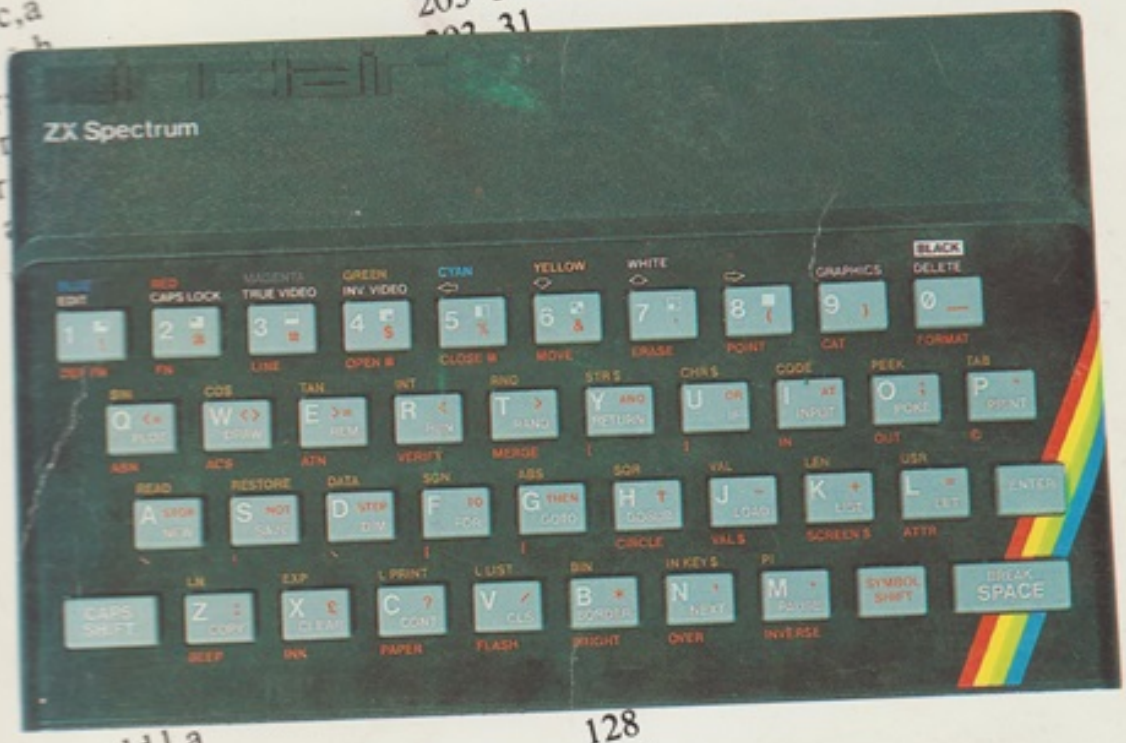
40 BEST MACHINE CODE ROUTINES for the ZX SPECTRUM

WITH EXPLANATORY TEXT



JOHN HARDMAN
ANDREW HEWSON

40 Best Machine Code Routines for the ZX Spectrum

John Hardman
and
Andrew Hewson

First Edition 1982
Copyright © Hewson Consultants 1982

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical including photocopying, recording, or any information storage and retrieval system without permission in writing from the publisher.

The programs and routines in this volume are the copyright of the publisher and are for the personal and private use of the purchaser only.

ZX Spectrum is a registered trademark of Sinclair Research Ltd, Cambridge, UK.

The contents of the ZX Spectrum 16K ROM are the copyright property of Sinclair Research Ltd.

ISBN 0-907912-03-6

Printed and bound in Great Britain for Hewson Consultants, 60A St. Mary's St, Wallingford, Oxon, by Powage Press, Aspley Guise, Milton Keynes.

Acknowledgments

To my family for their patience but especially to Debbie, John and Mairi for their unending encouragement.

J.H.

With thanks to Janet, Gordon and Louise.

A.D.H.

CONTENTS

Section A

		Page
1.	**Introduction**	3
	Why Use Machine Code	5
	How to Learn Machine Code	6
2.	**Internal Structure of the ZX Spectrum**	7
	The Memory Map	7
	PEEK and POKE	8
	Display File	10
	Attributes	11
	Printer Buffer	12
	Basic Program Area	12
	Five Byte Numeric Form	13
	Variables Area	13
	ROM Routines	14
3.	**Z80A Machine Language**	18
	Bits	18
	Bytes	19
	Addresses	20
	Z80A Registers	20
	Accumulator	21
	Flag Register	21
	Counting Registers	22
	Address Registers	23
	Index Registers	23
	Stack Pointer	24
	Program Counter	24
	Exchange Registers	25
	About the Instruction Set	25
	Glossary of Machine Code Instructions	26
	No Operation	26
	Load	26
	Push and Pop	27
	Exchange	27
	8 Bit Add and Subtract	27
	8 Bit and Or and Xor	27
	Compare	28
	8 Bit Increment and Decrement	28
	16 Bit Increment and Decrement	28
	16 Bit Add and Subtract	28
	Jump Call and Return	28
	Bit Instructions	29
	Rotate Left Digit	29
	Rotate Right Digit	29
	Accumulation Operations	29
	Restart	30
	Block Handling	30

CONTENTS (continued)

Section B

4. **Introduction** 33
   Machine Code Loader 34

5. **Scroll Routines** 39
   Scroll Attributes Left 39
   Scroll Attributes Right 40
   Scroll Attributes Up 41
   Scroll Attributes Down 42
   Left Scroll by One Character 44
   Right Scroll by One Character 45
   Up Scroll by One Character 46
   Down Scroll by One Character 48
   Left Scroll by One Pixel 50
   Right Scroll by One Pixel 51
   Up Scroll by One Pixel 52
   Down Scroll by One Pixel 54

6. **Display Routines** 57
   Merge Pictures 57
   Screen Invert 58
   Invert Character Vertically 59
   Invert Character Horizontally 60
   Rotate Character Clockwise 61
   Attribute Change 64
   Attribute Swap 65
   Region Filling 66
   Shape Tables 73
   Screen Magnify and Copy 77

7. **Routines to Manipulate Programs** 85
   Delete Block of Program 85
   Token Swap 86
   Rem Kill 88
   Rem Create 91
   Compact Program 94
   Load Machine Code into Data Statements 96
   Convert Lowercase to Upper Case 101

8. **Toolkit Routines** 103
   Renumber 103
   Memory Left 111
   Program Length 112
   Line Address 113
   Copy Memory 114
   Zero all Variables 116
   List Variables 119
   Search and List 122
   Search and Replace 126
   ROM Search 129
   Instr$ 131

Appendix A 137

Section A

1. INTRODUCTION

The aim of this volume is to provide both the beginner and the experienced computer user with a ready source of reference on a number of useful, interesting or entertaining machine code routines for the ZX Spectrum. To this end the book is divided into two sections. The Section A describes the features of the Spectrum which are of interest to the machine code programmer—what is meant by a machine code routine, the important internal features and routines of the Spectrum and the structure of the machine language itself.

The Section B presents the routines themselves. They are laid out in a standard format which is explained in detail at the beginning of the section. The routines are complete in themselves so that they can be loaded individually without reference to any other routines.

It is not necessary to understand how a routine works in order to use it because each routine can be loaded using the simple M/C Loader listed at the beginning of Section B. Hence if you are really impatient to use, say the List Variables routine simply turn to the relevant page, enter and RUN the M/C Loader and enter the decimal numbers listed in the column headed "Numbers to be entered". When all the numbers are loaded compare the value of the Check sum PRINTed by the M/C Loader with the value given with the routine. If they are the same you can be sure that the numbers have been entered correctly (unless you have made two or more errors which cancel out exactly). The routine is now ready for you to use.

If you have not got enough confidence to try a long routine like List Variables but are still keen to get started on machine code as soon as possible then choose a shorter routine. That way if you get lost or make too many mistakes you will not have spent so much time doing so. The routine to Scroll Attributes Down is ideal. Once again it is simply a question of entering and RUNning the M/C Loader listed at the beginning of Section B and copying the numbers given in the column headed "Numbers to be entered". When you have finished make sure that the Check sum is correct.

If you are happy to defer use of the machine code routines then read on. The remainder of this chapter introduces the basic ideas and goes on to explain in more detail the layout of the remainder of the book. The beginner is advised to read this information carefully but the more experienced user will only require to skim through it.

The Z80A microprocessor which drives the ZX Spectrum does not directly understand Basic words like PRINT, IF, TAB etc. Instead it obeys a special language of its own called machine code. The instructions in the Sinclair ROM which give the Spectrum its "personality" are written in this special language and they consist of a large number of routines for entering, listing, interpreting and executing the particular dialect of Basic which the Spectrum uses. In effect, the routines are groups of "WHAT TO DO IF" instructions. For example they tell the Z80A WHAT TO DO IF the next Basic command is the word PRINT, and then WHAT TO DO IF the next item is a variable name; and THEN WHAT TO DO IF the next item is a comma etc, etc.

Machine code consists of a sequence of positive whole numbers, each less than 256, and it dictates the action of the Z80A by setting eight switches according to the pattern of the binary equivalent of the number. The binary equivalent of 237 for example, is 11101101 so that when 237 is encountered the eight switches are set to on, on, on, off, on, on, off, on respectively.

Just because the machine obeys the binary version of the number there is no need for humans to consider the instruction in this form. We are used to using decimals and so it is this form that the M/C Loader in Section B recognises. However, even a string of decimal numbers is difficult to interpret and so the decimals are usually converted yet again to a special *assembly language* which is a bit cryptic but not too difficult to use with practice. Each routine in Section B is listed both in assembly language and in decimal numbers.

Assembly language is so-called because a special program called an *assembler* can be conveniently used to bring together or *assemble* many machine code instructions to form a new program. Assemblers are sophisticated programs because machine code is very extensive and they are usually written in machine code themselves. Several such assemblers are now available for the Spectrum and, whilst the routines in this book could be loaded using an assembler, it should be emphasised that the M/C Loader is quite adequate for the purpose.

One number only is required to specify the simpler Z80A instructions. The instruction to copy the contents of register c to register d is decimal 81 for example. (The meaning of the word register is explained in more detail in chapter 3. For the moment it is sufficient to think of c and d as akin to Basic variables). For these instructions there is a one-to-one correspondence between the decimal number and the assembly language version so that decimal 81, for example, is written in assembly language as ld d, c. "ld", by the way, stands for "load". Many assembly language instructions consist of similar simple abbreviations and for this reason they are often called mnemonics.

More complex instructions require two, three or even four numbers before they are completely specified in which case a single assembler mnemonic is used to represent them all. Table 1.1 lists a few examples of the numbers, their mnemonics and a brief explantation.

Ref	*Decimal*	*Assembly language*	*Comment*
(a)	81	ld d,c	Load d with the contents of c
(b)	14 27	ld c, 27	Put the number 27 into c
(c)	14 13	ld c, 13	Put the number 13 into c
(d)	33 27 52	ld hl, 13339	Put 13339 into the hl register pair. Note 27 + 256*52 = 13339; 27 is put into l; 52 is put into h
(e)	221 33 27 52	ld ix, 13339	Put 13339 into the ix register pair

Table 1.1 *Some examples of machine code instructions for the Z80A*

Line (a) of the table is the example of ld d, c discussed above. Lines (b) and (c) show how a positive whole number less than 255 may be loaded into a register using two successive numbers—the first specifies the action to be performed and the second specifies the number to be loaded. Line (d) shows how a large whole number may be loaded into two registers, h and l, together. This time it is the second and third numbers which specify the number to be loaded. The final example in line (e) illustrates a four number code for loading a large whole number into the ix register pair. Notice how three out of the four numbers also appear in line (d). In effect the first number specifies the ix instead of the hl pair.

The structure of machine language is explained in more detail in chapter 3 and a full list of the assembler mnemonics is given in appendix A. A more important question which should be answered at this point is:

Why use Machine Code?

With any programming language on any computer it always seems to be the case that there are tasks which the user wishes the machine to perform which cannot be conveniently written in the language available or which when written in that language, are very slow to execute. The ZX Spectrum is no exception in this regard.

Consider for example the problem of saving the entire screen display at the top of RAM or copying it back again perhaps with the intention of creating a cartoon effect by "flipping" between various displays. The display file and the attributes together occupy 6912 bytes and so it is necessary to move RAMTOP down to 32768 – 6912 = 25856 on the 16K machine to provide enough space for the copy of the display outside the Basic area (65536 – 6912 = 58624 on the 48K machine). The following simple Basic program will save the screen display but it takes a long time—about 70 seconds:

```
10 FOR i = 0 to 6911
20 POKE 25856 + i, PEEK (16384 + i)
30 NEXT i
```

The reason it takes so long is that the Spectrum spends most of its time decoding the commands before executing them. A certain amount of time is also spent converting numbers between the two byte integer form which the Z80A understands and the five byte decimal form in which the loop counter is held and also in performing five byte arithmetic. The steps are as follows:

1) Add i to 16384.
2) Convert the result to two byte form.
3) Retrieve the contents of the PEEK address.
4) Add i to 25856.
5) Convert the result to two byte form.
6) Store the value retrieved in the POKE address.

7) Add one to the value of i and store the result.
8) Subtract i from 6911. If the result is positive or zero go to 1).

Each time a pass is made through the loop the Spectrum must decode each command afresh because it retains no memory of the previous operations. It is easy to see that the computer spends over 99% of the time preparing to perform the task rather than performing the task itself. Therefore it is no surprise to know that a machine code routine to save the screen executes more or less instantaneously. An example routine is given in Section B.

How to Learn Machine Code

The machine language of the Z80A microprocessor is very complex and to understand all its facilities requires a good reference book, a lot of thought and much practice. There are several books available. The standard reference is *How to Program the Z80* by Rodney Zaks, published by Sybex and available through Radio Shack (ie Tandy Stores), ISBN No. 0-89588-057-1. It contains a great deal of information about the hardware organisation of the microprocessor as well as listing full details of the instruction set. The beginner might find it rather formidable because it runs to more than 600 pages.

A rather more readable account is contained in *Z80 and 8080 Assembly Language Programming* by Kathe Spracklen, published by Hayden, ISBN No. 0-8104-5167-0. The book starts at a more elementary level and covers the more important software aspects and ignores the hardware almost entirely.

This book is intended to act not only as an introduction to machine code for the beginner but also to be useful to the more experienced user. It gives the reader a strong incentive to learn machine code by providing him with routines which he can incorporate into his own Basic or machine code programs, with or without adaption.

Most of the routines depend intimately on the structure of the ZX Spectrum and so the next chapter covers the topic in some detail. It covers, for instance, the form of the display file, the program area and the variables area, explains the layout of Basic program lines and introduces five byte floating point arithmetic. The reader of Section B is assumed to be familiar with the contents of this chapter.

The third chapter explains Z80 machine language in some detail, describing many items which are assumed later on. It contains a glossary of the instruction set which covers most of the salient facts without reproducing the detailed coverage given in Zaks' book.

2. INTERNAL STRUCTURE OF THE ZX SPECTRUM

A computer is a machine which is capable of storing a sequence of instructions and then executing them. Clearly to do so it requires a memory in which the instructions can be stored. The ZX Spectrum contains two distinct types of memory. The first type is read-only-memory (ROM) which contains the fixed set of instructions implanted in the machine by the manufacturer. The second type is random-access-memory (RAM).

Random-access-memory is the notepad of the Spectrum. When the computer is performing a task it is continually looking at what is in RAM ("reading" from memory) and altering the contents of RAM ("writing" to memory). The Spectrum does not use its notepad haphazardly. Different parts of RAM are used to store different sorts of information. A Basic program entered by the user for example, is stored in one part of RAM, whilst the variables used by the program are stored elsewhere. The size of the notepad is limited and so the machine is careful to allocate just enough space and no more to the information that it holds. Thus the spare space is always collected in one place so that if, for example, the user wants to add a line to his program the information in RAM can be shuffled along using up some of the spare space to accommodate the extra line.

Most of this chapter is devoted to explaining in detail how the Spectrum organises RAM because many of the routines in Section B are designed to manipulate RAM. Therefore if the reader is to understand the design of the routines as opposed to using them blindly, he must understand the contents of this chapter. The chapter covers the display file, the attributes, the printer buffer, the system variables, the program area and the variables area. The final section describes the routines in ROM which are referred to in Section B.

The Memory Map

There are 16384 memory locations in RAM in the unexpanded ZX Spectrum (the expanded version contains a further 32768 locations making 49152 in all). Each location can hold a single whole number between 0 and 255 inclusive and is identified by its address which is a positive whole number.

Addresses 0 to 16383 are assigned to the fixed form of memory, the ROM, and so the first address assigned to RAM is 16384. Table 2.1 is the memory map of the Spectrum which shows how RAM is used starting at address 16384. The display file, for example, which holds the information which is currently shown on the screen, occupies locations 16384 to 22527. The attributes, which determine the colour, brightness etc of the screen display, follow immediately afterwards in locations 22528 to 23295.

The first five starting addresses in column 1 of table 2.1 are all fixed because the display file, the attributes etc all occupy a fixed amount of space. The fifth area is assigned to the microdrive maps. If a microdrive is attached to the Spectrum this area contains information on the layout

of the data in the microdrive. If a microdrive is not attached, the area is not needed in which case the sixth area, channel information, is placed immediately after the fourth, the system variables, in line with the Spectrum practice of saving space wherever possible. Hence the starting address of the channel information area and all subsequent areas is not fixed but can "float" up and down RAM.

The Spectrum keeps track of the starting address of all these areas by storing the current value of each address within the system variables area. The system variables area lies before the microdrive map at locations 23552 to 23733 inclusive and so there is no question of this area also moving up and down RAM! The address within the area which holds the starting address of all the floating areas is listed in column two of table 2.1. The address of the Basic program area, for example, is held at 23635 within the system variables area.

Starting address or system variable name	*Location of system variable*	*Memory contents*
16384	—	Display file
22528	—	Attributes
23296	—	Printer buffer
23552	—	System variables
23734	—	Microdrive map
CHANS	23631	Channel Information
PROG	23635	Basic program
VARS	23627	Variables
E_LINE	23641	Command/line being editted
WORKSP	23649	Data being INPUT
STKBOT	23651	Calculator stack
STKEND	23653	Spare
sp	—	Machine stack and GOSUB stack
RAMTOP	23730	User machine code routines
UDG	23675	User defined graphics
P— AMT	23732	End of RAM

Table 2.1 *The memory map. The stack pointer, sp is held not in RAM but in the sp register in the Z80A microprocessor.*

Referring to each system variable by the address at which it is held is rather awkward and so each is given a name—PROG in the case of the location which holds the address of the Basic program area. These names are for the users convenience only as they are not recognised by the Spectrum. Thus entering the line:

PRINT PROG

will cause the error message "2 Variable not found" to be PRINTed unless a Basic variable called PROG has been generated coincidentally by a

program or by the user. The value of such a Basic variable would have nothing to do with the value of the PROG system variable.

PEEK and POKE

The memory map is the key to understanding the use of RAM by the Spectrum but the keys to exploring the RAM are the Basic keywords, PEEK and POKE, which allow the user to look at and alter respectively, the contents of each memory location.

PEEK is a function of the form:

PEEK address

The address can be a positive whole number between 0 and 65535 or an arithmetic expression which when evaluated gives such a positive number. It is important to enclose an arithmetic expression in brackets because:

PEEK 16384 + 2

is interpreted as 2 added to the result of:

PEEK 16384

whereas:

PEEK (16384 + 2)

is interpreted as:

PEEK (16386)

The value returned by the PEEK function is the number currently held at the address in question which will always be a positive whole number between 0 and 255 inclusive. It was explained above that the PROG system variable is held at address 23635 but the value of PROG, being an address in RAM, is always much larger than 255 therefore two adjacent addresses, 23635 *and* 23636, are needed to hold it. The value of PROG can be PRINTed by entering:

PRINT "PROG = "; PEEK 23635 + 256* PEEK 23636

All addresses are held in two adjacent locations in this fashion and can be inspected by entering:

PRINT PEEK *first address* + 256* PEEK *subsequent address*

For example if a Spectrum is used without a microdrive attached the microdrive map area will be non-existent and the channel information will follow immediately after the system variables area. Thus the value of the CHANS system variable will be the same as the starting address of the microdrive map, were it to exist, ie 23734. CHANS is held at 23631 *and* 23632 and so entering:

PRINT PEEK 23631 + 256* PEEK 23632

will yield the value 23734.

The PEEK function can be used to look at the contents of any location in memory including the fixed instructions in ROM. It is therefore

a very important tool. PEEKing any location will not cause the Spectrum to crash or corrupt a program or variables. Very occasionally the results of a PEEK can be misleading because the contents of the location being PEEKed may alter during or immediately after the execution of the instruction. For example, if the contents of the locations which are assigned to the top left hand corner of the screen display are PEEKed and the results PRINTed in the top lefthand corner of the screen the information will already be out-of-date by the time the user views it.

The POKE command is altogether more dangerous than the PEEK function because by invoking it the user is interfering in the functioning of the Spectrum. Thus it is quite possible to make nonsense of the information in RAM using this command causing the machine to crash or to halt and display an error code.

The form of the command is:

POKE *address, number*

Once again the address is a positive whole number between 0 and 65535 inclusive or an arithmetic expression which gives such a number when evaluated. In this case it is not essential to enclose an arithmetic expression in brackets because POKE is a command not a function and therefore cannot be evaluated as a whole. The number POKEd into the location must lie between 0 and 255 inclusive.

The Spectrum will accept and execute a POKE command to put a number into an address in ROM (ie an address between 0 and 16383) but the number will never reach its destination. This fact can be demonstrated by RUNning the following program:

```
10 PRINT PEEK 0
20 POKE 0,92
30 PRINT PEEK 0
```

Lines 10 and 30 will each PRINT the value 243 which happens to be the contents of location 0. Line 20 has no effect.

The Display File

The normal display consists of 24 lines each containing 32 characters. We have seen that the display file occupies locations 16384 to 22527 ie 6144 locations in total, therefore the number of locations used per character is:

6144/(24*32) = 8

The easiest way to get an overall impression of how the display is organised is PRINT a picture on the screen, SAVE the screen to tape, clear the screen and LOAD the picture back again. The program P2.1 SAVEs and LOADs the screen in this manner using graphics character 5 to create the original picture.

When the picture is reLOADed from tape it becomes clear that the display is divided into three sections of eight character lines each. Each


```
100 FOR i=0 TO 703
110 PRINT " ";
120 NEXT i
130 SAVE "Picture"SCREEN$
140 CLS
150 INPUT "Rewind the cassette,
play it and then press a key ";
$
160 LOAD "Picture"SCREEN$
```

Program P2.1. *A program to SAVE, clear and LOAD the screen.*

character line is further divided into eight lines of *pixels*. Surprisingly, the Spectrum does not LOAD, the eight pixel lines which form the first character line followed by the eight pixel lines which form the second character line etc. Instead it LOADs the top pixel lines of the first eight character lines, followed by the next pixel lines of the same eight character lines and so on. The top section of the display, consisting of eight character lines and the final eight character lines, form the middle and the bottom of the display respectively.

Another way of understanding the form of the display is to consider where the eight bytes which are used to form the character at the top left hand corner of the screen are held. The first byte forms the topmost eighth of the character and is located at the beginning of display file at address 16384. A few moments experimentation shows that:

POKE 16384,0

blanks out the top line of eight pixels which form the top of the first character whereas:

POKE 16384, 255

causes all the pixels to be illuminated. POKEing numbers between 0 and 255 causes a speckled effect.

The line of eight pixels which is second from top in the first character on the screen is not formed from the number held at location 16385, rather this location is used for the top line of pixels in the adjacent character. There are 32 characters in a line and 8 in a section so that the second from top line of eight pixels in the first character is formed from the number held at location:

16384 + 32*8 = 16640

A similar argument applies to the remaining six lines of eight pixels therefore the form of the character at the top left hand corner of the screen is dictated by the contents of addresses:

16384, 16640, 16896, 17152, 17408, 17664, 17920, 18176

Program P2.2 allows the user to experiment by POKEing various numbers into these eight locations.

Every location in the display file controls the condition of eight pixels on the screen. This control is exerted by converting the number which is held at a given location to its binary form and then setting the eight pixels

```
  10 REM Routine to set characte
r at top LHS of screen"
  20 INPUT "A character is forme
d from eight bytes, each lying b
etween 0 and255 inclusive. Enter
 number of  byte (0 to 7) ";n
  30 IF n<0 OR n>7 OR n<>INT n T
HEN BEEP ,2,24: GO TO 20
  40 INPUT "Enter contents of by
te ";m
  50 IF m<0 OR m>255 OR m<>INT m
 THEN BEEP ,2,24: GO TO 40
  60 POKE 16384+8*32*n,m
```

Program P2.2. *A program to construct the character at the top left hand corner of the screen.*

according to the zero/one pattern of the eight binary digits. For example, 240 when converted to binary is:

11110000

Therefore if a location contains the number 240, four of the eight corresponding pixels will be illuminated and the remaining four will be blank.

To summarise, the display file consists of 6144 locations with eight locations assigned to each character position. Each location dictates the condition of a horizontal bar of eight pixels. The locations assigned to a given character position do not occur adjacent to one another, instead the display is divided into eight sections and within each section 256 locations separate the constituent bytes of each position.

The Attributes

The contents of the display file determine only which pixels are illuminated on the screen. The colour of the PAPER and INK and the BRIGHT and FLASH conditions are determined by the attributes. The attributes area occupies locations 22528 to 23295 with one location being assigned to each of the 768 character positions. In contrast to the display file the locations are assigned to the character positions in the obvious fashion, ie starting at the top left hand corner and working from left to right and top to bottom.

Each location sets both the INK and PAPER of the position to which it is assigned to one of the eight colours shown above the top line of keys on the Spectrum keyboard. It also determines whether the position is BRIGHT and whether it FLASHes. The four parameters are encoded using the following calculation:

Attribute value = 128*FLASH + 64*BRIGHT + 8*PAPER + INK

FLASH and BRIGHT take the value one if the appropriate condition applies and zero otherwise and PAPER and INK take the value of the required colour as shown on the keyboard (red is 2 for example). Program P2.3 decodes the attribute, ie given an attribute value it PRINTs the corresponding PAPER and INK colours etc.


```
  10 REM Atrribute decoder
  20 DATA "Black  ","Blue   ","R
ed   ","Magenta","Green  ","Cya
n   ","Yellow","White  ","Bright
 ","Flash"
  30 DIM c$(8,7)
  40 FOR i=1 TO 8
  50 READ c$(i)
  60 NEXT i
 100 REM Atrribute decoder
 110 INPUT "Enter a number betwe
en 0 and 255This program decodes
 its inter- pretation in the att
ributes file";n
 120 IF n<0 OR n>255 OR n<>INT n
 THEN BEEP ,2,24: GO TO 110
 200 PRINT "Ink colour is ";c$(1
+n-8*INT (n/8))
 210 PRINT "Paper colour is ";c$
(1+INT (n/8)-8*INT (n/64))
 220 IF INT (n/64)=1 OR INT (n/6
4)=3 THEN PRINT "Character is BR
IGHT"
 230 IF n>127 THEN PRINT "Charac
ter will FLASH"
 300 PRINT AT 6,0;"▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
▌▌▌▌▌▌▌▌▌▌▌▌▌▌▌
```

Basic Program Area

If a microdrive is attached to the Spectrum the beginning of the Basic program area must be determined by referring to the PROG system variable which is located at 23635. In the absence of a microdrive the area starts at 23755. In these comments it is assumed that a microdrive is not attached.

The four line program P2.4 PRINTs the contents of the 18 locations at the beginning of the program area as shown in figure F2.1. These 18 locations are used to hold the first line ie:

10 REM Peek program

Much can be learnt about the method of encoding programs by studying figure F2.1.

```
10 REM Peek program
20 FOR i=23755 TO 23772
30 PRINT i,PEEK i
40 NEXT i
```

Program P2.4. *A program to PRINT the contents of the first eighteen locations in the program area.*

```
23755        0
23756        10
23757        14
23758        0
23759        234
23760        80
23761        101
23762        101
23763        107
23764        32
23765        112
23766        114
23767        111
23768        103
23769        114
23770        97
23771        109
23772        13
```

Figure F2.1. *The form in which the line:*
10 REM Peek program
is held in the program area.

The line number, 10, is stored in the first two locations in the form:

line number = 256*PEEK *first address* + PEEK *second address*

Notice that the Z80A convention of multiplying the contents of the second address by 256 and adding it to the contents of the first is not applied.

The convention is applied to the next two locations, 23757 and 23758 which together hold the length of the remainder of the line starting at location 23759. The number stored in this case is:

14 + 256*0 = 14

Hence the next line starts at location:

23759 + 14 = 23773

Location 23759 itself holds the number 234 which is the character code of REM. The next 12 locations hold the character codes of the eleven letters and a space in the title:

Peek program

Finally location 23772 contains 13 which is the code for the ENTER character indicating that this is the end of the line. Table 2.2. summarises the method of encoding programs in the program area.

Locations	*Contents*
1 and 2	line number stored in the reverse order to the Z80A convention
3 and 4	length of the line excluding the first four locations.
5	the command code.
Final	the ENTER character, number 13.

Table 2.2. *The method used to encode program lines.*

An item which is omitted from the table is a description of the method which is used to store values occurring in the program. The method can be explored by substituting the line:

10 LET a = 1443

in program P2.4 Figure 2.2 shows the result of RUNning the program in this form.

```
23755        0
23756        10
23757        14
23758        0
23759        241
23760        97
23761        51
23762        49
23763        52
23764        52
23765        51
23766        14
23767        0
23768        0
23769        163
23770        5
23771        0
23772        13
```

Figure 2.2. *The form in which the line:*
10 LET a = 1443
is held in the program area.

Locations 23755 to 23758 are as before. They are followed by the codes for LET,a, = and the four digits in turn which together form the number 1443. The next item in location 23766 is 14, the character code which indicates that the subsequent five locations hold the number 1443 in numerical form. The line is terminated at location 23772 by the ENTER character as before.

Five Byte Numeric Form

Five memory locations are used to store each number which appears in a Basic Program (except line numbers as we have already seen). Whole numbers between – 65535 and 65535 are stored in a manner akin to the Z80A convention. For these numbers the first two locations and the last each contain zero and the third and fourth hold the number in the form:

number = PEEK *third location* + 256*PEEK *fourth location*

Thus, for example, 16553 is held in five locations as:

0 0 169 64 0

because

169 + 256*64 = 16553

Non-integer numbers are held in floating point form as a exponent in the first location and a mantissa in the following four ie:

number = mantissa * 2↑ exponent

The first location of the mantissa is also used to determine the sign of the number. If the location contains a value in the 0 to 127 range the number is positive, if not it is negative.

Program P2.5 can be used to reconstruct a non-integer number from its five component values.

```
  10 PRINT "Enter the exponent a
nd the four bytes of the mantiss
a. All entries to lie between 0
and 255 inclusive"
  20 INPUT e,a,b,c,d
  30 PRINT ,,"Exponent = ";e
  40 PRINT "Mantissa = ",a,,b,,c
,,d
  50 PRINT ,,"The number = ";(2*
(a<128)-1)*2↑(e-160)*(((256*(a+1
28*(a<128))+b)*256+c)*256+d)
```

Program P2.5. *This program reconstructs a non-integer number from its five component values.*

The Variables Area

The variables area starts at the location held in the VARS system variable which is itself held at 23627. Whenever a new variable is declared either in a program or from the keyboard an appropriate amount of space is created for it in this area.

All variable names must begin with a letter and no distinction is made between upper and lower case. These restrictions enable the Spectrum to manipulate the character code of the leading letter of each variable so that it can distinguish the six permitted types of variable by inspecting the range within which the code lies. All numeric variables with single character names, for example, have codes in the range 97 to 122; the letter a being 97; b being 98; c being 99 etc. Similarly, numeric arrays have codes in the range 129 to 153; a being 129; b being 130; c being 131 etc. The code ranges are summarised in table 2.3. The length of each type of variable is also shown in table 2.3.

Variable type	*Character code range*	*Length in variables area*
Numeric (single character name)	97 to 122	6
Numeric (multiple character name)	161 to 186	5 + name length
Numeric array	129 to 154	4 + 2* number of dimensions + 5* total number of elements
Control variable of a FOR-NEXT loop	225 to 250	18
String	65 to 90	3 + string length
Character array	193 to 218	4 + 2* number of dimensions + total number of elements

Table 2.3. *Variables, the range of character codes and the variable lengths.*

ROM Routines

Some of the routines in Section B use routines in the ROM as follows:

rst 16

PRINT the contents of the accumulator

call 3976

Insert the character held in the accumulator at the address in RAM held in the hl register pair.

call 4210

Delete one character at the address in RAM held in the hl register pair.

call 6326

If the accumulator holds the number character (14) set the zero flag and increment the hl register pair five times.

call 6510

Return in hl the address in RAM of the line whose line number was passed to the routine in hl.

3. Z80A MACHINE LANGUAGE

This chapter opens by explaining some of the more important words like *bit, byte, address* and *register,* which are taken for granted in the remainder of the book. The number and variety of the Z80A registers is then examined with particular reference to a small number of example instructions. Finally a summary of the instruction set is presented.

Perhaps the most difficult aspect for the newcomer to machine code programming is the number of new words and concepts which must be absorbed. Therefore before embarking on the main part of the chapter let us examine one instruction as an example of what is to come. Consider the following compound instruction which is to be found in many of the routines in Section B:

ld hl (23627)

The instruction is read as *load* the *hl register pair* with the *bytes* held at *addresses* 23627 and 23628. Each of the words in italics is explained in more detail in this chapter.

The instruction is conveyed in the form of three decimal numbers—42, 75, 92. The first number means:

ld hl, ( )

ie. load the hl register pair with the contents of two consecutive memory addresses. The addresses in question are specified by the second and third numbers using the calculation:

lower address = first number + 256*second number
higher address = lower address + 1
or in this case:
lower address = 75 + 256*92 = 23627
higher address = 23627 + 1 = 23628

The word load is just another way of saying copy and h and l can be thought of as two special locations within the Z80A which are used for holding numbers. Thus the whole instruction means copy the contents of 23627 into register l and 23628 into register h. Notice that the *l*ower address is the source for *l* and the *h*igher address is the source for *h*.

Bits

A *bit* is the fundamental unit of computer memory because it can exist in only one of two states. The two states can be thought of as representing ON or OFF; TRUE or FALSE; YES or NO; UP or DOWN; MALE or FEMALE or any other pair of logically opposite conditions. The mechanism by which a computer memory works is not really important to us but in the Spectrum the state of a bit is memorised by setting a microscopic solid-state switch either ON or OFF as appropriate.

The usual notation is to think of one state as the ZERO state and the other as the ONE state. A bit is considered to be 'set' when it is in the

state representing ONE and to be 'reset' otherwise. This notation allows us to speak of a given bit pattern in terms of its binary equivalent and by converting the binary number to a decimal each bit pattern can be given a unique positive integer decimal number.

For example consider 8 bits of which the rightmost four are set and the four leftmost are reset. Such a bit pattern is illustrated in table 3.1.

Switch	Off	Off	Off	Off	On	On	On	On
Setting	Reset	Reset	Reset	Reset	Set	Set	Set	Set
Binary Pattern	0	0	0	0	1	1	1	1
Bit Number	7	6	5	4	3	2	1	0

Table 3.1. *A group of 8 bits with the leftmost four reset and the rightmost four set.*

The binary pattern can be converted to decimal if it is remembered that, in a binary number, the rightmost column is the units column; the next to the left is the twos column; the next to the left again is the fours column and so on, doubling at each move to the left. The decimal equivalent of 00001111 is therefore:

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

because there is 1 in each of the ones, twos, fours and eights columns and 0 in the remainder.

It is obviously inconvenient to refer to bits as 'the rightmost' or as 'the second from the left' and so the convention is adopted of numbering the bits from the right starting at zero. It is not entirely coincidental that when this convention is used the bit number is also the number to which 2 must be raised to give the value of the column.

ie $2^{\text{bit number}}$ = column value

Bit 3, for example, appears in the eights column and $2^3 = 8$.

Bytes

The Z80A microprocessor which lies at the heart of the ZX Spectrum operates on eight bits at a time. (The term "operates" covers all the different tasks which are built into the instruction set like addition, subtraction, rotation, logical AND etc. The form of these instructions is explained in detail later in this chapter). Thus although a bit is the fundamental unit of computer memory, bits are usually manipulated together in groups of eight. These groups of eight bits are called a *byte* (pronounced bite).

Each of the bytes in RAM can be used to hold a single positive whole number lying between 0 and 255 inclusive by setting or resetting the eight bits in the byte according to the binary equivalent of the number. The byte in table 3.1 for example, holds decimal 15.

There are 16384 bytes in the read only memory (ROM) in the ZX Spectrum and it is the contents of these bytes together with the electronic organisation which give the computer its characters. The contents are

imprinted in the ROM when the Spectrum is manufactured and cannot subsequently be changed. It is for this reason that the memory is called read only memory—the contents can be read but they cannot be overwritten.

The unexpanded Spectrum contains a further 16384 bytes of random access memory (RAM). The term random access is something of a misnomer. It does not mean that memory is used haphazardly, rather it means that any byte can be reached (ie accessed) immediately at any time. This facility contrasts with those of a sequential access memory like a cassette tape for which it is necessary to move along the memory medium until the particular portion required is reached.

To the uninitiated, 16384 does not seem to be a convenient number of bytes to use. In fact it is a very convenient number because $2^{14} = 16384$ (ie 16384 is equal to 2 multiplied by itself 14 times). In the computer world, powers of 2 are "round numbers" just as powers of ten—hundreds, thousands, millions—are "round numbers" in everyday life. A particularly important "round number" is 1024 which is 2 to the power of 10. 1024 is sufficiently close to one thousand to justify using the letter K to represent it. (K is used for a thousand in the metric system as in kilogramme—Kg, kilometer—Km etc). Thus 1024 is written as 1K and 16384, which is 16×1024, is written as 16K.

Addresses

A computer must be able to identify each location in its memory so that it may copy to and from the right location. Hence each location is given a unique *address*. An address is a positive whole number, greater than or equal to zero.

Many of the Z80A instructions are of the form "copy the contents of the following address into such-and-such a register or register pair". The instruction:

ld hl, (23627)

which was described at the beginning of this chapter is of this form. The address following the instruction is held in two bytes and so the number of locations which the processor can access uniquely is limited to the number of addresses which can be held in two bytes. This number is the same as the number of different bit patterns which can be adopted by the 16 bits which make up the two address bytes ie $2^{16} = 65536$.

A two byte address is interpreted in the form:

address = first byte + 256*second byte

The two bytes are sometimes called the low and high bytes respectively. The two byte form of 16384 for example (the beginning of RAM in the Spectrum), is low byte = 0; high byte = 64 because:

0 + 256*64 = 16384

The Z80A Registers

A computer does not alter the contents of memory directly when it is executing a program, rather it copies the contents of a location in memory

into a *register* and operates on the contents of the register. Registers have a similar function in machine language to that of variables in Basic in that they are used to store numbers and can be used to control a decision. They differ from Basic variables in that they are limited in number and they exist within the processor itself and not in RAM. Also they only hold one byte, or two bytes in the case of a register pair.

The Z80A is a powerful microprocessor because it has several registers and so it can hold several numbers at once thereby reducing the need to make time-consuming transfers between the processor and memory. Most of the registers have one or more special features.

The Accumulator Register—a

The accumulator is the most important register because most of the arithmetic instructions, addition for example, and the logical instructions, eg logical OR, operate on the contents of this register. In fact it gains its name because the result of several succesive operations accumulates in the a register.

Some of the instructions which refer to the accumulator use a second register or a memory address as a source of data. For example, add a,b instructs the processor to add the contents of the b register to the a register, leaving the result in a.

The Flag Register—f

Most of the registers occur in pairs in the sense that some instructions operate on two registers together. The *f* or flag register is paired with the *a* register in this sense although the link is rather tenuous because it is limited to the push, pop and exchange instructions.

The f register is rather different from all the others because the eight individual bits in the register are used as so-called flags to record and control the sequence of program execution. Each flag is used to indicate that either one of two logically opposite events has occurred, for example the zero flag indicates whether the result of the last addition, subtration etc was zero. Only four of the eight flags are of interest to most users. Their features are summarised in table 3.2.

The Sign flag is the simplest. By convention if a byte is being used to represent a signed number then bit seven is used to hold the sign, being set when the number is negative and reset otherwise. The sign flag reflects the sign of the last result.

The Zero flag is set if the result of the last operation is zero. It is also used by comparison instructions which are in effect subtraction instructions for which the result is discarded.

The carry flag records the overflow which occurs if the result of an addition is too large to record in the register and if a "borrow" occurs on subtraction. There are also some rotation instructions in which the bits in a register are rotated to the left or to the right with bit 7 and 0 being rotated to or from the carry flag.

Flag	*Mnemonic*	*Mnemonic when reset*	*Use*
Sign	*M*	*P*	Set when the last result is negative.
Zero	Z	NZ	Set when the last result is zero or a match occurred.
Carry	C	NC	Set when the last result is too large to be fully recorded in one byte (or two bytes for operations on register pairs).
Parity/Overflow	PE	PO	Parity—set when the last result had odd parity. Overflow—set when an operation changes bit seven as a result of an overflow from other bits.

Table 3.2. *The four flags which control most of the operations of the Z80A.*

The Parity/Overflow flag is really two flags in one. It is used as an overflow flag by arithmetic instructions to indicate if bit seven has been affected by a carry or a borrow generated by bit six. It is therefore used to check if the sign bit has been corrupted. Logical instructions use the same flag to indicate the parity of the result. (The parity of a binary number is the number of bits set to one. If the number is even the parity is said to be even, if it is odd, the parity is said to be odd). The flag is set if the parity of a result is even.

The effect of some instructions depends on the current setting of particular flags. For example the instruction:

```
jr z, d
```

causes the Z80A to jump over the next d instructions if the zero flag is set. If the zero flag is not set the processor executes the next instruction in sequence as usual. Thus the flag register is important because it allows the processor to make decisions and branch to another part of the program.

The Counting Registers—b and c

The b register and to some extent the c register with which it is paired is available for a number of purposes but its most important use is as a counter. We have already seen how the flow of a program can be controlled by the use of the zero flag in the jr z,d instruction. Another instruction:

```
djnz d
```

also use the zero flag to allow loops to be constructed in machine code using b as a counter in an analogous fashion to FOR-NEXT loops in Basic.

When the instruction is encountered the Z80A decrements the contents of the b register, ie reduces the contents by one. If the result is zero then

the next instruction in the sequence is executed. If the result is not zero the routine jumps d instructions. If the programmer uses a negative value for d the jump goes back earlier in the program and assuming there are no other branches, the processor will eventually encounter the same instruction again. Thus by loading the b register with a suitable value initially and setting the displacement, d, appropriately, a section of code can be executed a given number of times.

The b register holds one byte only and so it can be set to any number between 0 and a maximum of 255. Hence at most 255 passes can be made through the same section of code using this mechanism.

There are no similar facilities for making more than 255 passes through a loop, but there are a limited number of very powerful instructions which use all 16 bits of the bc register pair as a counter up to 65535. An example is the instruction:

```
cpdr
```

When it is encountered the Z80A:

1) decrements bc by one;

2) decrements the contents of hl (hl is another register pair)—see below:

3) compares the contents of the accumulator, a, with the contents of the location in memory whose address is held in hl.

The processor repeats these actions until either a match is found between a and the memory contents or until bc = 0. Thus this instruction can be used to search through memory for an address containing a particular number.

The Address Register—de and hl

The d and e registers do not have any individual function and are mostly used as temporary, rapidly accessible memory. They may also be used together to hold the address of a location in memory which is currently of interest.

The main function of the h and l registers is together to hold the address of a location in memory and we have already seen how certain powerful instructions make use of hl for this purpose. h stands for high byte and l stands for low byte and the address is held in the form:

address = 256*h + l

giving a maximum of 65536 unique addresses (ie 0 to 65535 inclusive).

The Index Registers—ix and iy

The ix and iy registers are each 16 bit registers and can only be used as such, in contrast to the bc, de and hl registers which we have met so far which can be used in pairs as 16 bit registers or individually as 8 bit registers. ix and iy are generally used in a similar fashion to the hl register pair although the instructions which drive them require one more byte of storage compared to the equivalent hl instructions.

For example:

add hl,bc

is a one byte instruction which causes the Z80A to add the contents of the hl and bc register pairs and leave the result in hl. The same instruction using ix ie:

add ix, bc

is a two byte instruction.

ix and iy have one further property which is not available to hl and that is that they can be used with a displacement, d. This means that an instruction which references (ix + d) does not use the memory location whose address is held in ix. Rather d is added to the value in ix to give a new address and the instruction then uses the corresponding memory location. It is this property which leads ix and iy to be called index registers.

The Stack Pointer—sp

The stack is an area at or near the top of RAM which is used for the temporary storage of the contents of pairs of registers. It is designed to grow down the RAM as it is filled and to shrink back up the RAM as it is emptied. The bottom of the stack is fixed and, in the ZX Spectrum, it lies immediately below the location pointed to by the RAMTOP system variable. The top of the stack is *below* the bottom of the stack because it grows downwards and shrinks upwards. The address of the current location of the top of the stack in the sp register.

Transfers to and from the stack are made by means of push and pop instructions. For example:

push hl

causes the processor to:

1) decrement sp;
2) copy the contents of h to the location pointed to by sp;
3) decrement sp;
4) copy the contents of l to the location pointed to by hl.

The pop instruction is the exact reverse. In this manner the most recent pair of values pushed on to the stack are always the values which are popped off again. This provides a simple and convenient method of storing the contents of registers temporarily, perhaps whilst a subroutine is called. Provided the register pairs are popped in the reverse order to that in which they were originally pushed, no problems will arise.

The Program Counter—pc

The program counter, pc is a very important 16 bit register because it holds the address in memory of the next instruction to be executed.

The normal flow of events when an instruction is executed is as follows:

1) Copy the contents of the location pointed to by pc into a special register within the processor.

2) If the instruction is held in several bytes, increment pc and copy the contents of the next location into a second special register.
3) Increment pc so that it points to the next instruction to be executed.
4) Execute the instruction which has just been copied in.

A jump instruction such as djnz d or jr z, d alters the normal flow of events by altering pc during step 4). Note that this alteration occurs after pc has been incremented so the value of a displacement, d, should always be calculated relative to the position of the instruction following the one containing the displacement.

The Exchange Registers—af', bc', de' hl'

The Z80A possesses duplicates of each of the a,b,c,d,e,h and l registers. The duplicates are distinguished by the use of a prime, for example a' is the duplicate a register. No instructions operate on these duplicates directly but exchange instructions are available to swap two or more registers out of use and to bring their duplicates into use in their stead.

Exchange instructions are executed very rapidly, much more rapidly than push and pop instructions for example. The contents are not physically copied from one register to the other. Rather a set of internal switches are changed so that the prime register is used by subsequent instructions and the original register becomes dormant.

About the Instruction Set

There are more than 600 elements in the Z80A instruction set as listed in appendix A. As there are only 256 different arrangements of 8 bits (because $2^8 = 256$) less than half the insructions can be held in one byte. The remaining instructions are held in two or even three bytes. The first byte of a two byte instruction is either 203, 221, 237 or 253 (CD, DD, ED, or FD in hexadecimal). The first two bytes of a three byte instruction are either 221, 203, or 253, 203, (DD, CB or FD, CB in hexadecimal).

Some instructions are followed by a one byte displacement, d, or a one byte number, n, or a two byte number or address, nn, to which the instruction refers. In this way a single instruction can occupy as many as four bytes in total. For example the instruction:

jr nz, d

which we have already met requires one byte to hold the instruction itself (32 in decimal, 20 in hexadecimal) and a second byte to hold the displacement, d.

In this chapter all instructions are referred to by their assembly language mnemonic or Op Code. The mnemonics are an abbreviated way of describing each instruction and are for human convenience only. The Spectrum will not recognise the mnemonics except through the medium of an assembler program.

Certain conventions are followed as listed here:

1) Single registers are referred to by their letter eg b. Register pairs are named in alphabetical order eg bc.

2) A displacement, d, is taken to be positive if it lies in the range 0 to 127 and negative if it lies between 128 and 255. Larger or smaller numbers are not allowed.

   The negative value is calculated by subtracting d from 256. For example the unconditional relative jump instruction:

   jr d

   causes a jump forward 8 bytes if d = 8 and a jump backwards 8 bytes if d = 248 (= 256 – 8). Remember when calculating a displacement that a jump is made from the address of the first byte following the instruction.

3) A single byte number, n, lies in the range 0 to 255 inclusive.

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

5) nn in brackets — viz (nn) — means "the contents of the location at address nn", whereas nn without brackets means "the number nn". Thus

   ld hl, (23627)

   means load the hl register pair with the contents of locations 23627 and 23628 whereas:

   ld hl, 23627

   means load hl with the number 23627. Similarily (hl) means "the contents of the location at the address held in hl" whereas hl without brackets means "the number in hl".

6) The destination of the result of an instruction is always given first. For example:

   add a,b

   means "add the contents of b to the contents of a and leave the result in a.

Glossary of Machine Code Instructions

This section presents a summary of most of the Z80A instruction set. Some of the more specialised instructions for dealing with interrupts etc have been omitted.

No Operation nop

This is the simplest instruction and as its name implies the processor does nothing when it is encountered. It can be very useful when debugging a routine because it can be substituted temporarily for a suspect instruction without altering the functioning of the remainder of the routine. It can also be used to plug gaps introduced when making small alterations to existing programs or to cause a delay in execution particularly if it is incorporated into a suitable loop. The decimal code is 0.

Load ld

Load instructions are used to move one byte or two bytes between registers and between registers and memory. There are more than one hundred

different load instructions which is more than any other single class. They fall into eight groups:

1) 8 bit register to register.
   The contents of any of the registers a,b,c,d,e,h, or I can be copied to one another.

2) 8 bit memory to register.
   (hl), (ix + d) or (iy + d) can be copied to any of the registers a,b,c,d,e,h or l. (bc), (de) or (nn) can be copied to a.

3) 8 bit register to memory.
   a, b, c, d, e, h or l can be copied to (hl), (ix + d) or (iy + d). a can be copied to (bc), (de) or (nn).

4) 8 bit register to memory immediate.
   An immediate is a number read from the program itself rather than from a register or from another address in memory. A number, n, can be loaded into a, b, c, d, e, h, l, (hl), (ix + d) or (iy + d).

5) 16 bit register to register.
   The contents of hl, ix or iy can be copied to sp.

6) 16 bit memory to register.
   (nn) can be copied to bc, de, hl, ix, iy or sp.

7) 16 bit register to memory.
   bc, de, hl, ix, iy or sp can be copied to (nn).

8) 16 bit register immediate.
   nn can be loaded into bc, de, hl, ix, iy or sp.

Push and Pop push, pop

A push instruction copies the contents of a named 16 bit register to the stack and decrements the stack pointer twice. A pop instruction does the reverse so the two instructions can be used to save register values and re-load them later in the program. The register pairs af, bc, de, hl, ix and iy can each be pushed and popped.

Exchange ex

Exchanges can be made between hl and de, hl and (sp), ix and (sp), iy and (sp), af and af' and between bcdehl and bcdehl' (a single instruction swaps all six 8 bit registers).

8 Bit Add and Subtract add, sub, etc

a, b, c, d, e, h, l, (hl), n, (ix + d) and (iy + d) can be added or subtracted to or from the a register with or without the carry flag. Instructions involving the carry flag end in c.

8 Bit And, Or and Xor and, etc

a, b, c, d, e, h, l, (hl), n, (ix + d) and (iy + d) can be combined with the a register using any of the three logical operators. And sets each bit in the result which was set in both sources; Or sets each bit which was set in either or both sources and Xor sets each bit which was set in one or other source but not those which were set in both.

Compare cp

Compare is like subtract except that only the flags and not the contents of a are affected. a, b, c, d, e, h, l, (hl), n, (ix + d) and (iy + d) can be compared with the accumulator.

8 Bit Increment and Decrement inc, dec

a, b, c, d, e, h, l, (hl), (ix + d) and (iy + d) can be incremented or decremented.

16 Bit Increment and Decrement inc, dec

bc, de, hl, ix, iy and sp can be incremented or decremented.

16 Bit Add and Subtract add, sub, etc

bc, de, hl, ix can be added with or without carry or subtracted with carry only to or from hl. bc, de, sp, ix can be added without carry to ix. bc, de, sp and iy can be added without carry to iy.

Jump, Call and Return

The flag register, f, contains a carry flag, c, a parity flag, p, which is set if a result is even parity, a sign flag, s, which is set if a result is negative, an overflow flag, v, which is set on overflow, and a zero flag, z, which is set on a zero result. These flags can be used to control jumps, subroutine calls and subroutine returns.

1) Jump jp or jr

The following jumps to address nn are possible:
absolute jump (jp); jump on zero or not zero (jp z) and (jp nz); jump on carry or not carry (jp c and jp nc); jump on positive or negative (jp p and jp m); jump on p/v = 1 or p/v = 0 (jp pe and jp po).

The following relative jumps to an address d relative to the current position are available where d is interpreted as lying in the range −128 to 127: absolute relative jump (jr); relative jump on zero or not zero (jr z and jr nz); relative jump on carry or not carry (jr c and jr nc).

Jumps can also be made to the addresses held in hl, ix or iy (jp (hl), jp (ix), jp (iy)). The djnz instruction decrements the b register and jumps to d if b is non zero.

2) Call call

This instruction serves a similar function to the Basic GOSUB command. If the call condition is met then the program transfers to the instruction held in address nn. The following calls may be made: absolute call (call); call on zero or not zero (call z and call nz); call on carry or not carry (call c and call nc); call on positive or negative (call p and call m); call on p/v = 1 or p/v = 0 (call pe and call po).

3) Return ret

This instruction serves a similar function to the Basic RETURN command. Return conditions are available to match each call condition and returns can also be made from the interrupt and the non-maskable interrupt. (reti and retn).

Bit Instructions

The eight bits in each register are numbered from 0 to 7 from right to left. Each of the following operations can be performed on the a, b, c, d, e, h, l registers and on (hl), (ix + d) and (iy + d).

1) Bit Test bit

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

2) Bit Set set

Any bit can be set.

3) Bit Reset res

Any bit can be reset.

4) Rotate Left rl

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

5) Rotate Right rr

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

6) Rotate Left Circular rlc

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

7) Rotate Right Circular rrc

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

8) Shift Left Arithmetic sla

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

9) Shift Right Arithmetic sra

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

10) Shift Right Logical srl

As shift right arithmetic but with bit 7 reset.

Rotate Left Digit rld

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

Rotate Right Digit rrd

Bits 0 to 3 of a are copied to bits 4 to 7 of (hl); bits 4 to 7 of (hl) are copied to bits 0 to 3 of (hl); bits 0 to3 of (hl) are copied to bits 0 to 3 of a.

Accumulator Operations

1) Complement a cpl

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

2) Negate a neg

Complement a and add one.

3) Complement carry — cpl
Sets the carry flag if it is reset, resets it otherwise.

4) Set Carry — scf
Sets the carry flag.

5) Decimal adjust — daa
Corrects a after bcd addition and subtraction.

Restart

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

Block Handling

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

1) Load and increment — ldi
Move one byte from (hl) to (de). Increment hl and de and decrement bc.

2) Load, increment and repeat — ldir
Move one byte from (hl) to (de). Increment hl and de and decrement bc. Repeat until bc = 0

3) Load and decrement — ldd
Move one byte from (hl) to (de) and decrement hl, de and bc.

4) Load, decrement and repeat — lddr
Move one byte from (hl) to (de) and decrement hl, de and bc. Repeat until bc = 0.

5) Compare and increment — cpi
Compare a and (hl). Increment hl and decrement bc.

6) Compare, increment and repeat — cpir
Compare a and (hl). Increment hl and decrement bc. Repeat until a = (hl) or bc = 0.

7) Compare and decrement — cpd
Compare a and (hl). Decrement hl and bc.

8) Compare, decrement and repeat — cpdr
Compare a and (hl). Decrement hl and bc. Repeat until a = (hl) or bc = 0.

Section B

4. INTRODUCTION

The 40 machine code routines in Section B are listed in a standard format for ease of use. This introduction explains the format and presents a BASIC program which can be used to load the routines into memory.

Length:

This is the length in bytes of the routine.

Number of variables:

The execution of some of the routines can be controlled by altering the values one or more variables passed to the routine via the printer buffer.

Check sum:

Each routine is presented as a sequence of positive whole numbers to be POKEd into successive locations in memory. The check sum (ie the sum of all the numbers forming the routine) is given so that the user can ensure that he has loaded the routine correctly.

Operation:

A brief explanation is given of the task performed by the routine.

Variables:

The names, length and location in the printer buffer of each variable are defined. A variable which is one byte long must be a positive whole number between 0 and 255 inclusive and is passed from BASIC or from the keyboard by using:

POKE *location, value*

A two byte variable is passed using two commands:

POKE *location, value*—256*INT (*value*/256)

POKE *location* + 1, INT (*value*/256)

The locations used are in the printer buffer.

Call:

Routines are called using the USR function which must be incorporated into a command. If the machine code routine does not pass a value back to BASIC on completion then the RAND command is recommended as in:

RAND USR address

If the value in the bc register pair is to be returned then either:

LET A = USR address

or

PRINT USR address

is recommended depending on whether the value returned is to be stored in a BASIC variable or PRINTed on the screen.

Error Checks:
The checks made by the routine for illogical or conflicting variable values etc are explained.

Comments:
Simple variants on the main routines are explained.

Machine Code Listing:
The routine is presented in assembly language with the absolute form in the third column headed "Numbers to be entered". To load the routine the numbers in the third column are POKEd in sequence into memory. All the numbers are in decimal.

How it works:
The mode of operation of the routine is explained with references to the machine code listing.

Machine Code Loader

Almost all the machine code routines in this volume are *relocatable* meaning that they will function correctly no matter where in RAM they are located. If a routine is not relocatable then the comments paragraph explains how it must be altered if it is to be stored at a location other than that intended for it.

We have seen in Section A, chapter 2 that the Spectrum uses various parts of RAM for different functions and that the area between the locations pointed to by the RAMTOP and UDG system variables is intended for the storage of machine code routines.

Program BP can be used to load, alter and move a machine code routine. With it the user can reset the RAMTOP pointer to give more space for a routine; enter a routine from the keyboard; step forwards or backwards through the routine to correct an error and insert or delete parts of the routine.

When the program is RUN it PRINTS the lowest address at which a routine can be stored, ie one more than RAMTOP, and the amount of space available between that address and the end of RAM.

In the 16K machine the lowest address is 32600 unless the user has altered the RAMTOP system variable. Similarly in the 48K machine the lowest address is normally 65368.

The 168 bytes at the end of RAM are normally reserved for user defined graphics characters but the program allows the user to overwrite this area if he wishes. Alternatively he can choose a new lowest possible address which the program then puts into the RAMTOP pointer using the CLEAR command. The program will not accept an address lower than 27000 because the routine would then trespass on the space required by the program itself. The program asks for the address at which the routine is to start. Thus the

user can reserve space for several routines and then load them each separately.

Having given the user an opportunity to change his selection if he is not satisfied, the program PRINTS the main display. Figure BF1 shows the form of the display when the "Screen Invert" routine has been loaded at location 32000. The first column is the address, the second is the contents of the address and the third is the check sum. The "Screen Invert" routine is 18 bytes long and its check sum is 1613. It therefore occupies locations 32000 to 32017 and the check sum for location 32017, ie the sum of the contents of locations 32000 to 32017, is 1613.

When the main display is shown the user's attention is drawn to one location because the decimal contents FLASH. It is called the *current* location and initally it is the selected start address of the routine. The user enters a whole number between 0 and 255 inclusive which the program POKEs into the current location and then the following address becomes the current location. In this way an entire routine can be POKEd into place, the main display being updated, and scrolled if necessary, at each step.

The user may choose not to enter a number but to select an option from those summarised in table BT1 instead. These facilities allow corrections to be made.

Code	*Option*
b	Move the current location backwards by one address.
b *number*	Move the current location backwards by *number* addresses.
f	Move the current location forwards by one address.
f *number*	Move the current locations forwards by *number* address.
i *number*	Insert *number* bytes each containing zero at the current location.
d *number*	Delete *number* bytes at the current location.
t	Terminate program.

Table BT1. *Options available for editing machine code.*

Program BP. *Machine Code Loader*

```
 100 GO SUB 8100
 200 REM *****Calculate memory a
vailble
 210 LET min=1+PEEK 23730+256*PE
EK 23731
 220 LET p=PEEK 23732+256*PEEK 2
3733
 230 LET t=p-min+1
 400 REM *****Get start address
 410 PRINT "Lowest possible star
t = ";min,,"Maximum space availl
ble = ";t
 420 INPUT "Do you wish to chang
e the loweststart address (Y or
N) ? ";z$
```

```
430 IF z$="Y" OR z$="y" THEN GO
TO 7000
440 INPUT "Enter address at whi
ch to start loading machine code
";a
450 IF a<min OR a>p THEN BEEP .
2,24: GO TO 440
500 GO SUB 8100
510 LET t=t-a+min
520 PRINT "You can use up to ";
t;" bytes",,,
530 LET u=PEEK 23675+256*PEEK 2
3676
540 IF a<u AND u<p THEN PRINT "
If you use more than ";u-a;" byt
es, you will overwrite the user
defined graphics area."
550 IF a>=u THEN PRINT "You wil
l overwrite the user defined gra
phics area."
560 INPUT "Is that OK (Y or N)
? ";z$
570 IF z$="N" OR z$="n" THEN GO
TO 7000
580 IF z$<>"Y" AND z$<>"y" THEN
BEEP .2,24: GO TO 560
700 REM *****Go ahead and load
710 LET l=a
750 GO SUB 8200
760 INPUT "Enter number,b,f,i,d
or t ";z$
770 IF z$="" THEN BEEP .2,24: G
O TO 760
780 LET a$=CHR$ (CODE z$(1)-32*
(z$(1)>"E"))
790 GO TO 600+200*(a$="B")+300*
(a$="F")+400*(a$="I")+500*(a$="D
")+500*(a$="T")
800 LET x=VAL z$
810 IF l>p THEN BEEP .2,24: GO
TO 750
820 IF x<0 OR x>255 OR x<>INT x
THEN BEEP .2,24: GO TO 750
830 POKE l,x
840 LET l=l+1
850 GO TO 740
1000 REM *****Move forwards
1010 LET l=l-1
1020 IF LEN z$>1 THEN LET l=l+1-
VAL z$(2 TO )
1030 IF l<a THEN LET l=a
1040 GO TO 740
1100 REM *****Move backwards
1110 LET l=l+1
1120 IF LEN z$>1 THEN LET l=l-1+
VAL z$(2 TO )
1130 IF l>p THEN LET l=p
1140 GO TO 740
1200 REM *****Insert
1210 IF LEN z$=1 THEN LET n=1: G
O TO 1225
1220 LET n=VAL z$(2 TO ): IF n<1
```



```
 OR n>p-l OR n<>INT n THEN BEEP
.2,24: GO TO 740
1225 CLS : GO SUB 8100: PRINT TA
B 6;"Inserting in progress"
1230 FOR j=p TO l+n STEP -1
1240 POKE j,PEEK (j-n)
1250 NEXT j
1260 FOR j=l TO l+n-1
1270 POKE j,0
1280 NEXT j
1290 GO TO 740
1300 REM *****Delete
1310 IF LEN z$=1 THEN LET n=1: G
O TO 1330
1320 LET n=VAL z$(2 TO ): IF n<1
 OR n>p-l OR n<>INT n THEN BEEP
.2,24: GO TO 740
1330 IF n<0 OR n>p-l THEN BEEP .
2,24: GO TO 1320
1340 CLS : GO SUB 8100: PRINT TA
B 6;"Deleting in progress"
1350 FOR j=l TO p-n
1360 POKE j,PEEK (j+n)
1370 NEXT j
1380 GO TO 740
1400 STOP
1401 PRINT AT 21,7;"Program term
inated"
1410 STOP
7000 REM *****Reset RAMTOP
7010 INPUT "Enter new start addr
ess ";a
7020 IF a<27000 OR a>p THEN BEEP
.2,24: GO TO 7010
7030 CLEAR a-1
7040 RUN
7999 STOP
8100 CLS
8110 PRINT TAB 6;"Machine code l
oader",,,
8120 RETURN
8200 REM *****Print memory
8210 GO SUB 8100
8220 PRINT "Address    Decimal
Check sum"
8230 LET c=0
8240 LET s=l-8: IF s<a THEN LET
s=a: GO TO 8280
8250 FOR j=a TO s-1
8260 LET c=c+PEEK j
8270 NEXT j
8280 LET f=s+17: IF f>p THEN LET
 f=p
8290 FOR j=s TO f
8300 LET c=c+PEEK j
8310 PRINT AT j-s+3,1;j;TAB 12;P
EEK j;TAB 22;c
8320 NEXT j
8400 LET pos=l-s+3
8410 PRINT AT pos,12; FLASH 1;PE
EK l
8420 RETURN
```

Machine code loader

Address	Decimal	Check sum
32000	33	33
32001	0	33
32002	64	97
32003	1	98
32004	0	98
32005	24	122
32006	22	144
32007	255	399
32008	122	521
32009	150	671
32010	119	790
32011	35	825
32012	11	836
32013	120	956
32014	177	1133
32015	32	1165
32016	247	1412
32017	201	1613

Figure BF1. *The display produced by the Machine Code loader when the Screen Invert routine has been loaded at location 32000.*

5. SCROLL ROUTINES

Scroll Attributes Left

Length: 23
Number of Variables: 1
Check sum: 1574

Operation

This routine scrolls the attributes of all the characters on the screen left by one character.

Variables

Name	*Length*	*Location*	*Comment*
new attr	1	23296	The attribute to enter the rightmost column

Call

RAND USR address

Error Checks

None

Comments

This routine is useful for highlighting areas of text and graphics. To scroll only the top 22 lines the 24* should be changed to 22.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22528	33 0 88
	ld a, (23296)	58 0 91
	ld c, 24	14 24*
next line	ld b, 31	6 31
next char	inc hl	35
	ld e, (hl)	94
	dec hl	43
	ld (hl),e	115
	inc hl	35
	djnz, next char	16 249
	ld (hl),a	119
	inc hl	35
	dec c	13
	jr nz, next line	32 242
	ret	201

How it works

The hl register pair is loaded with the address of the attributes area. The accumulator is loaded with the value of the attribute to be entered in the right hand column. The c register is loaded with the number of lines to be scrolled, so that it can be used as a line counter. The b register is set to one less than the number of characters per line, to be used as a counter.

hl is incremented to point to the next attribute and this is loaded into the e register. hl is decremented and is then POKEd with the value in e. hl is incremented again to point to the next attribute. The b register is decremented, and if it does not hold zero a jump is made back to 'next char'. hl now points to the right hand column, and this is POKEd with the value in the accumulator. hl is incremented to point to the start of the next line. The line counter in the c register is decremented. If the resultant value is not zero the routine loops back to 'next line'.

The routine then returns to BASIC.

Scroll Attributes Right

Length: 23
Number of Variables: 1
Check sum: 1847

Operation

This routine scrolls the attributes of all the characters on the screen right by one character.

Variables

Name	*Length*	*Location*	*Comment*
new attr	1	23296	The attribute to enter the left-most column.

Call

RAND USR address

Error Checks

None

Comments

This routine is useful for highlighting areas of text and graphics. To scroll only the top 22 lines change the 24* to 22.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 23295	33 255 90
	ld a, (23296)	58 0 91
	ld c, 24	14 24*
next line	ld b, 31	6 31
next char	dec hl	43
	ld e, (hl)	94
	inc hl	35
	ld (hl),e	115
	dec hl	43
	djnz, next char	16 249
	ld (hl),a	119
	dec hl	43
	dec c	13
	jr nz, next line	32 242
	ret	201

How it works

The hl register pair is loaded with the address of the last byte of the attribute area. The accumulator is loaded with the value of the attribute to enter the left hand column. The c register is loaded with the number of lines to be scrolled, so that this can be used as a line counter. The b register is set to one less than the number of characters per line, for use as a counter.

hl is decremented to point to the next attribute. The value of this attribute is loaded into the e register. hl is incremented, and is then POKEd with the value in the e register. hl is decremented again to point to the next attribute. The counter in the b register is decremented, and if this does not hold zero the routine loops back to 'next char'.

hl now points to the leftmost column, and this is POKEd with the value in the accumulator. hl is decremented to point to the right end of the next line. The line counter is decremented, and if this does not hold zero the routine loops back to 'next line'.

The routine then returns to BASIC.

Scroll Attributes Up

Length: 21
Number of Variables: 1
Check sum: 1591

Operation

This routine scrolls the attributes of all the characters on the screen upwards by one character.

Variables

Name	*Length*	*Location*	*Comments*
new attr	1	23296	The attribute to enter the bottom line.

Call

RAND USR address

Error Checks

None

Comments

This routine is useful for highlighting areas of text or graphics. To scroll the top 22 lines only, change the 224* to 160.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22560	33 32 88
	ld de, 22528	17 0 88
	ld bc, 736	1 224* 2
	ldir	237 176
	ld a, (23296)	58 0 91
	ld b, 32	6 32
next char	ld (de),a	18
	inc de	19
	djnz next char	16 252
	ret	201

How it works

hl is loaded with the address of the second line of attributes, de is loaded with the address of the first line and bc is loaded with the number of bytes to be moved.

The bc bytes starting at hl are copied to de, using the 'ldir' instruction. This results in de pointing to the bottom line of attributes. The accumulator is loaded with the code of the attribute to be entered into the bottom line. The b register is then loaded with the number of characters in one line, to be used as a counter.

de is POKEd with the value in the accumulator, and then incremented to point to the next byte. The counter is decremented, and if it does not hold zero the routine loops back to 'next char'. The routine then returns to BASIC.

Scroll Attributes Down

Length: 21

Number of Variables: 1

Check sum: 2057

Operation

This routine scrolls the attributes of all the characters on the screen downwards by one character.

Variables

Name	*Length*	*Location*	*Comments*
new attr	1	23296	The attribute to enter the top line

Call

RAND USR address

Error Checks

None

Comments

This routine is useful for highlighting areas of text and graphics. To scroll only the top 22 lines the following changes must be made:

223* to 159
255** to 191
224*** to 160

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 23263	33 223* 90
	ld de, 23295	17 255** 90
	ld bc, 736	1 224*** 2
	lddr	237 184
	ld a, (23296)	58 0 91
	ld b, 32	6 32
next char	ld (de),a	18
	dec de	27
	djnz next char	16 252
	ret	201

How it works

The hl register pair is loaded with the address of the last attribute on the 23rd line. de is loaded with the address of the last attribute on the 24th line. bc is loaded with the number of bytes to be moved. Then the 'lddr' instruction moves the bc bytes ending at hl so that they end at de. This results in de holding the address of the last attribute on the first line.

The accumulator is then loaded with the value of the attribute to enter the top line. The b register is loaded with the number of bytes in the top line, to be used as a counter. de is POKEd with the value in the accumulator, and de is decremented to point to the next byte. The counter is decremented, and if it does not hold zero a jump is made to 'next char'.

The routine then returns to BASIC.

Left Scroll by One Character

Length: 21
Number of Variables: O
Check Sum: 1745

Operation
This routine scrolls the contents of the display file one character to the left.

Call
RAND USR address

Error Checks
None

Comments
This routine is useful when using the screen as a 'window' showing just a small area of a larger display area. The 'window' being moved using scroll routines.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 16384	33 0 64
	ld a,l	85
	ld a,192	62 192
next line	ld b,31	6 31
next byte	inc hl	35
	ld e, (hl)	94
	dec hl	43
	ld (hl),e	115
	inc hl	35
	djnz next byte	16 249
	ld (hl),d	114
	inc hl	35
	dec a	61
	jr nz, next line	32 242
	ret	201

How it works
The hl register pair is loaded with the address of the display file, and the d register is set to zero. The accumulator is loaded with the number of lines on the scrren. The b register is set to one less than the number of characters per line, as this is the number of bytes to be copied.

hl is incremented to point to the next byte, and the e register is loaded with its value. hl is decremented and POKEd with the value in e. hl is

incremented to address the next byte, and the counter in the b register is decremented. If this does not hold zero the routine loops back to 'next byte'.

If the b register holds zero, the last byte of the line has been copied, and hl points to the right most byte. This is then POKEd with zero, and hl incremented to point to the next line. The line counter in the accumulator is decremented and if this does not hold zero a jump is made to 'next line'.

The routine then returns to BASIC.

Right Scroll by One Character

Length: 22
Number of Variables: O
Check sum: 1976

Operation
This routine scrolls the contents of the display file one character to the right.

Call
RAND USR address

Error Checks
None

Comments
This routine is useful when using the screen as a 'window' showing just a small area of a larger display area. The 'window' being moved using scroll routines.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22527	33 255 87
	ld d, O	22 O
	ld a, 192	62 192
next line	ld b, 31	6 31
next byte	dec hl	43
	ld e, (hl)	94
	inc hl	35
	ld (hl),e	115
	dec hl	43
	djnz next byte	16 249
	ld (hl),d	114
	dec hl	43
	dec a	61
	jr nz, next line	32 242
	ret	201

How it works

The hl register pair is loaded with the address of the last byte of the display file, and the d register is set to zero. The accumulator is loaded with the number of lines on the screen. The b register is set to one less than the number of characters per line, to be used as a counter.

The hl register pair is decremented to point to the next byte, and its value is loaded into the e register. hl is then incremented and POKEd with the value in e. hl is decremented to point to the next byte, and the counter in the b register is decremented. If the b register does not hold zero the routine loops back to 'next byte'.

If the b register does hold zero, hl points to the leftmost byte of the line. This is then POKEd with zero, and hl is decremented to point to the next line. The counter in the accumulator is then decremented and if this does not hold zero, a jump is made to 'next line'.

The routine then returns to BASIC.

Up Scroll by One Character

Length: 68

Number of Variables: O

Check sum: 6328

Operation

This routine scrolls the contents of the display file upwards by eight pixels.

Call

RAND USR address

Error Checks

None

Comments

None

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 16384	33 0 64
	ld de, 16416	17 32 64
save	push hl	229
	push de	213
	ld c,23	14 23
next line	ld b,32	6 32
copy byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121
	and 7	230 7

	cp 1	254 1
	jr nz, next byte	32 2
	sub a	151
	ld (de),a	18
next byte	inc hl	35
	inc de	19
	djnz copy byte	16 241
	dec c	13
	jr z, restore	40 19
	ld a,c	121
	and 7	230 7
	cp O	254 O
	jr z, next block	40 22
	cp 7	254 7
	jr nz, next line	32 225
	push de	213
	ld de, 1792	17 0 7
	add hl,de	25
	pop de	209
	jr next line	24 217
restore	pop de	209
	pop hl	225
	inc d	20
	inc h	36
	ld a,h	124
	cp 72	254 72
	jr nz, save	32 204
	ret	201
next block	push hl	229
	ld hl, 1792	33 0 7
	add hl,de	25
	ex de,hl	235
	pop hl	225
	jr next line	24 198

How it works

The hl register pair is loaded with the address of the display file, and de is loaded with the address of the byte eight lines down. hl and de are saved on the stack. The c register is loaded with one less than the number of 'PRINT lines' on the screen. The b register is loaded with the number of bytes in one line of the display, to be used as a counter.

The accumulator is loaded with the byte addressed by de and this is POKEd into hl. The accumulator is loaded with the contents of the c register and if this holds, 1,9 or 17 then de is POKEd with zero. hl and de are incremented to point to the next bytes, the counter in the b register is decremented, and if this does not hold zero a jump is made to 'copy byte'.

The line counter in the c register is then decremented. If this holds zero a jump is made to 'restore'. If c holds 8 or 16 then a jump is made to 'next block'. If c does not hold 7 or 15 then the routine loops to 'next line'. 1792 is added to hl, so that hl points to the next block of the screen. The routine then jumps to 'next line'.

At 'restore' de and hl are retrieved from the stack, and 256 is added to each. Thus, de and hl point one line below the position that they held on the previous loop. If hl holds 18432 the routine returns to BASIC, otherwise a jump is made to 'save'. At 'next block', 1792 is added to de so that de points to the next block of the screen. The routine then loops to 'next line'.

Down Scroll by One Character

Length: 73
Number of Variables: O
Check sum: 7987

Operation
This routine scrolls the contents of the display file downwards by eight pixels.

Call
RAND USR address

Error Checks
None

Comments
None

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22527	33 255 87
	ld de, 22495	17 223 87
save	push hl	229
	push de	213
	ld c, 23	14 23
next line	ld b,32	6 32
copy byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121
	and 7	230 7
	cp 1	254 1
	jr nz, next byte	32 2
	sub a	151
	ld (de),a	18

next byte	dec hl	43
	dec de	27
	djnz copy byte	16 241
	dec c	13
	jr z, restore	40 21
	ld a,c	121
	and 7	230 7
	cp O	254 O
	jr z, next block	40 24
	cp 7	254 7
	jr nz, next line	32 225
	push de	213
	ld de, 1792	17 0 7
	and a	167
	sbc hl,de	237 82
	pop de	209
	jr next line	24 215
restore	pop de	209
	pop hl	225
	dec d	21
	dec h	37
	ld a,h	124
	cp 79	254 79
	ret z	200
	jr save	24 201
next block	push hl	229
	ld hl, 1792	33 0 7
	ex de, hl	235
	and a	167
	sbc hl,de	237 82
	ex de, hl	235
	pop hl	225
	jr next line	24 193

How it works
The hl register pair is loaded with the address of the last byte of the display file, and de is loaded with the address of the byte eight lines up. hl and de are saved on the stack. The c register is then loaded with one less than the number of 'PRINT lines' on the screen. The b register is loaded with the number of bytes in one line of the display, to be used as a counter.

The accumulator is loaded with the byte addressed by de, and this is POKEd into hl. The accumulator is loaded with the contents of the c register, and if this holds 1,9 or 17 then de is POKEd with zero. hl and de are then decremented to point to the next bytes of the display. The counter in the b register is decremented and if this does not hold zero a jump is made to 'copy byte'.

The line counter in the c register is decremented, and if this holds zero a jump is made to 'restore'. If c holds 8 or 16 then a jump is made to 'next block'. If c does not hold 7 or 15 the routine loops to 'next line'. 1792 is then subtracted from hl, so that hl points to the next block of the screen. The routine jumps to 'next line'.

At 'restore' de and hl are retrieved from the stack, and 256 is subtracted from both. Thus, de and hl point one line above the position that they held on the previous loop. If hl holds 20479 the routine returns to BASIC, otherwise a jump is made to 'save'. At 'next block' 1792 is subtracted from de, so that de points to the next block of the screen. The routine then loops to 'next line'.

Left Scroll by One Pixel

Length: 17
Number of Variables: O
Check sum: 1828

Operation
This routine scrolls the contents of the display file one pixel to the left.

Call
RAND USR address

Error Checks
None

Comments
This routine gives a smoother movement than left scroll by one character but eight calls are required to move the display by one full character.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22527	33 255 87
	ld c, 192	14 192
next line	ld b, 32	6 32
	or a	183
next byte	rl (hl)	203 22
	dec hl	43
	djnz next byte	16 251
	dec c	13
	jr nz, next line	32 245
	ret	201

How it works
The hl register pair is loaded with the address of the last byte of the display file, and the c register is loaded with the number of lines in the display file to be used as a line counter. The b register is loaded with the number of bytes in one line, for use as a counter. The carry flag is then set to zero.

The byte addressed by hl is then rotated one bit to the left, the carry flag being copied into the rightmost bit, and the leftmost bit being copied into the carry flag. The hl register pair is decremented to point to the next byte and the counter in the b register is decremented. If this does not hold zero the routine loops back to 'next byte'. The line number is decremented, and if this is not equal to zero the routine jumps back to 'next line'.

The routine then returns to BASIC.

Right Scroll by One Pixel

Length: 17
Number of Variables: O
Check sum: 1550

Operation
This routine scrolls the contents of the display file one pixel to the right.

Call
RAND USR address

Error Checks
None

Comments
This routine gives a smoother movement than Right Scroll by One Character but eight calls are required to move the display by one full character.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 16384	33 0 64
	ld c, 192	14 192
next line	ld b, 32	6 32
	or n	183
next byte	rr (hl)	203 30
	inc hl	35
	djnz next byte	16 251
	dec c	13
	jr nz, next line	32 245
	ret	201

How it works

The hl register pair is loaded with the address of the display file, and the c register is loaded with the number of lines in the display to be used as a line counter. The b register is loaded with the number of bytes in one line, to be used as a counter. The carry flag is then set to zero. The byte addressed by hl is then rotated one bit to the right, the carry flag being copied into the leftmost bit, and the rightmost bit being copied into the carry flag. The hl register pair is incremented to point to the next byte and the counter in the b register is then decremented. If this does not hold zero the routine loops back to 'next byte'. The line counter is decremented, and if this is not equal to zero the routine jumps back to 'next line'.

The routine then returns to BASIC.

Up Scroll by One Pixel

Length: 91
Number of Variables: O
Check sum: 9228

Operation

This routine scrolls the contents of the display file upwards by one pixel.

Call

RAND USR address

Error Checks

None

Comments

None

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 16384	33 0 64
	ld de, 16640	17 0 65
	ld c, 192	14 192
next line	ld b, 32	6 32
next byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121
	cp 2	254 2
	jr nz, next byte	32 2
	sub a	151
	ld (de),a	18

next byte	inc de	19
	inc hl	35
	djnz copy byte	16 243
	push de	213
	ld de, 224	17 224 O
	add hl,de	25
	ex (sp),hl	227
	add hl,de	25
	ex de,hl	235
	pop hl	225
	dec c	13
	ld a,c	121
	and 7	230 7
	cp O	254 O
	jr nz, subtract	32 10
	push de	213
	ld de, 2016	17 224 7
	and a	167
	sbc hl,de	237 82
	pop de	209
	jr next block	24 14
subtract	cp 1	254 1
	jr nz, next block	32 10
	push hl	229
	ex de,hl	235
	ld de, 2016	17 224 7
	and a	167
	sbc hl,de	237 82
	ex de,hl	235
	pop hl	225
next block	ld a,c	121
	and 63	230 63
	cp O	254 O
	jr nz, add	32 6
	ld a,7	62 7
	add a,h	132
	ld h,a	103
	jr next line	24 187
add	cp 1	254 1
	jr nz, next line	32 183
	ld a,7	62 7
	add a,d	130
	ld d,a	87
	ld a,c	121
	cp 1	254 1
	jr nz, next line	32 174
	ret	201

How it works

The hl register pair is loaded with the address of the display file, and the de register pair is loaded with the address of the first byte of the second line of the display. The c register is loaded with the number of lines in the display. The b register is loaded with the number of bytes in one line, to be used as a counter.

The accumulator is loaded with the byte addressed by de. This is then POKEd into the address in hl. The accumulator is loaded with the contents of the c register. If this contains the value two, de points to the bottom line of the screen, and so this is POKEd with zero. de and hl are then incremented to point to the next bytes. The counter in the b register is then decremented and if it does not hold zero the routine loops to 'copy byte'.

224 is added to both the hl and de register pairs, so that they point to the next line of the display. The line counter, in the c register, is decremented. If the value in c is not a multiple of eight a jump is made to 'subtract'. 2016 is subtracted from hl, and a jump made to 'next block'. This is to point hl at the next set of eight lines.

At 'subtract', if the value (c—1) is not a multiple of eight a jump is made to 'next block', otherwise 2016 is subtracted from de so that de points at the next set of eight lines. At 'next block', if c is a multiple of 64, 1792 is added to hl, and a jump is made to 'next line' so that hl points to the next block of 64 lines. At 'add', if (c—1) is a multiple of 64, 1792 is added to de so that de points to the next block of 64 lines. If c does not hold 1 the routine jumps to 'next line', otherwise the routine returns to BASIC.

Down Scroll by One Pixel

Length: 90
Number of Variables: O
Check sum: 9862

Operation

This routine scrolls the contents of the display file downwards by one pixel.

Call

RAND USR address

Error Checks

None

Comments

None

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22527	33 255 87
	ld de, 22271	17 255 86
	ld c, 192	14 192
next line	ld b, 32	6 32
copy byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121
	cp 2	254 2
	jr nz, next byte	32 2
	sub a	151
	ld (de),a	18
next byte	dec de	27
	dec hl	43
	djnz, copy byte	16 243
	push de	213
	ld de, 224	17 224 O
	and a	167
	sbc hl,de	237 82
	ex (sp), hl	227
	and a	167
	sbc hl,de	237 82
	ex de,hl	235
	pop hl	225
	dec c	13
	ld a,c	121
	and 7	230 7
	cp O	254 O
	jr nz, add	32 8
	push de	213
	ld de, 2016	17 224 7
	add hl,de	25
	pop de	209
	jr next block	24 11
add	cp 1	254 1
	jr nz, next block	32 7
	push hl	229
	ld hl, 2016	33 224 7
	add hl,de	25
	ex de,hl	235
	pop hl	225
next block	ld a,c	121
	and 63	230 63
	cp O	254 O
	jr nz, subtract	32 6

	ld a,h	124
	sub 7	214 7
	ld h,a	103
	jr next line	24 188
subtract	cp 1	254 1
	jr nz, next line	32 184
	ld a,d	122
	sub 7	214 7
	ld d,a	87
	ld a,c	121
	cp 1	254 1
	jr nz, next line	32 175
	ret	201

How it works

The hl register pair is loaded with the address of the last byte of the display file, and the de register pair is loaded with the address of the byte one line above the last byte. The c register is loaded with the number of lines in the display. The b register is loaded with the number of bytes in one line, to be used as a counter.

The accumulator is loaded with the byte addressed by de. This is then POKEd into the address stored in hl. The accumulator is loaded with the contents of the c register. If this contains the value two, de points to the top line of the screen, and so this is POKEd with zero. de and hl are then decremented to point to the next bytes. The counter in the b register is decremented, and if it does not hold zero the routine loops to 'copy byte'.

224 is subtracted from both hl and de, so that they point to the next line of the display. The line counter in the c register, is decremented. If the value in c is not a multiple of eight a jump is made to 'add'. 2016 is then added to hl, and a jump is made to 'next block'. This is to point hl at the next block of eight lines.

At 'add' if the value (c—1) is not a multiple of eight a jump is made to 'next block'. 2016 is then added to de so that de points at the next set of eight lines. At 'next block', if c is a multiple of 64, 1792 is subtracted from hl so that hl points to the next block of 64 lines, and a jump is made to 'next line'. At 'subtract', if (c—1) is a multiple of 64, 1792 is subtracted from de, so that de points to the next block of 64 lines. If c does not hold one, the routine jumps to 'next line', otherwise the routine returns to BASIC.

6. DISPLAY ROUTINES

Merge Pictures

Length: 21
Number of Variables: 1
Check sum: 1709

Operation

This routine merges a picture stored in RAM (using the 'Copy' routine elsewhere in this book) with the current screen display. The attributes are not changed.

Variables

Name	*Length*	*Location*	*Comment*
screen store	2	23296	address in RAM of stored picture

Call

RAND USR address

Error Checks

None

Comments

To merge pictures the routine should be used as listed. However, interesting effects can be produced by replacing 'or (hl) 182' instruction by 'xor (hl) 174' or 'and (hl) 166'.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 16384	33 0 64
	ld de, (23296)	237 91 0 91
	ld bc, 6144	1 0 24
next byte:	ld a, (de)	26
	or (hl)	182
	ld (hl),a	119
	inc hl	35
	inc de	19
	dec bc	11
	ld a,b	120
	or c	177
	jr nz, next byte	32 246
	ret	201

How it works

The hl register pair is loaded with the address of the display file and the de register pair is loaded with the length of the display, so that it can be used as a counter.

The accumulator is loaded with the byte at the address stored in de, and this is logically 'OR'ed (see Glossary) with the next byte of the display file. The resultant value is then loaded back into the display.

hl and de are moved onto the next position, and the counter is decremented. If the counter is not zero the routine then loops back to repeat the process on the next byte.

Screen Invert

Length: 18
Number of Variables: O
Check sum: 1613

Operation

Inverts all of the display file—where a point is on it is turned off, and where a point is off it is turned on.

Call

RAND USR address

Error Checks

None

Comments

This routine can be used in games programs to produce an effective explosion. The effect is increased if this routine is called several times, with some form of sound added.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 16384	33 0 64
	ld bc, 6144	1 0 24
	ld d, 255	22 255
next byte	ld a,d	122
	sub (hl)	150
	ld (hl),a	119
	inc hl	35
	dec bc	11
	ld a,b	120
	or c	177
	jr nz, next byte	32 247
	ret	201

How it works

The hl register pair is loaded with the address of the display file and bc is loaded with its length. The d register is set to 255. Each time the routine loops back to 'next byte' the accumulator is loaded from d. This method is used, rather than the 'ld a, 255' instruction because 'ld a,d' takes approximately half the time taken by the 'ld a, 255' instruction. The value of the byte stored at hl is subtracted from the accumulator, and the result is then loaded back into the same byte, thus inverting it.

hl is incremented to point to the next byte, and the counter, bc, is decremented. If the counter is not zero the routine loops back to 'next byte'. If the counter is zero, the routine returns to BASIC.

Invert Character Vertically

Length: 20
Number of Variables: 1
Check sum: 1757

Operation

This routine inverts a character vertically eg an up-arrow would become a down-arrow and vice versa.

Variables

Name	*Length*	*Location*	*Comment*
chr. start	2	23296	address of character data in RAM

Call

RAND USR address

Error Checks

None

Comments

This routine is useful in games such as 'Minefield' and 'Puckman' because symbols can change direction without using more than one character.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld d,h	84
	ld e,l	93
	ld b,8	6 8
next byte	ld a, (hl)	126
	inc hl	35

	push af	245
	djnz next byte	16 251
	ld b,8	6 8
replace	pop af	241
	ld (de),a	18
	inc de	19
	djnz replace	16 251
	ret	201

How it works

The hl register pair is loaded with the address of the character data in RAM. This is then copied into de. The b register is set to 8 to be used as a counter.

For each byte, the accumulator is loaded with the present value, hl is incremented to point to the next byte, and the accumulator is pushed on to the stack. The counter is decremented, and if it is not zero the routine loops back to repeat the process for the next byte. The b register is then re-loaded with 8 for use as a counter again.

For each byte, the accumulator is popped from the stack. and poked into the address stored in de. de is incremented to point to the next byte and the counter is decremented. If this is not zero the routine loops back to 'replace'. A return is then made to BASIC.

Invert Character Horizontally

Length: 19
Number of Variables: 1
Check sum: 1621

Operation

This routine inverts a character horizontally eg a left-arrow becomes a right-arrow and vice versa.

Variables

Name	*Length*	*Location*	*Comment*
chr. start	2	23296	address of character data in RAM

Call

RAND USR address

Error Checks

None

Comments

None

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld a, 8	62 8
next byte	ld b, 8	6 8
next pixel	rr (hl)	203 30
	rl c	203 17
	djnz next pixel	16 250
	ld (hl),c	113
	inc hl	35
	dec a	61
	jr nz, next byte	32 243
	ret	201

How it works

The hl register pair is loaded with the address of the character data in RAM, and the accumulator is loaded with the number of bytes to be inverted. The b register is loaded with the number of bits in each byte, to be used as a counter.

The byte at the address in hl is rotated to the right so that the right-most bit is copied into the carry flag. The c register is rotated leftwards so that the carry flag is copied into the rightmost bit. The counter stored in the b register is decremented. If the counter is not zero, a jump is made back to 'next pixel'. The inverted byte, which is stored in the c register, is POKEd back to the address that it originally came from.

hl is incremented to point to the next byte, and the accumulator is decremented. If the accumulator does not hold zero a jump is made back to 'next byte'.

A return is then made to BASIC.

Rotate Character Clockwise

Length: 42
Number of Variables: 1
Check sum: 3876

Operation

This routine rotates a character through 90° clockwise eg an up-arrow becomes a right-arrow.

Variables

Name	*Length*	*Location*	*Comment*
chr. start	2	23296	address of character data in RAM

Call
RAND USR address

Error Checks
None

Comments
This routine is useful in games and in serious applications eg labelling graphs.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld e, 128	30 128
next bit	push hl	229
	ld c, O	14 0
	ld b,1	6 1
next byte	ld a, e	123
	and (hl)	166
	cp O	254 0
	jr z, not set	40 3
	ld a,c	121
	add a,b	128
	ld c,a	79
not set	sla b	203 32
	inc hl	35
	jr nc, next byte	48 242
	pop hl	225
	push bc	197
	srl e	203 59
	jr nc, next bit	48 231
	ld de,7	17 7 0
	add hl,de	25
	ld b,8	6 8
replace	pop de	209
	ld (hl),e	115
	dec hl	43
	djnz replace	16 251
	ret	201

How it works
Each character consists of an 8×8 group of pixels, each of which can be turned on ($=1$) or off ($=0$). Consider any bit B_2 of byte B_1 in Figure B1. The data held at the location (B_2, B_1) in the matrix will be

$$\begin{pmatrix} N_1 & N_3 \\ N_2 & N_4 \end{pmatrix}$$

where:
N_1 = the byte at which the pixel (B_2, B_1) will be inserted after rotation.
N_2 = the bit in N_1 at which it will be inserted.
N_3 = the value that the bit currently represents.
N_4 = the value of the bit N_2.

Each byte of the rotated character will be built up one at a time, by adding the values of all the bits N_2 that will be in the new byte.

The hl register is loaded with the address of the first byte of the character in RAM. The e register is loaded with the value of the byte which has bit 7 on and bits 0-6 off ie 128. The hl register is saved on the stack. The c register, to which data will be added giving the new value of the byte being built, is loaded with zero. The b register is loaded with the value of the byte which has bit O on and bits 1-7 off, ie 1.

The accumulator is loaded with the contents of the e register, (N_3). This is 'AND'ed with the byte whose address is stored in hl. If the result is

1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	1
0 1	0 1	0 1	0 1	0 1	0 1	0 1	0 1	
1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	2
1 2	1 2	1 2	1 2	1 2	1 2	1 2	1 2	
1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	3
2 4	2 4	2 4	2 4	2 4	2 4	2 4	2 4	
1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	4
3 8	3 8	3 8	3 8	3 8	3 8	3 8	3 8	Byte (B_1)
1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	5
4 16	4 16	4 16	4 16	4 16	4 16	4 16	4 16	
1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	6
5 32	5 32	5 32	5 32	5 32	5 32	5 32	5 32	
1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	7
6 64	6 64	6 64	6 64	6 64	6 64	6 64	6 64	
1 128	2 64	3 32	4 16	5 8	6 4	7 2	8 1	8
7 128	7 128	7 128	7 128	7 128	7 128	7 128	7 128	
7	6	5	4	3	2	1	0	

Bit (B_2)

Figure B1. *Key to the character rotation routine*

zero a jump is made to 'not set', as the pixel addressed by e and hl is turned off. If it is turned on the accumulator is loaded with the present value of the byte, (N_1). The b register, (N_4), is added to the accumulator and this is loaded into c. The b register is then adjusted to point to the next bit of N_1. hl is increased to point to the next byte, (B_1). If the byte N_1 is not complete the routine loops back to 'next byte'.

hl is retrieved from the stack, to point to the first byte of the character again. bc is saved on the stack, holding the value of the last byte to be completed in c. The e register is adjusted to address the next bit of each byte. If the rotation is not complete a jump is made to 'next bit'.

de is loaded with 7, and this is added to hl so that hl points to the last byte of data. The b register is loaded with the number of bytes to be retrieved from the stack. For each byte, the new value is copied into e, and this is POKEd into hl. hl is decremented to point to the next byte, and the counter in the b register is decremented. If the counter does not hold zero a jump is made to 'replace'.

The routine then returns to BASIC.

Attribute Change

Length: 21
Number of Variables: 2
Check sum: 1952

Operation
This routine alters the attributes of all the characters on the screen in a specified manner eg the ink colour could be changed, the whole screen could be set to flash etc.

Variables

Name	*Length*	*Location*	*Comment*
data saved	1	23296	bits of attribute not to be altered
new data	1	23297	new bits to be inserted into attribute

Call
RAND USR address

Error Checks
None

Comments
Individual bits of the attributes, of each character, can be changed, by using the machine code instructions 'and' and 'or'.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22528	33 0 88
	ld bc, 768	1 0 3
	ld de, (23296)	237 91 0 91
next byte	ld a, (hl)	126
	and e	163
	or d	178
	ld (hl), a	119
	inc hl	35
	dec bc	11
	ld a,b	120
	or c	177
	jr nz, next byte	32 246
	ret	201

How it works
The hl register pair is loaded with the address of the attributes area, and the bc register pair is loaded with the number of characters in the display. The d register is loaded with the value 'new data', and the e register is loaded with 'data saved'.

The accumulator is loaded with the byte addressed by hl, and the bits are adjusted according to the values of the d and e registers. The result is POKEd back into hl. hl is incremented to point to the next byte, and the counter in bc is decremented. If bc does not hold zero the routine loops to 'next byte'.

The routine then returns to BASIC.

Attribute Swap

Length: 22
Number of Variables: 2
Check sum: 1825

Operation
This routine searches the attributes area for a certain value, and replaces every occurrence by another value.

Variables

Name	*Length*	*Location*	*Comment*
old value	1	23296	Value of byte to be replaced
new value	1	23297	New value of replaced byte

Call
RAND USR address

Error Checks
None

Comments
This routine is useful for highlighting areas of text and graphical characters.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, 22528	33 0 88
	ld bc, 768	1 0 3
	ld de, (23296)	237 91 0 91
next byte	ld a, (hl)	126
	cp e	187
	jr nz, no change	32 1
	ld (hl), d	114
no change	inc hl	35
	dec bc	11
	ld a,b	120
	or c	177
	jr nz, next byte	32 245
	ret	201

How it works
The hl register pair is loaded with the address of the attributes area, and bc is loaded with the number of characters on the screen. The e register is loaded with the 'old value', and the d register is loaded with the 'new value'.

The accumulator is loaded with the byte addressed by the hl register pair. If the accumulator holds the value of the e register the byte addressed by hl is POKEd with the contents of the d register. hl is then incremented to point to the next byte, and the counter in bc is decremented. If bc does not hold zero, a jump is made to 'next byte'.

The routine then returns to BASIC.

Region Filling

Length: 263
Number of Variables: 2
Check sum: 26647

Operation
This routine 'shades' an area of the screen bounded by a line of pixels on the edge of the screen.

Variables

Name	*Length*	*Location*	*Comment*
x co-ord	1	23296	x co-ordinate of start position
y co-ord	1	23297	y co-ordinate of start position

Call
RAND USR address

Error Checks
If the y co-ordinate is more than 175, or POINT (x,y) = 1 the routine returns to BASIC immediately.

Comments
This routine is not relocatable, the start address being 31955. To copy this routine to another address use the method given for the 'RENUMBER' routine. If 31955 is used for the start address of this routine and 32218 is used as the start address of 'RENUMBER' they may be held in RAM simultaneously. When shading very irregular shaped regions, a large amount of spare RAM is needed. If this is not available the routine may crash.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld a,h	124
	cp 176	254 176
	ret nc	208
	call subroutine	205 143* 125*
	and (hl)	166
	cp O	254 O
	ret nz	192
	ld bc, 65535	1 255 255
	push bc	197
right	ld hl, (23296)	42 0 91
	call subroutine	205 143* 125*
	and (hl)	166
	cp O	254 O
	jr nz, left	32 9
	ld hl, (23296)	42 0 91
	inc l	44
	ld (23296),hl	34 0 91
	jr nz, right	32 236
left	ld de, O	17 0 0
	ld hl, (23296)	42 0 91
	dec l	45
	ld (23296),hl	34 0 91

plot	ld hl, (23296)	42 0 91
	push hl	229
	call subroutine	205 143* 125*
	or (hl)	182
	ld (hl),a	119
	pop hl	225
	ld a,h	124
	cp 175	254 175
	jr z, down	40 44
	ld a,e	123
	cp O	254 O
	jr nz, reset	32 16
	inc h	36
	call subroutine	205 143* 125*
	and (hl)	166
	cp O	254 O
	jr nz, reset	32 7
	ld hl, (23296)	42 0 91
	inc h	36
	push hl	229
	ld e,l	30 1
reset	ld hl, (23296)	42 0 91
	ld a,e	123
	cp 1	254 1
	jr nx, down	32 15
	inc h	36
	call subroutine	205 143* 125*
	and (hl)	166
	cp O	254 O
	jr z, down	40 6
	ld e, O	30 0
	jr down	24 2
long jump	jr right	24 167
down	ld hl, (23296)	42 0 91
	ld a,h	124
	cp O	254 O
	jr z, next pixel	40 40
	ld a,d	122
	cp O	254 O
	jr nz, restore	32 16
	dec h	37
	call subroutine	205 143* 125*
	and (hl)	166
	cp O	254 O
	jr nz, restore	32 7
	ld hl, (23296)	42 0 91
	dec h	37

	push hl	229
	ld d,l	22 1
restore	ld a,d	122
	cp 1	254 1
	jr nz, next pixel	32 14
	ld hl, (23296)	42 0 91
	dec h	37
	call subroutine	205 143* 125*
	and (hl)	166
	cp O	254 O
	jr z, next pixel	40 2
	ld d, O	22 O
next pixel	ld hl, (23296)	42 O 91
	ld a,l	125
	cp O	254 O
	jr z, retrieve	40 12
	dec l	45
	ld (23296),hl	34 0 91
	call subroutine	205 143* 125*
	and (hl)	166
	cp O	254 O
	jr z, plot	40 129
retrieve	pop hl	225
	ld (23296),hl	34 0 91
	ld a, 255	62 255
	cp h	188
	jr nz, long jump	32 177
	cp l	189
	jr nz, long jump	32 174
	ret	201
subroutine	push bc	197
	push de	213
	ld a, 175	62 175
	sub h	148
	ld h,a	103
	push hl	229
	and 7	230 7
	add a, 64	198 64
	ld c,a	79
	ld a,h	124
	rra	203 31
	rra	203 31
	rra	203 31
	and 31	230 31
	ld b, a	71
	and 24	230 24

```
        ld d,a          87
        ld a,h          124
        and 192         230 192
        ld e,a          95
        ld h,c          97
        ld a,l          125
        rra             203 31
        rra             203 31
        rra             203 31
        and 31          230 31
        ld l,a          111
        ld a,e          123
        add a,b         128
        sub d           146
        ld e,a          95
        ld d, O         22 O
        push hl         229
        push de         213
        pop hl          225
        add hl,hl       41
        add hl,hl       41
        add hl,hl       41
        add hl,hl       41
        add hl,hl       41
        pop de          209
        add hl,de       25
        pop de          209
        ld a,e          123
        and 7           230 7
        ld b, a         71
        ld a, 8         62 8
        sub b           144
        ld b,a          71
        ld a,l          62 1
rotate  add a,a         135
        djnz rotate     16 253
        rra             203 31
        pop de          209
        pop bc          193
        ret             201
```

How it works

This routine plots horizontal lines of adjacent pixels called 'RUNS' within areas bounded by illuminated pixels. Each RUN is remembered by 'stacking' the co-ordinates of the rightmost pixel of the RUN. Starting from the specified co-ordinates, the routine fills in each RUN, noting the positions of any unfilled RUNS above or below. On completing one RUN,

the last set of co-ordinates noted are retrieved and the corresponding RUN is filled in. The process is repeated until there are no more unfilled RUNS.

Figure B2 illustrates the technique. The squares represent illuminated pixels, x marks the starting position within the area to be shaded and * marks the rightmost pixels of RUNS.

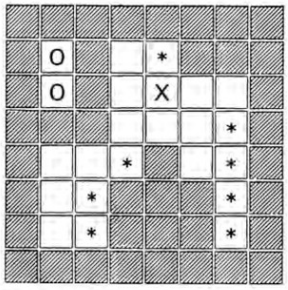


Figure B2. *An illustration of the technique used for filling a region. Grey squares are already illuminated and define the region to be shaded. X is the starting position, * are the starts of RUNS and O remain unshaded.*

The routine shades the horizontal line containing the starting position and saves on the stack the positions of the starts of the RUNS in the lines immediately above and below. It next shades the line above and then the line below noting in the latter case that two more RUNS start on the next line down and so on. Any position within the area to be shaded may be selected as the starting position but note that the two pixels marked with zeros are left untouched because they are separated from the area being shaded.

The h register is loaded with the y co-ordinate specified, and the l register is loaded with the x co-ordinate. If the value of the y co-ordinate is more than 175 the routine returns to BASIC. The 'subroutine' is called returning the address in memory of the bit (x,y). If this bit is 'on' the routine returns to BASIC.

The number 65535 is PUSHed onto the stack to mark the first value saved. Later in the routine, if a number is retrieved from the stack, it is used as a pair of co-ordinates. However, if the number is 65535, a return is made to BASIC as the routine will have finished.

The h register is loaded with the y co-ordinate, and the l register is loaded with the x co-ordinate. The 'subroutine' is called, returning in hl the address of the bit (x,y). If this bit is 'on' a jump is made to 'left'. Otherwise the x co-ordinate is incremented, and a loop to 'right' if x is not equal to 256.

At 'left', de is set to zero. The d and e registers are to be used as flags, d for down and e for up. The x co-ordinate is decremented. The subroutine is called, and the point (x,y) is plotted. If the y co-ordinate is 175 the routine jumps to 'down'. If the 'up flag' is set to one a jump is made to 'reset'. If the bit (x,y + 1) is 'off' the values of x and y + 1 are saved on the stack, and the 'up flag' set to one.

At 'reset', if the 'up flag' is set to zero a jump is made to 'down'. If the bit (x,y + 1) is 'on' the 'up flag' is set to zero. At 'down', if the y co-ordinate is zero a jump is made to 'next pixel'. If the 'down flag' is set to one a jump is made to 'restore'. If the bit (x,y—1) is 'off' the values of x and y—1 are saved on the stack, and the 'down flag' set to one.

At 'restore', if the 'down flag' is set to zero a jump is made to 'next pixel'. If the bit (x,y—1) is 'on' the 'down flag' is set to zero. At 'next pixel', if the x co-ordinate is zero the routine jumps to 'retrieve'. The x co-ordinate is decremented, and if the new bit (x,y) is 'off' a jump is made to 'plot'. At 'retrieve', an x and y co-ordinate are removed from the stack. If x and y both equal 255 then the routine returns to BASIC as the region has been completely filled. Otherwise the routine loops back to 'right'.

The subroutine has to calculate the address of the bit (x,y) in memory. In BASIC this address would be:

$$16384 + \text{INT}(Z/8) + 256 \times (Z - 8 \times \text{INT}(Z/8)) + 32 \times (64 \times \text{INT}(Z/64) + \text{INT}(Z/8) - 8 \times \text{INT}(Z/64))$$

where Z = 175—Y

The bc and de register pairs are saved on the stack. The accumulator is loaded with 175 and the Y co-ordinate is subtracted from this. The result is copied back into the h register. hl is then saved on the stack. The left five bits of the accumulator are set to zero, and 64 is added. The result is copied into the c register. When multiplied by 256 this gives $16384 + 256 \times (Z - 8 \times \text{INT}(Z/8))$. The accumulator is loaded with Z, and this is divided by eight, the result being copied into the b register. This result is INT (Z/8). Setting the rightmost three bits to zero produces the value $8 \times \text{INT}(Z/64)$, this being loaded into the d register.

The accumulator is loaded with Z, and the six rightmost bits are set to zero, producing the value $64 \times \text{INT}(Z/64)$. This is loaded into the e register. The value in the c register is copied into h. The accumulator is loaded with the x co-ordinate, this is divided by eight, and the result is copied into l.

The accumulator is then loaded with the value in e, and the contents of b are added. The value in d is subtracted and the result loaded into de. The hl register pair is saved on the stack, and then loaded with the value in de. This is multiplied by 32, de is retrieved from the stack and added to hl. Thus, hl now holds the address of the bit (x,y).

The accumulator is loaded with the original value of x. Setting the left five bits to zero, produces the value $x - 8 \times \text{INT}(x/8)$. The b register is then loaded with eight minus the value of the accumulator, to be used as

a counter. The accumulator is set to one, and this is multiplied by two b—1 times.

At this point a single bit should be set in the accumulator, which corresponds to the bit (x,y) addressed by hl. de and bc are then retrieved from the stack, and the subroutine then returns to the main routine.

Shape Tables

Length: 196
Number of Variables: 2
Check sum: 20278

Operation

This routine plots a shape of any size on the screen.

Variables

Name	*Length*	*Location*	*Comment*
X start	1	23296	X co-ordinate of first pixel
Y start	1	23297	Y co-ordinate of first pixel

Call

RAND USR address

Error Checks

If A$ does not exist, has zero length, or does not contain any shape information, the routine returns to BASIC immediately. This also happens if Y start is more than 175.

Comments

This is a useful method of storing shapes in memory to be plotted at speed on the screen.

The method for using this routine is:
(i) LET A$ = "shape information"
(ii) POKE 23296, X co-ordinate of first pixel
(iii) POKE 23297, Y co-ordinate of first pixel
(iv) RAND USR address

The shape information is a string of characters, which have the following meanings:

"O"	Plot point
"5"	decrease X co-ordinate
"6"	decrease Y co-ordinate
"7"	increase Y co-ordinate
"8"	increase X co-ordinate

Any other characters are ignored.

The routine includes a 'wrap-round' facility. ie if the X co-ordinate moves off the left of the screen it appears on the right etc.

To change the routine to use a string other than A$, change the 65* to the code of the upper case character of the name of the string.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23627)	42 75 92
next variable	ld a, (hl)	126
	cp 128	254 128
	ret z	200
	bit 7,a	203 127
	jr nz, for next	32 23
	cp 96	254 96
	jr nc, number	48 11
	cp 65	254 65*
	jr z, found	40 35
string	inc hl	35
	ld e, (hl)	94
	inc hl	35
	ld d, (hl)	86
add	add hl,de	25
	jr increase	24 5
number	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
increase	inc hl	35
	jr next variable	24 225
for next	cp 224	254 224
	jr c, next bit	56 5
	ld de, 18	17 18 0
	jr add	24 236
next bit	bit 5,a	203 111
	jr z, string	40 228
next byte	inc hl	35
	bit 7,(hl)	203 126
	jr z, next byte	40 251
	jr number	24 228
found	inc hl	35
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	70
	inc hl	35

	ex de,hl	235
	ld a, (23297)	58 1 91
	cp 176	254 176
	ret nc	208
again	ld hl, (23296)	42 0 91
	ld a,b	120
	or c	177
	ret z	200
	dec bc	11
	ld a, (de)	26
	inc de	19
	cp 48	254 48
	jr nz, not plot	32 78
	push bc	197
	push de	213
	ld a, 175	62 175
	sub h	148
	ld h,a	103
	push hl	229
	and 7	230 7
	add a,64	198 64
	ld c,a	79
	ld a,h	124
	rra	203 31
	rra	203 31
	rra	203 31
	and 31	230 31
	ld b,a	71
	and 24	230 24
	ld d,a	87
	ld a,h	124
	and 192	230 192
	ld e,a	95
	ld h,c	97
	ld a,l	125
	rra	203 31
	rra	203 31
	rra	203 31
	and 31	230 31
	ld l,a	111
	ld a,e	123
	add a,b	128
	sub d	146
	ld e,a	95
	ld d, O	22 O
	push hl	229
	push de	213

```
          pop hl          225
          add hl,hl       41
          add hl,hl       41
          add hl,hl       41
          add hl,hl       41
          add hl,hl       41
          pop de          209
          add hl,de       25
          pop de          209
          ld a,e          123
          and 7           230 7
          ld b,a          71
          ld a, 8         62 8
          sub b           144
          ld b,a          71
          ld a,1          62 1
rotate    add a,a         135
          djnz rotate     16 253
          rra             203 31
          pop de          209
          pop bc          193
          or (hl)         182
          ld (hl),a       119
here      jr again        24 165
not plot  cp 53           254 53
          jr nz, down     32 1
          dec l           45
down      cp 54           254 54
          jr nz, up       32 8
          dec h           37
          ld a,h          124
          cp 255          254 255
          jr nz, save     32 19
          ld h, 175       38 175
up        cp 55           254 55
          jr nz, right    32 8
          inc h           36
          ld a,h          124
          cp 176          254 176
          jr nz, save     32 7
          ld h, O         38 O
right     cp 56           254 56
          jr nz, save     32 1
          inc l           44
save      ld (23296),hl   34 0 91
          jr here         24 215
```


How it works

The address of the string, A$ is found using an adaption of the first section of the 'Instr$' routine.

The length of the string is loaded into bc, and the address of the first character of A$ is loaded into de. The accumulator is set to the value Y start, and if this is more than 175 the routine returns to BASIC. The h register is loaded with the Y co-ordinate, and l is loaded with the X co-ordinate. If the value of the bc register pair is zero the routine then returns to BASIC, because the end of the string has been reached. bc is decremented, to indicate that another character has been operated on. The next character is loaded into the accumulator, and de is incremented to point to the following byte. If the accumulator does not hold 48 a jump is made to 'not plot'. The point (X,Y) is plotted using the 'subroutine' from the "Region Filling" routine. The routine then jumps back to 'again'.

At 'not plot', if the accumulator holds 53, the X co-ordinate is decremented. At 'down' if the accumulator does not hold 54, a jump is made to 'up'. The Y co-ordinate is decremented, and if this then holds —1 the Y co-ordinate is then set to 175.

At 'up', if the accumulator does not hold 55 a jump is made to 'right'. The Y co-ordinate is incremented, and if it is 176 the Y co-ordinate is then set to O. At 'right', if the accumulator holds 56, the X co-ordinate is incremented. At 'save', the X and Y co-ordinates are POKEd into memory, and the routine loops to 'here'.

Screen Magnify and Copy

Length: 335
Number of Variables: 8
Check Sum: 33663

Operation

This routine copies a section of the display to another area on the screen, magnifying the copy in the x or y planes.

Variables

Name	*Length*	*Location*	*Comment*
upper y co-ord	1	23296	y co-ordinate of top row
lower y co-ord	1	23297	y co-ordinate of bottom row
right x co-ord	1	23298	x co-ordinate of rightmost column
left x co-ord	1	23299	x co-ordinate of leftmost column
horizontal scale	1	23300	magnification in x plane
vertical scale	1	23301	magnification in y plane

new left co-ord	1	23302	x co-ordinate of leftmost column of area to be copied to
new lower co-ord	1	23303	y co-ordinate to bottom row of area to be copied to

Call

RAND USR address

Error Checks

The routine returns to BASIC immediately if any of the following conditions are true:

(i) horizontal scale = O
(ii) vertical scale = O
(iii) upper co-ord greater than 175
(iv) new lower co-ord greater than 175
(v) lower y co-ord greater than upper y co-ord
(vi) left x co-ord greater than right x co-ord

However, to keep the routine short, there is no check that ensures that the copied section fits on the screen. If it does not, the routine may 'crash'. The routine also requires a large amount of spare RAM, and if this is not available, the routine may 'crash'.

Comments

This routine is not relocatable, due to the existence of a 'Plot/Point' subroutine. It is located at address 65033, and hence can only be used on machines with 48K of RAM. The routine can be re-positioned in memory, using the procedure given for the 'Renumber' routine. However, if large areas to the screen are to be copied, then a lot of spare RAM is needed, and so the start address should be as high as possible.

If the copied area of the display is to be the same size as the original, the scales should be set to one, to double the size load the scales with two, to triple the size load the scales with three etc.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld ix, 23296	221 33 0 91
	ld a, 175	62 175
	cp (ix + O)	221 190 0
	ret c	216
	cp (ix + 7)	221 190 7
	ret c	216
	sub a	151
	cp (ix + 4)	221 190 4
	ret z	200
	cp (ix + 5)	221 190 5
	ret z	200
	ld hl, (23296)	42 0 91

	ld b, 1	69
	ld a,l	125
	sub h	148
	ret c	216
	ld (23298),a	50 0 91
	ld e,a	95
	ld hl, (23298)	42 2 91
	ld c,l	77
	ld a,l	125
	sub h	148
	ret c	216
	ld (23298),a	50 2 91
	push bc	197
	ld l,a	111
	ld h, O	38 O
	inc hl	35
	push hl	229
	pop bc	193
	inc e	28
add	dec e	29
	jr z, remainder	40 3
	add hl,bc	9
	jr add	24 250
remainder	ld a,l	125
	and 15	230 15
	ld b,a	71
	pop hl	225
	ld c,l	77
	jr nz, save	32 2
full	ld b,16	6 16
save	push hl	229
	call subroutine	205 13* 255*
	and (hl)	166
	jr z, off	40 2
	ld a, 1	62 1
off	pop hl	225
	rra	203 31
	rl e	203 19
	rl d	203 18
	ld a,l	125
	cp (ix + 3)	221 190 3
	jr z, next row	40 6
	dec l	45
next bit	djnz save	16 231
	push de	213
	jr full	24 226

next row	ld l,c	105
	ld a,h	124
	cp (ix + 1)	221 190 1
	jr z, copy	40 3
	dec h	37
	jr next bit	24 241
copy	push de	213
	ld b, O	6 O
	ld h,b	96
	ld l,b	104
reset	ld (23306),hl	34 10 91
	ld a,b	120
	or a	183
	jr nz, retrieve	32 3
	pop de	209
	ld b, 16	6 16
retrieve	sub a	151
	dec b	5
	rr d	203 26
	rr e	203 27
	rl a	203 23
	push de	213
	push bc	197
	push af	245
	ld h,l	38 1
loop	ld l, l	46 1
preserve	ld (23304),hl	34 8 91
	ld a, (23307)	58 11 91
	ld hl, O	33 0 0
	ld de, (23301)	237 91 5 91
	ld d,l	85
multiply	or a	183
	jr z, calculate	40 6
	add hl,de	25
	dec a	61
	jr multiply	24 249
long jump	jr reset	24 208
calculate	ld a, (23303)	58 7 91
	add a,l	133
	ld hl, (23304)	42 8 91
	add a,l	133
	dec a	61
	push af	245
	ld a, (23306)	58 10 91
	ld hl, O	33 0 0
	ld de, (23300)	237 91 4 91

	ld d, l	85
repeat	or a	183
	jr z, continue	40 4
	add hl,de	25
	dec a	61
	jr repeat	24 249
continue	ld a, (23302)	58 6 91
	add a,l	133
	ld hl, (23305)	42 9 91
	add a, l	133
	dec a	61
	ld l,a	111
	pop af	241
	ld h,a	103
	pop af	241
	push af	245
	or a	183
	jr nz, plot	32 7
	call subroutine	205 13* 255*
	cpl	47
	and (hl)	166
	jr Poke	24 4
Plot	call subroutine	205 13* 255*
	or (hl)	182
Poke	ld (hl),a	119
	ld hl, (23304)	42 8 91
	inc l	44
	ld a, (23301)	58 5 91
	inc a	60
	cp l	189
	jr nz, preserve	32 165
	inc h	36
	ld a, (23300)	58 4 91
	inc a	60
	cp h	188
	jr nz, loop	32 155
	pop af	241
	pop bc	193
	pop de	209
	ld hl, (23306)	42 10 91
	inc l	44
	ld a, (23298)	58 2 91
	inc a	60
	cp l	189
	jr nz, long jump	32 164
	ld l, O	46 O
	inc h	36

```
            ld a, (23296)     58 0 91
            inc a             60
            cp h              188
            jr nz, long jump  32 154
            ret               201

subroutine  push bc           197
            push de           213
            ld a, 175         62 175
            sub h             148
            ld h,a            103
            push hl           229
            and 7             230 7
            add a, 64         198 64
            ld c,a            79
            ld a,h            124
            rra               203 31
            rra               203 31
            rra               203 31
            and 31            230 31
            ld b,a            71
            and 24            230 24
            ld d,a            87
            ld a,h            124
            and 192           230 192
            ld e,a            95
            ld h, c           97
            ld a,l            125
            rra               203 31
            rra               203 31
            rra               203 31
            and 31            230 31
            ld l,a            111
            ld a,e            123
            add a,b           128
            sub d             146
            ld e,a            95
            ld d, O           22 O
            push hl           229
            push de           213
            pop hl            225
            add hl,hl         41
            add hl,hl         41
            add hl,hl         41
            add hl,hl         41
            add hl,hl         41
            pop de            209
            add hl,de         25
```



```
            pop de            209
            ld a,e            123
            and 7             230 7
            ld b,a            71
            ld a,8            62 8
            sub b             144
            ld b,a            71
            ld a, 1           62 1

rotate      add a,a           135
            djnz rotate       16 253
            rra               203 31
            pop de            209
            pop bc            193
            ret               201
```

How it works

ix is loaded with the address of the printer buffer, for use as a pointer to the variables. If the upper y co-ordinate or the new lower y co-ordinate is more than 175 the routine returns to BASIC. If the horizontal scale or the vertical scale is zero a return is made to BASIC.

The h register is loaded with the lower y co-ordinate, and the l register is loaded with the upper y co-ordinate. The l register is copied into both the b register and the accumulator. The h register is subtracted from the accumulator, and the routine returns to BASIC, if the result is negative.

The value of the accumulator is then POKEd into location 23298, for use as a counter. The bc register pair is then saved on the stack.

The hl register is loaded with the value in the accumulator, incremented, and copied into the bc register. bc is added to hl, e times, the resulting value in hl being the number of pixels on the screen to be copied. The accumulator is loaded with the value in the l register, and the four leftmost bits are set to zero. The result is copied into the b register, to be used as a counter.

The hl register pair is retrieved from the stack, and the l register is copied into the c register. If the b register holds zero, it is loaded with sixteen, this being the number of bits in a register pair. The 'subroutine' is then called, and the accumulator is loaded with the value POINT (l,h). The de register pair is rotated to the left and the value of the accumulator is loaded into the rightmost bit of the e register.

If the l register equals the left x co-ordinate the routine jumps to 'next row'. Otherwise the l register is decremented, followed by the b register. If the b register does not hold zero, the routine loops to 'save' to feed the next bit into the de register pair. If the b register does hold zero, the de register pair is pushed on to the stack and a jump is made to 'full'.

At 'next row' the l register is loaded with the right x co-ordinate, and the accumulator is loaded with the value in the h register. If the value of the accumulator equals the lower y co-ordinate, a jump is made to 'copy' because the last pixel to be copied has been fed into de. Otherwise, the h

register is decremented to point to the next row, and the routine loops to 'next bit'.

At 'copy', de is pushed on to the stack, and the b, h and l registers are all set to zero for use as counters. The hl register pair is POKEd into addresses 23306/7, so that hl can be used as a counter for further loops without using the stack. If the b register holds zero, de is retrieved from the stack and the b register is reset to sixteen, indicating the number of pixels stored in de. The b register is decremented to indicate that a bit of information is to be removed from de. The rightmost bit of the e register is loaded into the accumulator, and the de register pair is rotated to the right. de, bc and af are all pushed on to the stack, while some calculations are performed.

The h and l registers, are both loaded with one, for use as counters, and hl is POKEd into addresses 23304/5. The accumulator is loaded with the value of the byte at address 23307, this being one of the counters saved earlier. The de register pair is loaded with the vertical scale. This is then multiplied by the value in the accumulator, and the result fed into hl. This is added to the new lower y co-ordinate, in the accumulator. The byte at address 23304 is then added to the accumulator, and the result is decremented.

The accumulator now holds the y co-ordinate of the next pixel to be plotted. This is saved on the stack, whilst the x co-ordinate is calculated by a very similar process. The x co-ordinate, when calculated, is loaded into the l register. The y co-ordinate is retrieved from the stack and loaded into the h register. The accumulator is set to the last value held on the stack. If this is one, then the point (x,y) should be plotted, otherwise it should be 'unplotted'. The subroutine is called, and the appropriate action taken.

The hl register pair is loaded with the loop counters, stored at addresses 23304/5. The l register is incremented and if this does not hold the value (1 + vertical scale) the routine loops to 'preserve'. The h register is incremented and if this does not hold the value (1 + horizontal scale) a jump is made to 'loop'.

The af, bc and de register pairs are retrieved from the stack and the hl register pair is loaded with the second set of loop counters, which are stored at addresses 23306/7. The l register is incremented, and a jump is made to 'reset' if the result does not equal (right x co-ord—left x co-ord + 1). The l register is set to zero, this being the original value of the loop counter. The h register is then incremented, and the routine loops to 'reset' if the result does not equal (upper y co-ord—lower y co-ord + 1). The routine returns to BASIC.

The 'subroutine' is identical to that used in the 'Region Fill' routine.

7. ROUTINES TO MANIPULATE PROGRAMS

Delete block of program

Length: 42
Number of Variables: 2
Check sum: 5977

Operation

This routine deletes blocks of BASIC program between two lines specified by the user.

Variables

Name	*Length*	*Location*	*Comment*
start line no	2	23296	First line to be deleted
end line no	2	23298	Final line to be deleted

Call

RAND USR address

Error Checks

If any of the following errors occur then the routine stops without deleting any of the BASIC program:

(i) Final line number is less than first line number;
(ii) there is no BASIC program between the two given lines;
(iii) either or both of the specified line numbers are zero.

Comments

This routine is quite slow to delete a large block of program lines but nonetheless it is much quicker to use it than to delete the lines by hand. Do not enter line numbers greater than 9999.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld de, (23298)	237 91 2 91
	ld a,h	124
	or l	181
	ret z	200
	ld a,d	122
	or e	179
	ret z	200
	push de	213
	call 6510	205 110 25
	ex (sp), hl	227

	inc hl	35
	call 6510	205 110 25
	pop de	209
	and a	167
	sbc hl,de	237 82
	ret z	200
	ret c	216
	ex de,hl	235
next chr:	ld a,d	122
	or e	179
	ret z	200
	push de	213
	push hl	229
	call 4120	205 24 16
	pop hl	225
	pop de	209
	dec de	27
	jr next chr	24 243

How it works

The hl and de register pairs are loaded with the start and end line numbers respectively. The values are checked and if either or both are zero, the routine returns to BASIC.

The ROM routine at address 6510 is then called and it returns the address of the first line. It is then called again to find the address of the character after the "ENTER" in the final line. The hl register pair is set to the difference in the two addresses, and if this is zero or negative, the routine returns to BASIC.

The contents of the hl register pair are copied into de to be used as a counter. If the counter is zero the routine has finished, if not then the ROM routine at address 4120 is called which deletes one character. The routine then loops back to 'next chr'.

Token Swap

Length: 46

Number of Variables: 2

Check sum: 5000

Operation

Changes every occurence of a specified character in a BASIC program to another specified character. eg all PRINT statements could be changed to LPRINTS.

Variables

Name	*Length*	*Location*	*Comment*
chr old	1	23296	character to be replaced
chr new	1	23297	character to be entered

Call

RAND USR address

Error Checks

If there is no BASIC program in memory or if either of the specified characters have codes less than 32, the routine returns to BASIC.

Comments

This routine is very fast but obviously, the longer the BASIC program, the longer it takes to run.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld, bc, (23296)	237 75 0 91
	ld a,31	62 31
	cp b	184
	ret nc	208
	cp c	185
	ret nc	208
	ld hl, (23635)	42 83 92
next chr:	inc hl	35
	inc hl	35
	inc hl	35
check:	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	ret nc	208
	add hl,de	25
	inc hl	35
	ld a, (hl)	126
	inc hl	35
	cp 13	254 13
	jr z, next chr	40 237
	cp 14	254 14
	jr nz, compare	32 3
	inc hl	35
	jr next chr	24 230
compare:	dec hl	43
	cp c	185
	jr nz, check	32 229
	ld (hl), b	112
	jr check	24 226

How it works

The b and c registers are loaded with the new and old characters respectively. If either character has a code less than 32 then the routine returns to BASIC.

The hl register pair is loaded with the address of the start of the basic program. The hl pair is then increased and compared with the address of the variables area. If hl is not less than the address of the variables the routine returns to BASIC.

The hl pair is incremented to point to the next character. The code of this character is loaded into the accumulator, and hl is incremented again. If the value of the accumulator is 13 or 14 (ENTER or NUMBER) the routine jumps back to next chr and hl is increased to point to the next character. If the accumulator does not hold 13 or 14, the value stored is compared with 'chr old'. If a match is found this character is replaced by 'chr new'.

The routine then jumps back to check for the end of the program.

REM Kill

Length: 132
Number of Variables: O
Check sum: 13809

Operation

This routine deletes all 'REM' statements in a BASIC program in memory.

Call

RAND USR address

Error Checks

If there is no BASIC program in memory the routine will return without doing anything.

Comments

The ROM routine which is used to delete characters is not very fast and so this routine may take some time to run.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23635)	42 83 92
	jr check	24 31
next line	push hl	229
	inc hl	35
	inc hl	35
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	70

next chr	inc hl	35
	ld a,(hl)	126
	cp 33	254 33
	jr c, next chr	56 250
	cp 234	254 234
	jr nz, search	32 26
	inc bc	3
	inc bc	3
	inc bc	3
	inc bc	3
	pop hl	225
delete line	push bc	197
	call 4120	205 24 16
	pop bc	193
	dec bc	11
	ld a,b	120
	or c	177
	jr nz, delete line	32 246
check	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	ret nc	208
	add hl,de	25
	jr next line	24 214
search	inc hl	35
	ld a, (hl)	126
	cp 13	254 13
	jr nz, not enter	32 8
enter found	pop hl	225
	add hl,bc	9
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr check	24 231
not enter	cp 14	254 14
	jr nz, not number	32 7
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr search	24 231
not number	cp 33	254 33
	jr c, search	56 227
	cp 34	254 34

```
            jr nz, not quote     32 8
find quote  inc hl               35
            ld a, (hl)           126
            cp 34                254 34
            jr nz, find quote    32 250
            jr search            24 215
not quote   cp 58                254 58
            jr nz, search        32 211
            ld d,h               84
            ld e, l              93
find enter  inc hl               35
            ld a, (hl)           126
            cp 13                254 13
            jr z, enter found    40 209
            cp 33                254 33
            jr c, find enter     56 246
            cp 234               254 234
            jr nz, not quote     32 236
            ld h,d               98
            ld l,e               107
delete chr  push bc              197
            call 4120            205 24 16
            pop bc               193
            dec bc               11
            ld a, (hl)           126
            cp 13                254 13
            jr nz, delete chr    32 245
            pop hl               225
            inc hl               35
            inc hl               35
            ld (hl),c            113
            inc hl               35
            ld (hl),b            112
            dec hl               43
            dec hl               43
            dec hl               43
            jr check             24 160
```

How it works

The hl register pair is loaded with the address of the start of the BASIC program area, and a jump is made to the routine which checks for the end of the program area. If the end has been reached a return to BASIC is made.

The routine jumps to 'next line'. This section saves the address in hl on the stack for later use, and then loads bc with the length of the BASIC line that has been encountered. The 'next chr' routine increments the address in hl and loads the accumulator with the character stored at that address. If this

character has a code less than 33, indicating that it is a space or control character, the routine jumps back to repeat this section again. If the character encountered is not the REM token a jump is made to 'search'.

If a REM has been found the bc register is increased by four so that it can be used as a counter, and hl is removed from the top of the stack. Then bc characters are deleted at address hl using the ROM routine at address 4120. The routine then 'falls through' to the 'check' routine again.

If a jump is made to the 'search' routine hl is incremented to point to the next character and this is loaded into the accumulator. If this is an ENTER character hl is restored from the stack, increased to point to the start of the next line, and a jump is made to 'check'.

If the accumulator holds the NUMBER character (14) hl is increased to point to the first character after the stored number and the search process is repeated.

A check is then made for characters whose codes are less than 33, and if one is found a jump is made back to 'search'. If a quote character (34) is found, the routine loops until a second quote is found and then the search is continued. If the character found is not a colon, indicating a multi-statement line, the search is repeated. hl is then copied into de to save the address of the colon, and then hl is incremented to point to the next character. If this character is an ENTER a jump is made to 'enterfound', otherwise if it is a control character or space the routine loops back to 'find enter'.

If the character is not a REM token a jump is made back to 'not quote'. If a REM token is found hl is loaded with the address of the last colon encountered, and then all the characters from hl to the next ENTER token are deleted. The pointers for the line are corrected, hl is set to the start of the line and a jump is made back to 'check'.

REM Create

Length: 85

Number of Variables: 3

Check sum: 9526

Operation

This routine creates a REM statement at a specified line containing a given number of characters. The character is chosen by the user.

Variables

Name	*Length*	*Location*	*Comment*
line number	2	23296	line at which REM is to be inserted
number char.	2	23298	number of characters after REM
char. code	1	23300	code of characters after REM

Call

RAND USR address

Error Checks

If the line number given is zero, more than 9999, or a line with the same number already exists the routine returns to BASIC.

Comments

This routine does not check that enough memory is free for the new line to be inserted. Therefore, this should be done before running this routine, by calling the 'Memory Left' routine elsewhere in this book.

The characters to be entered after the REM should preferably have codes more than 31 as the control characters (0-31) may confuse the LIST routine in the ROM.

The ROM routine which is called to insert characters is fairly slow, so this routine can take a long time.

The REM statement created using this routine, can be used to store machine code or data which has to be POKEd into place.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld, a,h	124
	or l	181
	ret z	200
	ld de, 10000	17 16 39
	and a	167
	sbc hl,de	237 82
	ret nc	208
	add hl,de	25
	push hl	229
	call 6510	205 110 25
	jr nz, create	32 2
	pop hl	225
	ret	201
create	ld bc, (23298)	237 75 2 91
	push bc	197
	push bc	197
	ld a, 13	62 13
	call 3976	205 136 15
	inc hl	35
	pop bc	193
next chr	push bc	197
	ld a,b	120
	or c	177

	jr z, insert REM	40 11
	ld a, (23300)	58 4 91
	call 3976	205 136 15
	inc hl	35
	pop bc	193
	dec bc	11
	jr next chr	24 240
insert REM	pop bc	193
	ld a, 234	62 234
	call 3976	205 136 15
	inc hl	35
	pop bc	193
	inc bc	3
	inc bc	3
	ld a,b	120
	push bc	197
	call 3976	205 136 15
	pop bc	193
	inc hl	35
	ld a,c	121
	call 3976	205 136 15
	inc hl	35
	pop bc	193
	ld a,c	121
	push bc	197
	call 3976	205 136 15
	pop bc	193
	inc hl	35
	ld a,b	120
	jp 3976	195 136 15

How it works

The hl register pair is loaded with the specified line number. This is compared with zero, and if a match is found the routine returns to BASIC. Also if hl contains a number longer than 9999 (the highest possible line number), a return is made to BASIC.

A ROM routine is called which returns in hl the address of the line whose number was previously in hl. If the zero flag is set, a line already exists there, and so the routine returns to BASIC.

If the zero flag is not set a jump is made to 'create'. bc is loaded with the number of characters to be inserted after the 'REM' and this number is saved on the stack. The accumulator is then loaded with 13, which is the code of the ENTER character. The ROM routine at address 3976 is then called to insert the ENTER character. The bc register is retrieved from the stack. After re-saving bc on the stack, bc is tested to see if any more characters have to be inserted. If not, a jump is made to "insert REM". If

another character has to be inserted, the accumulator is loaded with the specified code and the ROM routine at 3976 is used to insert it. The counter (bc) is decremented and the routine loops back to test if bc is zero. Once the routine reaches "insert REM", a REM token is inserted using the same ROM routine. bc is then loaded with the length of the new line, and the pointers for that line are created. The line number is then removed from the stack and this is finally inserted before returning to BASIC.

Compact Program

Length: 71
Number of Variables: O
Check sum: 7158

Operation

This routine deletes all unnecessary control characters and spaces in a BASIC program, thus increasing the amount of spare RAM available.

Call

RAND USR address

Error Checks

If there is no BASIC program in memory the routine returns to BASIC immediately.

Comments

This routine assumes that all REM statements have already been removed from the BASIC program. However, if this is not so, the computer will not 'crash'. The time taken for the routine to finish is proportional to the length of the BASIC program in memory.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23635)	42 83 92
next line	inc hl	35
	inc hl	35
check	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	ret nc	208
	add hl,de	25
length	push hl	229
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	70

next byte	inc hl	35
load	ld a, (hl)	126
	cp 13	254 13
	jr nz, number	32 8
restore	pop hl	225
	ld (hl),c	113
	inc hl	35
	ld (hl),b	112
	add hl,bc	9
	inc hl	35
	jr next line	24 227
number	cp 14	254 14
	jr nz, quote	32 7
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr next byte	24 231
quote	cp 34	254 34
	jr nz, control	32 12
find quote	inc hl	35
	ld a, (hl)	126
	cp 34	254 34
	jr z, next byte	40 221
	cp 13	254 13
	jr z, restore	40 223
	jr find quote	24 244
control	cp 33	254 33
	jr nc, next byte	48 211
	push bc	197
	call 4120	205 24 16
	pop bc	193
	dec bc	11
	jr load	24 204

How it works

The hl register pair is loaded with the address of the BASIC program. hl is then incremented twice, so that it points to the two bytes holding the length of the next line. The de register pair is loaded with the address of the variables area. If hl is not less than de the routine returns to BASIC because the end of the program area has been reached.

The address in hl is saved on the stack, bc is loaded with the length of the present line and hl is incremented to point to the next byte in the line. The byte at hl is then loaded into the accumulator. If the accumulator does not hold thirteen a jump is made to 'number'.

To reach 'restore' the end of the present line must have been found. The address of the line 'pointers' is loaded from the stack into hl, and the present length inserted. The line length is added to hl, hl is incremented and the routine loops back to 'next line'.

If the routine reaches 'number' the accumulator is checked to see if it holds the NUMBER character (14). If so, hl is increased by five so that the following number is not changed, and a jump is made to 'next byte'.

If the accumulator does not hold the code for a quote character the routine jumps to 'control'. If a quote has been found the routine loops until the end of the line is reached, or another quote. In the former case a jump is made to 'restore', in the latter case the jump is to 'next byte'.

At 'control' the character is checked to see if it has a code less than thirty-three. If not the routine loops to 'next byte'.

If a space or control character has been found the ROM routine at address 4120 is called to delete it. The line length, which is held in bc, is decremented and a jump is made to 'load'.

Load Machine Code into Data Statements

Length: 179
Number of Variables: 2
Check sum: 19181

Operation

This routine produces a DATA statement at line one in a BASIC program and then fills it with data PEEKed from memory.

Variables

Name	*Length*	*Location*	*Comment*
data start	2	23296	address to be copied from
data length	2	23298	number of bytes to copy

Call

RAND USR address

Error Checks

If the number of bytes to be copied is zero or there is already a line one the routine returns to BASIC immediately. The routine does not check that there is enough memory available for the new line, and so this must be done manually.

The routine requires ten bytes per byte of data, plus five for line numbers, pointers, etc. However, the ROM routine used also uses a large workspace, so always take this into account. If there is not enough memory available, the line pointers will not be set correctly and the BASIC listing will be corrupted.

Comments

The time taken by this routine is proportional to the length of memory to be copied.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld de, (23296)	237 91 0 91
	ld bc, (23298)	237 75 2 91
	ld a,b	120
	or c	177
	ret z	200
	ld hl, (23635)	42 83 92
	ld a, (hl)	126
	cp O	254 0
	jr nz, continue	32 6
	inc hl	35
	ld a, (hl)	126
	cp 1	254 1
	ret z	200
	dec hl	43
continue	push hl	229
	push bc	197
	push de	213
	sub a	151
	call 3976	205 136 15
	ex de,hl	235
	ld a, l	62 1
	call 3976	205 136 15
	ex de,hl	235
	call 3976	205 136 15
	ex de,hl	235
	call 3976	205 136 15
	ex de,hl	235
	ld a, 228	62 228
	call 3976	205 136 15
	ex de,hl	235
next byte	pop de	209
	ld a, (de)	26
	push de	213
	ld c,47	14 47
hundreds	inc c	12
	ld b, 100	6 100
	sub b	144
	jr nc, hundreds	48 250
	add a,b	128
	ld b,a	71

```
            ld a,c          121
            push bc         197
            call 3976       205 136 15
            ex de,hl        235
            pop bc          193
            ld a,b          120
            ld c,47         14 47
tens        inc c           12
            ld b,10         6 10
            sub b           144
            jr nc, tens     48 250
            add a,b         128
            ld b,a          71
            ld a,c          121
            push bc         197
            call 3976       205 136 15
            pop bc          193
            ex de,hl        235
            ld a,b          120
            add a,48        198 48
            call 3976       205 136 15
            ex de,hl        235
            ld a,14         62 14
            ld b, 6         6 6
next zero   push bc         197
            call 3976       205 136 15
            pop bc          193
            ex de,hl        235
            sub a           151
            djnz next zero  16 247
            pop de          209
            push hl         229
            dec hl          43
            dec hl          43
            dec hl          43
            ld a, (de)      26
            ld (hl),a       119
            pop hl          225
            inc de          19
            pop bc          193
            dec bc          11
            ld a,b          120
            or c            177
            jr z, enter     40 10
            push bc         197
            push de         213
            ld a, 44        62 44
```



```
            call 3976       205 136 15
            ex de, hl       235
            jr next byte    24 173
enter       ld a,13         62 13
            call 3976       205 136 15
            pop hl          225
            ld bc, 0        1 0 0
            inc hl          35
            inc hl          35
            ld d,h          84
            ld e,l          93
            inc hl          35
pointers    inc hl          35
            inc bc          3
            ld a, (hl)      126
            cp 14           254 14
            jr nz, end?     32 12
            inc bc          3
            inc bc          3
            inc bc          3
            inc bc          3
            inc bc          3
            inc hl          35
            inc hl          35
            inc hl          35
            inc hl          35
            inc hl          35
            jr pointers     24 237
end?        cp 13           254 13
            jr nz pointers  32 233
            ld a,c          121
            ld (de),a       18
            inc de          19
            ld a,b          120
            ld (de),a       18
            ret             201
```

How it works

The de register pair is loaded with the address of the bytes to be copied, and the bc register pair is loaded with the number of bytes to be copied. If bc holds zero the routine returns to BASIC immediately.

The hl register pair is loaded with the address of the BASIC program. The accumulator is loaded with the byte stored at the address in hl. This is the high byte of the line number. If this does not hold zero, line one does not already exist and so the routine jumps to 'continue'. If the high byte does hold zero, the accumulator is loaded with the low byte. If this is set to one, line one already exists, and so the routine returns to BASIC.

The address of the high byte of the line number is saved on the stack. The number of bytes to be copied is saved, followed by the address of the data.

The accumulator is then loaded with zero—the high byte of the new line number. Calling the ROM routine at address 3976 then inserts the character, held in the accumulator, at the address stored in hl. hl is set to the value held before this operation. The accumulator is loaded with one, and this is inserted three times. The first one is the low byte of the line number, the next two being the line pointer. The accumulator is then loaded with the code of the 'DATA' token and this is inserted.

The address of the next byte of data is retrieved from the stack and loaded into de. The accumulator is loaded with this byte, and de is stacked again. The c register is loaded with one less than the code for the character "O". The c register is incremented and the b register is loaded with 100. The b register is subtracted from the accumulator and if the result is not negative the routine loops back to 'hundreds'.

The b register is added once to the accumulator so that the accumulator holds a positive value. This value is then loaded into the b register. The accumulator is loaded with the contents of c, and bc is saved on the stack. The ROM routine at address 3976 then inserts the character, held in the accumulator, at the address stored in hl. The bc register pair is retrieved from the stack and the accumulator is loaded with the value of the b register. The above process is then repeated for b = 10. The accumulator is then increased by 48 and the resulting character is inserted.

The above routine has inserted the decimal value, of the byte of data encountered, into the DATA statement. The binary representation must now be inserted. This is marked by the NUMBER token, chr 14, which is entered first, followed by five zeros. The value of the byte being copied is POKEd to replace the third zero. de is then incremented to point to the next byte of data. The number of bytes to be copied is copied from the stack into bc, and this is decremented. If the result is zero a jump is made to 'enter', otherwise the bc and de register pairs are re-stacked, a comma is inserted in the DATA statement, and the routine loops to 'next byte'.

At 'enter' an ENTER token is inserted to mark the end of the DATA statement. hl is loaded with the address of the start of the line, and bc is set to zero. hl is increased to point to the low byte of the line pointer, and this new address is copied into de. hl is incremented to point to the high byte of the line pointer. hl and bc are then incremented, and the accumulator is loaded with the character at the address stored in hl.

If the accumulator holds 14, a number has been found and so both hl and bc are increased by five to point to the first character after the number, the routine then looping to 'pointers'.

If the accumulator does not hold 14, and it does not hold 13 a jump is made back to 'pointers'.

To reach this stage the ENTER token marking the end of the line must have been encountered. bc now holds the line length and so this is POKEd into the line pointer, the address of which is stored in de.

The routine then returns to BASIC.

Convert Lower Case to Upper Case

Length: 41
Number of Variables: O
Check sum: 4683

Operation

This routine converts all lower case characters in a BASIC program to upper case or vice versa.

Call

RAND USR address

Error Checks

If there is no BASIC program in memory the routine returns to BASIC immediately.

Comments

To change this routine so that it converts from upper case to lower case change the numbers marked as below:

96* to 64
90** to 122

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23635)	42 83 92
	ld de, (23627)	237 91 75 92
jump	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
changed	inc hl	35
next byte	and a	167
	sbc hl,de	237 82
	ret nc	208
	add hl,de	25
	ld a, (hl)	126
	cp 13	254 13
	jr z, jump	40 241
	cp 14	254 14
	inc hl	35
	jr z, jump	40 236
	sub 96	214 96*
	jr c, next byte	56 237
	sub 26	214 26

jr nc, next byte	48 233
add a,90	198 90**
dec hl	43
ld (hl), a	119
jr changed	24 226

How it works

The hl register pair is loaded with the address of the BASIC program and de is loaded with the address of the variables area. hl is increased to jump over the line number/pointers. If hl is not less than de, the routine returns to BASIC, as the end of the program has been reached.

The accumulator is loaded with the byte stored at hl. If this byte is an ENTER character the routine loops back to 'jump'. If the byte is the NUMBER token, the routine also loops back to 'jump', having already incremented hl. Thus the five bytes after the character 14 are avoided.

Ninety six is subtracted from the accumulator. If the result is negative the routine jumps to 'next byte' because the character cannot be a lower case letter. Twenty six is then subtracted from the accumulator. If the result is not negative a jump is made to 'next byte' as the character has a code too high to be a lower case letter. Ninety is then added to the accumulator to give the code of the corresponding upper case letter. hl is decremented to point to the character that is to be replaced. This address is POKEd with the value in the accumulator and a jump is made to 'changed'.

8. TOOLKIT ROUTINES

Renumber

Length: 382
Number of Variables: 2
Check sum: 41423

Operation

This routine renumbers a BASIC program including any GOTO, GOSUBs etc.

Variables

Name	*Length*	*Location*	*Comment*
first line no	2	23296	The number of the first line when RUN
step	2	23298	The difference between consecutive line numbers.

Call

RAND USR address

Error Checks

If the number of the first line is zero, or the step is zero the routine returns to BASIC immediately. If there is no BASIC program in RAM the routine returns to BASIC. Any calculated line numbers (eg GOTO 7*A); numbers including decimal points (eg GOTO 7.8); numbers less than zero (eg GOTO—1) or numbers more than 9999 (eg GOTO 20170) are ignored. If the step is too large, line numbers may be repeated and the program corrupted. The routine increases the length of the BASIC program in RAM so a check should always be made that there is some spare RAM.

Comments

The time taken by this routine is proportional to the length of the BASIC program in RAM.

The routine is not relocatable and should normally be entered at memory location 32218. The position can be changed by following this procedure:

(i) Let X = new address —32218
(ii) Let H = INT (x/256)
Let L = x — 256*h
(iii) For every pair of numbers in the listing marked '*'
Let LI = L + the first number
Let HI = H + the second number

If LI is more than 255 Let HI = HI + 1
let LI = LI—256

Replace the pair of numbers by LI and HI.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld a, h	124
	or L	181
	ret z	200
	ld hl, (23298)	42 2 91
	ld a,h	124
	or L	181
	ret z	200
	ld hl, (23635)	42 83 92
	ld de, (23296)	237 91 0 91
next line	call check	205 76* 127*
	jr nc, find GOTO	48 22
	ld b, (hl)	70
	ld (hl),d	114
	inc hl	35
	ld c, (hl)	78
	ld (hl),e	115
	inc hl	35
	ld (hl),c	113
	inc hl	35
	ld (hl),b	112
	inc hl	35
	push hl	229
	ld hl, (23298)	42 2 91
	add hl,de	25
	ex de, hl	235
	pop hl	225
	call end of line	205 65* 127*
	jr next line	24 229
find GOTO	ld hl, (23635)	42 83 92
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
search	call find	205 235* 126*
	jp nc, restore	210 184* 126*
	ld d,h	84
	ld e, l	93
	ld b, O	6 0

next digit	inc b	4
	inc hl	35
	ld a, (hl)	126
	cp 46	254 46
	jr nz, continue	32 3
find next	ex de,hl	235
	jr search	24 236
continue	cp 14	254 14
	jr nz, next digit	32 242
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	ld a, (hl)	126
	cp 58	254 58
	jr z, found	40 4
	cp 13	254 13
	jr nz, find next	32 234
found	ld a, b	120
compare	cp 4	254 4
	jr z, calculate	40 16
	jr nc, find next	48 227
	push de	213
	ld h,d	98
	ld l,e	107
	push af	245
	ld a, 48	62 48
	call 3976	205 136 15
	pop af	241
	inc a	60
	pop de	209
	jr compare	24 236
calculate	ld b,d	66
	ld c,e	75
	push de	213
	ld hl, O	33 O O
	ld de, 1OOO	17 232 3
	call add	205 226* 126*
	ld de, 100	17 100 0
	call add	205 226* 126*
	ld e, 10	30 10
	call add	205 226* 126*
	ld a, (bc)	10
	sub 48	214 48

	ld e, a	95
	add hl,de	25
	ld b,h	68
	ld c,l	77
	ld hl, (23635)	42 83 92
find line	inc hl	35
	inc hl	35
end of prog	call check	205 76* 127*
	jr c, exists	56 3
	pop hl	225
	jr search	24 153
exists	ld a, (hl)	126
	cp c	185
	jr nc, next byte	48 7
	inc hl	35
wrong line	inc hl	35
	call end of line	205 65* 127*
	jr find line	24 235
next byte	inc hl	35
	ld a, (hl)	126
	cp b	184
	jr c, wrong line	56 245
	dec hl	43
	dec hl	43
	ld c, (hl)	78
	dec hl	43
	ld h, (hl)	102
	ld l, c	105
	pop bc	193
	push bc	197
	push hl	229
	ld de, 1000	17 232 3
	call insert	205 212* 126*
	ld de, 100	17 100 0
	call insert	205 212* 126*
	ld e, 10	30 10
	call insert	205 212* 126*
	ld e, 1	30 1
	call insert	205 212* 126*
	inc bc	3
	sub a	151
	ld (bc),a	2
	inc bc	3
	ld (bc),a	2
	inc bc	3
	pop hl	225

	ld a,l	125
	ld (bc),a	2
	inc bc	3
	ld a,h	124
	ld (bc),a	2
	inc bc	3
	sub a	151
	ld (bc),a	2
	pop hl	225
	jp search	195 15* 126*
restore following	ld hl, (23635)	42 83 92
	inc hl	35
	inc hl	35
	call check	205 76* 127*
	ret nc	208
	ld b,h	68
	ld c,l	77
	call end of line	205 65* 127*
	push hl	229
	and a	167
	sbc hl,bc	237 66
	dec hl	43
	dec hl	43
	ld a,l	125
	ld (bc),a	2
	inc bc	3
	ld a,h	124
	ld (bc),a	2
	pop hl	225
	jr following line	24 231
insert	ld a, 48	62 48
subtract	and a	167
	sbc hl,de	237 82
	jr c, poke	56 3
	inc a	60
	jr subtract	24 248
poke	add hl,de	25
	ld (bc),a	2
	inc bc	3
	ret	201
add	ld a, (bc)	10
	inc bc	3
	sub 47	214 47
repeat	dec a	61
	ret z	200
	add hl,de	25

```
            jr repeat          24 251
find        ld a, (hl)         126
            call check         205 76* 127*
            ret nc             208
            cp 234             254 234
            jr nz not REM      32 13
find ENTER  inc hl             35
            ld a, (hl)         126
            cp 13              254 13
            jr nz, find ENTER  32 250
increase    inc hl             35
            inc hl             35
            inc hl             35
            inc hl             35
            inc hl             35
            jr find            24 234
not REM     cp 34              254 34
            jr nz, not string  32 9
next character inc hl          35
            ld a, (hl)         126
            cp 34              254 34
            jr nz, next character 32 250
            inc hl             35
            jr find            24 221
not string  cp 13              254 13
            jr z, increase     40 232
            call 6326          205 182 24
            jr z, find         40 212
            cp 237             254 237
            jr z, check digit  40 27
            cp 236             254 236
            jr z, check digit  40 23
            cp 247             254 247
            jr z, check digit  40 19
            cp 240             254 240
            jr z, check digit  40 15
            cp 229             254 229
            jr z, check digit  40 11
            cp 225             254 225
            jr z, check digit  40 7
            cp 202             254 202
            jr z, check digit  40 3
            inc hl             35
            jr find            24 181
check digit inc hl             35
```



```
            ld a, (hl)         126
            cp 48              254 48
            jr c, find         56 175
            cp 58              254 58
            jr nc, find        48 171
            ret                201
end of line ld a, (hl)         126
again       call 6326          205 182 24
            jr z, again        40 251
            cp 13              254 13
            inc hl             35
            jr nz, end of line 32 245
check       push hl            229
            push de            213
            ld de, (23627)     237 91 75 92
            and a              167
            sbc hl,de          237 82
            pop de             209
            pop hl             225
            ret                201
```

How it works

The hl register pair is loaded with the first line number. If this is zero, the routine returns to BASIC. hl is then loaded with the step, and if this is zero the routine returns to BASIC.

hl is loaded with the address of the BASIC program, and de is set to the first line number. The subroutine 'check' is then called, and if the end of the BASIC program has been reached, a jump is made to 'find GOTO'. bc is loaded with the old number of the line encountered, and the number is replaced by de, bc is then copied into the line pointers.

hl is saved on the stack, is loaded with the step, and increased by de. The result is copied into de, being the next line number. hl is then retrieved from the stack, and the subroutine 'end of line' increases it, so that it points to the next line. The routine then loops back to 'next line'.

At 'find GOTO', hl is loaded with the address of the BASIC program and this is increased to point to the first character of the first line. The subroutine 'find' is then called. If there are no more GOTOs, GOSUBs etc left to alter, the routine jumps to 'restore'. Otherwise, on return from the subroutine hl holds the address of the first digit after the GOTO, GOSUB, etc. This is copied into de, and the b register is set to zero. The b register is used to count the number of digits in the following number.

The b register is incremented, and hl is increased to point to the next character and this character is then loaded into the accumulator. If the character is a decimal point hl is loaded with de and the routine loops back to search, to find the next GOTO. If the character is not the NUMBER token the routine loops to 'next digit'.

hl is increased to point to the character following the NUMBER. If this is not a colon or an ENTER token the routine jumps back to 'find next' as the GOTO being tested has a calculated destination. The accumulator is loaded with the value in the b register. If this is four the routine jumps to 'calculate'; if it is more than four a jump is made back to 'find next' as line numbers more than 9999 are invalid.

de is then saved on the stack, and copied into hl. The accumulator is then saved on the stack, and loaded with the code of the zero character. This is inserted at the address in hl by the ROM routine at address 3976. The accumulator is retrieved from the stack and incremented. It then holds the new number of digits in the line number. de is retrieved from the stack, and the routine loops to 'compare'.

At 'calculate', the address in de is copied into bc, and then saved on the stack. hl is loaded with zero, and de is loaded with 1OOO. The subroutine 'add' is then called, to add to hl the number of thousands in the line number under scrutiny. This is then repeated for the hundreds, tens and units, thus loading hl with the line number. The bc register pair is loaded with the result.

The hl register pair is loaded with the address of the BASIC program. The 'check' subroutine is called, and if the end of the program area has been reached, the routine retrieves hl from the stack and jumps to 'search', because the destination of the GOTO does not exist. If the byte addressed by hl is less than the value of the c register, hl is increased to point to the next line, and a jump made to 'find line'. Otherwise hl is incremented to point to the next byte of the line number under test. If this is less than the value of the b register a jump is made to 'wrong line'.

To reach this stage the destination of the GOTO must have been found. hl is decreased to point to the start of the line, and then loaded with its new line number. bc is loaded with the address on the stack, and then hl is saved on the stack. bc now holds the address to which the line number is to be copied. de is loaded with 1OOO, and the subroutine 'insert' is called. This calculates the number of thousands in hl, adds 48 to produce a readable digit, and POKEs the value into bc. bc is then increased to point to the next character. This process is repeated for the hundreds, tens and units.

The binary representation of the line number is then built up; bc is increased to point to the character after the NUMBER token and the next two bytes are POKEd with zero. hl is then retrieved from the stack, and POKEd into the following two bytes. The fifth byte of the number is POKEd with zero. hl is retrieved from the stack, and the routine jumps to 'search' to repeat the process for the next GOTO.

At 'restore', hl is loaded with the address of the BASIC program area, and then incremented twice to address the pointers of the following line, which actually hold the old line number. The subroutine 'check' is called, and if the end of the BASIC program has been reached the routine returns to BASIC. bc is loaded with the address in hl, and the subroutine 'end of line' is called. This returns one plus the address of the ENTER token in hl. hl is saved on the stack. bc is subtracted from hl, and then

hl is decremented twice producing the new line pointers which are POKEd on to bc and bc + 1. hl is retrieved from the stack, and a jump made to 'following line'.

Subroutines

Insert:

The accumulator is loaded with the code of the zero character. de is subtracted from hl, and if the result is negative, a jump is made to 'poke'. Otherwise the accumulator is incremented and a loop made to 'subtract'.

At 'poke', de is added to hl to produce a positive value. bc is POKEd with the value in the accumulator, and then incremented to point to the next byte. A return is then made.

Add:

The accumulator is loaded with the byte addressed by bc, and bc is incremented to point to the next byte. 47 is subtracted from the accumulator. The accumulator is decremented and if the result is zero, a return is made. Otherwise de is added to hl, and the routine loops to 'repeat'.

Find:

The accumulator is loaded with the byte addressed by hl. The 'check' subroutine is then called, and if the end of the BASIC program has been reached a return is made. If the character in the accumulator is not the 'REM' token a jump is made to 'not REM'. hl is incremented repeatedly until the end of the line is found. hl is increased to point to the first character of the next line, and a jump is made to 'find'.

At 'not REM', if the accumulator does not hold the code of the quote character, a jump is made to 'not string'. Otherwise hl is incremented repeatedly until a second quote symbol is found. hl is incremented once more to point to the next character, and a jump is made back to 'find'.

At 'not string', if the accumulator holds the ENTER token a loop is made to 'increase', if it holds the NUMBER token the routine loops to 'find'. If none of the GOSUB, GOTO, RUN, LIST, RESTORE, LLIST, LINE instructions has been found, hl is incremented and a jump made to 'find'. hl is incremented, and the accumulator loaded with the next character. If this is not in the range 48-57 the routine jumps to 'find'. The routine then returns.

End of Line:

The accumulator is loaded with the byte addressed by hl. If this is the NUMBER token hl is increased and a loop made to 'again'. hl is incremented. If the accumulator does not hold the ENTER token the routine jumps to 'end of line'. A test is made to see if the end of the BASIC program has been reached, and the routine then returns.

Memory Left

Length: 14

Number of Variables: O

Check sum: 1443

Operation
Returns the amount of spare RAM in bytes.

Call
PRINT USR address

Error Checks
None

Comments
This routine should be called before using any routines that may increase the program length, to ensure that there is enough spare RAM.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, O	33 O O
	add hl, sp	57
	ld de, (23653)	237 91 101 92
	and a	167
	sbc hl,de	237 82
	ld b,h	68
	ld c,l	77
	ret	201

How it works
The hl register pair is set to zero, and the address of the end of spare RAM is added to it (the address is stored in sp). The de register pair is loaded with the address of the start of spare RAM, and is subtracted from hl. hl is copied into bc, and the routine returns to BASIC.

Program Length

Length: 13
Number of Variables: O
Check sum: 1544

Operation
Returns the length of BASIC program, in bytes.

Call
PRINT USR address

Error Checks
None

Comments
None

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23627)	42 75 92
	ld de, (23635)	237 91 83 92
	and a	167
	sbc hl,de	237 82
	ld b,h	68
	ld c,l	77
	ret	201

How it works
The hl register pair is loaded with the address of the variables area, and de is loaded with the address of the BASIC program. de is subtracted from hl, to give the program length. hl is copied into bc, and the routine returns to BASIC.

Line Address

Length: 29
Number of Variables: 1
Check sum: 2351

Operation
Returns the address of the first character after the 'REM' token in a specified line.

Variables

Name	*Length*	*Location*	*Comment*
line number	2	23296	line number which should contain 'REM'

Call
LET A = USR address

Error Checks
If the specified line does not exist or it is not a REM statement, the routine will return the value zero.

Comments
This routine can be used to find the address at which machine-code should be POKEd to be positioned in a REM statement.

When called, the variable A (any variable could be used) is set to the address, or zero if an error occurs. Do not enter line numbers more than 9999.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld a,h	124
	or l	181
	jr z, error	40 5
	call 6510	205 110 25
	jr z, continue	40 4
error	ld bc, O	1 0 0
	ret	201
continue	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	ld a, 234	62 234
	cp (hl)	190
	jr nz, error	32 243
	inc hl	35
	ld b,h	68
	ld c,l	77
	ret	201

How it works

The hl register pair is loaded with the specified line number. If this number is zero a jump is made to 'error' otherwise the ROM routine at address 6510 is called on return from this subroutine. hl is set to the address of the line. If the zero flag is set a jump is made to 'continue'. If the zero flag is not set, the line does not exist, and the routine falls through to 'error' where bc is loaded with zero and the routine returns to BASIC.

If the routine reaches 'continue' hl is increased by four to point to the first instruction in the specified line. If this instruction does not have a code of 234 a jump is made to 'error'. If the instruction is a 'REM' hl is increased to point to the next character. The value of hl is then copied into bc and the routine returns to BASIC.

Copy Memory

Length: 33
Number of Variables: 3
Check sum: 4022

Operation

This routine copies an area of memory from one address to another.

Variables

Name	*Length*	*Location*	*Comment*
start	2	23296	address to be copied from
destination	2	23298	address to be copied to
length	2	23300	number of bytes to be copied

Call

RAND USR address

Error Checks

None

Comments

This routine can be used to produce animated 'films' by the following method:

(i) produce the first screen of information
(ii) copy the display to above RAMTOP
(iii) repeat for further screens.
(iv) copy the screens back one at a time in rapid succession.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld de, (23298)	237 91 2 91
	ld bc, (23300)	237 75 4 91
	ld a,b	120
	or c	177
	ret z	200
	and a	167
	sbc hl,de	237 82
	ret z	200
	add hl,de	25
	jr c, lddr	56 3
	ldir	237 176
	ret	201
lddr	ex de,hl	235
	add hl,bc	9
	ex de,hl	235
	add hl,bc	9
	dec hl	43
	dec de	27
	lddr	237 184
	ret	201

How it works

The hl register pair is loaded with the address of the first byte of memory to be copied, de is loaded with the address that it is to be copied to, and bc is loaded with the number of bytes to be copied. If bc is zero or hl = de then the routine returns to BASIC. If hl is more than de, the section of memory is copied using the 'ldir' instruction, and then the routine returns to BASIC.

If de is more than hl, bc—1 is added to both register pairs, the memory is copied using the 'lddr' instruction, and the routine returns to BASIC.

Zero all Variables

Length: 108

Number of Variables: O

Check sum: 10717

Operation

All numeric variables are given the value zero, all dimensioned strings are filled with spaces, and non-dimensioned strings are set to length zero (null strings).

Call

RAND USR address

Error Checks

If there are no variables in memory the routine returns to BASIC immediately.

Comments

This routine is a useful debugging aid.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23627)	42 75 92
check	ld a, (hl)	126
	cp 128	254 128
	ret z	200
	ld de, 1	17 1 0
	bit 7,a	203 127
	jr nz, next bit	32 32
	bit 5,a	203 111
	jr z, 'string'	40 9
zero	ld b,5	6 5
next byte	inc hl	35
	ld (hl),d	114
	djnz next byte	16 252

	add hl,de	25
	jr check	24 232
string	inc hl	35
	ld c,(hl)	78
	ld (hl),d	114
	inc hl	35
	ld b,(hl)	70
	ld (hl),d	114
	inc hl	35
delete	ld a,b	120
	or c	177
	jr z, check	40 221
	push bc	197
	call 4120	205 24 16
	pop bc	193
	dec bc	11
	jr delete	24 244
next bit	bit 6,a	203 119
	jr nz, bit 5	32 45
	bit 5,a	203 111
	jr z, array	40 7
number	inc hl	35
	bit 7, (hl)	203 126
	jr z, number	40 251
	jr zero	24 213
array	sub a	151
find length	puch af	245
	inc hl	35
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	70
	inc hl	35
	push hl	229
	ld l, (hl)	110
	ld h, d	98
	add hl,hl	41
	pop de	209
find elements	inc de	19
	dec bc	11
	dec hl	43
	ld a,h	124
	or l	181
	jr nz, find elements	32 249
	dec bc	11
rub out	inc de	19
	dec bc	11

	pop af	241
	push af	245
	ld (de),a	18
	ld a,b	120
	or c	177
	jr nz, rub out	32 247
	pop af	241
restore	inc de	19
	rx de,hl	235
	jr check	24 164
bit 5	bit 5,a	203 111
	jr z, string array	40 5
	ld de, 14	17 14 0
	jr 'zero'	24 170
string array	ld a, 32	62 32
	jr find length	24 210

How it works

The hl register pair is loaded with the address of the start of the variables area. The accumulator is loaded with the byte stored at hl. If the value of this byte is 128 the routine returns to BASIC, because the code 128 marks the end of the variables. The de register is loaded with the value one for use later in the routine. If bit 7 of the accumulator is set to one, the routine jumps to 'next bit' then, if bit 5 is set to zero, the routine jumps to 'string'.

To reach 'zero' without jumping ahead in the routine, the variable found must be a number whose name is one letter long. The b register is set to five, to be used as a counter, hl is incremented to point to the next byte and this is POKEd with zero. The counter is decremented, and if zero has not been reached the routine loops back to 'next byte'. de is then added to hl to point to the next variable and a jump is made back to 'check'.

If the routine reaches 'string' hl is incremented to point to the bytes holding the length of string found. The old length is loaded into bc to be used as a counter, and the new length is set to zero. hl is again incremented to point to the text of the string. If the counter is set to zero, hl now points to the next variable and so a jump is made back to 'check'. If not, then bc is saved on the stack and the ROM routine at address 4120 is called to delete one character. The counter is then retrieved from the stack, decremented, and a jump is made back to 'delete'.

At 'next bit', bit six of the accumulator is checked. If it is set to one a jump is made to 'bit 5' as a string array or FOR/NEXT control variable has been found. If it is set to zero, and bit five is set to zero a jump is made to 'array'.

To reach 'number' the variable found must be a number with a name more than one character long. The hl register pair is incremented to point to the next byte, and this is repeated until a byte is encountered with bit

seven set to one. When this is found the routine jumps to 'zero' to load with variable with nought.

If the routine reaches 'array' the accumulator is loaded with zero, because this is the value which the elements must be set to later.

At 'find length' the accumulator is saved on the stack and hl is incremented to point to the bytes holding the array length. This is copied into bc for use as a counter. hl is agian incremented, so that it now points to the byte holding the number of dimensions, and then hl is saved on the stack. hl is loaded with the number of dimensions and this is multiplied by two. de is set to the address saved on the stack, then de is incremented hl times and bc is decremented (hl + 1) times. de is then incremented and bc decremented again. de now points to the next element of the array and bc holds the number of bytes left before the end is reached. The accumulator is retrieved from the stack and this is POKEd into de. The counter in bc is decremented, and if it does not hold zero the routine jumps back to 'rub out'. The value in hl is then adjusted to point to the next variable, and a jump is made to 'check'.

At 'bit 5' a test is made to see if a string array has been encountered. If so, the accumulator is set to the code for a space and a jump is made to 'find length'. To reach this point, the variable must be a FOR/NEXT control variable. de is set to 14 so that adding this to (hl + 5) points to the next variable. The routine then jumps back to 'zero'.

List Variables

Length: 94
Number of Variables: O
Check sum: 10295

Operation

This routine lists the names of all the variables presently in memory.

Call

RAND USR address

Error Checks

If there are no variables in memory the routine returns to BASIC immediately.

Comments

This is a useful aid for program debugging, particularly with long or complex programs.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	res O, (IY + 2)	253 203 2 134
	ld hl, (23627)	42 75 92

next variable	ld a, 13	62 13
	rst 16	215
	ld a, 32	62 32
	rst 16	215
	ld a, (hl)	126
	cp 128	254 128
	ret z	200
	bit 7,a	203 127
	jr z, bit 5	40 62
	bit 6,a	203 119
	jr z, next bit	40 31
	bit 5,a	203 111
	jr z, string array	40 9
	sub 128	214 128
	ld de, 19	17 19 0
print	rst 16	215
	add hl,de	25
	jr next variable	24 225
string array	sub 96	214 96
	rst 16	215
	ld a, 36	62 36
brackets	rst 16	215
	ld a, 40	62 40
	rst 16	215
	ld a, 41	62 41
pointers	inc hl	35
	ld e, (hl)	94
	inc hl	35
	ld d, (hl)	86
	inc hl	35
	jr print	24 234
next bit	bit 5,a	203 111
	jr z, array	40 19
	sub 64	214 64
	rst 16	215
next character	inc hl	35
	ld a, (hl)	126
	bit 7,a	203 127
	jr nz, last character	32 3
	rst 16	215
	jr next character	24 247
last character	sub 128	214 128
jump	ld de, 6	17 6 0
	jr print	24 211
array	sub 32	214 32

	jr brackets	24 216
bit 5	bit 5,a	203 111
	jr nz, jump	32 243
	add a, 32	198 32
	rst 16	215
	ld a, 36	62 36
	jr pointers	24 211

How it works

Bit O of the byte at address 23612 is reset to ensure that any characters PRINTed appear in the top part of the screen. hl is loaded with the address of the variables area. The accumulator is loaded with the ENTER token and this is PRINTed using the ROM routine at address 16. The accumulator is then loaded with the code for a space and this is PRINTed using the same routine.

The accumulator is loaded with the byte stored at the address in hl. If the value of this is 128 the routine returns to BASIC because the end of the variables area has been reached.

If bit 7 of the accumulator is set to zero the routine jumps to 'bit 5' because a string, or a number whose name is one letter only, has been encountered. bit 6 of the accumulator is tested. If it is set to zero a jump is made to 'next bit' because an array, or a number whose name is more than one letter, has been found. If bit 5 of the accumulator is zero the routine jumps to 'string array'.

The routine reaches this point if the variable found is the control of a FOR/NEXT loop, 128 is subtracted from the accumulator, the result being the code of the character to be PRINTed. de is loaded with 19 to point to the next variable when added to hl, the character in the accumulator is PRINTed, de is added to hl, and the routine loops back to 'next variable'.

If the routine reaches 'string array' 96 is subtracted from the accumulator, to give the code of the name of the array found. This is PRINTed using the ROM routine. A dollar sign and an open-bracket are then PRINTed, and the accumulator is loaded with the code of a close-bracket. hl is increased to point to the bytes holding the length of the array. This is loaded into de, so that adding to hl gives the address of the next variable. A jump is made to 'print' where the close-bracket is PRINTed and de is added to hl.

At 'next bit', bit 5 of the accumulator is tested. If it is set to zero, a jump is made to 'array'. If it is set to one, a number has been found whose name is longer than one letter. 64 is subtracted from the accumulator and the resulting character is PRINTed. Then the routine loops, PRINTing each character encountered, until one is found with bit 7 set to one. 128 is subtracted from the character code, de is loaded with the displacement to the next variable, and the routine jumps to 'print'.

If an array is found 32 is subtracted from the accumulator to give the correct code, and a jump is made to 'brackets'.

At 'bit 5', if a number has been found whose name is one letter only, the routine loops back to 'jump'.

To get to this section, the variable encountered must be a string. Subtracting 32 from the accumulator gives the code to be PRINTed. Finally the accumulator is loaded with the code for a dollar sign, and a jump is made to 'pointers'.

Search and List

Length: 155
Number of Variables: 2
Check sum: 17221

Operation

This routine searches through a BASIC program and lists every line containing a string of characters specified by the user.

Variables

Name	*Length*	*Location*	*Comments*
data start	2	23296	address of first byte of data
string length	1	23298	number of characters in string

Call

RAND USR address

Error Checks

If there is no BASIC program in memory or the string is zero characters in length, the routine returns to BASIC immediately.

Comments

The time taken by this routine is proportional to both the length of the string and the length of the BASIC program.

The string to be searched for should be POKEd above RAMTOP and the address of the first byte of the string POKEd into 23296/7. The string length should be stored in 23298.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	res O, (IY + 2)	253 203 2 134
	ld ix, (23296)	221 42 0 91
	ld hl, (23635)	42 83 92
restart	ld a, (23298)	58 2 91
	ld e, a	95
	cp O	254 0

	ret z	200
	push hl	229
restore	push ix	221 229
	pop bc	193
	ld d, O	22 0
	inc hl	35
	inc hl	35
	inc hl	35
check	inc hl	35
	push de	213
	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	add hl,de	25
	pop de	209
	jr c, enter	56 4
	pop hl	225
	ret	201
long jump	jr restart	24 223
enter	ld a, (hl)	126
	cp 13	254 13
	jr nz, number	32 5
	inc hl	35
	pop bc	193
	push hl	229
	jr restore	24 221
number	call 6326	205 182 24
	jr nz, compare	32 8
	dec hl	43
different	push ix	221 229
	pop bc	193
	ld d, O	22 0
	jr check	24 216
compare	ld a, (bc)	10
	cp (hl)	190
	jr nz, different	32 245
	inc bc	3
	inc d	20
	ld a,d	122
	cp e	187
	jr nz, check	32 206
	ld a, 13	62 13
	rst 16	215
	pop hl	225
	push hl	229
	ld b, (hl)	70

```
                inc hl                  35
                ld l, (hl)              110
                ld h, b                 96
                ld de, 1000             17 232 3
                ld a, 47                62 47
thousands       inc a                   60
                and a                   167
                sbc hl,de               237 82
                jr nc, thousands        48 250
                add hl, de              25
                rst 16                  215
                ld de, 100              17 100 0
                ld a, 47                62 47
hundreds        inc a                   60
                and a                   167
                sbc hl, de              237 82
                jr nc, hundreds         48 250
                add hl,de               25
                rst 16                  215
                ld de, 10               17 10 0
                ld a, 47                62 47
tens            inc a                   60
                and a                   167
                sbc hl,de               237 82
                jr nc, tens             48 250
                add hl, de              25
                rst 16                  215
                ld a,l                  125
                add a, 48               198 48
                rst 16                  215
                pop hl                  225
                inc hl                  35
                inc hl                  35
                inc hl                  35
next character  inc hl                  35
                ld a, (hl)              126
line end        cp 13                   254 13
                jr nz, chr 14           32 4
                rst 16                  215
                inc hl                  35
                jr long jump            24 155
chr 14          call 6326               205 182 24
                jr z, line end          40 243
                cp 32                   254 32
                jr c, next character    56 237
                rst 16                  215
                jr next character       24 234
```


How it works

Bit O of the byte stored at address 23612 is reset to ensure that any characters PRINTed appear in the top part of the screen. ix is loaded with the address of the first byte of data. This allows the address to be loaded into other register pairs, using less references to the printer buffer. hl is loaded with the address of the BASIC program.

The accumulator is loaded with the length of the string and this is copied into the e register. If the length is zero the routine returns to BASIC immediately. The address in hl is saved on the stack, holding the position in memory of the line currently being searched.

The address of the data is copied from ix into bc to be more accessible. The d register is loaded with zero, ie the number of characters found that match the data entered. The hl register pair is increased by three to point to the high byte of the line pointer. hl is incremented to point to the next character. The de register pair is saved on the stack.

de is loaded with the address of the variables area, and this is subtracted from hl. If the result is negative the routine jumps to 'enter' after restoring hl and retrieving de from the stack. If the result was positive the stack is restored to its original size and the routine returns to BASIC, as the end of the BASIC program had been reached.

At 'enter' the accumulator is loaded with the byte stored at the address in hl. If this is not the ENTER token a jump is made to 'number'. If the ENTER token is found hl is increased to point to the start of the next line. The address of the previous line is removed from the stack and is replaced by the new value in hl. Then a jump is made to 'restore'.

At 'number' the ROM routine at address 6326 is called. If the character in the accumulator is the NUMBER token, hl is increased to point to the first character after the binary representation of the number found by the ROM routine. If the NUMBER token is not found the routine jumps to 'compare', otherwise hl is decremented and the routine 'falls through' to 'different'. bc is copied from ix, the number of characters found is reset to zero, and a jump is made to 'check'.

At 'compare' the accumulator is loaded with the byte stored at the address in bc. If this is not the same as the byte stored at the address in hl, the routine loops back to 'different'. bc is incremented to point to the next data byte, and the number of characters found is increased. If this is not equal to the length of the string the routine loops back to 'check'.

The accumulator is loaded with the code of the ENTER token and this is PRINTed using the ROM routine at address 16. The address of the line to be PRINTed is loaded from the stack into hl. The line number is then copied into hl via the b register. de is loaded with 1000 and the accumulator is loaded with one less than the code of the "O" character. The accumulator is incremented and de is repeatedly subtracted from hl until hl is negative. Then de is added once to hl to produce a positive remainder. The character in the accumulator is then PRINTed.

The above process is then repeated for de = 100 and de = 10. Then the remainder is loaded into the accumulator, 48 is added, and the resultant character is PRINTed.

The address of the start of the line is retrieved from the stack, and loaded into hl. Then hl is increased to point to the high byte of the line pointer, hl is incremented, and the byte at hl is loaded into the accumulator. If this byte is not the ENTER token a jump is made to 'chr 14', otherwise the ENTER is PRINTed, hl is incremented, and a jump is made back to 'restart'.

At 'chr 14', the ROM routine at address 6326 is called. If the character in the accumulator is the NUMBER token, hl is increased to point to the first character after the number found, this character is loaded into the accumulator, and a jump is made to 'line end'. Then, if the character in the accumulator has a code less than 32, the routine loops back to 'next character'. If the code is more than 31 the character found is PRINTed and a jump is made to 'next character'.

Search and Replace

Length: 85
Number of Variables: 3
Check sum: 8518

Operation

This routine searches a BASIC program for a string of characters, and replaces every occurrence by another string of the same length.

Variables

Name	*Length*	*Location*	*Comment*
old data start	2	23296	address of string to be replaced
string length	1	23298	length of string to be replaced
new data start	2	23299	address of replacement string

Call

RAND USR address

Error Checks

If the string length is zero or there is no BASIC program in memory the routine returns to BASIC immediately.

Comments

The time taken by this routine is dependent on the string length and the length of the BASIC program in memory.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld ix, (23296)	221 42 0 91
	ld hl, (23635)	42 83 92
	ld a, (23298)	58 2 91
	ld e, a	95
	cp O	254 O
	ret z	200
	dec hl	43
new line	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr reset	24 23
check	inc hl	35
	push de	213
	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	add hl,de	25
	pop de	209
	ret nc	208
	ld a, (hl)	126
	cp 13	254 13
	jr z, new line	40 233
	call 6326	205 182 24
	jr nz, compare	32 8
	dec hl	43
reset	push ix	221 229
	pop bc	193
	ld d, O	22 0
	jr check	24 226
compare	ld a, (bc)	10
	cp (hl)	190
	jr nz, reset	32 245
	inc bc	3
	inc d	20
	ld a,d	122
	cp e	187
	jr nz, check	32 216
	push hl	229
	ld d, O	22 0
	and a	167
	sbc hl,de	237 82
	ld d,e	83
	ld bc, (23299)	237 75 3 91
	inc d	20

```
next char   inc hl        35
            dec d         21
            jr z, finish  40 5
            ld a, (bc)    10
            ld (hl),a     119
            inc bc        3
            jr next char  24 247
finish      pop hl        225
            jr reset      24 215
```

How it works

ix is loaded with the address of the string to be searched for. This should be above RAMTOP. hl is loaded with the address of the program area, and the accumulator is loaded with the length of the string. The length is copied into the e register for use later in the program. If the string length is zero the routine returns to BASIC. hl is adjusted to point to the high byte of the next BASIC line pointer and a jump is made to 'reset'.

At 'check' hl is incremented to point to the next character. de is saved on the stack, and loaded with the address of the variables area. If hl is not less than de the end of the program has been reached and so, after restoring de from the stack, the routine returns to BASIC.

The accumulator is loaded with the character addressed by hl. If this is the ENTER token the routine loops back to 'newline'. If the accumulator does not hold the NUMBER token (character 14) a jump is made to 'compare', otherwise hl is increased by five, so that hl points to the fifth byte of the number found.

At 'reset' bc is loaded with the address of the string to be searched for. The d register is set to zero to hold the number of characters in the string found so far. The routine then loops back to 'check'.

At 'compare' the accumulator is loaded with the character in the string that is pointed to by bc. If this is different to the byte addressed by hl, the routine jumps to 'reset'. bc is incremented to point to the next character in the string, and the counter in the d register is incremented. If this is not equal to the length of the string the routine loops back to 'check'.

If the string has been found hl is saved on the stack, so that the routine starts searching for the next occurrence from this address. de is loaded with the length of the string and this is subtracted from hl to give one less than the start address. The string length is then loaded into d for use as a counter. bc is loaded with the start address of the new string, and the d register is incremented. The hl register is incremented to point to the next location and the counter is decremented. If the counter holds zero hl is retrieved from the stack and a jump is made to 'reset' to find the next occurrence. The accumulator is loaded with the character pointed to by bc and this is POKEd into hl. bc in incremented to point to the next character and the routine loops back to 'next char'.

ROM Search

Length: 58
Number of Variables: 3
Check sum: 6533

Operation

This routine searches the ROM for a pattern of bytes specified by the user.

Variables

Name	*Length*	*Location*	*Comment*
search start	2	23296	start of memory to be searched
string length	1	23298	number of bytes in string
data start	2	23299	address of string in RAM

Call

PRINT USR address

Error Checks

If the length of the string is zero the routine returns to BASIC immediately, giving the address of the start of the data. If the string is not found the value of 65535 is returned.

Comments

When writing machine code programs this routine can be used to find subroutines in the ROM, if the user already knows how part of the routine is written.

As most of the Spectrum ROM is adapted from the ZX81, programs originally written for the ZX81 can be easily adapted. For example, the 'Line Address' routine calls the ROM routine at address 6510. On the ZX81, the routine starts at address 2520. Disassembling this routine gives:

```
push hl
ld hl, program
ld d,h*
ld e,l*
pop bc*
call 09EA
```

The three bytes marked by an asterisk are the same on the Spectrum, and may be found using the search routine. In fact the routine returns the address 6514 which is four plus the start address of the required ROM routine.

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*
	ld hl, (23296)	42 0 91
	ld de, (23298)	237 91 2 91
restart	ld bc, (23299)	237 75 3 91
	ld a,e	123
	cp O	254 O
	ret z	200
	push hl	229
	ld d, O	22 O
compare	ld a, (bc)	10
	cp (hl)	190
	jr z, match	40 25
	pop hl	225
	inc hl	35
	push de	213
	push hl	229
	ld hl, 16384	33 0 64
	l d, O	22 0
	and a	167
	sbc hl,de	237 82
	inc hl	35
	pop de	209
	and a	167
	sbc hl,de	237 82
	ex de,hl	235
	pop de	209
	jr nz, restart	32 220
	ld bc, 65535	1 255 255
	ret	201
match	inc d	20
	ld a,d	122
	cp e	187
	jr nz, next byte	32 2
	pop bc	193
	ret	201
next byte	inc hl	35
	inc bc	3
	jr compare	24 216

How it works

The hl register pair is loaded with the address of the first location in memory that is to be checked. To find the first occurrence in the ROM, this should be set to zero. The e register is loaded with the number of bytes in the string being searched for. The bc register pair is loaded with the address of the

string, entered by the user, in RAM. The accumulator is loaded with the string length, and if this is zero the routine returns to BASIC.

The address in hl is saved on the stack. The accumulator is loaded with the byte pointed to by the bc register pair. If this is the same as the byte pointed to by hl a jump is made to match. If the two bytes are different, hl is loaded with the address on the stack. This is then incremented to point to the next location in memory.

The de and hl registers are saved on the stack, hl is loaded with the address of the first byte of RAM, and de is loaded with the string length. de is subtracted from hl to give the highest possible start address for the string. This is incremented to point to the first address at which the string could not be held.

The address on the top of the stack is loaded into de and this is subtracted from hl. The result of this operation is remembered whilst hl is loaded with the contents of de, and de is loaded with the number on the stack. If the result was zero, bc is loaded with 65535 and the routine returns to BASIC as the string does not exist in the ROM. If the result was not zero the routine loops back to 'restart'.

At 'Match' the d register is incremented to hold the number of bytes found that have matched. If this equals the length of the string bc is retrieved from the stack and the routine returns to BASIC. If the d register did not hold the length of the string, hl and bc are both incremented to point to the next bytes and the routine loops to 'compare'.

Instrs

Length: 168

Number of Variables: O

Check sum: 19875

Operation

This routine returns the position of a substring (B$) in a main string (A$), or zero if an error occurs.

Call

Let P = USR address

Error Checks

If either string does not exist, the length of the substring is zero, or the length of the substring is more than the length of the main string, the routine returns the value zero.

If an error does not occur, but the substring cannot be found in the main string, the routine also returns to zero.

Comments

On return from the machine code routine the variable P (any other variable could be used) will hold the return value. The strings referred to cannot

be DIMensioned as character arrays. To change the strings used, the asterisked numbers should be altered. The 66* is the substring, the 65* being the mainstring. To alter these, replace the numbers by the codes of the characters required (A to Z = 65 to 90).

Machine Code Listing

Label	*Assembly language*	*Numbers to be entered*	
	sub a	151	
	ld b,a	71	
	ld c,a	79	
	ld d, a	87	
	ld e,a	95	
	ld hl, (23627)	42 75 92	
next variable	ld a, (hl)	126	
	cp 128	254 128	
	jr z, not found	40 95	
	bit 7,a	203 127	
	jr nz, for-next	32 41	
	cp 96	254 96	
	jr nc, number	48 29	
	cp 65	254 65*	MAINSTR$
	jr nz, substring	32 2	
	ld d,h	84	
	ld e,l	93	
substring	cp 66	254 66*	SUBSTR$
	jr nz, check	32 2	
	ld b,h	68	
	ld c,l	77	
check	ld a,d	122	
	or e	179	
	jr z, string	40 4	
	ld a, b	120	
	or c	177	
	jr nz, found	32 38	
string	push de	213	
	inc hl	35	
	ld e, (hl)	94	
	inc hl	35	
	ld d, (hl)	86	
add	add hl,de	25	
	pop de	209	
	jr increase	24 5	
number	inc hl	35	
	inc hl	35	
	inc hl	35	

	inc hl	35
	inc hl	35
increase	inc hl	35
	jr next variable	24 206
for-next	cp 224	254 224
	jr c, next bit	56 6
	push de	213
	ld de, 18	17 18 0
	jr add	24 234
next bit	bit 5,a	203 111
	jr z, string	40 225
next byte	inc hl	35
	bit 7, (hl)	203 126
	jr z, next byte	40 251
	jr number	24 227
found	ex de,hl	235
	inc hl	35
	inc hl	35
	push hl	229
	push hl	229
	inc bc	3
	push bc	197
	ld a, (bc)	10
	ld e,a	95
	inc bc	3
	ld a, (bc)	10
	ld d,a	87
	or e	179
	jr z, zero length	40 11
	push de	213
	ld a, (hl)	126
	dec hl	43
	ld l, (hl)	110
	ld h,a	103
	and a	167
	sbc hl,de	237 82
	jr nc, continue	48 8
	pop bc	193
zero length	pop bc	193
	pop bc	193
error	pop bc	193
not found	ld bc, O	1 0 0
	ret	201
continue	pop ix	221 225
	pop bc	193

```
        ex de,hl        235
        pop hl          225
        inc bc          3
        inc bc          3
save    inc hl          35
        push hl         229
        push bc         197
        push ix         221 229
        push de         213
compare ld a, (bc)      10
        cp (hl)         190
        jr z, match     40 12
        pop de          209
        pop ix          221 225
        pop bc          193
        pop hl          225
        ld a,d          122
        or e            179
        jr z, error     40 225
        dec de          27
        jr save         24 234
match   inc hl          35
        inc bc          3
        push hl         229
        dec ix          221 43
        push ix         221 229
        pop hl          225
        ld a,h          124
        or l            181
        pop hl          225
        jr nz, compare  32 227
        pop de          209
        pop de          209
        and a           167
        sbc hl,de       237 82
        pop de          209
        pop de          209
        pop de          209
        and a           167
        sbc hl,de       237 82
        ld b,h          68
        ld c,l          77
        ret             201
```

How it works

The accumulator, the bc register pair and the de register pair are all loaded with zero. Later in the routine bc will be set to the address of B$ and de will

be set to the address of A$. hl is loaded with the address of the variables area.

The accumulator is loaded with the byte addressed by hl. If the accumulator holds 128 the routine jumps to 'not found' as the end of the variables area has been reached. If bit 7 of the accumulator is set to one a jump is made to 'for-next' as the variable found is not a string or a number whose name is one letter only. If the accumulator holds a number larger than 95 a jump is made to 'number'.

To reach this stage, a string must have been found. If the accumulator holds 65, A$ has been located and the contents of hl are copied into de. If the accumulator holds 66, B$ has been found and hl is copied into bc. If de does not hold zero and bc does not hold zero, both strings have been located, and so the routine jumps to 'found'.

If the routine reaches 'string' de is saved on the stack, and loaded with the length of the string encountered. This is added to the address of the high byte of the string pointers, and stored in hl. de is retrieved from the stack, and a jump is made to 'increase'.

At 'number', hl is incremented five times to point to the last byte of any number encountered. hl is then incremented to point to the next variable, and a jump made to 'next variable'.

At 'for next', if the accumulator holds a number below 224 a jump is made to 'next bit' as the variable encountered is not a FOR—NEXT loop control. If the value in the accumulator is more than 223, eighteen is added to hl to point to the last byte of the control, and the routine loops to 'increase'.

If the routine reaches 'next bit' and bit 5 of the accumulator is set to zero a jump is made to 'string', to load hl with the address of the following variable, as an array has been found.

If the routine reaches 'next byte' a number has been found whose name is more than one character in length. Thus hl is increased until it points to the last character of the variable name, and then a jump is made to 'number'.

At 'found' hl is loaded with the address of A$ and this is incremented twice to give the address of the high byte of the pointers. This value is then saved on the stack twice. bc is incremented to point to the low byte of the pointers for B$. The address in bc is then saved on the stack. de is loaded with the length of B$, and if this is zero a jump is made to 'zero length'. de is then PUSHed onto the stack. hl is loaded with the length of A$, and if this is not less than de the routine jumps to 'continue'. The stack is then restored to its original size, bc is loaded with zero, and the routine returns to BASIC.

At 'continue' ix is set to the length of B$, and bc is set to the address of the low byte of the pointers for B$. de is loaded with the difference in lengths of A$ and B$, and hl is loaded with the address of the high byte of the pointers for A$. bc is then incremented twice to give the address of the first character in B$. hl is incremented to point to the next character of A$.

hl, bc, ix and de are then saved on the stack. The accumulator is loaded with the byte addressed by bc, and if this is the same as the byte addressed by hl a jump is made to 'match'. de, ix, bc and hl are then retrieved from the stack. If de holds zero a jump is made to 'error' as B$ does not occur in A$. The counter in de is then decremented, and the routine loops back to 'save'.

If the routine reaches 'match' hl and bc are both incremented to point to the next characters of A$ and B$ respectively. hl is then saved on the stack. The counter in ix is decremented and, after retrieving hl from the stack, if ix does not hold zero a jump is made back to 'compare'.

To reach this stage, an occurrence of B$ must have been found in A$. The length of B$ is then subtracted from hl, and then the address of the high byte of the pointers for A$ is subtracted from hl. The result is the position in A$ of B$. This is copied into the bc register pair and the routine returns to BASIC.

APPENDIX A

There are two main tables of instructions in this appendix. Table A2. lists the one byte instructions and those two byte instructions which are preceded by 203 (hexadecimal CB) or 237 (hexadecimal ED). Table A3. lists the index register instructions.

There are many patterns in the instruction set. For example, the registers are almost always in the order b, c, d, e, h, l, (hl), a as in, for example, the group of 8 bit register to register load instruction, numbers 64 to 127. Similarly, the index register codes mimic the hl codes, being preceded by 221 (hexadecimal DD) when referring to the ix index register and by 233 (hexadecimal FD) when referring to the iy index register.

Some of the instructions are qualified by one or more of the following:

n a one byte integer between 0 and 255 inclusive

d a one byte displacement between 0 and 255 inclusive (index register instructions) or between −127 and 128 (jump instructions). Negative values of d are obtained by subtracting the positive value from 256.

nn a two byte integer between 0 and 65535 inclusive. The most significant byte lies second, for example 16384 (= 0 + 256*64) is held as 0,64.

Qualifiers are always placed in the byte or bytes following the instruction to which they refer, except in three byte index register instructions (columns 5 and 6 of table A.3) in which case they are placed between the second and third bytes. See table A1 for examples.

Table A1 *Some examples of the Z80A instruction set. Column one refers to the appropriate column in tables A2 and A3.*

Table and column number	*General form*	*Specific example*	*Decimal*			
A2,3	—	inc b	4			
A2,3	ld e,n	ld e,25	30	25		
A2,3	ld a, (nn)	ld a, (23296)	58	0	91	
A2,4	—	res 2,d	203	92		
A2,5	ld (nn),de	ld (23760),de	237	53	208	92
A3,3	—	add ix,bc	221	9		
A3,3	ld (ix + d),n	ld (ix + 193),5	221	54	193	5
A3,4	—	add iy,bc	253	9		
A3,4	ld (nn), iy	ld (23760),iy	253	34	208	92
A3,5	rrc (ix + d)	rrc (ix + 5)	221	203	5	14
A3,6	rrc (iy + d)	rrc (iy + 5)	253	203	5	14

Table A2. *Z80A instructions except for index register codes (see table A3).*

Decimal	*Hex*	*Op Code*	*After 203 (hex CB)*	*After 237 (hex ED)*
0	00	nop	rlc b	
1	01	ld bc,nn	rlc c	

2	02	ld (bc),a	rlc d
3	03	inc bc	rlc e
4	04	inc b	rlc h
5	05	dec b	rlc l
6	06	ld b,n	rlc (hl)
7	07	rlca	rlc a
8	08	ex,af,af'	rrc b
9	09	add hl,bc	rrc c
10	0A	ld a, (bc)	rrc d
11	0B	dec bc	rrc e
12	0C	inc c	rrc h
13	0D	dec c	rrc l
14	0E	ld c,n	rrc (hl)
15	0F	rrca	rrc a
16	10	djnz d	rl b
17	11	ld de,nn	rl c
18	12	ld (de),a	rl d
19	13	inc de	rl e
20	14	inc d	rl h
21	15	dec d	rl l
22	16	ld d,n	rl (hl)
23	17	rla	rl a
24	18	jr d	rr b
25	19	add hl,de	rrc
26	1A	ld a, (de)	rr d
27	1B	dec de	rr e
28	1C	inc e	rr h
29	1D	dec e	rr l
30	1E	ld e,d	rr (hl)
31	1F	rra	rr a
32	20	jr nz,d	sla b
33	21	ld hl,nn	sla c
34	22	ld (nn),hl	sla d
35	23	inc hl	sla e
36	24	inc h	sla h
37	25	dec h	sla l
38	26	ld h,n	sla (hl)
39	27	daa	sla a
40	28	jr z,d	sra b
41	29	add hl,hl	sra c
42	2A	ld hl, (nn)	sra d
43	2B	dec hl	sra e
44	2C	inc l	sra h
45	2D	dec l	sra l
46	2E	ld l,n	sra (hl)
47	2F	cpl	sra a
48	30	jr nc,d	

49	31	ld sp,nn		
50	32	ld (nn),a		
51	33	inc sp		
52	34	inc (hl)		
53	35	dec (hl)		
54	36	ld (hl),n		
55	37	scf		
56	38	jr c,d	srl b	
57	39	add hl,sp	srl c	
58	3A	la d, (nn)	srl d	
59	3B	dec sp	srl e	
60	3C	inc a	srl h	
61	3D	dec a	srl l	
62	3E	ld a,n	srl (hl)	
63	3F	ccf	srl a	
64	40	ld b,b	bit 0,b	in b, (c)
65	41	ld b,c	bit 0,c	out (c),b
66	42	ld b,d	bit 0,d	sbc hl,bc
67	43	ld b,e	bit 0,e	ld (nn),bc
68	44	ld b,h	bit 0,h	neg
69	45	ld b,l	bit 0,l	retn
70	46	ld b, (hl)	bit 0, (hl)	im 0
71	47	ld b,a	bit 0,a	ld i,a
72	48	ld c,b	bit 1,b	in c, (c)
73	49	ld c,c	bit 1,c	out (c),c
74	4A	ld c,d	bit 1,d	adc hl,bc
75	4B	ld c,e	bit 1,e	ld bc,(nn)
76	4C	ld c,h	bit 1,h	
77	4D	ld c,l	bit 1,l	reti
78	4E	ld c, (hl)	bit 1, (hl)	
79	4F	ld c,a	bit 1,a	ld r,a
80	50	ld d,b	bit 2,b	in d, (c)
81	51	ld d,c	bit 2,c	out (c),d
82	52	ld d,d	bit 2,d	sbc hl,de
83	53	ld d,e	bit 2,e	ld (nn),de
84	54	ld d,h	bit 2,h	
85	55	ld d,l	bit 2,l	
86	56	ld d, (hl)	bit 2, (hl)	im 1
87	57	ld d,a	bit 2,a	ld a,i
88	58	ld e,b	bit 3,b	in e,(c)
89	59	ld e,c	bit 3,c	out (c),e
90	5A	ld e,d	bit 3,d	adc hl,de
91	5B	ld e,e	bit 3,e	ld de,(nn)
92	5C	ld e,h	bit 3,h	
93	5D	ld e,l	bit 3,1	
94	5E	ld e, (hl)	bit 3, (hl)	im 2
95	5F	ld e,a	bit 3,a	ld a,r

96	60	ld h,b	bit 4,b	in h, (c)
97	61	ld h,c	bit 4,c	out (c),h
98	62	ld h,d	bit 4,d	sbc hl,hl
99	63	ld h,e	bit 4,e	ld (nn),hl
100	64	ld h,h	bit 4,h	
101	65	ld h,l	bit 4,l	
102	66	ld h, (hl)	bit 4, (hl)	
103	67	ld h,a	bit 4,a	rrd
104	68	ld l,b	bit 5,b	in l,(c)
105	69	ld l,c	bit 5,c	out (c),l
106	6A	ld l,d	bit 5,d	adc hl,hl
107	6B	ld l,e	bit 5,e	ld hl,(nn)
108	6C	ld l,h	bit 5,h	
109	6D	ld l,l	bit 5,l	
110	6E	ld l, (hl)	bit 5, (hl)	
111	6F	ld l,a	bit 5,a	rld
112	70	ld (hl),b	bit 6,b	in f,(c)
113	71	ld (hl),c	bit 6,c	
114	72	ld (hl),d	bit 6,d	sbc hl,sp
115	73	ld (hl),e	bit 6,e	ld (nn),sp
116	74	ld (hl),h	bit 6,h	
117	75	ld (hl),l	bit 6,l	
118	76	halt	bit 6, (hl)	
119	77	ld (hl),a	bit 6,a	
120	78	ld a,b	bit 7,b	in a,(c)
121	79	ld a,c	bit 7,c	out (c),a
122	7A	ld a,d	bit 7,d	adc hl,sp
123	7B	ld a,e	bit 7,e	ld sp, (nn)
124	7C	ld a,h	bit 7,h	
125	7D	ld a,l	bit 7,l	
126	7E	ld a, (hl)	bit 7, (hl)	
127	7F	ld a,a	bit 7,a	
128	80	add a,b	res 0,b	
129	81	add a,c	res 0,c	
130	82	add a,d	res 0,d	
131	83	add a,e	res 0,e	
132	84	add a,h	res 0,h	
133	85	add a,l	res 0,l	
134	86	add a, (hl)	res 0, (hl)	
135	87	add a,a	res 0,a	
136	88	adc a,b	res 1,b	
137	89	adc a,c	res 1,c	
138	8A	adc a,d	res 1,d	
139	8B	adc a,e	res 1,e	
140	8C	adc a,h	res 1,h	
141	8D	adc a,l	res 1,l	
142	8E	adc a, (hl)	res 1, (hl)	

143	8F	adc a,a	res 1,a	
144	90	sub b	res 2,b	
145	91	sub c	res 2,c	
146	92	sub d	res 2,d	
147	93	sub e	res 2,e	
148	94	sub h	res 2,h	
149	95	sub l	res 2,l	
150	96	sub (hl)	res 2, (hl)	
151	97	sub a	res 2,a	
152	98	sbc a,b	res 3,b	
153	99	sbc a,c	res 3,c	
154	9A	sbc a,d	res 3,d	
155	9B	sbc a,e	res 3,e	
156	9C	sbc a,h	res 3,h	
157	9D	sbc a,l	res 3,1	
158	9E	sbc a, (hl)	res 3, (hl)	
159	9F	sbc a,a	res 3,a	
160	A0	and b	res 4,b	ldi
161	A1	and c	res 4,c	cpi
162	A2	and d	res 4,d	ini
163	A3	and e	res 4,e	outi
164	A4	and h	res 4,h	
165	A5	and l	res 4,1	
166	A6	and (hl)	res 4, (hl)	
167	A7	and a	res 4,a	
168	A8	xor b	res 5,b	ldd
169	A9	xor c	res 5,c	cpd
170	AA	xor d	res 5,d	ind
171	AB	xor e	res 5,e	outd
172	AC	xor h	res 5,h	
173	AD	xor l	res 5,l	
174	AE	xor (hl)	res 5, (hl)	
175	AF	xor a	res 5,a	
176	B0	or b	res 6,b	ldir
177	B1	or c	res 6,c	cpir
178	B2	or d	res 6,d	inir
179	B3	or e	res 6,e	otir
180	B4	or h	res 6,h	
181	B5	or l	res 6,1	
182	B6	or (hl)	res 6, (hl)	
183	B7	or a	res 6,a	
184	B8	cp b	res 7,b	lddr
185	B9	cp c	res 7,c	cpdr
186	BA	cp d	res 7,d	indr
187	BB	cp e	res 7,e	otdr
188	BC	cp h	res 7,h	
189	BD	cp l	res 7,l	

190	BE	cp (hl)	res 7, (hl)
191	BF	cp a	res 7,a
192	C0	ret nz	set 0,b
193	C1	pop bc	set 0,c
194	C2	jp nz, nn	set 0,d
195	C3	jp nn	set 0,e
196	C4	call nz,nn	set 0,h
197	C5	push bc	set 0,l
198	C6	add a,n	set 0, (hl)
199	C7	rst 0	set 0,a
200	C8	ret z	set 1,b
201	C9	ret	set 1,c
202	CA	jp z,nn	set 1,d
203	CB	—	set 1,e
204	CC	call z,nn	set 1,h
205	CD	call nn	set 1,l
206	CE	adc a,n	set 1, (hl)
207	CF	rst 8	set 1,a
208	D0	ret nc	set 2,b
209	D1	pop de	set 2,c
210	D2	jp nc,nn	set 2,d
211	D3	out (n),a	set 2,e
212	D4	call nc,nn	set 2,h
213	D5	push de	set 2,l
214	D6	sub,n	set 2, (hl)
215	D7	rst 16	set 2,a
216	D8	ret c	set 3,b
217	D9	exx	set 3,c
218	DA	jp c,nn	set 3,d
219	DB	in a,(n)	set 3,e
220	DC	call c,nn	set 3,h
221	DD	—	set 3,l
222	DE	sbc a,n	set 3, (hl)
223	DF	rst 24	set 3,a
224	E0	ret po	set 4,b
225	E1	pop hl	set 4,c
226	E2	jp po,nn	set 4,d
227	E3	ex (sp),hl	set 4,e
228	E4	call po,nn	set 4,h
229	E5	push hl	set 4,l
230	E6	and n	set 4, (hl)
231	E7	rst 32	set 4,a
232	E8	ret pe	set 5,b
233	E9	jp (hl)	set 5,c
234	EA	jp pe,nn	set 5,d
235	EB	ex de,hl	set 5,e
236	EC	call pe,nn	set 5,h

237	ED	—	set 5,l
238	EE	xor n	set 5, (hl)
239	EF	rst 40	set 5,a
240	F0	ret p	set 6,b
241	F1	pop af	set 6,c
242	F2	jp p,nn	set 6,d
243	F3	di	set 6,e
244	F4	call p,nn	set 6,h
245	F5	push af	set 6,l
246	F6	or n	set 6, (hl)
247	F7	rst 48	set 6,a
248	F8	ret m	set 7,b
249	F9	ld sp,hl	set 7,c
250	FA	jp m,nn	set 7,d
251	FB	ei	set 7,e
252	FC	call m,nn	set 7,h
253	FD	—	set 7,l
254	FE	cp n	set 7, (hl)
255	FF	rst 56	set 7,a

Table A3 *Index register codes. Columns 3 and 5 refer to the ix register. Columns 4 and 6 refer to the iy register. All the instructions in this table mimic the instructions for the hl register pair in table A2.*

Decimal	*Hex*	*After 221 (hex DD)*	*After 253 (hex FD)*	*After 221, 203 (hex DD, CB)*	*After 253, 203 (hex FD, CB)*
6	06			rlc (ix + d)	rlc (iy + d)
9	09	add ix, bc	add iy,bc		
14	OE			rrc (ix + d)	rrc (iy + d)
22	16			rl (ix + d)	rl (iy + d)
25	19	add ix,de	add iy,de		
30	IE			rr (ix + d)	rr (iy + d)
33	21	ld ix,nn	ld iy,nn		
34	22	ld (nn),ix	ld (nn),iy		
35	23	inc ix	inc iy		
38	26			sla (ix + d)	sla (iy + d)
41	29	add ix,ix	add iy,iy		
42	2A	ld ix,(nn)	ld iy,(nn)		
43	2B	dec ix	dec iy		
46	2E			sra (ix + d)	sra (iy + d)
52	34	inc (ix + d)	inc (iy + d)		
53	35	dec (ix + d)	dec (iy + d)		
54	36	ld (ix + d),n	ld (iy + d),n		
57	39	add ix,sp	add iy,sp		
62	3E			srl (ix + d)	srl (iy + d)
70	46	ld b, (ix + d)	ld b, (iy + d)	bit 0,(ix + d)	bit 0,(iy + d)
78	4E	ld c,(ix + d)	ld c,(iy + d)	bit 1,(ix + d)	bit 1,(iy + d)

86	56	ld d,(ix + d)	ld d,(iy + d)	bit 2,(ix + d)	bit 2,(iy + d)
94	5E	ld e,(ix + d)	ld e,(iy + d)	bit 3,(ix + d)	bit 3,(iy + d)
102	66	ld h,(ix + d)	ld h,(iy + d)	bit 4,(ix + d)	bit 4,(iy + d)
110	6E	ld l,(ix + d)	ld l,(iy + d)	bit 5,(ix + d)	bit 5,(iy + d)
112	70	ld (ix + d),b	ld (iy + d),b		
113	71	ld (ix + d),c	ld (iy + d),c		
114	72	ld (ix + d),d	ld (iy + d),d		
115	73	ld (ix + d),e	ld (iy + d),e		
116	74	ld (ix + d),h	ld (iy + d),h		
117	75	ld (ix + d),l	ld (iy + d),l		
118	76			bit 6, (ix + d)	bit 6, (iy + d)
119	77	ld (ix + d),a	ld (iy + d),a		
126	7E	ld a,(ix + d)	ld a,(iy + d)	bit 7, (ix + d)	bit 7,(iy + d)
134	86	add a,(ix + d)	add a,(iy + d)	res 0,(ix + d)	res 0,(iy + d)
142	8E	adc a,(ix + d)	adc a,(iy + d)	res 1,(ix + d)	res 1,(iy + d)
150	96	sub (ix + d)	sub (iy + d)	res 2,(ix + d)	res 2,(iy + d)
158	9E	sbc a,(ix + d)	sbc a,(iy + d)	res 3,(ix + d)	res 3,(iy + d)
166	A6	and (ix + d)	and (iy + d)	res 4,(ix + d)	res 4,(iy + d)
174	AE	xor (ix + d)	xor (iy + d)	res 5,(ix + d)	res 5,(iy + d)
182	B6	or (ix + d)	or (iy + d)	res 6,(ix + d)	res 6,(iy + d)
190	BE	cp (ix + d)	cp (iy + d)	res 7, (ix + d)	res 7, (iy + d)
198	C6			set 0,(ix + d)	set 0,(iy + d)
206	CE			set 1,(ix + d)	set 1,(iy + d)
214	D6			set 2,(ix + d)	set 2,(iy + d)
222	DE			set 3,(ix + d)	set 3,(iy + d)
225	E1	pop ix	pop iy		
227	E3	ex (sp),ix	ex (sp),iy		
229	E5	push ix	push iy		
230	E6			set 4,(ix + d)	set 4,(iy + d)
233	E9	jp ix	jp iy		
238	EE			set 5,(ix + d)	set 5,(iy + d)
246	F6			set 6,(ix + d)	set 6,(iy + d)
249	F9	ld sp,ix	ld sp,iy		
254	FE			set 7,(ix + d)	set 7,(iy + d)