETI:

This is a special 'return' instruction for use with a maskable interrupt routine. The effect of the instruction is to return with the same maskable interrupt state as occurred before the maskable interrupt.

RETN:

This 'return' instruction is similar to 'RETI' but it is applicable at the end of a non-maskable interrupt routine.

Group 18. Miscellaneous instructions.

There are six further instructions to be mentioned. They are:

mnemonic	instruction hex.	mnemonic	instruction hex.
CPL	2F	CCF	3F
NEG	ED 44	HALT	76
SCF	37	DAA	27

Each of these instructions will now be discussed in turn.

CPL:

This is a simple instruction that complements the A register. It therefore sets any bit that is reset and vice versa. This operation is called 1's complementing. No major flags are affected.

NEG:

This instruction 2's complements the contents of the A register. By so doing it is similar to the two operations of 1's complementation and incrementation.

A 'NEG' instruction does affect the major flags. The sign and zero flags depend on the 'result'. The carry flag will be reset is the original value was zero otherwise it is reset and the overflow/parity flag is set if the original value was hex. 80 otherwise it is reset.

SCF:

This instruction sets the carry flag.

CCF:

This instruction complements the carry flag.

HALT:

This is a special instruction that makes the microprocessor stop executing instructions until an interrupt occurs. In the SPECTRUM system the only interrupts that occur are the maskable interrupts. Hence, as long as the interupts are enabled, a 'HALT' operation will be terminated by the next maskable interrupt signal. The PAUSE command uses this fact to count 1/50 ths. of a second.

DAA:

This is the 'decimally adjust the A register' instruction. In 'binary coded arithmetic' (BCD) the decimal numbers 0-9 are represented by the binary nibbles 0000-1001 and the nibbles 1010-1111 are not used. Therefore:

The byte 0000 0000 represents the number 0.

The byte 0011 1001 represents the number 39, etc.

This instruction converts bytes in absolute binary form to the BCD form.

The sign flag and the zero flag are affected by the 'result' and the overflow/parity flag is set by even parity. The effect on the carry flag depends on whether there is BCD overflow, in addition operation, or BCD borrow in subtraction operations.

6. UNDERSTANDING — Demonstration Machine Code Programs

6.1 Introduction

The principal aim of this chapter is to get the reader over the initial problems associated with writing machine code programs.

The instructions of the Z80 instruction set have all been listed in chapter 5 but no mention was made as to how machine code programs can be run on the SPECTRUM system. This present chapter contains a series of demonstration programs that show, in a simple manner, how the instructions can be used.

In chapter 8 there will be further advice about how machine code routines can be written to take full advantage of the SPECTRUMS's colour and high resolution display.

There are now three points to be discussed:

a) Choose an area of RAM.

A user-written machine code program must have allocated to it the required number of memory locations. In the SPECTRUM system there are several areas of the memory that may be used but the demonstration programs in this chapter will be allocated a part of the 'spare' RAM — location 32,000 and onwards. A machine code program in this area can be SAVEd and LOADed to and from tape as a block of data, if required.

b) Enter the bytes of the machine code.

There is no provision in the SPECTRUM system for the entering of machine code other than to use the command POKE. However, the operand of this command may be expressed as a decimal number, a binary number or an expression.

The following lines are therefore all valid:

```
    10 POKE 32000,201
or, 10 POKE 32000,BIN 11001001
or, 10 LET A=201: POKE 32000,A
```

and each may prove to be useful on certain occasions.

The recommended method, however, is to describe the machine code instructions in their appropriate hexadecimal characters. The following **Hex loader** routine will be used throughout this chapter.

```
10 LET D=32000: REM Hex loader
20 DEF FN A(A$,B)=CODE A$(B)-4
   8-7*(CODE A$(B) > 57)
30 DEF FN C(A$)=16*FN A(A$,1)+
   FN A(A$,2)
40 READ A$: IF A$ <> "**" THEN P
   OKE D,FN C(A$): LET D=D+1: GO TO 40
```

The above **Hex loader** will read a DATA list in which each pair of hexadecimal characters is enclosed by quotes and separated from other pairs by commas. The pair of characters — ** — is used as a terminator.

For example:

```
5Ø DATA "ØØ", "Ø1", "Ø2", "**"
RUN
```

would lead to location 32,000 holding 'Ø', location 32,001 holding '1' and location 32,002 holding '2'.

c) Execute the user-written machine code.

The BASIC command 'USR number' allows for execution of the SPECTRUM's monitor program to be stopped and user-written machine code to be followed instead. It is important to ensure that the operand of USR is set to the required location, that the user-written program does end with a 'return' instruction if it is indeed the user's wish to 'return to BASIC' and that the value returned by the USR command is handled appropriately.

The commonly used forms of USR are:

PRINT USR n — which prints the decimal value of the contents of the BC register pair.

RANDOMIZE USR n — which affects the random number generator.

LET A=USR n — which uses a further variable.

each of these forms is useful on occasions.

The programs that follow will be described in assembly format and a DATA list produced for use with the **Hex loader** given above.

The instructions are arranged in the same order as given in chapter 5.

6.2 The programs

Group 1. The NO OPERATION instruction. *(see also page 72)*

This instruction is very simple to use and a machine code program containing one or more NO OPERATION lines followed by a 'return' instruction does constitute a complete machine code program.

The following assembly listing shows an elementary machine code program using a single NO OPERATION instruction.

address	machine code	mnemonic	comment
7D00	00	NOP	A single NO OPERATION.
7D01	C9	RET	The 'return to BASIC'.

The following BASIC program shows this assembly listing being changed into the required form for the SPECTRUM.

Program 1. 'NOP'

Lines **1Ø — 4Ø**. The **Hex loader** as given above.

```
50 DATA "00","C9","**"
60 LET A=USR 32000
70 PRINT "The NOP machine code
   program has been executed succes
   sfully"
```

The reader is advised to get a good understanding of Program 1 before proceeding further.

Group 2. The instructions for loading registers with constants.
(see also page 72)

The 'USR number' command returns the value of the BC register pair as an absolute 16-bit number and the 'PRINT' command will display this number as a decimal number in the range 0—65,535.

In the following assembly listing the B and C registers are loaded with constants using instructions from this group and the BASIC program 2 uses this machine code routine.

address	machine code	mnemonic		comment
7D00	06 00	LD	B,+oo	Make the B register zero
7D02	0E 00	LD	C,+xx	The user will enter different values.
7D04	C9	RET		'Return to BASIC'.

In program 2 the instructions 'LD B,+dd' & 'LD C,+dd' are both used and the user is invited to enter different values as the operand of the latter instruction.

Program 2. 'LD B,+dd' & 'LD C,+dd'.

Lines 10 — 40. The **Hex loader** as given above.

```
50 DATA "06","00","0E","00"
51 DATA "C9","**"
60 INPUT "Enter a value for th
   e C register (0—255 only)",N
70 CLS
80 POKE 32003,N
90 PRINT AT 10,0;"The C regist
   er now holds";CHR$ 32;USR 32000
100 GO TO 60
```

The next program illustrates the 'LD BC,+dddd' instruction. So first the assembly listing:

address	machine code	mnemonic		comment
7D00	01 00 00	LD	BC,+xxxx	The user will enter different values here.
7D03	C9	RET		'Return to BASIC'.

In program 3 the instruction 'LD BC,+dddd' is used and the user is invited to enter different values. The value entered has to be split into 'high' and 'low' parts before it can be POKEd into the appropriate memory locations.

Program 3. 'LD BC,+dddd'

```
Lines 10 – 40. The Hex loader as given above.
50 DATA "01","00","00"
51 DATA "C9","**"
60 INPUT "Enter value for the
   BC register pair (0–65535 only)"
   ;CHR$ 32;N
70 CLS
80 POKE 32001,N–256*INT (N/256
   ): POKE 32002,INT (N/256)
90 PRINT AT 10,0;"BC register
   pair now holds";CHR$ 32;USR 3200
   0
100 GO TO 60
```

Group 3. Register copying and exchanging instructions. *(see also page 73)*
The register-to-register copying instructions can be demonstrated by loading registers, other than the B & C registers, with constants and then copying those constants to the B & C registers for returning to the user.

The next program shows the 'LD C,L' instruction being used.

address	machine code	mnemonic		comment
7D00	06 00	LD	B,+00	Make the B register zero.
7D02	2E 00	LD	L,+xx	Enter different values.
7D04	4D	LD	C,L	Copy 'L to C'.
7D05	C9	RET		'Return to BASIC'.

In program 4 the user is invited to enter different values into the L register and then have them returned in the C register.

Program 4. 'LD C,L'

```
Lines 10 – 40. The Hex loader as given above.
50 DATA "06","00","2E","00"
51 DATA "4D","C9","**"
60 INPUT "Enter a value for th
   e L register (0–255 only)";CHR$ 3
   2;N
70 CLS
80 POKE 32003,N
90 PRINT AT 10,0;"The C regist
```

```
er now holds"; CHR$ 32;USR 32000
100 GO TO 60
```

The reader is encouraged to try other instructions from this group.

For example:

```
50 DATA "06","00","0E","00"
51 DATA "65","7C","57","5F"
52 DATA "4B","C9","***"
```

is perfectly valid. Do you see just what is being done here? The instruction 'EX DE,HL' can also be included in program 4. For example by including the following routine.

address	machine code	mnemonic		comment
7D00	26 00	LD	H,+00	Make the H register zero.
7D02	2E 00	LD	L,+xx	Enter different values.
7D04	EB	EX	DE,HL	Move value to DE.
7D05	42	LD	B,D	Move D to B and
7D06	4B	LD	C,E	E to C.
7D07	C9	RET		'Return to BASIC'.

This would give a DATA list as follows:

```
50 DATA "26","00","2E","00"
51 DATA "EB","42","4B"
52 DATA "C9","***"
```

and can be used with program 4.

The last two instructions in this group are 'EX AF,A'F" and 'EXX'. In the SPECTRUM system the user is perfectly allowed to use the alternate register set with the exception of H' & L' that contain the 'return address to BASIC'. The reader might like to try including these instructions in program 4.

For example:

```
50 DATA "06","00","2E","00"
51 DATA "08","D9","D9","08"
52 DATA "4D","C9","***"
```

or something more ambitious.

(Line 51 simply switches all the main registers and then switches them back, unaltered.)

Group 4. Instructions for the loading of registers with data copied from a memory location. *(see also page 75.)*

The instructions in this group allow the user to load registers with copies of the data contained in addressed locations. This addressing can be 'absolute', 'indirect' or 'indexed'.

The first routine shows absolute addressing. A location, 31,999, is given the name STORE and a 'LD A,(addr)' instruction is used to fetch the contents of STORE using the following routine. In the listing the first line 'equates' STORE with the location hex. 7CFF, decimal 31,999.

```
        STORE    equ. 7CFF
```

address	machine code	mnemonic		comment
7D00	06 00	LD	B,+00	Make the B register zero.
7D02	3A FF 7C	LD	A,(STORE)	Fetch current value.
7D05	4F	LD	C,A	Move A to C.
7D06	C9	RET		'Return to BASIC'.

In program 5 the user is invited to enter different values to be held in the location 'STORE'. The program then uses the above routine to return the value to the user.

Program 5. 'LD A,(addr)'

Lines 1Ø – 4Ø. The **Hex loader** as given above.

```
5Ø DATA "Ø6","ØØ"
51 DATA "3A","FF","7C"
52 DATA "4F","C9","**"
6Ø INPUT "Enter a value for lo
   cation STORE(Ø–255 only)"; CHR$ 3
   2;N
7Ø CLS
8Ø POKE 31999,N
9Ø PRINT AT 1Ø,Ø;"The location
STORE now holds";CHR$ 32;USR 32
   ØØØ
1ØØ GO TO 6Ø
```

The second routine shows indirect addressing. In this routine the address of STORE is first loaded into the HL register pair before a 'LD C,(HL)' instruction is used.

```
        STORE    equ. 7CFF
```

address	machine code	mnemonic		comment
7D00	06 00	LD	B,+00	Make the B register zero.
7D02	21 FF 7C	LD	HL,+STORE	Make the HL register pair point to STORE.
7D05	4E	LD	C,(HL)	Fetch the current value.
7D06	C9	RET		'Return to BASIC'.

In program 6 the user is once again invited to enter different values to be held in the location 'STORE'.

Program 6. 'LD C,(HL)'

Lines 10 – 40. The **Hex loading** as given above.

```
50 DATA "06","00"
51 DATA "21","FF","7C"
52 DATA "4E","C9","**"
```

Lines 60 – 100 as given in program 5.

The third routine shows indexed addressing. In the routine the location 31,936, hex. 7CC0, is called BASE and the location STORE, 31,999, hex. 7CFF, is considered as 'BASE + 3F'. A 'LD C,(IX+d)' instruction is then used to get the current value of STORE.

```
BASE    equ. 7CC0
STORE   equ. BASE+3F
```

address	machine code	mnemonic		comment
7D00	06 00	LD	B,+00	Make the B register zero.
7D02	DD 21 C0 7C	LD	IX,+BASE	Set the IX register pair.
7D06	DD 4E 3F	LD	C,(BASE+3F)	Fetch the current value.
7D09	C9	RET		'Return to BASIC'.

In program 7 the user is once again invited to enter different values to be held in the location 'STORE'.

Program 7. 'LD C,(IX+d)'

Lines 10 – 40. The **Hex loader** as given above.

```
50 DATA "06","00"
51 DATA "DD","21","C0","7C"
52 DATA "DD","4E","3F"
53 DATA "C9","**"
```

Lines 60 – 100 as given in program 5.

In program 7 the IX register pair has been used but it is quite possible to use the IY register pair if preferred. The mnemonics will need to be changed so as to read IY instead of IX and the machine code bytes DD changed to read FD. Note that the maskable interrupt will need to be disabled whilst the IY holds a new value.

Group 5. Instructions for loading locations in memory with data copied from registers, or with constants. *(see also page 78.)*

The instructions in this group allow the user to load addressed locations with data copied from registers or with constants. Once again 'absolute', 'indirect' and 'indexed' addressing are possible.

The first routine shows absolute addressing. In the routine the absolute address for the location 31,999, hex. 7CFF, and named 'STORE' is used in a 'LD (addr),A' instruction.

```
STORE   equ. 7CCF
```

address	machine code	mnemonic		comment
7D00	3E 00	LD	A,+xx	Enter different values.
7D02	32 FF 7C	LD	(STORE),A	Enter the current value into STORE.
7D05	C9	RET		'Return to BASIC'.

In program 8 the user is again invited to enter different values for passing to the location STORE. The machine code routine is executed by using RANDOMIZE USR 32000 and the value in STORE recovered by using PEEK 31999.

Program 8. 'LD (addr),A'

Lines 10 – 40. The **Hex loader** as given above.

```
50 DATA "3E","00"
51 DATA "32","FF","7C"
52 DATA "C9","**"
60 INPUT "Enter a value for lo
   cation STORE(0–255 only)";CHR$ 3
   2;N
70 CLS
80 POKE 32001,N: RANDOMIZE USR
   32000
90 PRINT AT 10,0;"The location
   STORE now holds"; CHR$ 32; PEEK 3
   1999
100 GO TO 60
```

The second routine shows indirect addressing. In the routine the HL register pair is set to point to the location STORE and a 'LD (HL),E' instruction is used to make the transfer.

```
STORE    equ. 7CFF
```

address	machine code	mnemonic		comment
7D00	1E 00	LD	E,+xx	Enter different values.
7D02	21 FF 7C	LD	HL,+STORE	Make the HL register pair point to STORE.
7D05	73	LD	(HL),E	Make the transfer.
7D06	C9	RET		'Return to BASIC'.

In program 9 the user is again invited to enter values to be transferred to STORE and read back by PEEK 31999.

Program 9. 'LD (HL),E'

Lines 10 – 40. The **Hex loader** as given above.

```
50 DATA "1E","00"
51 DATA "21","FF","7C"
```

52 DATA "73","C9","***"

Lines 6Ø – 1ØØ as given in program 8.

In the third routine indexed addressing is used. On this occasion location 32,061, hex. 7D3D, is called BASE and the location STORE is therefore considered as 'BASE – 3E'.

```
BASE    equ. 7D3D
STORE   equ. BASE–3E
```

address	machine code	mnemonic		comment
7D00	3E 00	LD	A,+xx	Enter different values.
7D02	DD 21 3D 7D	LD	IX,+BASE	Make the IX register pair point to BASE.
7D06	DD 77 C2	LD	(IX–3E),A	'IX–3E' is taken as 'IX+C2'.
7D09	C9	RET		'Return to BASIC'.

In program 10 the user is again invited to enter values to be transferred to STORE and read back by PEEK 31999.

Program 10. 'LD (IX+d),A'

Lines 1Ø – 4Ø. The **Hex loader** as given above.

5Ø DATA "3E","ØØ"

51 DATA "DD","21","3D","7D"

52 DATA "DD","77","C2"

53 DATA "C9","***"

Programs 5 – 10 have all used single byte numbers but the reader is urged to adapt the programs to use 2-byte numbers and the corresponding register pair instructions.

Group 6. The addition instructions. *(see also page 80)*

The instructions in this group allow the user to add values together (ADD), increment values (INC) and add with carry (ADC).

The first routine shows an 'ADD A,B' instruction being used.

address	machine code	mnemonic		comment
7D00	00	NOP		For use later.
7D01	3E 00	LD	A,+xx	Enter two different
7D03	06 00	LD	B,+xx	values into these registers.
7D05	80	ADD	A,B	Make the addition.
7D06	06 00	LD	B,+00	Make the B register zero.
7D08	4F	LD	C,A	Transfer the 'result'.
7D09	C9	RET		'Return to BASIC'.

In program 11 the user is invited to enter two numbers. These numbers are transferred to the A & B register and added together in absolute binary arithmetic. The result is returned by the USR function.

Program 11. 'ADD A,B'

Lines 1Ø– 4Ø. The **Hex loader** as given above.

```
5Ø DATA "ØØ","3E","ØØ"
51 DATA "Ø6","ØØ","8Ø"
52 DATA "Ø6","ØØ","4F"
53 DATA "C9","**"
6Ø INPUT "Enter a first value
   (Ø–255)";CHR$ 32;F
7Ø INPUT "Enter a second value
   (Ø–255)";CHR$ 32;S
8Ø CLS
9Ø POKE 32ØØ2,F: POKE 32ØØ4,S
1ØØ PRINT AT 1Ø,5;F;CHR$ 32;"AD
   D";CHR$ 32;S;CHR$ 32;"=";CHR$.32
   ;USR 32ØØØ
   11Ø GO TO 6Ø
```

The second routine for this group of instructions uses an 'INC BC' instruction.

address	machine code	mnemonic	comment
7D00	01 00 00	LD BC,+xxxx	Make the BC register pair hold different values.
7D03	03	INC BC	Increment the value.
7D04	C9	RET	'Return to BASIC'.

In program 12 the user is invited to enter a number in the range 0 – 65,535. This number is then split into high and low parts and POKEd into locations 7D01 & 7D02. The BC register pair is then incremented and the value returned by the USR function.

Note the effect of entering the value 65,535.

Program 12. 'INC BC'

Lines 1Ø – 4Ø. The **Hex loader** as given above.

```
5Ø DATA "Ø1","ØØ","ØØ"
51 DATA "Ø3","C9","**"
6Ø INPUT "Enter a value (Ø–655
   35)";CHR$ 32;N
7Ø POKE 32ØØ2, INT (N/256)
8Ø POKE 32ØØ1,N–256*INT (N/256
   )
9Ø CLS
1ØØ PRINT AT 1Ø,Ø;N;CHR$ 32;"in
   crements to give";CHR$ 32;USR 32
   ØØØ
11Ø GO TO 6Ø
```

The third routine shows an 'ADC A,B' instruction being used. (This is the same routine as used in program 11 but changed to include the 'ADC A,B' instruction rather than the 'ADD A,B'.)

address	machine code	mnemonic	comment
7D00	37	SCF	Set the carry flag.
7D01	3E 00	LD A,+xx	Enter different values
7D03	06 00	LD B,+xx	into these registers.
7D05	88	ADC A,B	Make the addition with carry.
7D06	06 00	LD B,+00	Make the B register zero.
7D08	4F	LD C,A	Transfer the 'Result'.
7D09	C9	RET	'Return to BASIC'.

This routine is used in program 13. in which the user is invited to add two numbers together **with carry.** The carry flag is always set.

Try the effect of changing the instruction 'SCF' to 'AND A' (hex. A7) which will give carry reset.

Program 13. 'ADC A,B'

Lines 1Ø – 4Ø. The **Hex loader** as given above.

```
5Ø DATA "37","3E","00"
51 DATA "Ø6","ØØ","88"
52 DATA "Ø6","ØØ","4F"
53 DATA "C9","**"
6Ø INPUT "Enter a first value
   (Ø–255)";CHR$ 32;F
7Ø INPUT "Enter a second value
    (Ø–255)";CHR$ 32;S
8Ø CLS
9Ø POKE 32ØØ2,F: POKE 32ØØ4,S
1ØØ PRINT AT 1Ø,Ø;"With carry s
   et";CHR$ 32;F;CHR$ 32;"ADC";CHR$
32;S,CHR$ 32;"=";CHR$ 32;USR 32ØØØ
```

Group 7. The subtraction instructions. *(see also page 82)*

The instructions in this group allow the user to subtract values from each other (SUB), decrement values (DEC) and subtract with carry (SBC).

The first routine for this group of instructions uses a 'SUB B' instruction. The state of the carry flag after the subtraction is determined and the value, 0 or 1, is saved in the location STORE.

```
        STORE   equ. 7CFF
```

address	machine code	mnemonic	comment
7D00	00	NOP	For use later.

7D01	3E 00	LD	A,+xx	Enter two values
7D03	06 00	LD	B,+xx	into these registers.
7D05	90	SUB	B	The subtraction.
7D06	06 00	LD	B,+00	Make the B register zero.
7D08	4F	LD	C,A	Transfer the 'result'.
7D09	3E 00	LD	A,+00	Make the A register zero.
7D0B	CE 00	ADC	A,+00	Add with carry.
7D0D	32 FF 7C	LD	(STORE),A	Transfer the 'value of the carry' to STORE.
7D10	C9	RET		'Return to BASIC'.

In program 14 the above routine is called after the user has entered values for the A & B registers. The value of the carry flag is obtained by using PEEK 31999.

Program 14. 'SUB B'

```
Lines 10 — 40. The Hex loader as given above.
50 DATA "00","3E","00"
51 DATA "06","00","90"
52 DATA "06","00","4F"
53 DATA "3E","00","CE","00"
54 DATA "32","FF","7C"
55 DATA "C9","**"
60 INPUT "Enter a first value
   (0—255)";CHR$ 32;F
70 INPUT "Enter a second value
   (0—255)";CHR$ 32;S
80 CLS
90 POKE 32002,F: POKE 32004,S
100 PRINT AT 10,0;F;CHR$ 32;"SU
   B":CHR$ 32;S;CHR$ 32;"=";CHR$ 32
   ;USR 32000;CHR$ 32;"with carry";
   CHR$ 32;"set" AND PEEK 31999;"re
   set" AND NOT PEEK 31999
110 GO TO 60
```

Try the above program with numbers that are 'less than', 'greater than' and 'equal' to each other.

The second group of instructions in this group contains the DEC instructions.

The 'DEC BC' instruction can be demonstrated by making the appropriate amendments to program 12.

They are:

Change Line 51 & 100 to read:

```
51 DATA "ØB","C9","**"
1ØØ PRINT AT 1Ø,Ø;N;CHR$ 32;"de
   crements to give";CHR$ 32;USR 32
   ØØØ
```

The third subgroup of instructions in this group contains the SBC instructions.

The 'SBC A,B' instruction can be demonstrated by making the appropriate amendments to program 14.

They are:

Change lines 51 & 1ØØ to read:

```
5Ø DATA "37","3E","ØØ"          for carry set.
or, 5Ø DATA "A7","3E","ØØ"      for carry reset.
   51 DATA "Ø6","ØØ","98"       'SBC A,B' is hex. 98.
   1ØØ change "SUB" to read "SBC"
```

Once again try the effect of numbers that are 'less than', 'greater than' and 'equal' to each other.

Note that there is no difference in the results between 'SUB' and 'SBC with carry reset'.

Group 8. The Compare instructions. *(see also page 85)*

The compare instructions are in effect the same as the SUB instructions apart from the fact that the A register is unchanged. The carry flag is affected and can subsequently be tested.

The instruction 'CP B' can be demonstrated by once again making the appropriate changes to program 12.

They are:

Change lines S1 & 1ØØ to read:

```
51 DATA "Ø6","ØØ","B8"          'CP B' is hex. B8.
1ØØ PRINT AT 1Ø,Ø;F;CHR$ 32;"CP
   ";CHR$ 32;S;CHR$ 32;"gives carry
   ";CHR$ 32;
```

and add lines 1Ø1 & 1Ø2.

```
1Ø1 RANDOMIZE USR 32ØØØ
1Ø2 PRINT "re" AND NOT PEEK 319
     99;"set"
```

Group 9. The Logical instructions. *(see also page 86)*

The instructions in this group allow the user to logically AND, OR and XOR two 8-bit numbers.

The following routine shows a 'AND B' instruction being used to logically AND two values entered by the user.

```
        STORE    equ. 7CFF
```

address	machine code	mnemonic		comment
7D00	3E 00	LD	A,+xx	Enter two values
7D02	06 00	LD	B,+xx	into these registers.
7D04	A0	AND	B	Logical AND.
7D05	32 FF 7C	LD	(STORE),A	Copy result to STORE.
7D08	C9	RET		'Return to BASIC'.

In program 15 the above routine is used. The user is asked to input two values in turn. As each value is entered it is displayed in its binary form. The result is collected from STORE by using — PEEK 31999, and also displayed in binary form.

Program 15. 'AND B'

Lines 1Ø — 4Ø. The **Hex loader** as given above.

```
5Ø DATA "3E","ØØ"
51 DATA "Ø6","ØØ"
52 DATA "AØ"
53 DATA "32","FF","7C"
54 DATA "C9","**"
6Ø INPUT "Enter a first value
   (Ø—255)";CHR$ 32;F
7Ø CLS
8Ø POKE 32ØØ1,F
9Ø PRINT AT 8,4;: GO SUB 3ØØ
1ØØ PRINT AT 1Ø,8;"AND"
11Ø INPUT "Enter a second value
   (Ø—255)";CHR$ 32;S
12Ø POKE 32ØØ3,S
13Ø PRINT AT 12,4;: LET F=S: GO
   SUB 3ØØ
14Ø PRINT AT 14,7;"gives"
15Ø RANDOMIZE 32ØØØ
16Ø PRINT AT 16,4;: LET F=PEEK
   31999: GO SUB 3ØØ
17Ø GO TO 6Ø

3ØØ REM Binary of F
31Ø FOR N=7 TO 1 STEP —1
32Ø LET P=2 ↑N
33Ø PRINT CHR$ (48+INT (F/P));C
   HR$ 32;
34Ø LET F=F—INT (F/P)*P
35Ø NEXT N
36Ø PRINT INT F
370 RETURN
```

Program 15 can be adapted to demonstrate the instructions 'OR B' and 'XOR B'.

For 'OR B' the required changes are to lines 52 & 10Ø as follows:

```
52 DATA "BØ"
10Ø PRINT AT 1Ø,8;"OR"
```

and for 'XOR B' they are:

```
52 DATA "A8"
10Ø PRINT AT 1Ø,8;"XOR"
```

The reader is encouraged to become familiar with these three logical operations.

Group 10. The Jump instructions. *(see also page 89)*

The seventeen machine code instructions in this group allow the user to make jumps from one part of a machine code routine to another. The jumps can be relative, i.e. —128 to +127 decimal locations from the present Program Counter location, or absolute, i.e. an address of a location is given. A jump can also be made conditional on the state of a major flag although it is only with the absolute jump instructions that there is a full range of instructions available.

The following routine will be used in program 16 to demonstrate the instructions in this group.

```
        NEXT    equ. 7D0F
```

address	machine code	mnemonic	comment
7D00	3E 00	LD A,+xx	Enter two values and compare
7D02	FE 00	CP +xx	them so as to set the flags.
7D04	01 00 00	LD BC,+0000	Set BC to zero.
7F07	18 06	JR NEXT	Jump forward to 'NEXT'.
7D09	00	NOP	For later use.
7D0A	C9	RET	'Return' if no jump made.
7D0B	00 00 00 00	—	Four unused locations.
7D0F	03	INC BC	Increment BC as jump made.
7D10	C9	RET	'Return to BASIC'.

In the above routine the BC register pair will be returned holding zero if 'no jump' is made but will contain '1' if a 'jump' is made.

Program 16 shows this routine being used to demonstrate the 'JR e' instruction. This instruction is unconditional so the jump will be regardless of the result of the 'test' entered by the user.

Program 16. 'JR e'

Lines 1Ø — 4Ø. The **Hex loader** as given above.

```
50 DATA "3E", "00"
51 DATA 'FE", "00"
52 DATA "01","00","00"
53 DATA "18","06","00"
54 DATA "C9"
55 DATA "00","00","00","00"
56 DATA "03","C9","***"
60 PRINT AT 4,0;"Instruction -
   ",
70 PRINT " 'JR  e' "
80 PRINT AT 8,0;"Test -",
90 INPUT "Enter a first value
   (0-255)";CHR$ 32;F
100 PRINT F;CHR$ 32;"CP"; CHR$ 3
    2;
110 INPUT "Enter a second value
   (0-255)";CHR$ 32;S
120 PRINT S
130 PRINT AT 12,0;"JUMP -",
140 POKE 32001,F: POKE 32003,S
150 LET R=USR 32000: PRINT "NO"
    AND NOT R;"YES" AND R
160 PRINT AT 21,0;"Any key to c
    ontinue"
170 PAUSE 50
180 IF INKEY$="" THEN GO TO 180
190 CLS: GO TO 60
```

In the above program the user is asked to enter two values. A 'compare' operation is then made and a jump if it is needed. The variable R will be zero if 'no jump' is taken and '1' if it is.

The program has been written so that the user can, by amending only lines 53 & 70, demonstrate fourteen of the seventeen jump instructions.

For the relative jump instructions the changes are:

'JR NZ,e'	–	53 DATA "20","06","00" 70 PRINT " 'JR NZ,e' "
'JR Z	–	53 DATA "28","06","00" 70 PRINT " 'JR Z,e' "
'JR NC,e'	–	53 DATA "30","06","00" 70 PRINT "'JR NC,e' "
'JR C,e'	–	53 DATA "38","06","00" 70 PRINT " 'JR C,e' "

The absolute jump instructions are demonstrated by changing the machine code routine so that an absolute jump to location 'NEXT' at 7D0F is made.

The instruction line will now read — using 'JP addr':

address	machine code	mnemonic	comment
7D07	C3 0F 7D	JP NEXT	Jump forward to 'NEXT'.

The changes to program 16 will be:

'JP addr' — 53 DATA "C3","ØF","7D"
70 PRINT " 'JP addr' "

'JP NZ,addr' — 53 DATA "C2","ØF","7D"
70 PRINT " 'JP NZ,addr' "

'JP Z,addr' — 53 DATA "CA","ØF","7D"
70 PRINT " 'JP Z,addr' "

'JP NC,addr' — 53 DATA "D2","ØF","7D"
70 PRINT " 'JP NC,addr' "

'JP C,addr' — 53 DATA "DA","ØF","7D"
70 PRINT " 'JP C,addr' "

'JP PO,addr' — 53 DATA "E2","ØF","7D"
70 PRINT " 'JP PO,addr' "

'JP PE,addr' — 53 DATA "EA","ØF","7D"
70 PRINT " 'JP PE,addr' "

'JP P,addr' — 53 DATA "F2","ØF","7D"
70 PRINT " 'JP P,addr' "

'JP M,addr' — 53 DATA "FA","ØF","7D"
70 PRINT " 'JP M,addr' "

The remaining three instructions in this group use indirect addressing and it is left as an exercise for the interested reader to ammend program 16 to show these instructions.

Group 11. The 'DJNZ,e' instruction. *(see also page 95)*

The instruction 'DJNZ,e' is a most useful instruction and can readily be used to produce simple 'loops' in a machine code program.

To use a 'DJNZ,e' instruction the programmer has first to specify the number of loops required and this number has to be copied to the B register. The 'work' of the loop can now be done and the 'DJNZ,e' instruction is put after the 'work' to act in a 'NEXT B' manner.

The following four line BASIC program will print out the alphabet, in capitals, over and over again. The prompt 'scroll?' appears when the screen is full.

```
10 FOR A=65 TO 90
20 PRINT CHR$ A;
30 NEXT A
40 GO TO 10
```

The following machine code routine shows this same operation being performed. But note that the programmer can only use 'STEP −1'

```
        LOOP    equ. 7D02
```

address	machine code	mnemonic	comment
7D00	06 1A	LD B,+1A	There are to be dec. 25 loops back from 'DJNZ,e'.
7D02	3E 5B	LD A,+5B	'A' is hex. 5B−1A.
7D04	90	SUB B	Value is in the A register.
7D05	D7	RST 0010	Prints the character — see later.
7D06	10 FA	DJNZ,LOOP	Back for another letter if
7D08	C9	RET	needed; otherwise 'return'.

Program 17 uses the above routine to print the alphabet on the T.V. screen. Repeated calls to 'USR' 32000' will repeat the printing operation over and over again.

Program 17. 'DJNZ,e'

Lines 10 — 40. The **Hex loader** as given above.

```
50 DATA "06","1A"
51 DATA "3E","5B"
52 DATA "90"
53 DATA "D7"
54 DATA "10","FA"
55 DATA "C9","**"
60 PRINT
70 RANDOMIZE USR 32000
80 GO TO 70
```

Note: The inclusion of line 60 is important as it opens the channel to the 'main' area of the display. If this instruction, or a similar one, is omitted then the printing will be made in the 'edit' area.

Group 12. The Stack instructions. *(see also page 96)*

There are five subgroups of instructions in this group. The first subgroup contains the 'PUSH' and 'POP' instructions and the second subgroup the 'stack exchanging' instructions.

The following routine is used in program 18 to demonstrate instructions from the first subgroup.

address	machine code	mnemonic	comment
7D00	21 00 00	LD HL,+xxxx	Enter different values.
7D03	E5	PUSH HL	'PUSH' this value.
7D04	C1	POP BC	Now it is in BC.
7D05	C9	RET	'Return to BASIC'.

In program 18 the user is asked to enter a value. This value is then copied into the HL register pair. It is subsequently passed to the machine stack. Finally it is 'POPped' into the BC register pair.

Program 18. 'PUSH HL' & 'POP BC'

```
Lines 10 – 40. The Hex loader as given above.
50 DATA "21","00","00"
51 DATA "E5","C1"
52 DATA "C9","**"
60 INPUT "Enter a value (0—655
   35)";CHR$ 32;F
70 POKE 32001,F—INT (F/256)*25
   6: POKE 32002,INT (F/256)
80 CLS
90 PRINT AT 10,0;"Value taken
   off stack =";CHR$ 32;USR 32000
100 GO TO 60
```

The reader is encouraged to try alternative routines in the above program. But care must be taken to ensure that the number of 'PUSHes' balances the number of 'POPs'.

The third and fourth subgroups of instructions contain the 'CALL' and 'RET' instructions.

The following machine code routine can be used to demonstrate the different 'CALL' instructions.

```
NEXT    equ. 7D0F
```

address	machine code	mnemonic	comment
7D00	3E 00	LD A,+xx	Enter two values and
7D02	FE 00	CP +xx	compare them.
7D04	01 00 00	LD BC,+0000	The BC registers hold zero.
7D07	CD 0F 7D	CALL NEXT	Call the subroutine.
7D0A	C9	RET	'Return to BASIC'
7D0B	00 00 00 00	—	Four unused locations.
7D0F	03	INC BC	BC now holds '1'.
7D10	C9	RET	Return from subroutine.

The above routine can be used in program 16 with the following changes for the different instructions.

For all instructions:

```
130 PRINT AT 12,0;"CALL —",
```

'CALL addr'	—	53 DATA "CD","0F","7D" 70 PRINT "'CALL addr' "
'CALL NZ,addr'	—	53 DATA "C4","0F","7D" 70 PRINT " 'CALL NZ,addr' "
'CALL Z,addr'	—	53 DATA "CC","0F","7D" 70 PRINT " 'CALL Z,addr' "
'CALL NC,addr'	—	53 DATA "D4","0F","7D" 70 PRINT " 'CALL NC,addr' "
'CALL C,addr'	—	53 DATA "DC","0F","7D" 70 PRINT " 'CALL C,addr' "
'CALL PO,addr'	—	53 DATA "E4","0F","7D" 70 PRINT "'CALL PO,addr' "
'CALL PE,addr'	—	53 DATA "EC","0F","7D" 70 PRINT " 'CALL PE,addr' "
'CALL P,addr'	—	53 DATA "F4","0F","7D" 70 PRINT "'CALL P,addr' "
'CALL M,addr'	—	53 DATA "FC","0F","7D" 70 PRINT " 'CALL M,addr' "

The following machine code routine can be used to demonstrate the 'RET' instructions. However it has been left as an exercise for the reader to make the necessary amendments to program 16.

address	machine code	mnemonic	comment
7D00	3E 00	LD A,+xx	Enter two values and
7D02	FE 00	CP +xx	compare them.
7D04	01 01 00	LD BC,+0001	Make the BC pair hold '1'.
7D07	D8	RET C	Return if carry set.
7D08	0B	DEC BC	BC will now hold zero.
7D09	C9	RET	An ordinary return.

Note: In the above routine the 'RET C' instruction is being demonstrated. Also the logic has been changed so that program 16 will give 'YES' if the conditional return is made.

The 'RST' instructions will be discussed in chapters 7 & 8.

Group 13. The Rotation instructions. *(see also page 99)*

There are a large number of Rotation instructions in the Z80 instruction set.

The following routine is used in program 19 to demonstrate the seven types of rotation that involve the C register.

```
        STORE    equ. 7CFF
address machine code  mnemonic        comment
7D00    AF            XOR A           Clear the A register.
7D01    FE 00         CP  +xx         Compare against a value.
                                      Zero gives carry reset;
                                      '1' gives carry set.
7D03    06 00         LD  B,+00       Make the B register zero.
7D05    0E 00         LD  C,+xx       Give the C register a value.
7D07    CB 01         RLC C           Make a rotation.
7D09    3E 00         LD  A,+00       Make the A register zero.
7D0B    CE 00         ADC A,+00       Will transfer the state of
7D0D    32 FF 7C      LD  (STORE),A   the carry flag to STORE.
7D10    C9            RET             'Return to BASIC'.
```

In program 19 the user is asked to enter '0' or '1' in order to make the carry flag reset or set. The next prompt asks for a 'value'. This is copied to the C register and rotated as required. The 'result' is printed together with the current value of the carry flag.

Program 19. 'RLC C'

Lines 10 — 40. The **Hex loader** as given above.

```
50 DATA "AF","FE","00"
51 DATA "06","00","0E","00"
52 DATA "CB","01"
53 DATA "3E","00","CE","00"
54 DATA "32","FF","7C"
55 DATA "C9","**"
60 PRINT AT 2,0;"Instruction -
   ",
70 PRINT " RLC C' "
80 INPUT "Carry reset or set?
   (0/1)";CHR$ 32;C
90 POKE 32002,C
100 PRINT AT 6,0;"CARRY -",C
110 INPUT "Enter a value (0-255
    )";CHR$ 32;F
120 POKE 32006,F
130 PRINT AT 10,0;"Initial valu
    e";: GO SUB 300
140 LET F=USR 32000
150 PRINT AT 14,0;"Final value
    ,: GO SUB 300
160 PRINT AT 18,0;"CARRY -", PEE
```

```
    K 31999
170 PRINT AT 21,0;"Any key to c
    ontinue"
180 PAUSE 50
190 IF INKEY$="" THEN GO TO 190
200 CLS: GO TO 60
300 REM Binary of F
310 FOR N=7 to 1 STEP -1
320 LET P=2↑N
330 PRINT CHR$ (48+INT (F/P));C
    HR$ 32;
340 LET F=F-INT (F/P)*P
350 NEXT N
360 PRINT INT F
370 RETURN
```

Program 19 can be used to demonstrate the seven instructions that use the C register.

The required changes are:

'RLC C'	–	52 DATA "CB","01" 70 PRINT " 'RLC C' "
'RRC C'	–	52 DATA "CB","09" 70 PRINT " 'RRC C' "
'RL C'	–	52 DATA "CB","11" 70 PRINT " 'RL C' "
'RR C'		52 DATA "CB","19" 70 PRINT " 'RR C' "
'SLA C'	–	52 DATA "CB","21" 70 PRINT " 'SLA C' "
'SRA C'	–	52 DATA "CB","29" 70 PRINT " 'SRA C' "
'SRL C'	–	52 DATA "CB","39" 70 PRINT " 'SRL C' "

If wished the routine in program 19 can be changed to demonstrate the four single byte instructions.

Group 14. The 'Bit handling' instructions. (see also page 102 *)*

The instructions in this group can be split into three subgroups.

The RES & SET instructions are not commonly used instructions and no BASIC program is given here to demonstrate their use. The reader however is

welcome to write simple programs that use these instructions.

The BIT instructions are by far the most useful instructions in this group and the following program shows 'BIT 7,H' being used in an impressive manner.

Program 20 is a 'Binary' printing program. The user is asked to enter an integer in the range 0—65,535. The machine code routine is then called to print out the binary form.

The number given by the user is transferred to the HL register pair and bit 7 of the H register is read and the required 'Ø' or '1' is printed. The H & L registers are then shifted leftwards, the carry from L being picked up by H. Once again bit 7 of the H register is read and the digit printed. This operation of reading and shifting is performed sixteen times to build up the complete binary number.

The routine is:

```
        LOOP    equ. 7D05
        PRINT   equ. 7D0C
```

address	machine code	mnemonic	comment
7D00	21 00 00	LD HL,+xxxx	The value entered.
7D03	06 10	LD B,+10	There are 16 bits.
7D05	CB 7C	BIT 7,H	Test the lefthand bit.
7D07	3E 30	LD A,'Ø'	Prepare to print zero.
7D09	28 01	JR Z,PRINT	Jump if zero needed.
7D0B	3C	INC A	Now the character '1'.
7D0C	D7	RST 0010	Print the 'Ø' or the '1'.
7D0D	3E 20	LD A,'sp.'	Prepare to print a 'space'.
7D0F	D7	RST 0010	Print the 'space'.
7D10	CB 15	RL L	Rotate L.
7D12	CB 14	RL H	Rotate H picking up carry.
7D14	10 EF	DJNZ,LOOP	Back until finished.
7D16	C9	RET	'Return to BASIC'

Program 20 below shows this routine being used.

Program 20. 'BIT 7,H'

Lines 1Ø1 — 4Ø. The **Hex loader** as given above.

```
5Ø DATA "21","ØØ","ØØ"
51 DATA "Ø6","1Ø","CB","7C"
52 DATA "3E","3Ø","28","Ø1"
53 DATA "3C","D7"
54 DATA "3E","2Ø","D7"
55 DATA "CB","15","CB","14"
56 DATA "1Ø,"EF","C9","**"
6Ø INPUT "Enter a value (Ø—655
```

```
   35)"; CHR$ 32;F
70 POKE 32001,F-INT (F/256)*25
   6
80 POKE 32002,INT (F/256)
90 CLS
100 PRINT AT 10,8;"Binary of";C
    HR$ 32;F
110 PRINT AT 12,14;"is"
120 PRINT AT 14,0;
130 RANDOMIZE 32000
140 GO TO 60
```

Group 15. The Block handling instructions. (see also page 102)

The instructions in this group allow the user to move blocks of data or to search blocks of data.

Of all the instructions in the group the 'LDIR' instruction is by far the most common.

The following routine uses the 'LDIR' instruction to copy the 'top third' of the display area to the 'middle third'. This means that whenever the routine is called all the bytes in locations hex. 4000–47FF are copied to locations 4800–4FFF. The user can see that this has occurred as all the characters that were in lines 0–7 became duplicated in lines 8–15.

The routine is:

address	machine code	mnemonic	comment
7D00	21 00 40	LD HL,+4000	The top-left of the display.
7D03	11 00 48	LD DE,+4800	The start of line 8.
7D06	01 00 08	LD BC,+0800	There are 2,048 locations.
7D09	ED B0	LDIR	Move the block.
7D0B	C9	RET	'Return to BASIC'.

Program 21 uses this routine.

Program 21. 'LDIR'

Lines 10 – 40. The **Hex loader** as given above.

```
50 DATA "21","00","40"
51 DATA "11","00","48"
52 DATA "01","00","08"
53 DATA "ED","B0"
54 DATA "C9","**"
60 INPUT "Enter your character
   s – 8 lines";C$
70 PRINT C$
80 RANDOMIZE USR 32000
```

Note that a 'LDIR' or 'LDDR' can be used to duplicate 'thirds' of the display but that the physical arrangement prevents these instructions being used for other types of copying that involve whole character areas.

The following routine demonstrates the 'CPIR' instruction. Program 22 that uses the routine gives the address of the first location in the 16K ROM of the SPECTRUM that holds a specific byte. When the program is run it can be seen that all the values from 0—255 decimal are to be found in the ROM but that the value decimal 154 has its first occurrence at location 11,728 and is therefore a rarely used value in this particular ROM.

The routine is:

address	machine code	mnemonic	comment
7D00	3E 00	LD A,+xx	The matching value.
7D02	01 FF 3F	LD BC,+3FFF	The top location of the ROM.
7D05	21 00 00	LD HL,+0000	The first location of the ROM.
7D08	ED B1	CPIR	Search the ROM.
7D0A	44	LD B,H	Move the high address byte.
7D0B	4D	LD C,L	Move the low address byte.
7D0C	0B	DEC BC	Point to the matching
7D0D	C9	RET	location and return.

Program 22 that uses this routine is:

Program 22. 'CPIR'

Lines 1Ø — 4Ø. The **Hex loader** as given above.

```
5Ø DATA "3E","ØØ"
51 DATA "Ø1","FF","3F"
52 DATA "21","ØØ","ØØ"
53 DATA "ED","B1","44","4D"
54 DATA "ØB","C9","**"
6Ø FOR F = Ø TO 255
7Ø POKE 32ØØ1,F
8Ø PRINT F;TAB 4;"occurs first
   at loc.";CHR$ 32;USR 32ØØØ
9Ø NEXT F
```

The reader might like to try rewriting these programs using the non-automatic instructions.

There are no demonstration programs given here for the 'Input and Output instructions', 'the Interrupt instructions' or the small group of miscellaneous instructions, but once again the reader is encouraged to try writing appropriate programs.

7. UNDERSTANDING — An outline of the 16K monitor program

7.1 Why study the monitor program?

The SPECTRUM system is provided with a 16K ROM (read only memory) that provides:

i. An operating system.
ii. A BASIC interpreter.
iii. A character set of 96 characters.

The ROM holding the monitor program occupies locations decimal 0—16,383, hex. 0000—3FFF, and cannot be moved from this area in the standard SPECTRUM system. The machine code instruction at location zero is the first instruction to be executed when power is first applied to the system.

The monitor program of the SPECTRUM is well worth studying for the following reasons:

i. The subroutines in the monitor program are always available to be 'called' from user-written machine code programs. So doing will considerably shorten a machine code program.
ii. The monitor program shows how SINCLAIR RESEARCH have tackled certain problems and the tecnhiques used can be copied.
iii. The monitor program is a machine code program of some size and it is instructive to see how a large program can be structured.

The different parts of the monitor program will now be discussed in turn. First as they are used in a 'system view' and then as they occur in the monitor program itself.

7.2 A 'system view' of the monitor program.

The user of a SPECTRUM system is usually unaware that the microprocessor at the centre of the system follows the machine code program held in the 16K ROM from the moment that the power is first applied to the system. The only exception to this being when user-written machine code routines are being executed.

To the user the SPECTRUM appears as a machine that waits for the user to enter BASIC lines either in direct mode or with lines numbers so as to build up a BASIC program. Direct BASIC lines are executed immediately and may involve the interpretation of BASIC lines from a previously entered program. The SPECTRUM system is a little complicated as the BASIC interpreter is called to check the syntax of a BASIC line before it is stored in the program area — if it had a line number, or before it is interpreted properly — if it is a direct line.

The Operating System.

A study of the pathways through the monitor program starts, not unexpectedly, with the INITIALISATION routine that is entered when power is first applied, or a jump is made to location zero. (RANDOMIZE USR Ø)

INITIALISATION:

This routine occupies locations 0000–0007 & 11CB–12A1.

The main tasks of this routine are to check that the memory is available and to set the system variables to their required values. Further details will be given later in the chapter.

The INITIALISATION routine ends with the writing of the SINCLAIR copyright message on the bottom line of the display. Following this routine is the MAIN EXECUTION routine.

MAIN EXECUTION:

This routine occupies the locations 12A2–15AE.

In the SPECTRUM system and THE MAIN EXECUTION routine can be considered, as its name implies, to be the dominant routine in the monitor program.

It is this routine that calls, as necessary, the LIST command routine, the EDITOR routine and the SYNTAX CHECKER as BASIC lines are added to the program area. In the case of a direct BASIC line being entered then the LINE–RUN routine is called and that single line interpreted – this may involve the interpretation of the other BASIC lines. Then, on returning from the LINE–RUN routine the 'report' is produced. Reference is made to the table of error messages at locations 1391–1536 as required. A jump back to near the start of the MAIN EXECUTION routine leads to the EDITOR routine being called again but note that in this instance there is no call to the LIST command routine.

EDITOR:

This routine occupies the locations 0F2C–10A7.

The EDITOR routine allows the user to build up a BASIC line apparently at the bottom of the display. In reality the line is formed in the editing area and then copied, with tokens expanded, to the display area.

The user can enter either characters or cursor controllers and the appropr-ate subroutines are called as required.

Certain cursor controllers and ENTER lead to a return being made to the MAIN EXECUTION routine.

In the standard SPECTRUM system the only input to the EDITOR routine can be via the keyboard and the KEYBOARD–INPUT routine is called by the EDITOR routine. This call is vectored through the channel information area.

KEYBOARD–INPUT:

This routine occupies the locations 10A8–111C.

This routine gets the code of the last key to have been pressed by reading the system variable LAST–K.

Certain operations are performed within this routine, i.e. setting the CAPS LOCK & graphics flags and fetching the second byte of the colour control keys for the system variable K-DATA.

KEYBOARD:

The scanning of the keyboard is interrupt driven and occurs every 1/50 th. of a second. There are five separate subroutines involved with the main KEYBOARD routine occupying locations 02BF–030F.

The actual scanning of the keyboard is performed by the KEY–SCAN subroutine at locations 028E–02BE. This subroutine returns an appropriate key-value in the DE register pair which the other keyboard routines use to produce the required character code.

PRINT–OUTPUT:

In addition to the routines discussed above there are many other routines that also form part of the operating system of the SPECTRUM.

The PRINT–OUTPUT routine at location 09F4–0D4C is possibly the most important of these other routines.

This routine is in effect the routine called by using the machine code instruction 'RST 0010'. The address of location 09F4 being obtained from the channel information area.

The 'RST 0010' instruction will lead to the character, whose code is held in the A register, being printed either on the T.V. display or the printer. The flag that determines just where the output is to be made is repeatedly tested in the routine and the appropriate path taken. In the SPECTRUM system the PRINT–OUTPUT routine is very powerful as both character codes and control codes are handled by the same routine.

In the case of printing to the T.V. display the current print position has to be 'fetched' from the appropriate system variables, updated and then 'stored' once again. The print position indicates the line and column numbers of the character area being used, and also the corresponding address in the display file of the 'top-left pixel' of that character area. The PRINT–OUTPUT routie has subroutines that are called as necessary to transfer the sixty four bits of a character from the 'character set' to the requisite locations in the display file; another subroutine alters the attribute byte for that character area according to the values held in the appropriate system variables. Note that the ordinary characters are copied from the 'character set', user-defined characters from the UDG area but that 'graphics' are computed as needed.

A similar operation applies to the passing of character forms to the printer

buffer for subsequent output to the printer but obviously certain control codes are ignored.

In the SPECTRUM system every character that is printed on the T.V. display or the printer is handled by the PRINT–OUTPUT routine. The fact that the routine caters for many different conditions leads to the routine being fairly slow but nevertheless it is very useful.

The BASIC interpreter.

The interpreter part of the monitor program is called both for syntax checking and for line-execution. In the most part the same routines are used for both functions with the syntax/run flag being repeatedly tested to determine whether or not operations should be performed.

e.g. In considering the line – LET A=1

the syntax checker would check the syntax but the line-executer would actually give the value '1' to the variable 'A'.

In some ways therefore it can be convenient to consider the syntax checker as being totally separate in function from the line-executer whereas in reality it is not.

The different parts of the BASIC interpreter are:

The command tables:

These tables are to be found in locations 1A48–1B16.

In the SPECTRUM system there are fifty BASIC commands and the command tables contain the command class details, the required separator characters and the command routine addresses.

The controlling routine:

The part of the monitor program in locations 1B17–1C00 contain the controlling instructions that ensure the interpreter passes from one BASIC statement to another as required in the program.

The entry point for the SYNTAX CHECKER is location 1B17 and for the LINE–RUN operation location 1B8A.

The command class routines:

The part of the monitor program in locations 1C01–1CDD is principally concerned with the analysis of the parameters that follow BASIC commands.

e.g. The command NEXT is considered as being in class 4 as it is required to be followed by a single character variable.
The command POKE is considered as being in class 8 as it is required to be followed by two numeric expressions separated by a comma.

The command routines:

Most of the command routines are to be found in locations 1CDE–24FA.

The command routines concerned with input and output procedures are generally to be found in the operating system part of the monitor program.

There is a command routine for each of the fifty BASIC commands and it is the execution of these command routines that is the essence of BASIC interpretation.

The interpretation of a BASIC statement can be illustrated as follows:

Consider the statement — 1Ø CLS which is interpreted in a straightforward manner. First the command is considered. In the present case it is 'CLS,' a command that has no operand. The command routine for this command is found, by reference to the command tables, at location 0D6B. The command routine is then executed with the result that the display is cleared and the attribute bytes set as required. The controlling routine then procedes with the interpretation of the next statement.

Next the statement — 2Ø GO TO 5Ø
In this case the command is 'GO TO' and the command routine is at location 1E67. In this command routine the operand of the statement has to be identified and entered into the system variable NEWPPC so that the controlling routine considers the first statement of line 5Ø as the next statement for interpretation.

In this line — 2Ø GO TO 5Ø the characters between the BASIC command GO TO and the 'carriage return' character (or a colon) form an expression. In the present case the evaluation of the expression is simple as the value is obtained by evaluating the characters '5' and 'Ø' to give a result equal to decimal 50. However the BASIC interpretation of the SPECTRUM has a very sophisticated 'expression evaluator' and this will be discussed next.

The expression evaluator:
This routine occupies locations 24FB—28B1.

In the SPECTRUM system the result obtained from evaluating an expression can be either numeric or string. A numeric result will be returned by the expression evaluator as a five byte floating-point number at the top of the calculator stack. In the case of a string result the five bytes will represent a set of parameters that describe the string.

Expressions are evaluated from left to right with the different mathematical operations being given different precedent values. An operation with a higher precedence is performed before one with a similar or lower precedence. Certain operations, viz. FN, RND, PI, INKEY$, BIN,SCREEN$, ATTR & POINT, are performed actually within the expression evaluator but for all other operations the CALCULATOR is used.

When a BASIC variable is used in an expression the expression evaluator obtains the appropriate value for the variable by calling the appropriate variable identifying routine.

The variable handling routines:
This set of routines occupy the locations 28B2—2ACB.

The routines return the current value or parameters of a given variable in the variable area. In the case of an array variable the correct element or elements have to be identified before the correct values can be returned and in the case of a sliced string variable then the string parameters have to be restricted as required.

Miscellaneous arithmetic routines:
The part of the monitor program that occupies locations 2C88—2F9A contains a series of arithmetic routines. The most important of these are STACK—BC at 2D2B — that converts the current value held in the BC register pair to a floating-point number at the top of the calculator stack; and PRINT—FP at 2DE3 — that takes the top value off the calculator stack, converts it to its decimal form and prints it on the T.V. display or printer.

THE CALCULATOR:
This large and complicated routine occupies locations 2F9B—386D. It is normally called by using a 'RST 0028' instruction which acts as an indirect jump to location 335B. In essence the CALCULATOR consists of sixty six subroutines that each perform a different operation. The calling of these subroutines is not normally done using 'CALL' instructions but rather by using 'literals' with hex. values 00—41.

e.g. Literal '04' is equivalent to 'CALL 30CA' and leads to the top two values on the calculator stack being replaced with a single value that is their product. Therefore '04' is the literal controlling 'multiplication'.

and; literal '17' is equivalent to 'CALL 359C' and deals with the operation of string concatenation (i.e. A$+B$).

These 'literals' are included in a machine code routine as bytes of data (DEFB's — read as 'defined bytes') that follow the 'RST 0028' instruction. The final DEFB is always '38' which performs an 'end-calc' operation and thereby acts as a 'return' from the CALCULATOR.

The following example shows the CALCULATOR being used — interestingly in a recursive manner from within the CALCULATOR itself.

The 'tan' subroutine.

address	machine code	mnemonic	comment
37DA	EF	RST 0028	Call the calculator; a value is already on the calculator stack, i.e. x .
37DB	31	DEFB +31	'duplicate' — x,x
37DC	1F	DEFB +1F	'sin' — x, sin x

37DD	01	DEFB +01	'exchange' — sin x, x
37DE	20	DEFB +20	'cos' — sin x, cos x
37DF	05	DEFB +05	'division' — tan x
37E0	38	DEFB +38	'end-calc' — tan x
37E1	C9	RET	Now back to ordinary machine code and the value on the top of the calculator stack has gone from 'x' to 'tan x'.

The reason for the inclusion of this special system of using the instruction 'RST 0028' followed by 'literals' is that it shortens machine code routines. The example above uses only eight locations whereas the use of five 'CALL' instructions and a single 'JP' instruction would use eighteen locations.

It is quite possible to use the CALCULATOR in a user-written machine code program but care must be taken to ensure the calculator stack is kept balanced.

The character set

There is an 'unused area' from location 386E — 3CFF that precedes the 'character set' that occupies locations 3D00 — 3FFF.

The character set consists of ninety six character forms with each form using sixty four bits.

The following example shows one of the character forms.

address	**mnemonic**	**comment**
3D58	DEFB +00	'0 0 0 0 0 0 0 0'
3D59	DEFB +00	'0 0 0 0 0 0 0 0'
3D5A	DEFB +08	'0 0 0 0 1 0 0 0'
3D5B	DEFB +08	'0 0 0 0 1 0 0 0'
3D5C	DEFB +3E	'0 0 1 1 1 1 1 0'
3D5D	DEFB +08	'0 0 0 0 1 0 0 0'
3D5E	DEFB +08	'0 0 0 0 1 0 0 0'
3D5F	DEFB +00	'0 0 0 0 0 0 0 0'
		i.e. the character '+'.

A BASIC program that shows the binary forms is given next.

```
10 REM "Large character printe
   r"
20 FOR A=15616 TO 16376 STEP 8
30 FOR B=0 TO 7: LET F=PEEK (A
   +B): GO SUB 300: NEXT B
40 PRINT
50 PRINT TAB 2;"'";CHR$ (32+(A
```

```
    -15516)/8);""""
60 INPUT A$
70 CLS
80 NEXT A
90 STOP

300 REM Binary of F
310 FOR N=7 TO 1 STEP -1
320 LET P=2↑N
330 PRINT CHR$ (48+INT (F/P));
340 LET F=F-INT (F/P)*P
350 NEXT N
360 PRINT INT F
370 RETURN
```

Line 60 in the above program allows the user to step through the ninety six character forms one by one.

7.3 The different parts of the monitor program.

The monitor program will now be discussed section by section in the order in which they occur in the ROM.

0000 – 0007 'RST 0000' The very start. Disable the maskable interrupt, clear the A register, load the DE register pair with +FFFF – the top of RAM and jump forward to 11CB.

0008 – 000F 'RST 0008' The error routine. The machine stack will be cleared and the appropriate report given.

0010 – 0012 'RST 0010' the PRINT-A entry point. Jump forward to 15F2.

0018 – 0024 'RST 0018' & 'RST 0020'. Fetch the current character pointed to by CH-ADD, or the next one.

0028 – 0029 'RST 0028' Jump forward to 335B which is the starting address of the CALCULATOR.

0030 – 0037 'RST 0030' BC-SPACES. Jump forward to 169E to make space in the 'work space'.

0038 – 0052 The maskable interrupt routine. The real-time clock is updated and the keyboard scanned by a call to 02BF.

0066 – 0072 The non-maskable interrupt routine that will cause a system restart if location 5CB0 holds zero.

0095 – 0204 The token table. The following BASIC program shows this table.

```
1Ø REM Token table printer
2Ø FOR A=149 TO 516
3Ø LET B=PEEK A
4Ø IF B<128 THEN PRINT CHR$ B;
   : GO TO 6Ø
5Ø PRINT CHR$ (B-128)
6Ø NEXT A
```

0205 – 028D The key tables. There are six tables one for each of the possible modes. The most important table is the first (0205 – 022B) that holds the ASCII values for the capital letters and digits.

028E – 02BE The KEY–SCAN subroutine. A key-value is returned in the DE register pair. The zero flag is reset if too many keys are pressed at the same time. Normally the E register contains the key number – hex. 00–27, and the D register indicates which shift key is being pressed, if any.

02BF – 03B4 The KEYBOARD subroutines. A set of subroutines that handle the 'repeat' facility and decode the key-value to give the required character code. If a key is pressed and is accepted then its code is copied to the system variable LAST–K and bit 5 of FLAGS is set to indicate the presence of a new code.

03B5 – 03F7 The BEEPER subroutine. On entry the HL register pair is to hold the pitch of the required note and the DE register pair the duration. The lowest pitch is given by HL holding +FFFF and the value is roughly halved for each octave above this. The duration value is absolute and has to be increased as the pitch value is decreased if a note is to have the same overall duration.

03F8 – 046D The BEEP command routine. This routine makes extensive use of the CALCULATOR to change the 'duration' and 'pitch' into the appropriate values for the DE & HL register pairs. There is a table of twelve floating-point numbers at 046E – 04A8 for deriving the correct semi-tone values.

04AA – 04C1 In error these locations contain a subroutine applicable to the ZX81.

04C2 – 09F3 The SAVE, LOAD, VERIFY & MERGE command routines. The important parts of this section of the monitor program are:

04C2 – 053E The SAVE–BYTES subroutine. Pass 'DE' bytes, starting at the location '(IX)' to the cassette recorder together with the initial marker byte and the trailing parity byte.

053F – 0555 The SAVE/LOAD end subroutine.

0556 – 0604 The LOAD–BYTES subroutine. LOAD or match 'DE' bytes from the cassette player. Again the IX register pair points to the first loca-

tion of the destination area. The carry flag is reset for verifying but set for loading and merging.

The SAVE–BYTES & LOAD–BYTES subroutines are used for both the header part and the data part of the tape format.

0605 – 075F The entry point for all the subroutines is at 0605 and this part is concerned with the construction of the 'header' details in the 'work space' and is common to all four BASIC commands.

0760 – 096F This part deals with loading, verifying and merging. It calls LOAD–BYTES as required.

0970 – 09A0 This part deals with saving and is quite straightforward. The channel that allows printing in the lower screen is opened and the 'start tape' message is produced. There is then a 'wait for a key to be pressed'. After this the 'header' is sent out and then, after a delay of one second the 'data block' is sent out.

09A1 – 09F3 The cassette messages.

09F4 – 0D4C The PRINT–OUTPUT routine. Bit 1 of FLAGS is set when the printing is to appear on the printer but reset for the T.V. display.

The important parts of this section of the monitor program are:

09F4 – 0A0E Character codes that are 'printable' are distinguished from 'control' characters. In either case the current print position is fetched – see 0B03.

0A11 – 0A22 A look-up table for the 'control' characters i.e. characters hex. 06–17.

0A23 – 0AD8 The various routines for dealing with the 'control' characters.

0ADC – 0B02 The important STORE subroutine. The current print position is saved in the appropriate system variables. The print position may refer to the main part of the screen, the lower part of the screen or the printer buffer.

0B03 – 0B23 The equally important FETCH subroutine.

0B24 – 0BDA The PRINT–ANY character subroutine. This particular subroutine forms the essential part of the character printing routine of the SPECTRUM.

On entry the HL register pair holds the initial pixel address of where the character is to be printed – DF–CC or equivalent, the BC register pair holds the current line and column values – S–POSN or equivalent and the A register holds the character code.

The base address of the character form is then found whether it is in the character set, the UDG area or is a graphics character created in an Ad

Hoc manner in the calculator's memory area.

P–FLAG is also analysed to see if the character is to be printed in OVER and/or INVERSE modes.

Then, in the loop at 0BB7 – 0BC4 the actual character form is copied into the memory – either the display area or the printer buffer as is required.

0BDB – 0C09 The attribute setting routine. After a character has been printed on the T.V. display the attribute byte for the character area has to be set. This involves fetching the former attribute value and the system variables ATTR–T, MASK–T & P–FLAG. All these values are then manipulated together and the resultant value stored as the new attribute value.

0C0A – 0C54 The message and token printing subroutines. The entry point for message printing is 0C0A and for token printing 0C10. In the case of message printing the DE register pair has to point to the first location of the table of messages (must hold a value greater than hex. 7F) and the A register holds the number of the message to be printed – starting at zero. All the characters of the message will be printed until an 'inverted' character is found.

In the case of token printing the instruction line at 0C10 loads the DE register pair with the base address of the token table – 0095.

0C55 – 0D4C The SCROLL? subroutine. Whenever a character is printed on the T.V. display the current print position to be used is tested to see if the display should be scrolled. If it is then this routine is called to print the prompt message and act accordingly upon the next keystroke.

0D4D – 0D6A The 'set temporary colours' subroutine. This is an important little subroutine that is called on many occasions.

If the main part of the display area is being used then the system variables ATTR–P & MASK–P that hold the permanent colours are copied to ATTR–T & MASK–T. The system variable P–FLAG has its odd bits – the permanent bits, copied to its even bits – the temporary bits.

However, if the lower part of the display area is being used then the system variable ATTR–T takes the value of BORDCR and MASK–T is given the value zero. All of the temporary bits of P–FLAG are reset.

0D6B – 0EAB The CLS command routine. The operation of clearing the screen in the SPECTRUM involves setting all the locations of the display area to hold zero and the locations of the attribute area to hold specified values. The command routine makes use of the CL–LINE subroutine (0E44–0E87) to clear the twenty four 'lines' of the display area. Scrolling the screen involves the use of the CL–SCROLL subroutine (0E00 – 0E43).

0EAC – 0F2B The printer routines.

0EAC – 0ECC The COPY command routine. The dec. 176 pixel-lines are passed directly to the printer.

0ECD – 0EF3 The COPY–BUFF subroutine. The printer buffer's contents are passed to the printer.

0EF4 – 0F2B The PRINTER subroutine itself.

0F2C – 10A7 The EDITOR. This routine allows the user to build up a BASIC line in the 'editing area'. Following each keystroke that represents a printable character or token a 'beep' is produced and the ADD–CHAR subroutine (0F81 – 0F91) adds the appropriate code to the edit-line.

The editing keys are dealt with in a separate manner. Locations 0FA0 – 0FA8 holds a look-up table for the codes hex. 07 – 0F, and the section from 0FA9 – 10A7 contains the various handling routines for these keys.

Note: The EDITOR is also called from the INPUT command routine and allows the user to build up an INPUT-line in the work space.

10A8 – 111C The KEYBOARD–INPUT subroutine. This subroutine collects the value from LAST–K as long as bit 5 of FLAGS shows it is a new keystroke. A return is made with carry set and zero reset if the code is 'printable'.

The setting of the CAPS LOCK flag is handled by the code in locations 10DD – 10E5. Bit 3 of FLAGS 2 is complemented every time this section is executed.

The setting of MODE by the use of the GRAPHICS key and shift, or the SYMBOL SHIFT key and shift, is handled in locations 10E9 – 10F3.

The section from 10FA – 111C is concerned with the setting of K–DATA when a digit key has been pressed.

111D – 11B6 The ED–COPY subroutine. The edit-line or INPUT-line is build up in the editing area or 'work space' and this subroutine is used whenever such a line is to be displayed on the T.V. screen.

11B7 – 11CA The NEW command routine. This command performs a system restart operation but leaves RAMTOP, P–RAMT, RASP, PIP & UDG unchanged. It continues into INITIALISATION.

11CB – 12A1 The INITIALISATION routine. On entry the value in the A register will be zero for a full system restart but +FF for a NEW operation.

The parts of the INITIALISATION procedure will now be discussed.

11CC – 11CF The border of the T.V. display is made white.

11D0 – 11D9 The I register is set to hold +3F. This register is used in the generation of the T.V. scan signals.

11DA – 11EF The RAM–CHECK routine. All the locations from RAMTOP down to location 4000 are tested. On leaving the routine the HL register pair holds the address of the last location of the memory available.

11F0 – 11FF In the case of a NEW operation being performed this section is used to restore the former values of P–RAMT, RASP, PIP & UDG. When a system restart is being handled then this section is meaningless.

1200 – 1218 This part is used only when a system restart is being followed. The user-defined graphics are set to A–U by copying their forms from the character set to the UDG area. PIP, RASP and P–RAMT are also initialised.

1219 – 1234 The system variable CHARS is initialised to +3C00 and The machine stack organised. Interrupt mode 1 is selected, the IY register pair set to 5C3A and the maskable interrupt enabled. From this point on the keyboard will be scanned every 1/50th. of a second.

1235 – 1243 The initial channel information is copied to the channel information area.

1244 – 127B A series of system variables are given initial values. For example the permanent colour variables are set to give 'black ink', 'white paper' and 'white border'.

127C – 1285 The initial stream data is copied to the first fourteen locations of STRMS. This represents streams '–3' to '+3'.

1286 – 12A1 The printer buffer is cleared, the screen is cleared and finally the copyright message printed on the bottom line of the display.

12A2 – 15AE The MAIN EXECUTION routine. The various parts of this important routine will now be discussed.

12A2 – 12E1 The main loop for building up and subsequently listing a BASIC program made up of lines starting with line numbers. The syntax of each line is checked and only if the syntax is correct will the line be copied to the program area.

12E2 – 1302 A direct BASIC line, that has passed the syntax checking tests, will be interpreted. A 'return' to 1303 is made upon completion of the interpretation no matter what the reason.

1303 – 1390 The appropriate report is produced and the main execution loop entered again by jumping to 12AC.

1391 – 1554 The table of error messages. The SINCLAIR copyright message is also in this table.

1355 – 15AE This final part of the MAIN EXECUTION routine is used to copy a BASIC line from the 'editing area' to its appropriate place in the program area. Any existing copy of a line with the same line number as the new line is reclaimed.

15AF – 15C5 The initial channel data table.

15C6 – 15D2 The initial stream data table.

15D4 – 1651 The channel accessing routines. Entry to this set of routines at 15D4 has the effect of 'waiting for a key to be pressed', i.e. a jump is made repeatedly to the KEYBOARD–INPUT subroutine until the carry flag is set. Entry at 15EF or 15F2 has the effect of 'printing a character'. The output channel is normally to PRINT–OUTPUT. The first entry point is used for printing digits whereas the second entry point is used for single characters or tokens.

1652 – 16E4 A collection of subroutines.

1652 – 1654 The ONE–SPACE subroutine. A single space is made in a BASIC line either in the 'editing area' or the 'work space' as required.

1655 – 1663 The MAKE–ROOM subroutine. The current value of BC shows how many spaces are to be made after the location currently addressed by the HL register pair.

1664 – 168E The POINTERS subroutine. All the pointers from VARS to STKEND are changed as required.

168F – 169D The collect line number subroutine. For a given adrress of the start of a BASIC line the line number is collected in the DE register pair.

169E – 16AF The RESERVE subroutine. The required number of locatons is made available in the 'work space'.

16B0 – 16D8 A set of 'clearing' routines. Entry at 16B0 clears the edit-line, the temporary 'work space' and the calculator stack. Entry at 16BF clears the temporary 'work space' and the calculator stack. Whereas entry at 16C5 clears the calculator stack only.

16DB – 16E5 The indexer subroutine used in several instances to 'index' into a table.

16E5 – 1792 The CLOSE and OPEN command routines. An open channel will have a value that is other than zero in the appropriate STRMS location for that channel.

e.g. PRINT PEEK 23584 is normally zero which means channel 5 is closed.
But;
OPEN #5,"K": PRINT PEEK 23584 will give '1' and the channel is now open.
CLOSE #5: PRINT PEEK 23584 will give zero.

The two command routines check that channels are being OPENed and CLOSEd in the correct way.

1793 – 1794 In the standard SPECTRUM the use of the commands

FORMAT, MOVE, ERASE & CAT lead to the production of the error message 'invalid stream'.

1795 – 1A47 The LISTing routines. Entry at 1795 is used by the MAIN EXECUTION routine to produce an automatic listing, entry at 1795 by LLIST and entry at 17F9 by LIST itself.

The various subroutines in this section will now be discussed.

1855 – 18B5 The 'print a BASIC line' subroutine. This subroutine is called repeatedly by the controlling routines so as to print each line of the BASIC program. Initially the line number has to be printed. Then the line cursor if it is required. The subroutine at 18C1 – 18E0 is called to print a flashing character as necessary for when a BASIC line in the editing area a flashing character as necessary. Finally all the characters and tokens of the line are printed.

196E – 197F The LINE–ADDR subroutine. This subroutine is used to find the starting address of a given BASIC line in the program area.

19B8 – 19D4 The NEXT–ONE subroutine. The next BASIC line, or variable, is found by using this subroutine.

19E5 – 19FA The 'reclaiming' subroutine. Any reclaiming that has to be done is performed using this subroutine. First the pointers from VARS to STKEND have to be adjusted then the data from the requisite location to the end of the calculator stack is moved down.

1A1B – 1A47 The 'number' printing subroutines that are used to print the line numbers of BASIC lines either in a listing or in a report.

1A48 – 1B16 The COMMAND TABLES. There are two tables. The first is an offset table that indexes into the 'parameter' table. The full 'parameter table' is given below.

The 'parameter table'

address	command	command classes & separators	command routine address
1A7A	LET	01 = 02	(2AFF)
1A7D	GO TO	06 00	1E67
1A81	IF	06 THEN 05	1CF0
1A86	GO SUB	06 00	1EED
1A8A	STOP	00	1CEE
1A8D	RETURN	00	1F23
1A90	FOR	04 = 06 TO 06 05	1D03
1A98	NEXT	04 00	1DAB
1A9C	PRINT	05	1FCD
1A9F	INPUT	05	2089

1AA2	DIM	05	2C02
1AA5	REM	05	1BB2
1AA8	NEW	00	11B7
1AAB	RUN	03	1EA1
1AAE	LIST	05	17F9
1AB1	POKE	08 00	1E80
1AB5	RANDOMIZE	03	1E4F
1AB8	CONTINUE	00	1E5F
1ABB	CLEAR	03	1EAC
1ABE	CLS	00	0D6B
1AC1	PLOT	09 00	22DC
1AC5	PAUSE	06 00	1F3A
1AC9	READ	05	1DED
1ACC	DATA	05	1E27
1ACF	RESTORE	03	1E42
1AD2	DRAW	09 05	2382
1AD6	COPY	00	0EAC
1AD9	LPRINT	05	1FC9
1ADC	LLIST	05	17F5
1ADF	SAVE	0B	(0605)
1AE0	LOAD	0B	(0605)
1AE1	VERIFY	0B	(0605)
1AE2	MERGE	0B	(0605)
1AE3	BEEP	08 00	03F8
1AE7	CIRCLE	09 05	2320
1AEB	INK	07	(1C96)
1AEC	PAPER	07	(1C96)
1AED	FLASH	07	(1C96)
1AEE	BRIGHT	07	(1C96)
1AEF	INVERSE	07	(1C96)
1AF0	OVER	07	(1C96)
1AF1	OUT	08 00	1E7A
1AF5	BORDER	06 00	2294
1AF9	DEF FN	05	1F60
1AFC	OPEN #	06 , 0A 00	1736
1B02	CLOSE #	06 00	16E5
1B06	FORMAT	0A 00	1793
1B0A	MOVE	0A , 0A 00	1793
1B10	ERASE	0A 00	1793
1B14	CAT	00	1793

Note: Several of the command routine addresses are not to be found in the table. These have been given in brackets.

1B17 – 1C00 The controlling routine of the BASIC interpreter. In the case of the edit-line being checked for syntax errors the routine is entered at location 1B17 and the following steps taken.

i. The syntax flag is reset – bit 7 of FLAGS.

ii. Any line number present is checked for legality – the subroutine 'E–LINE number' at 19FB is used.

iii. The system variable that counts the statements in a line – SUBPPC, is initialised to zero.

iv. The system variable ERR–NR is initialised to hex.FF.

Then the statement-loop at 1B28 – 1B3C is entered and the syntax for each statement checked in turn. An indirect return via 1BB3 – 1BB6 is made if the syntax is proved correct. If any one of the syntax tests is failed then the return will be made via the appropriate error routine. Note that the error number is entered into ERR–NR but is unavailable.

When a direct BASIC line is being interpreted the entry point is 1B8A. The syntax flag – bit 7 of flags, will always be set. The statements of the line are then dealt with in turn by the statement-loop. A striaghtforward 'return' is made if there are no further BASIC lines to be interpreted, However, in the event of the direct BASIC line containing commands such as RUN, GO TO, CONTINUE, or when appropriate RETURN or NEXT, there may be further lines of BASIC to be interpreted before the 'return' is made upon the end of the BASIC program being reached.

In all cases when the statement–loop is entered a statement is interpreted in the following manner.

i. The BASIC command for that statement is identified and the requisite address in the command table computed.

ii. The first command class routine as specified in the parameter table is then executed.

iii. Further command class routines are executed, or characters matched against specified 'separators', until the stage is reached when the command routine address is fetched and the routine executed.

iv. When the last statement in a line has been interpreted then the next line is considered.

1C01 – 1C0C The command class table. This is an offset table that is used to find the base address of the various command class routines.

1C0D – 1CDD The command class routines. The requirements specified by the different command classes can be summarised as follows:

Class 00 – No further operands.

Class 01 — LET. A variable is required.
Class 02 — LET. An expression, numeric or string, to be given.
Class 03 — A numerical expression may be given. Zero to be used in case of default.
Class 04 — A single character variable must follow.
Class 05 — A set of items may be given.
Class 06 — A numeric expression must follow.
Class 07 — Handles colour items.
Class 08 — Two numeric expressions, separated by a comma, must follow.
Class 09 — As for class 8 but colour items may precede the expressions.
Class 0A — A string expression must follow.
Class 0B — Handle cassette routines.

The command class routines are fairly complicated and will not be discussed further. However locations 1CAD — 1CBD which contains part of the command class 07 routine is of particular interest. These locations contain the code that copies the 'current temporary system variables' to the corresponding permanent ones and can be called whenever this needs to be done.

1CDE — 24FA The command routines.
The routines in this section of the monitor program are again fairly complicated and will not be discussed further in detail. In chapter 8, however, reference to the various 'printing' routines will be made as they can be called from user-written programs.

24FB — 28B1 The' expression evaluator. This is a most interesting routine in the SPECTRUM as the evaluation of functions that do not require arguments are performed within the expression evaluator itself and not in the expected manner of having separate subroutines. As a result of the use of this method a programmer who wishes to use these function routines has to resort to calling the expression evaluator from the 'val/vals' routine in the CALCULATOR. (see chapter 8 for details).

There are two fundamental points to be made concerning the expression evaluator. The first is that the purpose of the routine is to evaluate the 'next expression' and produce a single result. This result if numeric will be a five byte floating-point number, however if the result is a string then a five byte set of parameters will be produced. In either case the five bytes constitutes a 'last value' and is returned by the routine as the topmost value on the calculator stack. The calculator stack will always be increased be 'one value' when the expression evaluator is used.

The second point to be made is that the expression evaluator uses precedence values and an operation of a higher precedence will be performed before an operation with a lower precedence. Operations with identical precedent

values will be performed in the order that they occur. In the SPECTRUM system the precedent values for operations that are yet to be done are saved on the machine stack and 'part answers' on the calculator stack. Also saved on the machine stack are the 'literals' that determine which CALCULATOR routine is to be used for the different operations.

The parts of the expression evaluator are:

24FB – 24FE The initial precedence marker – zero is saved on the machine stack.

24FF – 2794 The main loop of the expression evaluator. A pass through the loop is made for every item in the expression.

2530 – 2534 The SYNTAZ–Z subroutine. When syntax is being tested then the zero flag will be set.

2535 – 257F The SCREEN$ routine.

2580 – 2595 The ATTR routine.

2596 – 25AE A look-up table for the functions that do not require arguments.

25F8 – 2626 The RND routine.

2627 – 2634 The PI routine.

2634 – 2667 The INKEY$ routine.

2756 – 2758 The CALCULATOR is used to perform a specified operation on one or two operands as required.

2795 – 17AF A reference table used to convert arithmetic character codes to calculator 'literals'.

27B0 – 27BC The main precedence value table. (Priority table)

27BD – 28B1 The FN routine.

28B2 – 2995 The LOOK–VARS subroutine. This subroutine is called whenever a search of the variable area is to be be made. For a specified variable the address of the current value will be returned if that variable has already been used, but if not then the appropriate flags will be set.

2996 – 2A51 The STK–VARS subroutine. A complicated subroutine that is used when searching for simple string variables and array variables. The parameters of the string or the array element required are returned on the calculator stack.

2A52 – 2AB0 The SLICING subroutine. Any string can be sliced and this subroutine is called whenever a 'slice' is specified.

2AB1 – 2ACB The STK–STORE subroutine. A very useful routine that passes the current string parameters, in the A,B,C,D & E registers, to the

calculator stack. The stack is thereby extended by 'one value'.

In a set of string parameters the BC register pair hold the length of the string, the HL register pair the starting address of the string and the A register is normally unused and set to zero. On occasions the A register holds '1'and indicates elements of an array are being handled.

2AFF – 2BF0 The LET command routine. This is the actual assignment routine for the LET and INPUT commands. In the case of a simple numeric variable either the old value is overwritten or a new variable is constructed at the end of the current variable area. In the case of a simple string variable any old value will be reclaimed and a new variable constructed, once again at the end of the variable area. Finally if an array variable is being handled then the old entry will always be overwritten.

2BF1 – 2C01 The STK–FETCH subroutine. A call to this subroutine will result in the topmost value on the calculator stack being loaded into the A, B, C, D & E registers. The calculator stack is thereby reduced by 'one value'. In the SPECTRUM system the STK–STORE and the STK–FETCH subrouties are usually used to handle sets of string parameters however there is no reason why five byte floating-point numbers should not be handled. (But use the entry point 2AB2 for STK–STORE otherwise the fifth byte will be lost.)

2C02 – 2C87 The DIM command routine. A straightforward routine that sets up an array as specified. If a given array already exists then the old array will be reclaimed before the new array is constructed at the end of the variable area.

2C88 – 2F9A Miscellaneous arithmetic routines. The most important of these routines are:

2D22 – 2D27 The STK–DIGIT subroutine. A valid ASCII digit code – hex. 30–39, will be passed to the calculator stack as a floating-point number.

2D28 – 2D2A The STACK–A subroutine. The value held in the A register is passed to the calculator stack.

2D2B – 2D3A The STACK–BC subroutine. The value currently in the BC register pair is passed to the calculator stack.

Interestingly it is this subroutine that is used as the 'exiting' routine from a USR function. Hence the current value in the BC register pair becomes the 'last value' of the expression – 'USR number', and is returned. Note especially the IY register pair is re-initialised to +5C3A and therefore will always be correct upon returning from a USR function call. In a user-written machine code program care must be taken if the IY register pair holds a modified value and the STACK–BC (or STACK–A &

used.

2DA2 – 2DC0 The FP–TO–BC subroutine. The topmost value on the calculator stack is compressed into the BC register pair.

2DD5 – 2DE2 The FP–TO–A subroutine. The topmost value is compressed into the A register.

2DE3 – 2F9A The PRINT–FP subroutine. A long and very complicated subroutine that 'fetches' the topmost value off the calculator stack (thereby reducing it by 'one value') and prints the required number in either full decimal or E format. This subroutine will print a number for any five byte value irrespective of whether or not the value is genuinely numeric. (String values often give very strange numbers indeed!)

In performing its work this subroutine makes use of an Ad Hoc print buffer in the ten locations of MEM–3 & MEM–4 (system variables 5CA1 – 5CAA).

2F9B – 386D The CALCULATOR subroutine. The various parts of this important subroutine will be discussed in turn.

2F9B – 300E A set of miscellaneous arithmetic subroutines.

300F – 3013 'Literal 03' – 'subtract'. The first of the four main arithmetic routines.

In the subtraction routine the operation is performed between the top two values on the calculator stack. The topmost value being subtracted **from** the value underneath. the result will always be a single value and is the topmost value on the calculator stack. The stack is thereby reduced in size by 'one value'.

Subtraction is considered as the 'addition of a negated subtrahend'. i.e. a– b is taken as a+ (–b).

3014 – 30A8 'Literal 0F' – 'addition'. The two values at the top of the calculator stack are added together and replaced by the single result.

30CA – 31AE 'Literal 04' – 'multiply'. The two values at the top of the calculator stack are multiplied together.

31AF – 2313 'Literal 05' – 'division'. The two values at the top of the calculator stack are divided – the top value into the value underneath.

32C5 – 32D6 The calculator's table of constants. Five values are to be found in this table. The values are stored in a compressed form. The values are:

Zero, one, a half, a half of PI & ten.

These constants are available for use and the 'literals A0 – A5' will make the required constant the top value on the calculator stack. The stack thereby increases by 'one value'.

32D7 – 335A The calculator's table of addresses. This table contains the addresses of the sixty six routines that are called by using the calculator's 'literals'. The information held in the table is shown below.

'literal'	address	name	'literal'	address	name
00	368F	jump-true	21	37DA	tan
01	343C	exchange	22	3833	asn
02	33A1	delete	23	3843	acs
03	300F	subtract	24	37E2	atn
04	30CA	multiply	25	3713	ln
05	31AF	division	26	36C4	exp
06	3851	to-power	27	36AF	int
07	351B	or	28	384A	sqr
08	3524	no-&-no	29	3492	sgn
09	353B	no-l-eq	2A	346A	abs
0A	353B	no-gr-eq	2B	34AC	peek
0B	353B	nos-neql	2C	34A5	in
0C	353B	no-gtr	2D	34B3	usr-no
0D	353B	no-less	2E	361F	strs
0E	353B	nos-eql	2F	35C9	chrs
0F	3014	addition	30	3501	not
10	352D	str-&-no	31	33C0	duplicate
11	353B	str-l-eq	32	36A0	n-mod-m
12	353B	str-gr-eq	33	3686	jump
13	353B	strs-neql	34	33C6	stk-data
14	353B	str-gtr	35	367A	dec-jr-nz
15	353B	str-less	36	3506	less-0
16	353B	str-gtr	37	34F9	greater-0
17	359C	strs-add	38	369B	end-calc
18	35DE	vals	39	3783	get-argt
19	34BC	usr-s	3A	3214	truncate
1A	3645	read-in	3B	33A2	fp-calc-2
1B	346E	negate	3C	2D4F	e-to-fp
1C	3669	code	3D	3297	restack
1D	35DE	val	86 etc	3449	series-06 etc
1E	3674	len	A0 etc	341B	constants
1F	37B5	sin	C0 etc	342D	st-mem-0 etc.
20	37AA	cos	E0 etc	340F	get-mem-0 etc.

335B – 33A1 The controlling routine of the calculator.

33A2 – 33A8 'Literal 3B – 'fp-calc-2'. This is an important subroutine as it is used by the expression evaluator to perform arithmetic operations. In order to use this subroutine the 'literal' of the arithmetic operation

must be held in the B register before the 'RST 0028' instruction is used.

35DE – 361E 'Literal ID-val'. In general the subroutines in the calculator are rather complicated and beyond the scope of this book. However this particular subroutine is worth considering in some detail.
detail.

The steps are:

- The parameters of the string are fetched off the calculator stack.
- Sufficient room is made in the 'work space' for the string; and a carriage return character.
- The string is copied into the 'work space' and a carriage return character is added.
- Bit 7 of FLAGS is reset (the syntax flag) and the expression evaluator called.
- Then, as long as the expression was numeric the syntax flag will be set (execution) and the expression evaluator called a second time.
- The 'last value' on the calculator stack is then the required result.

36AF – 386D The 'function' subroutines. In the SPECTRUM system Chebyshev polynomials are used as required to evaluate the functions EXP, LN, SIN & ATN.

The tables of constants for these subroutines are to be found at:

36D6 – 36F6	–	EXP	– 8 constants.
3752 – 377E	–	LN	– 12 constants.
37BF – 37D6	–	SIN	– 6 constants.
3803 – 382E	–	ATN	– 12 constants.

These constants are stored in the compressed form and can be expanded by using the 'stk–data' subroutine. (Load the DE register with a destination address, the H'L' register pair with the base address of the constant and use 'CALL 33C6').

The compressed form will now be explained:

i. The first byte is divided by hex. 40 and the exponent will be—
   - when there is a remainder.
     The remainder + hex. 50
   - when there is no remainder.
     The second byte + hex. 50

ii. The quotient, which will be 0, 1, 2 or 3, shows how many further bytes are being specified. In all cases the number of bytes is the quotient + '1'.

As the mantissa is to have four bytes any unspecified bytes are set to zero.

The following examples (taken from the table of constants) illustrate the method used in the SPECTRUM.

Zero — compressed form 00 B0 00.
— The byte '00' is divided by '40'. There is no remainder so the exponent is 'B0+50' which is '00' and the mantissa is the third byte — '00' and three unspecified bytes.
i.e. 00 00 00 00 00

One — compressed form 40 B0 00 01.
As above the exponent will be '00'. The quotient is '1' so two further bytes are specified and the result is:
00 00 01 00 00
the integral form for one.

A half	— compressed form	30 00.
	expanded form	80 00 00 00 00.
PI/2	— compressed form	F1 49 0F DA A2
	expanded form	81 49 0F DA A2
Ten	— compressed form	40 B0 00 0A
	expanded form	00 00 0A 00 00

Using the compressed form as outlined any five byte number can be passed to the calculator stack. The bytes of the compressed form are put after the 'literal 34' — stk-data.

i.e. To make the value ten the 'last value' on the calculator stack use:

```
RST  0028          : Use the calculator.
34                 : Use 'stk-data'.
40 B0 00 0A        : The compressed form.
38                 : Use 'end-calc'.
```

'Literal 3D — 'restack' is also worth considering at this point. This subroutine changes a 'last value' that is in its integral form to the full floating-point form.

i.e. For the value ten.
00 00 0A 00 00 will be changed to 84 20 00 00 00

The subroutine has no effect on values that are already in floating-point form.

386E — 3CFF The character set. (see page 141 for details)

8. UNDERSTANDING — Using the monitor program's subroutines.

8.1 Introduction

The aim of this chapter is to show that machine code routines can be written for the SPECTRUM in a relatively easy manner by making use of the large number of subroutines that are always available in the monitor program.

In many ways the technique is based on BASIC as the appropriate machine code routines are developed to replace simple BASIC statements. These routines can then be called with a 'USR number' function. A larger machine code routine can be formed by 'joining a series of small machine code routines together'.

The author does not wish to discuss 'structured programming' in this book but would like to make the point that a successful program, whether in BASIC or machine code, will normally be constructed from a series of well defined 'tasks'. Each task having certain 'entry conditions' and producing a 'result'.

8.2 Hex. input

In chapter 6 a 'Hex. loader' program was given so that the reader could prepare a machine code routine in a DATA list and subsequently execute it. However the 'Hex. loader' is not really suitable for larger routines and the following program is preferable.

```
10 LET D=32000: REM Hex input
20 DEF FN A(A$,B)=CODE A$(B)-4
   8-7* (CODE A$,B) >57)
30 DEF FN C(A$)=16*FN A (A$,1)+
   FN A(A$,2)
40 DEF FN G$(F)=CHR$ (F+48+7*(
   F>9))
50 DEF FN H$(E)=FN G$(INT (E/1
   6))+FN G$(E-16*INT (E/16))
60 DIM A$(2)
70 PRINT FN H$(PEEK D);TAB 7;F
   N H$(INT (D/256));FN H$(D-256*IN
   T (D/256));
80 INPUT A$
90 LET L=1
100 IF A$(1)="U" THEN LET L=-1:
    GO TO 130
110 IF A$(1)="*" THEN GO TO 160
120 IF A$(1)<>CHR$ 32 THEN POKE
```

```
    D,FN C(A$)
130 PRINT TAB 16;FN H$(PEEK D)
140 LET D=D+L
150 GO TO 70
160 INPUT "Press any key to run
    the routine";A$
170 RANDOMIZE USR 32000
```

Notes: Use with CAPS LOCK SET.

- Enter a value as a pair of hexadecimal characters.
- Use ENTER by itself to step forward.
- Use "U" and ENTER to step backwards.
- Use "*" and ENTER to 'run the routine'.
- Line 160 gives the user a second chance!

By using the above program the reader will be able to enter, and check, a machine code routine fairly easily. The reader is well advised to SAVE the program before using it the first time as 'crashing' the system will occur often.

The reader is welcome to change the above program to suit any particular wish; e.g. hexadecimal characters in 'lower case', etc.

8.3 The BEEP command

In BASIC the BEEP command has the form:

BEEP duration, pitch

where the 'duration' must be a positive value not exceeding ten and the 'pitch' a positive or megative number denoting how far the note is from middle C.

In a machine code routine there are two distinct ways in which a 'beep' can be produced. The first is to call the BEEPER subroutine with the appropriate values in the DE & HL register pairs; whilst the second way is to call the BEEP command routine with the values for the 'duration' and 'pitch' on the calculator stack.

Both these methods will now be illustrated.

Method i.

Using the 'Hex input' program given above enter:

address	machine code	mnemonic	comment
7D00	11 05 01	LD DE,+0105	A duration of '1 second'.
7D03	21 66 06	LD HL,+0666	A pitch of 'middle C'.
7D06	CD B5 03	CALL BEEPER	Turn on the beeper.
7D09	C9	RET	'Return to BASIC'.

When the above routine is executed it will produce the same effect as 'BEEP 1,0'.

The values of DE & HL are found as follows:

- Consider a note of a given frequency 'f'.
  e.g. Middle C taken as 261.63 Hz. in the SPECTRUM.
- Then the duration required for a period 't' is simply 'f*t'. This goes in the DE register pair.
- The value for the HL register pair is given by:

  437,500/f – 30.125

Note that there is no 10 second limitation on the duration when this method is used.

Method ii.
Use the 'Hex input' program given above to enter:

address	machine code	mnemonic	comment
7D00	3E 01	LD A,+01	Duration of '1' second.
7D02	CD 28 2D	CALL STACK–A	Pass to calculator stack.
7D05	3E 00	LD A,+00	Pitch 'Ø'.
7D07	CD 28 2D	CALL STACK–A	Pass to calculator stack.
7D0A	CD F8 03	CALL BEEP	Turn on the beeper.
7D0D	C9	RET	'Return to BASIC'.

Again when this routine is executed the effect will be the same as 'BEEP 1,Ø'.

It is possible to use 'STACK–A' for integer values but 'stk-data' should be used for other values.

i.e. for 'BEEP 1.3,–1.12' it will be:

address	machine code	mnemonic	comment
7D00	EF	RST 0028	Use the CALCULATOR.
7D01	34	DEFB +34	'stk-data'.
7D02	F1	DEFB	Exponent '81'.
7D03	26 66 66 66	DEFBs	Mantissa. (= dec. 1.3)
7D07	34	DEFB +34	'stk-data'.
7D08	F1	DEFB	Exponent '81'.
7D09	8F 5C 28 F6	DEFBs	Mantissa. (= dec. –1.12)
7D0D	38	DEFB +38	'end-calc'.
7D0E	CD F8 03	CALL BEEP	Turn on the beeper.
7D11	C9	RET	'Return to BASIC'.

To produce a series of 'beeps' the values can be held in a table and collected as required.

8.4 SAVEing and LOADing

The SAVE–BYTES and LOAD–BYTES subroutines can be called from machine code in a straightforward manner.

In either case the IX register pair must hold the 'destination address' and

the DE register pair the 'byte count'.

e.g. to handle the bytes from 7E00 to 7EFF inclusively IX would hold 7E00 and DE would hold 0100.

The A register must be set to hold +FF to show that a block of data is being moved.

Finally, when LOADing the carry flag must be set.

The following routine will SAVE bytes.

address	machine code	mnemonic	comment
7D00	3E FF	LD A,+FF	Signify a data block.
7D02	DD 21 00 7E	LD IX,+START	Set the IX register pair.
7D06	11 00 01	LD DE,+COUNT	Set the DE register pair.
7D09	CD C2 04	CALL SAVE–BYTES	Perform the SAVEing.
7D0C	C9	RET	'Return to BASIC'.

Note that the bytes are SAVEd without a 'header' and will only LOAD if the byte count is known.

The LOADing routine is:

address	machine code	mnemonic	comment
7D00	37	SCF	Set the carry flag. It would be reset for VERIFY.
7D01	3E FF	LD A,+FF	Signify a data block.
7D03	DD 21 00 7E	LD IX,+START	Set the IX register pair.
7D07	11 00 01	LD DE,+ COUNT	Set the DE register pair.
7D0A	CD 56 05	CALL LOAD–BYTES	Perform the LOADing.
7D0D	C9	RET	'Return to BASIC'.

8.5 The 'colour items'

In the SPECTRUM system all of the attribute bytes have the following form:

Bit 7 – set for FLASH.
Bit 6 – set for BRIGHT.
Bits 5–3 – PAPER colour.
Bits 2–0 – INK colour.

This form applies to the 768 bytes in the attribute area and the system variables BORDER, ATTR–P, ATTR–T, MASK–P & MASK–T.

In addition to the attribute bytes the system variable P–FLAG is used to hold the permanent and temporary flags for PAPER 9, INK 9, INVERSE & OVER. The even numbered bits of P–FLAG – bits 6, 4, 2 & 0, are the temporary flags and the odd numbered bits – 7, 5, 3 & 1, are the permanent flags.

In most instances the SPECTRUM uses the temporary values when deciding the value for a location in the attribute area but on certain occasions, most importantly when following the CLS routines, the permanent values are used.

The BORDER command and the six 'colour items' will now be considered in turn.

BORDER:

At any time the colour of the border can be changed by use of an 'OUT (+FE),A' instruction. But this operation is normally coupled with the saving of the new colour value in bits 5–3 of BORDCR. Note that the other bits of BORDCR control the FLASHing, BRIGHTness and INK colour of the lower part of the display.

The following routine shows how the border colour can be changed and BORDCR set to store this new value.

```
        BORD–2  equ. 7D0D
address machine code  mnemonic          comment
7D00    3E 02         LD   A,'RED'      A red border.
7D02    D3 FE         OUT  (+FE),A      Effect the change.
7D04    07            RLCA              The colour of the
7D05    07            RLCA              border is moved
7D06    07            RLCA              to bits 5–3 of the A register.
7D07    CB 6F         BIT 5,A           The INK colour for the lower
7D09    20 02         JR   NZ,BORD–2    screen is to contrast.
7D0B    EE 07         XOR +07           White INK.
7D0D    32 48 5C      LD   (BORDCR),A   Set BORDCR.
7D10    C9            RET               'Return to BASIC'.
```

Or more simply:

```
7D00    3E 02         LD   A,'RED'      A red border.
7D02    CD 9B 22      CALL BORD–1       Set the border as directed.
7D05    C9            RET               'Return to BASIC'.
```

It is interesting to see how in the above routine the INK colour is made to contrast with the border colour.

PAPER:

The permanent PAPER colour is given by bits 5–3 of ATTR–P. The following routine shows for PAPER Ø – PAPER 7 just one way in which these bits can be changed without altering the other attribute values.

```
address machine code  mnemonic          comment
7D00    3A 8D 5C      LD   A,(ATTR–P)   Fetch ATTR–P.
7D03    0F            RRCA              Move bits 5–3
7D04    0F            RRCA              to bits 2–0.
7D05    0F            RRCA
7D06    E6 F8         AND,+F8           Discard the old colour.
7D08    C6 02         ADD A,+RED        Enter the new colour.
7D0A    07            RLCA              Rotate the
```

7D0B	07	RLCA	byte three times
7D0C	07	RLCA	to the left.
7D0D	32 8D 5C	LD (ATTR–P), A	Restore ATTR–P.
7D10	C9	RET	'Return to BASIC'.

The above method is not very efficient but it is a 'general method' that can be used in other situations.

PAPER 8 is handled separately and involves the setting of bits 5–3 of MASK–P. Whilst PAPER 9 involves the setting of bit 7 of P–FLAG.

INK:

The permanent INK colour is given by bits 2–0 of ATTR–P. The following routine shows how these bits can be changed for INK Ø – INK 7.

address	**machine code**	**mnemonic**	**comment**
7D00	3E F8	LD A,+F8	Prepare the mask.
7D02	FD A6 53	AND (ATTR–P)	Fetch bits 7–3 of ATTR–P.
7D05	C6 02	ADD A,+RED	The INK is red.
7D07	32 8D 5C	LD (ATTR–P),A	Restore ATTR–P.
7D0A	C9	RET	'Return to BASIC'.

Note how ATTR–P is on one occasion considered as being in location 'IY+53' and then in location '5C8D'.

INK 8 involves the setting of bits 2–0 of MASK–P. Whilst INK 9 involves the setting of bit 5 of P–FLAG.

FLASH:

The three states of FLASH can be programmed as follows:

FLASH Ø	– reset bit 7 of ATTR–P	– 'RES 7,(ATTR–P)'
FLASH 1	– set bit 7 of ATTR–P	– 'SET 7,(ATTR–P)'
FLASH 8	– set bit 7 of MASK–P	– 'SET 7,(MASK–P)'

BRIGHT:

Similarly the three states of BRIGHT:

BRIGHT Ø	– reset bit 6 of ATTR–P	– 'RES 6,(ATTR–P)'
BRIGHT 1	– set bit 6 of ATTR–P	– 'SET 6,(ATTR–P)'
BRIGHT 8	– set bit 6 of MASK–P	– 'SET 6,(MASK–P)'

In all cases when mode '8' requires cancelling the appropriate bits of MASK–P will have to be reset again.

OVER:

The two states of OVER can be specified by:

OVER 1	– set bit 1 of P–FLAG	– 'SET 1,(P–FLAG)'
OVER Ø	– reset bit 1 of P–FLAG	– 'RES 1,(P–FLAG)'

INVERSE:

The two states of INVERSE can be specified by:

INVERSE 1	— set bit 3 or P–FLAG	— 'SET 3,(P–FLAG)'
INVERSE Ø	— reset bit 3 of P–FLAG	— 'RES 3,(P–FLAG)'

Permanent v. Temporary:

As mentioned above most of the operations in the SPECTRUM use the temporary values; e.g. the 'RST 0010' operation. The CLS routine being the most important routine to use the permanent colours.

In order to copy the current permanent values to the temporary system variables use:

CALL TEMPS — 'CALL 0D4D'

and if the temporary values are to be used as the new permanent values use:

CALL PERMS — 'CALL 1CAD'

The reader might like at this stage to write machine code routines that are equivalent to:

PAPER 4: INK 3: BEEP 2,Ø

or; BEEP 1,Ø: BORDER 5: BEEP 2,12: BORDER 2

8.6 The CLS command and scrolling

One of the great advantages of using machine code instead of BASIC is that the user is not limited to using only the BASIC commands. In the SPECTRUM monitor program there are subroutines for clearing a part of the display area and for scrolling a part of it. These subroutines can only be used from a machine code routine.

CLS:

The complete operation of clearing the screen and setting all the attribute bytes to the 'permanent' values is obtained by using:

CALL CLS — 'CALL 0D6B'

but it is important to ensure that channel 'S' is open before the subroutine is called. The channel will need re-opening afterwards if further printing is to be done. The following routine does indeed clear the whole screen.

address	machine code	mnemonic	comment
7D00	3E 02	LD A,+02	Open channel 'S'.
7D02	CD 01 16	CALL CHAN–OPEN	
7D05	CD 6B 0D	CALL CLS	Now clear the screen.
7D08	C9	RET	'Return to BASIC'.

The above routine does work well and does have the advantage of using the permanent colours however it is probably simpler in many instances to use the CL–LINE subroutine.

CL–LINE subroutine:

This subroutine can be used to clear a specified number of lines of the screen display. The lines are counted from the bottom. Once again the permanent values are used when setting the attribute bytes.

Before a call is made to the routine the B register must hold a value in the range hex. 01-18; where hex.18 would mean the whole of the screen is to be cleared.

The following routine shows this subroutine being used:

address	machine code	mnemonic	comment
7D00	06 17	LD B,+17	Clear dec. 23 lines
7D02	CD 44 0E	CALL CL–LINE	leaving only the top one.
7D05	C9	RET	'Return to BASIC'.

CL–SCROLL:

This is an interesting subroutine as it enables the user to scroll a specified number of lines of the display. The subroutine ends by using CL–LINE to clear the bottom line and hence the attribute bytes for this line are given 'permanent' values.

The B register is again used to hold the value that specifies the number of lines to be handled but in this instance it is the 'actual number of lines –1'. i.e. the hex. range is 01–17. (A minimum number of two is required as the number of lines to be scrolled.)

The following routine shows this subroutine being used:

address	machine code	mnemonic	comment
7D00	3E 16	LD A,+16	Leave only the top line
7D02	CD 00 0E	CALL CL–SCROLL	unscrolled.
7D05	C9	RET	'Return to BASIC'.

Note that neither the CL–LINE subroutine nor the CL–SCROLL subroutine affect the current channel usage.

8.7 The printing subroutines

In order to use any of the printing subroutines to display characters the PRINT–OUTPUT subroutine must be made the current output routine. This can readily be achieved by opening channel 'S'. Therefore the instruction lines:

```
LD A,+02
CALL CHAN–OPEN
```

must be used before using any of the other subroutines.

'RST 0010':

In the SPECTRUM system all printing of characters to the screen is performed using this instruction. With channel 'S' open it has the effect of

using the PRINT–OUTPUT routine at 09F4 as the output routine.

The 'RST 0010' instruction is very powerful and can be used for the printing of any character, the changing of the current print position by the use of AT and TAB, the printing of expanded tokens and the temporary 'colour items'.

The following routine shows these uses:

address	machine code	mnemonic	comment
7D00	3E 02	LD A,+02	Open channel 'S'.
7D02	CD 01 16	CALL CHAN–OPEN	
7D05	06 18	LD B,+18	Clear the whole
7D07	CD 44 0E	CALL CL–LINE	display area.
7D0A	3E 16	LD A,+'AT'	
7D0C	D7	RST 0010	In effect
7D0D	3E 05	LD A,+05	PRINT AT 5,Ø;
7D0F	D7	RST 0010	
7D10	3E 00	LD A,+00	
7D12	D7	RST 0010	
7D13	3E 41	LD A,'A'	In effect
7D15	D7	RST 0010	PRINT "A";
7D16	3E 0D	LD A,'Cr'	In effect
7D18	D7	RST 0010	PRINT
7D19	3E F9	LD A,+F9	In effect
7D1B	D7	RST 0010	PRINT CHR$ 249;
7D1C	3E 0D	LD A,'Cr'	In effect
7D1E	D7	RST 0010	PRINT
7D1F	3E 11	LD A,'PAPER'	In effect
7D21	D7	RST 0010	PAPER 2;
7D22	3E 02	LD A,'RED'	(it is temporary)
7D24	D7	RST 0010	
7D25	3E 06	LD A,','	In effect
7D27	D7	RST 0010	PRINT ,
7D28	3E 42	LD A,'B'	In effect
7D2A	D7	RST 0010	PRINT "B";
7D2B	C9	RET	'Return to BASIC'

Note that in the above routine that a call to CL–LINE is made to clear the screen and that this does not set the print position to '0,0' (whereas CALL CLS would).

When using a 'RST 0010' instruction to 'print' a 'colour item' then two separate calls are required and when using a 'AT' or 'TAB' control character then three separate calls are required.

An alternative method of altering the print position is as follows;

- Load the BC register pair with the appropriate values for the new print position.
- CALL CL–SET & 'CALL 0DD9'; which enters the required values into S–POSN & DF–CC.

The values for the BC register pair for a position 'AT a,b;' are the B register holds hex. 18–a and the C register hex. 21–b.

e.g. 'PRINT AT 5,0;' requires:
'LD BC,+1321'
'CALL 0DD9'

Printing strings:

In the SPECTRUM strings of characters are always considered by having the DE register pair holding the address of the location containing the first character of the string and the BC pair holding a value equal to the number of characters in the string.

In the PRINT command routine the PR–STRING subroutine is used to print any string. The details of this subroutine are:

```
     PR–STRING    equ. 203C
PR–STRING         LD   A,B          : Fetch the 'high' count.
                  OR   C            : OR with the 'low' count.
                  DEC  BC           : Decrease the count.
                  RET  Z            : Return if count was zero.
                  LD   A,(DE)       : Fetch the character.
                  INC DE            : Move on a character.
                  RST  0010         : Print the character.
                  JR   PR–STRING    : Back for another?
```

Any string of characters can therefore be printed by:

- Load the start address into DE.
- Load the length into BC.
- CALL PR–STRING – 'CALL 203C'.

As the PR–STRING subroutine uses the 'RST 0010' instruction to actually print the characters a string may contain character codes, 'colour items', print position controllers and token codes.

As an example of the use of the PR–STRING subroutine try entering the following routine.

address	machine code	mnemonic	comment
7CF0	11 01 10 09		A string of
7CF4	16 0A 05 53		hex. OF length and
7CF8	70 65 63 74		starting at 7CF0
7CFC	72 75 6D		
....			
7D00	3E 02	LD A,+02	Open channel 'S'.
7D02	CD 01 16	CALL CHAN–OPEN	
7D05	06 18	LD B,+18	Clear the whole screen.
7D07	CD 44 0E	CALL CL–LINE	
7D0A	11 F0 7C	LD DE,+string	The start of the string.
7D0D	01 0F 00	LD BC,+length	The length of the string.
7D10	CD 3C 20	CALL PR–STRING	Now print the string.
7D13	C9	RET	'Return to BASIC'.

The above routine is similar in effect to:

CLS: PRINT PAPER 1; INK 9; AT 1Ø,5; "Spectrum"

Printing numbers:

The very powerful PRINT–FP subroutine is used to print the decimal form of any five byte floating-point number. This subroutine takes as its operand the topmost entry on the calculator stack. Note that when this subroutine is used the actual number is removed from the stack and lost.

The following routines uses the PRINT–FP subroutine to print the constant 'PI/2' from the calculator's table of constants.

address	machine code	mnemonic	comment
7D00	3E 02	LD A,+02	Open channel 'S'.
7D02	CD 01 16	CALL CHAN–OPEN	
7D05	CD 6B 0D	CALL CLS	Clear the whole screen.
7D08	3E 02	LD A,+02	Re-open channel 'S'.
7D0A	CD 01 16	CALL CHAN–OPEN	
7D0D	EF	RST 0028	Use the CALCULATOR.
7D0E	A3	DEFB +A3	Get the fourth constant.
7D0F	38	DEFB +38	'end-calc'.
7D10	CD E3 2D	CALL PRINT–FP	Print the topmost entry in decimal format.
7D13	C9	RET	'Return to BASIC'.

The use of the line:

RANDOMIZE USR 32ØØØ: PRINT 'PI/2 shows identical numbers. But it is interesting to consider the point that in obtaining 'PI/2', the constant also 'PI/2' was collected from the table of constants, doubled and then divided by two to finally give the result.

On many occasions however the use of the complete floating-point facility may not be required and it is indeed possible to print integers in the range dec. 0—9,999 by using the OUT—NUM subroutines that are normally used for producing the line numbers in a listing or report. When using OUT—NUM—1 (1A1B) the number has to be present in the BC register pair in the normal 'high-low' order. But when OUT—NUM—2 (1A28) is used the HL register pair is used to indirectly address the number that this time must be in 'low-high' order. Note that with this subroutine leading spaces are printed and this can be quite useful.

Care must be taken if these subroutines are used with numbers that might exceed the limit of dec. 9,999.

The following example shows the OUT—NUM—1 subroutine being used to print a number.

address	machine code	mnemonic	comment
7D00 — 7D0C	As given on page 169.		
7D0D	01 0F 27	LD BC,+270F	Decimal 9,999
7D10	CD 1B 1A	CALL OUT—NUM—1	Print the number.
7D13	C9	RET	'Return to BASIC'.

This completes the section on the printing subroutines and the reader is advised to try writing some longer machine code routines at this stage before considering the more special commands discussed in the next section.

8.8 PLOT, DRAW and CIRCLE

These three commands all deal with 'pixels' in the display area. The number of pixels that may readily be used is dec. 256*176 and represent the upper twenty two lines of the screen. The coordinates of the bottom lefthand pixel is taken as '0,0' and for the top righthand pixel they are '255,175'.

The system variables COORDS—x and COORDS—y hold the coordinates of the last pixel to have been used. These system variables are reset to '0,0' by the CLS command, or other commands such as RUN that do call the CLS command routine.

Note that in BASIC the three commands are normally used with 'FLASH 8; BRIGHT 8; PAPER 8'. i.e. MASK-T holding hex. F8.

PLOT:

This is by far the simplest of the three commands as it involves the identification of only a single bit in the display area and its subsequent setting or resetting.

There are three quite suitable entry points into the command routine but the third one is perhaps the easiest one to use.

i. CALL PLOT — 'CALL 22DC'. The entry requirements for entry to the PLOT command routine are to have the 'x' and the 'y' values on the top of the calculator stack. The 'x' value being underneath the 'y' value. The

use of STACK–A to place the values on the stack, in turn, is quite sensible.

ii. CALL PLOT–1 – 'CALL 22DF'. Entry at this point requires that the B register holds the 'y' value and the C register the 'x' value. After the pixel has been plotted the permanent colour values are copied automatically to the temporary system variables.

iii. CALL PLOT–BC – 'CALL 22E5'. Again the B register is to hold the 'y' value and the C register the 'x' value. This subroutine performs the actual PLOTting operation. The required bit is identified by calling PIXEL–ADD (22AA) and will then be set or reset according to the result of considering its present state and the value of P–FLAG. (Bit 0 of P–FLAG giving the present state of OVER and bit 2 the state of INVERSE.)

The following routine shows a single pixel being PLOTted.

address	machine code	mnemonic	comment
7D00 – 7D0C	As given on page 169.		
7D0D	01 64 32	LD BC,+3264	PLOT 100,50
7D10	CD E5 22	CALL PLOT–BC	
7D13	C9	RET	'Return to BASIC'.

DRAW:

The use of DRAW x,y is really an extension of the PLOT x,y command as the set of pixels that constitute a straight line is defined and manipulated.

DRAW x,y,a involves drawing an arc and is similar in many aspects to CIRCLE x,y,r.

– DRAW x,y

There are two suitable entry points for this command.

i. Enter at DRAW–1 (2477) with the values of 'x' & 'y' being the topmost entries on the calculator stack. The permanent colour values are copied to the temporary system variables after the line has been drawn.

ii Enter at DRAW–3 (24BA) with the B register holding 'ABS y', the C register holding 'ABS x', the D register holding 'SGN x' and the E register holding 'SGN y'.

The following routine shows a line being drawn. Note that the H'L' register pair has to be saved, and later restored, if a successful return to BASIC is to made.

address	machine code	mnemonic	comment
7D00 – 7D0C	As given on page 169.		
7D0D	D9	EXX	Save the value of
7D0E	E5	PUSH HL	H'L' on the machine stack.
7D0F	D9	EXX	

7D10	FD 36 43 64	LD (COORDS–x),+64	Set the last
7D14	FD 36 44 64	LD (COORDS–y),+64	coordinate values.
7D18	01 32 32	LD BC,+3232	This will give:
7D1B	11 01 01	LD DE,+0101	'DRAW 50,50'
7D1E	CD BA 24	CALL DRAW–3	Now draw the line.
7D21	D9	EXX	Restore the value
7D22	E1	POP HL	of H'L'.
7D23	D9	EXX	
7D24	C9	RET	

The above routine sets the 'last plot position' to '100,100' and then draws the line '+50,+50'.

– DRAW x,y,a

In order to draw an arc the three values 'x,y & a' have first to be placed on the calculator stack in either their long or their short five byte forms. Then a call can be made to DRAW–ARC (2394).

The following routine gives the same result as:

Draw 50,50,1 with the last plot position being '100,100'.

address	machine code	mnemonic	comment
7D00 – 7D0C	as given on page 169.		
7D0D	D9	EXX	Save the value of H'L'
7D0E	E5	PUSH HL	on the machine stack.
7D0F	D9	EXX	
7D10	FD 36 43 64	LD (COORDS–x),+64	Set the last
7D14	FD 36 44 64	LD (COORDS–y),+64	coordinate position.
7D18	EF	RST 0028	Use the CALCULATOR.
7D19	34	DEFB +34	'stk-data'.
7D1A	40 B0 00 32	DEFBs	(= +50 dec.)
7D1E	31	DEFB +31	'duplicate'.
7D1F	A1	DEFB +A1	'stk-one'.
7D20	38	DEFB +38	'end-calc'.
7D21	CD 94 23	CALL DRAW–ARC	Draw the actual arc.
7D24	D9	EXX	Restore the value
7D25	E1	POP HL	of H'L'.
7D26	D9	EXX	
7D27	C9	RET	'Return to BASIC'.

In the routine the values obtained for 'x,y & a' are dec. +50, +50 & +1.

CIRCLE:

Once again the three operands of the command have to be placed on the calculator stack before a call can be made to CIRCLE–1 (232D).

The following routine shows this being done. Note that there is no appreciable saving of time by drawing circles from machine code as it is the CIRCLE command routine that is slow.

address	machine code	mnemonic	comment
7D00 – 7D0C	As given on page 169.		
7D0D	D9	EXX	Save the value of H'L'
7D0E	E5	PUSH HL	on the machine stack.
7D0F	D9	EXX	
7D10	EF	RST 0028	Use the calculator.
7D11	34	DEFB +34	'stk-data'.
7D12	40 B0 00 64	DEFBs	(= 100 dec.)
7D16	31	DEFB +31	'duplicate'.
7D17	34	DEFB +34	'stk-data'.
7D18	40 B0 00 30	DEFBs	(= 48 dec).
7D1C	38	DEFB +38	'end-calc'.
7D1D	CD 2D 23	CALL CIRCLE–1	Draw the actual circle.
7D20	D9	EXX	Restore the value
7D21	E1	POP HL	of H'L'.
7D22	D9	EXX	
7D23	C9	RET	'Return to BASIC'.

The above routine produces the same result as:

'CIRCLE 100,100,48'

Note that the permanent colour values are copied to the temporary system variables whenever an arc or a circle is drawn using the above routines.

8.9 POINT, ATTR and SCREEN$

These three functions can be called from a machine code routine in a fairly easy manner as they each have a separate evaluation subroutine. They can also be called as functions by using 'VAL' but so doing does not appear to be worth considering.

POINT:

The coordinates of the pixel to be handled are entered into the BC register and the subroutine called as POINT–1 (22CE). As usual the B register holds the 'y' coordinate and the C register the 'x' coordinate.

The following routine shows this being done.

address	machine code	mnemonic	comment
7D00 – 7D0C	As given on page 169.		
7D0D	3E 41	LD A,'A'	Print an 'A'.
7D0F	D7	RST 0010	
7D10	3E 0D	LD A,'Cr'	Go on to the next
7D12	D7	RST 0010	line.

7D13	01 04 AE	LD BC,+AE04	This will be the same
7D16	CD CE 22	CALL POINT—1	as 'POINT (4,174)'.
7D19	CD E3 2D	CALL PRINT—FP	Show the result.
7D1C	C9	RET	'Return to BASIC'.

The above routine is the same as:

'PRINT "A";CHR$ 13;POINT (4,174)'

ATTR:

The 'line' number is entered into the C register and the 'column' number into the B register and the subroutine called at ATTR—1 (2583). The result of the test is returned as the top entry on the calculator stack.

The following routine shows this being done.

address	machine code	mnemonic	comment
7D00 — 7D0C	As given on page 169.		
7D0D	3E 11	LD A,'PAPER'	Make the paper colour
7D0F	D7	RST 0010	by 'cyan'. (Temporary)
7D10	3E 05	LD A,'cyan'	
7D12	D7	RST 0010	
7D13	3E FF	LD A,'COPY'	Print the token 'COPY'.
7D15	D7	RST 0010	
7D16	3E 0D	LD A,'cr'	Go on to the
7D18	D7	RST 0010	next line.
7D19	01 00 04	LD BC,+0400	This will be the same
7D1C	CD 83 25	CALL ATTR—1	as 'ATTR (Ø,4)'.
7D1F	CD E3 2D	CALL PRINT—FP	Show the result.
7D22	C9	RET	

The above routine is the same as:

'PRINT PAPER 5;CHR$ 255;CHR$ 13;ATTR (Ø,4)'

SCREEN$

Again the 'line' number is entered into the C register and the 'column' number into the B register. The subroutine can then be called by using CALL SCREEN$—1 (2538). The result is returned as the top entry on the calculator stack but note it is a set of string parameters.

The following routine shows this being done.

address	machine code	mnemonic	comment
7D00 — 7D0C	As given on page 169.		
7D0D	3E 7A	LD A,'z'	Print a character.
7D0F	D7	RST 0010	
7D10	3E 0D	LD A,'cr'	Go on to the next line.
7D12	D7	RST 0010	
7D13	01 00 00	LD BC,+0000	This will be the same as

```
7D16    CD 38 25    CALL SCREEN$-1 'SCREEN$ (0,0)'.
7D19    CD F1 2B    CALL FETCH     Collect the parameters.
7D1C    CD 3C 20    CALL PR-STRING Print the string.
7D1F    C9          RET            'Return to BASIC'.
```

The above routine is the same as:

'PRINT "z";CHR$ 13;SCREEN$ (0,0)

Remember that SCREEN$ will only search the character set and hence looks for codes in the range hex. 20–7F.

Note: See appendix iv; the 'SCREEN$' error. The double storing of the result does not occur when the entry point 'SCREEN$-1' is used.

8.10 PI, RND and INKEY$

These three functions are grouped together because their evaluation routines are included in the 'expression evaluator' and it is not possible to evaluate them in a direct manner. However by the use of the 'val/vals' routine in the CALCULATOR that calls the 'expression evaluator' it is feasible to consider evaluating:

```
      VAL CHR$ 167     for PI
      VAL CHR$ 165     for RND
and   VAL$ CHR$ 166    for INKEY$.
```

PI:

The actual evaluation procedure for PI is:

```
PI          equ. 262C
address     mnemonic        comment
262C        RST  0028       Use the CALCULATOR.
262D        A3              Stack 'PI/2'
262E        38              'end-calc'.
262F        INC  (HL)       Doubles 'PI/2' by increasing
.....                       the exponent.
```

This procedure takes only four locations and therefore is readily copied. It can be called as 'VAL CHR$ 167' if wished and the method is shown in the next routine.

```
address   machine code   mnemonic        comment
7D00 -
7D0C      As given on page 169.
-------------------------------------------------------
7D0D      3E A7          LD  A,'PI'      The code for PI.
7D0F      CD 28 2D       CALL STACK-A    It is stacked.
7D12      EF             RST 0028        Use the CALCULATOR.
7D13      2F             DEFB +2F        'chrs'.
7D14      38             DEFB +38        The string "PI" is now defined.
7D15      06 1D          LD  B,+1D       The literal for 'val'.
7D17      EF             RST 0028        Use the calculator.
7D18      3B             DEFB +3B        'fp-calc-2' the single
                                         operation routine.
```

```
7D19    38          DEFB +38          'end-calc'.
7D1A    CD E3 2D    CALL PRINT-FP     Show the result.
7D1D    C9          RET               'Return to BASIC'.
```

RND:

In the SPECTRUM the random numbers are obtained by:

i. Fetching the value of the system variable SEED.

ii. Changing the value according to the rules given below and restoring it in SEED.

iii. Dividing the value by 65,536.

When power is first applied to the SPECTRUM, or after a NEW, the value in SEED is zero. Then with the first call to the 'RND' routine the value of SEED becomes dec. 74.

This makes the first random number:

74 / 65,536 = .0011291504

The sequence for SEED is:

0, 74, 5624,28652,.....,0,74,.....

and all the numbers from zero to 65,536 are present.

The rules for changing the value of SEED are:

i. Add '1'.

ii. Multiply by '75'.

iii. Take the modulus 65,537. i.e. the remainder after dividing by 65,537.

iv. Subtract '1'.

The following BASIC program shows these changes.

```
10 INPUT "SEED?";CHR$ 32;SEED
20 PRINT "OLD VALUE =";CHR$ 32
    ;SEED
30 LET SEED=SEED+1
40 LET SEED=SEED*75
50 LET SEED=SEED-65537*INT (SE
    ED/65537)
60 LET SEED=SEED-1
70 PRINT "NEW VALUE =";CHR$ 32
    ;SEED
80 GO TO 30
```

In the monitor program the actual evaluation procedure for 'RND' is as follows:

```
RND       equ. 25FD
RND-END   equ. 2625
```

address	mnemonic	comment
25FD	LD BC,(SEED)	Fetch old value of SEED.
2601	CALL STACK–BC	Put it on the stack.
2604	RST 0028	Use the CALCULATOR.
2605	A1	Stack '1'.
2606	0F	SEED=SEED+1: 'addition'.
2607	34	Stack the
2608	37 16	value '75' dec.
260A	04	SEED=SEED*75: 'multiply'.
260B	34	Stack the
260C	80 41 00 00 80	value '65,537' dec.
2611	32	'n–mod–m'.
2612	02	'delete'. (The quotient)
2613	A1	Stack '1'.
2614	03	SEED=SEED–1: 'subtract'.
2615	31	'duplicate'.
2616	38	'end-calc'.
2617	CALL FP–TO–BC	'CALL 2DA2'.
261A	LD (SEED),BC	Enter new value of SEED.
261E	LD A,(HL)	Fetch the exponent.
261F	AND A	Test for zero.
2620	JR Z,RND–END	Jump if it is.
2622	SUB +10	Divide by 65,536.
2624	LD (HL),A	Restore exponent.
2625	.....	

The value for 'RND' is now the top value on the calculator stack.

The RND routine can be called from a machine code routine by considering the result of evaluating:

VAL CHR$ 165

The routine for evaluating PI given on page 175 can be used with the single alteration of:

7D0D 3E A5 : LD A,+A5

INKEY$:

There is a variety of ways in which the keyboard of the SPECTRUM can be read from machine code.

i. Indirectly by reading the system variable LAST–K.

ii. The keyboard can be scanned using 'IN' instructions in a manner identical, or similar, to that used in the KEY–SCAN subroutine at 028E.

iii. The subroutine KEY–SCAN can be called itself and the resultant key-values matched against known values.

iv. The actions taken in the INKEY$ routine at 2646 can be copied.

i.e.
```
CALL KEY-SCAN    : CALL 028E
LD C,+00         : Ensure 'K', 'L' or 'C' modes.
JR NZ,No-key     : Multiple keys were pressed.
CALL KEY-TEST    : Do not accept 'shift alone'. (031E)
JR NC,No-key     : or 'no keys'.
DEC D            : Ensure 'L' or 'C' modes.
LD E,A           : Move the key number.
CALL KEY-CODE    : CALL 0333:
```

and the character code is available in the A register.

v. The INKEY$ routine can be used by evaluating the expression 'VAL$ CHR$ 166'.

The following routine shows this last approach being used.

address	machine code	mnemonic	comment
7D00 – 7D0C	As on page 169.		
7D0D	11 00 08	LD DE,+0800	Read INKEY$ 2,048 times.
7D10	D5	PUSH DE	Save the counter.
7D11	3E A6	LD A,'INKEY$'	
7D13	CD 28 2D	CALL STACK-A	Stack the code.
7D16	EF	RST 0028	Use the CALCULATOR.
7D17	2F	DEFB +2F	'chrs'.
7D18	18	DEFB +18	'vals'. Note: 'vals' can be used whereas 'val' cannot be used this way.
7D19	38	DEFB +38	'end-calc'.
7D1A	CD F1 2B	CALL STK-FETCH	Fetch the parameters.
7D1D	CD 3C 20	CALL PR-STRING	No effect if null string.
7D20	CD BF 16	CALL X-TEMP	Clear the work space or it will encroach on this routine.
7D23	D1	POP DE	Fetch the counter.
7D24	7B	LD A,E	Go around the loop
7D25	B2	OR D	until the counter has
7D26	C8	RET Z	reached zero.
7D27	1B	DEC DE	Decrease the counter.
7D28	18 E6	JR LOOP	Back again.

Notes:

i. When this routine is run it has a similar effect to

```
FOR A=1 TO 2048: PRINT INKEY$;: NEXT A
```

ii. If wished locations 7D19–7D1F can be changed to:

```
1C, 38, CD, E3 2D, 00, 00
```

which will give:

```
FOR A=1 TO 2048: PRINT CODE INKEY$;: NEXT A
```

8.11 'BREAK'

On many occasions it is useful to allow an exit to be made from a machine code routine by scanning for the 'BREAK' key.

The following routine shows how this can be done.

```
        BREAK    equ. 7D0D
```

address	machine code	mnemonic	comment
7D00 – 7D0C	As given on page 169.		
7D0D	3E 7F	LD A,+7F	
7D0F	DB FE	IN A,(+FE)	Input address 7FFE.
7D11	1F	RRA	Bit 0 of A will be reset
7D12	D0	RET NC	if 'BREAK' pressed.
7D13	18 F8	JR BREAK	Back again.

In the routine the half row of the keyboard – BREAK to B – is repeatedly scanned. An exit will only occur when the bit that represents the 'BREAK' key becomes reset.

8.12 Conclusion

By this stage the reader should be confident about writing a machine code routine that involves the use of perhaps twenty to thirty instruction lines. Such a small routine can then be used within a BASIC program, being called by 'USR number'.

The whole operation can then be repeated for another 'task' and hopefully in the end a BASIC program will consist of only a single line:

```
10 RANDOMIZE USR .....
```

as no 'returns to BASIC' are made until the end of the machine code routine has been reached.

Appendix i.

Tables of Z80 machine code instructions

00 NOP	01 LD BC,+dddd	02 LD (BC),A	03 INC BC	04 INC B	05 DEC B	06 LD B,+dd	07 RLCA
10 DJNZ,e	11 LD DE,+dddd	12 LD (DE),A	13 INC DE	14 INC D	15 DEC D	16 LD D,+dd	17 RLA
20 JR NZ,e	21 LD HL,+dddd	22 LD (addr),HL	23 INC HL	24 INC H	25 DEC H	26 LD H,+dd	27 DAA
30 JR NC,e	31 LD SP,+dddd	32 LD (addr),A	33 INC SP	34 INC (HL)	35 DEC (HL)	36 LD (HL),+dd	37 SCF
40 LD B,B	41 LD B,C	42 LD B,D	43 LD B,E	44 LD B,H	45 LD B,L	46 LD B,(HL)	47 LD B,A
50 LD D,B	51 LD D,C	52 LD D,D	53 LD D,E	54 LD D,H	55 LD D,L	56 LD D,(HL)	57 LD D,A
60 LD H,B	61 LD H,C	62 LD H,D	63 LD H,E	64 LD H,H	65 LD H,L	66 LD H,(HL)	67 LD H,A
70 LD (HL),B	71 LD (HL),C	72 LD (HL),D	73 LD (HL),E	74 LD (HL),H	75 LD (HL),L	76 HALT	77 LD (HL),A

08 EX AF,A'F'	09 ADD HL,BC	0A LD A,(BC)	0B DEC BC	0C INC C	0D DEC C	0E LD C,+dd	0F RRCA
18 JR,e	19 ADD HL,DE	1A LD A,(DE)	1B DEC DE	1C INC E	1D DEC E	1E LD E,+dd	1F RRA
28 JR Z,e	29 ADD HL,HL	2A LD HL,(addr)	2B DEC HL	2C INC L	2D DEC L	2E LD L,+dd	2F CPL
38 JR C,e	39 ADD HL,SP	3A LD A,(addr)	3B DEC SP	3C INC A	3D DEC A	3E LD A,+dd	3F CCF
48 LD C,B	49 LD C,C	4A LD C,D	4B LD C,E	4C LD C,H	4D LD C,L	4E LD C,(HL)	4F LD C,A
58 LD E,B	59 LD E,C	5A LD E,D	5B LD E,E	5C LD E,H	5D LD E,L	5E LD E,(HL)	5F LD E,A
68 LD L,B	69 LD L,C	6A LD L,D	6B LD L,E	6C LD L,H	6D LD L,L	6E LD L,(HL)	6F LD L,A
78 LD A,B	79 LD A,C	7A LD A,D	7B LD A,E	7C LD A,H	7D LD A,L	7E LD A,(HL)	7F LD A,A

80 ADD A,B	81 ADD A,C	82 ADD A,D	83 ADD A,E	84 ADD A,H	85 ADD A,L	86 ADD A,(HL)	87 ADD A,A
90 SUB B	91 SUB C	92 SUB D	93 SUB E	94 SUB H	95 SUB L	96 SUB (HL)	97 SUB A
A0 AND B	A1 AND C	A2 AND D	A3 AND E	A4 AND H	A5 AND L	A6 AND (HL)	A7 AND A
B0 OR B	B1 OR C	B2 OR D	B3 OR E	B4 OR H	B5 OR L	B6 OR (HL)	B7 OR A
C0 RET NZ	C1 POP BC	C2 JP NZ,addr	C3 JP addr	C4 CALL NZ,addr	C5 PUSH BC	C6 ADD A,+dd	C7 RST 0000
D0 RET NC	D1 POP DE	D2 JP NC,addr	D3 OUT (+dd),A	D4 CALL NC,addr	D5 PUSH DE	D6 SUB +dd	D7 RST 0010
E0 RET PO	E1 POP HL	E2 JP PO,addr	E3 EX (SP),HL	E4 CALL PO,addr	E5 PUSH HL	E6 AND +dd	E7 RST 0020
F0 RET P	F1 POP AF	F2 JP P,addr	F3 DI	F4 CALL P,addr	F5 PUSH AF	F6 OR +dd	F7 RST 0030

88 ADC A,B	89 ADC A,C	8A ADC A,D	8B ADC A,E	8C ADC A,H	8D ADC A,L	8E ADC A,(HL)	8F ADC A,A
98 SBC A,B	99 SBC A,C	9A SBC A,D	9B SBC A,E	9C SBC A,H	9D SBC A,L	9E SBC A,(HL)	9F SBC A,A
A8 XOR B	A9 XOR C	AA XOR D	AB XOR E	AC XOR H	AD XOR L	AE XOR (HL)	AF XOR A
B8 CP B	B9 CP C	BA CP D	BB CP E	BC CP H	BD CP L	BE CP (HL)	BF CP A
C8 RET Z	C9 RET	CA JP Z,addr	CB see pages 100,103	CC CALL Z,addr	CD CALL addr	CE ADC A,+dd	CF RST 0008
D8 RET C	D9 EXX	DA JP C,addr	DB IN A,(+dd)	DC CALL C,addr	DD see page 185	DE SBC A,+dd	DF RST 0018
E8 RET PE	E9 JP (HL)	EA JP PE,addr	EB EX DE,HL	EC CALL PE,addr	ED see page 184	EE XOR +dd	EF RST 0028
F8 RET M	F9 LD SP,HL	FA JP M,addr	FB EI	FC CALL M,addr	FD see page 185	FE CP +dd	FF RST 0038

ED Instructions

ED 40 IN B,(C)	ED 50 IN D,(C)	ED 60 IN H,(C)		ED A0 LDI	ED B0 LDIR
ED 41 OUT (C),B	ED 51 OUT (C),D	ED 61 OUT (C),H		ED A1 CPI	ED B1 CPIR
ED 42 SBC HL,BC	ED 52 SBC HL,DE	ED 62 SBC HL,HL	ED 72 SBC HL,SP	ED A2 INI	ED B2 INIR
ED 43 (addr),BC	ED 53 (addr),DE	ED 63 (addr),HL	ED 73 (addr),SP	ED A3 OUTI	ED B3 OTIR
ED 44 NEG					
ED 45 RETN					
ED 46 IM 0	ED 56 IM 1				
ED 47 LD I,A	ED 57 LD A,I	ED 67 RRD			
ED 48 IN C,(C)	ED 58 IN E,(C)	ED 68 IN L,(C)	ED 78 IN A,(C)	ED A8 LDD	ED B8 LDDR
ED 49 OUT (C),C	ED 59 OUT (C),E	ED 69 OUT (C),L	ED 79 OUT (C),A	ED A9 CPD	ED B9 CPDR
ED 4A ADC HL,BC	ED 5A ADC HL,DE	ED 6A ADC HL,HL	ED 7A ADC HL,SP	ED AA IND	ED BA INDR
ED 4B LD BC,(addr)	ED 5B LD DE,(addr)	ED 6B LD HL,(addr)	ED 7B LD SP,(addr)	ED AB OUTD	ED BB OTRD
ED 4D RETI					
	ED 5E IM 2				
ED 4F LD R,A	ED 5F LD A,R	ED 6F RLD			

The Indexing Instructions

All the instructions using the IX register pair are prefixed 'DD' and those using the IY register pair are prefixed 'FD'.
In the following table read IY for IX and FD for DD if required.

DD 09		ADD	IX,BC	DD CB d 06		RLC	(IX+d)
DD 19		ADD	IX,DE	DD CB d 0E		RRC	(IX+d)
DD 21	+dddd	LD	IX,+dddd	DD CB d 16		RL	(IX+d)
DD 22	addr	LD	(addr),IX	DD CB D 1E		RR	(IX+d)
DD 23		INC	IX	DD CB d 26		SLA	(IX+d)
DD 29		ADD	IX,IX	DD CB d 2E		SRA	(IX+d)
DD 2A	addr	LD	IX,(addr)	DD CB d 3E		SRL	(IX+d)
DD 2B		DEC	IX	DD CB d 46		BIT	0,(IX+d)
DD 34	d	INC	(IX+d)	DD CB d 4E		BIT	1,(IX+d)
DD 35	d	DEC	(IX+d)	DD CB d 56		BIT	2,(IX+d)
DD 36	d +dd	LD	(IX+d),+dd	DD CB d 5E		BIT	3,(IX+d)
DD 39		ADD	IX,SP	DD CB d 66		BIT	4,(IX+d)
DD 46	d	LD	B,(IX+d)	DD CB d 6E		BIT	5,(IX+d)
DD 4E	d	LD	C,(IX+d)	DD CB d 76		BIT	6,(IX+d)
DD 56	d	LD	D,(IX+d)	DD CB d 7E		BIT	7,(IX+d)
DD 5E	d	LD	E,(IX+d)	DD CB d 86		RES	0,(IX+d)
DD 66	d	LD	H,(IX+d)	DD CB d 8E		RES	1,(IX+d)
DD 6E	d	LD	L,(IX+d)	DD CB d 96		RES	2,(IX+d)
DD 70	d	LD	(IX+d),B	DD CB d 9E		RES	3,(IX+d)
DD 71	d	LD	(IX+d),C	DD CB d A6		RES	4,(IX+d)
DD 72	d	LD	(IX+d),D	DD CB d AE		RES	5,(IX+d)
DD 73	d	LD	(IX+d),E	DD CB d B6		RES	6,(IX+d)
DD 74	d	LD	(IX+d),H	DD CB d BE		RES	7,(IX+d)
DD 75	d	LD	(IX+d),L	DD CB d C6		SET	0,(IX+d)
DD 77	d	LD	(IX+d),A	DD CB d CE		SET	1,(IX+d)
DD 7E	d	LD	A,(IX+d)	DD CB d D6		SET	2,(IX+d)
DD 86	d	ADD	A,(IX+d)	DD CB d DE		SET	3,(IX+d)
DD 8E	d	ADC	A,(IX+d)	DD CB d E6		SET	4,(IX+d)
DD 96	d	SUB	(IX+d)	DD CB d EE		SET	5,(IX+d)
DD 9E	d	SBC	A,(IX+d)	DD CB d F6		SET	6,(IX+d)
DD A6	d	AND	(IX+d)	DD CB d FE		SET	7,(IX+d)
DD AE	d	XOR	(IX+d)	DD E1		POP	IX
DD B6	d	OR	(IX+d)	DD E3		EX	(SP),IX
DD BE	d	CP	(IX+d)	DD E5		PUSH	IX
				DD E9		JP	(IX)
				DD F9		LD	SP,IX

Appendix ii

DECIMAL-HEXADECIMAL CONVERSION TABLE

DECIMAL 0–255 HEXADECIMAL 00–FF, Low byte

Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	2's C.	Dec.	Hex.	2's C.
0	00	64	40	128	80	−128	192	C0	−64
1	01	65	41	129	81	−127	193	C1	−63
2	02	66	42	130	82	−126	194	C2	−62
3	03	67	43	131	83	−125	195	C3	−61
4	04	68	44	132	84	−124	196	C4	−60
5	05	69	45	133	85	−123	197	C5	−59
6	06	70	46	134	86	−122	198	C6	−58
7	07	71	47	135	87	−121	199	C7	−57
8	08	72	48	136	88	−120	200	C8	−56
9	09	73	49	137	89	−119	201	C9	−55
10	0A	74	4A	138	8A	−118	202	CA	−54
11	0B	75	4B	139	8B	−117	203	CB	−53
12	0C	76	4C	140	8C	−116	204	CC	−52
13	0D	77	4D	141	8D	−115	205	CD	−51
14	0E	78	4E	142	8E	−114	206	CE	−50
15	0F	79	4F	143	8F	−113	207	CF	−49
16	10	80	50	144	90	−112	208	D0	−48
17	11	81	51	145	91	−111	209	D1	−47
18	12	82	52	146	92	−110	210	D2	−46
19	13	83	53	147	93	−109	211	D3	−45
20	14	84	54	148	94	−108	212	D4	−44
21	15	85	55	149	95	−107	213	D5	−43
22	16	86	56	150	96	−106	214	D6	−42
23	17	87	57	151	97	−105	215	D7	−41
24	18	88	58	152	98	−104	216	D8	−40
25	19	89	59	153	99	−103	217	D9	−39
26	1A	90	5A	154	9A	−102	218	DA	−38
27	1B	91	5B	155	9B	−101	219	DB	−37
28	1C	92	5C	156	9C	−100	220	DC	−36
29	1D	93	5D	157	9D	−99	221	DD	−35
30	1E	94	5E	158	9E	−98	222	DE	−34
31	1F	95	5F	159	9F	−97	223	DF	−33
32	20	96	60	160	A0	−96	224	E0	−32
33	21	97	61	161	A1	−95	225	E1	−31
34	22	98	62	162	A2	−94	226	E2	−30
35	23	99	63	163	A3	−93	227	E3	−29
36	24	100	64	164	A4	−92	228	E4	−28
37	25	101	65	165	A5	−91	229	E5	−27
38	26	102	66	166	A6	−90	230	E6	−26
39	27	103	67	167	A7	−89	231	E7	−25
40	28	104	68	168	A8	−88	232	E8	−24
41	29	105	69	169	A9	−87	233	E9	−23
42	2A	106	6A	170	AA	−86	234	EA	−22
43	2B	107	6B	171	AB	−85	235	EB	−21
44	2C	108	6C	172	AC	−84	236	EC	−20
45	2D	109	6D	173	AD	−83	237	ED	−19
46	2E	110	6E	174	AE	−82	238	EE	−18
47	2F	111	6F	175	AF	−81	239	EF	−17
48	30	112	70	176	B0	−80	240	F0	−16
49	31	113	71	177	B1	−79	241	F1	−15
50	32	114	72	178	B2	−78	242	F2	−14
51	33	115	73	179	B3	−77	243	F3	−13
52	34	116	74	180	B4	−76	244	F4	−12
53	35	117	75	181	B5	−75	245	F5	−11
54	36	118	76	182	B6	−74	246	F6	−10
55	37	119	77	183	B7	−73	247	F7	−9
56	38	120	78	184	B8	−72	248	F8	−8
57	39	121	79	185	B9	−71	249	F9	−7
58	3A	122	7A	186	BA	−70	250	FA	−6
59	3B	123	7B	187	BB	−69	251	FB	−5
60	3C	124	7C	188	BC	−68	252	FC	−4
61	3D	125	7D	189	BD	−67	253	FD	−3
62	3E	126	7E	190	BE	−66	254	FE	−2
63	3F	127	7F	191	BF	−65	255	FF	−1

DECIMAL-HEXADECIMAL CONVERSION TABLE

DECIMAL 0–65,280 HEXADECIMAL 00–FF, high byte

Decimal	Hex.	Decimal	Hex.	Decimal	Hex.	Decimal	Hex.
0	00	16,384	40	32,768	80	49,152	C0
256	01	16,640	41	33,024	81	49,408	C1
512	02	16,896	42	33,280	82	49,664	C2
768	03	17,152	43	33,536	83	49,920	C3
1,024	04	17,408	44	33,792	84	50,176	C4
1,280	05	17,664	45	34,048	85	50,432	C5
1,536	06	17,920	46	34,304	86	50,688	C6
1,792	07	18,176	47	34,560	87	50,944	C7
2,048	08	18,432	48	34,816	88	51,200	C8
2,304	09	18,688	49	35,072	89	51,456	C9
2,560	0A	18,944	4A	35,328	8A	51,712	CA
2,816	0B	19,200	4B	35,584	8B	51,968	CB
3,072	0C	19,456	4C	35,840	8C	52,224	CC
3,328	0D	19,712	4D	36,096	8D	52,480	CD
3,584	0E	19,968	4E	36,352	8E	52,736	CE
3,840	0F	20,224	4F	36,608	8F	52,992	CF
4,096	10	20,480	50	36,864	90	53,248	D0
4,352	11	20,736	51	37,120	91	53,504	D1
4,608	12	20,992	52	37,376	92	53,760	D2
4,864	13	21,248	53	37,632	93	54,016	D3
5,120	14	21,504	54	37,888	94	54,272	D4
5,376	15	21,760	55	38,144	95	54,528	D5
5,632	16	22,016	56	38,400	96	54,784	D6
5,888	17	22,272	57	38,656	97	55,040	D7
6,144	18	22,528	58	38,912	98	55,296	D8
6,400	19	22,784	59	39,168	99	55,552	D9
6,656	1A	23,040	5A	39,424	9A	55,808	DA
6,912	1B	23,296	5B	39,680	9B	56,064	DB
7,168	1C	23,552	5C	39,936	9C	56,320	DC
7,424	1D	23,808	5D	40,192	9D	56,576	DD
7,680	1E	24,064	5E	40,448	9E	56,832	DE
7,936	1F	24,320	5F	40,704	9F	57,088	DF
8,192	20	24,576	60	40,960	A0	57,344	E0
8,448	21	24,832	61	41,216	A1	57,600	E1
8,704	22	25,088	62	41,472	A2	57,856	E2
8,960	23	25,344	63	41,728	A3	58,112	E3
9,216	24	25,600	64	41,984	A4	58,368	E4
9,472	25	25,856	65	42,240	A5	58,624	E5
9,728	26	26,112	66	42,496	A6	58,880	E6
9,984	27	26,368	67	42,752	A7	59,136	E7
10,240	28	26,624	68	43,008	A8	59,392	E8
10,496	29	26,880	69	43,264	A9	59,648	E9
10,752	2A	27,136	6A	43,520	AA	59,904	EA
11,008	2B	27,392	6B	43,776	AB	60,160	EB
11,264	2C	27,648	6C	44,032	AC	60,416	EC
11,520	2D	27,904	6D	44,288	AD	60,672	ED
11,776	2E	28,160	6E	44,544	AE	60,928	EE
12,032	2F	28,416	6F	44,800	AF	61,184	EF
12,288	30	28,672	70	45,056	B0	61,440	F0
12,544	31	28,928	71	45,312	B1	61,696	F1
12,800	32	29,184	72	45,568	B2	61,952	F2
13,056	33	29,440	73	45,824	B3	62,208	F3
13,312	34	29,696	74	46,080	B4	62,464	F4
13,568	35	29,952	75	46,336	B5	62,720	F5
13,824	36	30,208	76	46,592	B6	62,976	F6
14,080	37	30,464	77	46,848	B7	63,232	F7
14,336	38	30,720	78	47,104	B8	63,488	F8
14,592	39	30,976	79	47,360	B9	63,744	F9
14,848	3A	31,232	7A	47,616	BA	64,000	FA
15,104	3B	31,488	7B	47,872	BB	64,256	FB
15,360	3C	31,744	7C	48,128	BC	64,512	FC
15,616	3D	32,000	7D	48,384	BD	64,768	FD
15,872	3E	32,256	7E	48,640	BE	65,024	FE
16,128	3F	32,512	7F	48,896	BF	65,280	FF

Appendix iii.

Currently available machine code handling programs.

At the present time the author has seen only three programs although in due course there will be many more produced.

i. SPDE (SPECTRUM disassembler and editor) — CAMPBELL SYSTEMS.
This is a magnificent program for the beginner,

The 'Hex input' program given in chapter 8 is a rudimentary editor program as compared to this most polished program.

SPDE allows the user to examine the contents of any of the locations of the SPECTRUM's memory. The contents are displayed in hexadecimal form and followed by the appropriate mnemonics. This forms the disassembler part of the program.

But SPDE also allows the user to enter machine code into memory by using either hexadecimal characters or ordinary characters. A user-written routine can then be executed.

SPDE is a pleasure to use. It has an excellent display format and allows for both forward and backward paging.

ii. SPECTRUM BUG — ARTIC COMPUTING.
This is a very complicated machine code handling program. Again it has a disassembler so that the mnemonics can be shown.

SPECTRUM BUG is really intended for the serious machine code programmer and therefore allows for the inclusion of 'break points' as well as the facility to examine the contents of the registers of the Z80.

SPECTRUM BUG is straightforward to use but it is not so 'user friendly' as SPDE.

iii. SPECTRUM MONITOR — PICTURESQUE.
This program is really very similar to SPECTRUM BUG and is perhaps a little easier to use. Again machine code programs can be disassembled (and printed if desired) and user written routines entered (with or without break points).

The display format of SPECTRUM MONITOR is a pleasant white on blue but even so it is not as good as the display of SPDE.

Future programs:
Francis Ainley will soon be producing a SPECTRUM version of his MACHINE CODE TEST TOOL and it should prove to be fairly successful.

Three firms, at least, will soon be producing assembler programs. The firms are: ARCTIC COMPUTING, BUG BYTE and PICTURESQUE, but it remains to be seen which product will be the most successful.

Appendix iv. SPECTRUM 'bugs'

The 16K monitor program is an excellent program but there are a few programming errors. The following list details twelve errors of which only the first two are really important.

i. The 'division' error (credit to Dr. Frank O'Hara)
Location hex. 3200 should contain hex. DA rather than hex. E1. This error in the 'division' routine leads to, for example;
0.5 having the floating-point form 7F 7F FF FF FF
but 1/2 " " " " " 80 00 00 00 00

ii. The '−65536' error. (credit to Dr. Ian Logan)
In the monitor program there is a failure to be consistent over this number. On some occasions it is taken as '00 FF 00 00 00' whilst on others it has its full floating-point form.
The best example of this error is given by;
PRINT INT −65536 which gives −1.

iii. The 'program name' subroutine.
The subroutine from 04AA–04C1 applies to the ZX81 and should have been deleted.

iv. The 'CHR$ 9' error.
In the PRINT–OUTPUT routine there is a section for handling 'CHR$ 9' – rightspace. However the programmer has failed to store the new print position so 'CHR$ 9' will only work if the next printing is at a newly defined place.
e.g. PRINT PAPER 2;CHR$ 9;AT 4,Ø;
does work but is not helpful.

v. The 'scroll?' error. (Also applies to 'start tape . .')
It is not possible to reply to a prompt message with CAPS LOCK, shift & GRAPHICS or shift & SYMBOL SHIFT without the previous edit-line being copied to the lower part of the screen. The error occurs in the KEYBOARD–INPUT routine that fails to recognise the 'prompt' situation.

vi. The 'current line cursor' error. (credit to Paul Harrison)
It is possible to get an edit-line containing a 'cursor'.
e.g. enter 1ØØ PRINT & ENTER
1Ø1 & ENTER
Shift & EDIT
A 'current line cursor' will appear in the edit-line because the 'edit-line' number +'1' equals the 'current line' number. The mistake is in the 'print a BASIC line subroutine'.

vii. The 'leading space' error.
There is an inconsistency over the printing of leading spaces before tokens.
e.g. PRINT CHR$ 255;CHR$ 13;CHR$ 255

and the leading space is present on the first occasion but suppressed on the second.

viii. The 'K-mode' error. (credit to Chris Thornton)
When the SPECTRUM is in K-mode a keyword will be printed if a suitable key is pressed. Unfortunately if the key is held down the keyword is repeated.

The mistake here is in the 'key repeat' subroutine that continues to supply the same code even though the mode has been changed to 'L'. The subroutine should test that bit 3 of flags has not been changed.

ix. The 'CHR$ 8' error. (credit to Dr. Frank O'Hara)
Location 0A33 should contain hex. 19 rather than hex. 18. 'Backspacing' works perfectly well whilst it is used in lines 1 to 21. It is however not possible to 'backspace' from the start of line 1 to the end of line 0 as the programmer has used the wrong limit. Indeed 'backspacing' from '0,0' leads to a variety of interesting results.

x. The 'SCREEN$' error. (credit Stephen Kelly and others.)
Location 257D should contain hex. C9 (RET) rather than hex. C3 (JP). As a result of the error the string obtained by using SCREEN$ is stored twice.

This can be shown by:

```
10 PRINT "123"
20 PRINT SCREEN$ (0,0)+SCREEN$ (0,1)
```

Which will give the string '22'.
This error can be avoided by the use of temporary string variables,
i.e.

```
20 LET S$=SCREEN$ (0,0)
30 LET T$=SCREEN$ (0,1)
40 PRINT S$+T$
```

xi. The 'STR$' error. (credit Tony Stratton)
When handling numbers in the range $-1 > n > 1$ (but not zero) the PRINT–FP routine puts an extra zero on the calculator stack thereby giving more 'results' than 'operations'.
Hence: PRINT "A"+STR$ 0.1 is evaluated as PRINT ""+STR$ 0.1
and PRINT 1+VAL STR$ 0.1 as PRINT 0+VAL STR$ 0.1 etc.
Again this error can be avoided by the use of temporary string variables when handling parameters of STR$ that are likely to give errors; or by placing STR$ before any binary operators.

xii. The 'CLOSE' error. (credit Martin Wren-Hilton)
Any attempt to CLOSE streams +04 to +0F without first opening the stream will lead to either i. a system restart – as a jump is made to location hex. 0000, or ii. the production of a strange report.

The reason for this error occurring is that the 'CLOSE stream look-up' table at hex. 1716 does not finish with an end marker as is customary at the end of such a table.

INDEX

Notes: BASIC commands or functions are given in capitals as usual, e.g. LIST.
System variables are labelled – '(SV)'.
Monitor program routines are labelled – '(ROM)'.
Z80 machine code instructions have – 'instructions'.

A
ABS 38 156
Absolute addressing 75 115
Absolute binary arithmetic 63
ACS 38 156
ADC instructions 79 118
ADD-CHAR (ROM) 146
ADD instructions 80 118
Address bus 11 12 48 47
Addressing modes 75
Alternate register set 54 75 114
AND 38
AND instructions 85 122
Arithmetic logic unit 56
ASN 38 156
Assembler 59
Assembly format 59
ATN 38 156
ATTR 38 174
Attribute area 14 15
ATTR-P (SV) 30 35 87 88 145 162
ATTR-T (SV) 146 162

B
BASIC interpreter 9 Ch.7
BASIC line format 17
BASIC program area 14 17 32
BC-SPACES (ROM) 142
BEEP 22 143 160
BEEPER (ROM) 144 160
BIN 39
BIT instructions 102 132
Block handling instructions 102 132
BORDCR (SV) 23 145 162 163
BORDER 22 163
BREAK key 179
BRIGHT 23 164

C
CALCULATOR (ROM) 140 154 161
Calculator stack 14 19 153 154
CALL instructions 97 128
Carry flag 80 90
CAPS LOCK key 137 146
CAT 23
CH-ADD (SV) 142
Channel information area 14 16 147
Channel usage 16 33 127 148 165 166
CHANS (SV) 14 16
Character set 9 135 137 141
Chebyshev polynomials 38 40 43 157
CHR$ 39 157 175
CIRCLE 23 170
CLEAR 19 24 38
CL-LINE (ROM) 145 165
CLOSE 24 148
CLS 24 145 165
CL-SET (ROM) 168
CL-SCROLL (ROM) 145 166
CODE 39 157 178
Command class routines (ROM) 138 151 152
Command class table (ROM) 151
Command routines (ROM) 138 152
Command table (ROM) 138 149
Compressed form of floating-point 157
CONTINUE 24 37
Control characters 45
COORDS (SV) 35 170
COPY 25 146
COS 39 157
Colour items 22 26 29 30 34 162
CP instructions 85 122

D
DATA 25 111
Data bus 11 12
DATADD (SV) 36
DEC instructions 83
DEF FN 25
DELETE 26
DF-CC (SV) 144 168
DIM 26 154
Display area – memory mapped 13 14
DJNZ instruction 95 126
DRAW 26 170

E
e 39
Editing area 14 18
EDITOR (ROM) 136 146
E-format 68
E-LINE (SV) 14 18
ERASE 26
ERR-NR (SV) 37 151
ERR-SP (SV) 37
EX instructions 74 75
EXP 39 156
Exponent 68
EXPRESSION EVALUATOR (ROM) 139 152

F
F register (flag) 54
FETCH (ROM) 144
FLAGS (SV) 144 146 151
FLASH 26 164
Floating-point representation 67
FN 39
FOR 27
FORMAT 28
FP-TO-A (ROM) 155
FP-TO-BC (ROM) 155
FRAMES (SV) 25

G
GO SUB 28 99
GO SUB stack 14 19 28 38
GO TO 28

H
Header 32
Hexadecimal coding 60
Hex input program 159
Hex loader program 110

I
I register (interrupt vector) 56 146
IF 29
IN 40
INC instructions 80 118
Indexed addressing 75 77 116
Indexing registers 55
Indirect addressing 75 115
INK 28 164
INKEY$ 40 175
IN instructions 106
INPUT 30
Instruction lines 58
Instruction register 51 50
INT 40 156
Integral representation 27 65
Interrupt instructions 107
INVERSE 30 165

J
JP instructions 89 124

K
K-DATA (SV) 137 146
KEYBOARD-INPUT (ROM) 136 146
Keyboard interrupt routine (ROM) 35 107
KEYBOARD routines (ROM) 137 143 177
KEY-SCAN (ROM) 137 143 177
Key tables (ROM) 143

L
LAST-K (SV) 137 146 177
LD instructions 72 80 111 118
LEN 40 156
LET 30 154
LINE-ADDR (ROM) 149
LINE-RUN (ROM) 136 138
LIST 31
Literals (calculator) 140 155 156

LLIST 32
LN 40 156
LOAD 32 143 161
LOOK-VARS (ROM) 153
Loudspeaker 8 34
LPRINT 32

M
Machine code instructions 56 Ch.5
Machine stack 14 19 56 96 127
MAIN EXECUTION (ROM) 136 147
Main registers (user registers) 51 52
MAKE-ROOM (ROM)
Mantissa 68
MASK-P (SV) 30 35 145 162
MASK-T (SV) 145 162
Memory area (calculator's) 145 155
Memory map 13 14
MERGE 32 143
Microdrive maps 14 16
Mnemonics 58
MODE (SV) 146
Monitor program 9 Ch.7
MOVE 33

N
NEW 33 146
NEWPPC (SV) 25 28
NEXT 33
NEXT-ONE (ROM) 149
Non maskable interrupt 142
NOP instruction 72 111
NOT 40
NSPPC (SV) 25 28

O
OLDPPC (SV) 25
One's complement arithmetic 64 109
ONE-SPACE (ROM) 148
OPEN 33 148
Operating system 9 Ch.7
OR 40
OR instructions 86 122
OSPCC (SV) 25
OUT 22 34
OUT instructions 106
OUT-NUM-1 (ROM) 170
OUT-NUM-2 (ROM) 170
OVER 34 164
Overflow/parity flag 94

P
PAL encoder 11
PAPER 34 163
Parameter table (ROM) 149
PAUSE 35
PEEK 41 156
Permanent colours 22 145 159
PERMS (ROM) 159
P-FLAG (SV) 30 34 35 145 162
PI 41 173
PIP (SV) 146
PLOT 35 170
POINT 41 173
POINTERS (ROM) 147
POKE 35
POP instructions 96 127
PPC (SV) 25
P-RAMT (SV) 14 20 146
Precedence table (priority) 153
PRINT 35 166
PRINT AT 35 46 167
Printer buffer 14 136 137
PRINT-FP (ROM) 140 155 169
Printing characters/tokens 166
Printing numbers 167
Printing strings 168
PRINT-OUTPUT (ROM) 136 137 166
PRINT TAB 36 46 167
PROG (SV) 14 17
Program counter 51 52 98
PR-STRING (ROM) 168
PUSH instructions 96 127

R
RAM-CHECK (ROM) 146
RAMTOP (SV) 14 20 24 146
Random access memory – RAM 8 9 13
RANDOMIZE 36
RASP (SV) 146
READ 36 110
Read only memory – (ROM) 8 9 13
Real time clock 25 108
RECLAIM (ROM) 149
REM 36
RESERVE (ROM) 148
RES instructions 102 133
'restack' (ROM) 158
Restarts (ROM) 142
RESTORE 36
RET instructions 98 129
RETURN 37
RND 41 175
ROM area 13 14
Rotation instructions 99 129
RST 0010 (ROM) see PRINT-OUTPUT
RST instructions 99
RUN 37

S
SAVE 37 143 161
SBC instructions 83 120
SCREEN$ 42 173
Scrolling 165
SEED (SV) 36 41
SET instructions 102 131
SGN 42 156
Sign bit 64 68 92
Sign flag 92
SIN 43 156
SLICING (ROM) 153
Spare memory area 14 19 110
S-POSN (SV) 144 168
SQR 43 156
Stack pointer 14 19 55 96
STACK-A (ROM) 154 161
STACK-BC (ROM) 140 154
STKBOT (SV) 14 19
'stk-data' (ROM) 157 161
STK-DIGIT (ROM) 154
STK-END (SV) 14 19 147
STK-FETCH (ROM) 154
STK-STORE (ROM) 153
STK-VAR (ROM) 153
STOP 37
STORE (ROM) 145
STRMS (SV) 147
STR$ 43 156
Structured programming 159
SUB instructions 83 120
SUBPPC (SV) 25 151
SYNTAX CHECKER (ROM) 136 138
Syntax flag (ROM) 138
SYNTAX-Z (ROM) 153
System variable area 14 16

T
Table of addresses (calculator's) (ROM 156
Table of constants (ROM) 155
TAN 43 156
Temporary colours 22 145 165
TEMPS (ROM) 165
Token table (ROM) 142
Two's complement arithmetic 64 94 109

U
UDG (SV) 14 44 146
Uncommitted logic array 8 11
User-defined graphics area 14 20 44 137 144 146
USR – number 43 111 156
USR – string 44 150

V
VAL 45 156 175
VAL$ 45 156 175
Variables area 14 18 31 32
Variable handling routines (ROM) 139
VARS (SV) 14 18 148
VERIFY 38 143

W
WORKSP (SV) 14 19
Work space 14 19

X
XORing 34
XOR instructions 86 122

Z
Z80 microprocessor Ch.1 & Ch.3

ANNOUNCING
The BEST Books For Your SPECTRUM



Dr. Ian Logan is the acknowledged leading authority on Sinclair computers. In this book, he gives a complete overview of the way the Spectrum operates, both for BASIC and machine language programming. A special section on the ROM operating system will give you insight into this computer as well as provide you with information on how to use many of the routines present in the ROM. This book is a must if you are serious about programming the Spectrum. Only £7.95.

Over the Spectrum is the book where you will find your dreams really do come true. If you want to know how to use the complete facility of the Spectrum, as well as have the full listing for over 30 Spectrum programs, this is the book for you. Fantastic programs such as the incredible *3D-Mazeman*, *Alien Invaders*, just to mention two. Games, utilities, educational and business programs are all in *Over the Spectrum*. Only £6.95.

This title speaks for itself, it's everything you need to understand about *Spectrum Machine Language* when you're just starting off. A must for all new Spectrum owners. Only £6.95.

After leading the way in Sinclair ZX81 software, we've produced the highest quality, most exciting Spectrum software available. From the three excellent books depicted above to fast-action games on cassette, we're providing the best choice in Sinclair Spectrum software today.

Whether it's for your new Spectrum or ZX81 Melbourne House has books and programs perfectly suited to your needs.

Send for your Spectrum or ZX81 catalogue today.

MELBOURNE HOUSE PUBLISHERS

send to:
U.S.A.: Melbourne House Software Inc.,
347 Reedwood Drive, Nashville TN 37217.

U.K. Melbourne House (Publishers) Ltd.,
Glebe Cottage, Glebe House, Station Road,
Cheddington, Leighton Buzzard, BEDS LU7 7NA

Australia & New Zealand:
Melbourne House (Australia) Pty Ltd,
Suite 4/75 Palmerston Crescent,
Sth. Melbourne 3205.

[ ] Please send me your Spectrum/ZX81 catalogue (please specify).
I enclose a stamped self-addressed envelope.

Name.....

Address.....

.....Code.....

UNDERSTANDING YOUR SPECTRUM

R E G I S T R A T I O N
C A R D

Please fill out this page and return it promptly in order that we may keep you informed of new software and special offers that arise. Simply cut along the dotted line and return it to the correct address selected from those overleaf.

Where did you learn of this product?

☐ Magazine. If so, which one?.....

☐ Through a friend

☐ Saw it in a Retail Store

☐ Other. Please specify.....

Which Magazines do you purchase?

Regularly:.....

Occassionally:.....

What Age are you?

☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ Over 25

We are continually writing new material and would appreciate receiving your comments on our product.

How would you rate this book?

☐ Excellent ☐ Value for money

☐ Good ☐ Priced right

☐ Poor ☐ Overpriced

Please tell us what software you would like to see produced for your computer.

_____

_____

_____

Name.....

Address.....

.....Code.....

Put this in a stamped envelope and send to:

In the United States of America return page to:
Melbourne House Software Inc., 347 Reedwood Drive, Nashville TN 37217.

In the United Kingdom return page to:
Melbourne House (Publishers) Ltd., Glebe Cottage, Glebe House, Station Road, Cheddington, Leighton Buzzard, Bedfordshire, LU7 7NA.

In Australia & New Zealand return page to:
Melbourne House (Australia) Pty. Ltd., Suite 4, 75 Palmerston Crescent, South Melbourne, Victoria, 3205.

Dr. Ian Logan is widely acknowledged as the leading authority on Sinclair computers. In this book, he gives you a complete overview of the way the Spectrum operates, both for BASIC and machine language programming, including numerous demonstration programs.

In Dr. Logan's own words *Understanding Your Spectrum* has three main aims: "*to explain, in simple terms, how the Spectrum works; *to teach Z80 machine code from first principles; and *to give details of 'monitor entry points' so that efficient programs can be written."

A special section of the ROM operating system will give you insight into the Spectrum and provide you with information on how to use many of the routines present in the ROM in your own programming.

This book is a must if you are serious about programming the Spectrum.



Melbourne House Publishers