MACHINE CODE EXTENSIONS FOR SPECTRUM BASIC

ROB BANKS

HEWSON CONSULTANTS

Machine Code Extensions for Spectrum Basic

by

Rob Banks

First edition 1985

ISBN 0-907912 04-4

Published by Hewson Consultants Ltd., 56B Milton Trading Estate, Milton, Oxon OX14 4RX.
Printed and bound in Great Britain by Powage Press, Milton Keynes.

PREFACE

by

Andrew Hewson

Author of the Help Line column in Sinclair User every month.

When Rob Banks first approached me with news of his Basic Extension system for the ZX Spectrum I was immediately intrigued. I had previously been aware that such systems were possible and had inspected several attempted implementations but I had never been satisfied with the results.

Rob's implementation achieves two important objectives. It is compact – the forty or so extended commands in this book require less than 4K of user RAM principally because of the extensive use made of Spectrum ROM routines. As a result the implementation also marries well with the "flow" of the Spectrum ROM – Rob's commands appear to the user as natural extensions of Spectrum Basic.

We have presented the material in book form rather than as a package on cassette in the belief that the majority of users are interested in understanding the software mechanisms involved rather than merely using the results. Thus the reader is invited to understand the techniques used so that he may develop and modify them for his own purposes if he so wishes. Nonetheless a cassette containing all the programs and routines listed in this book is available for those who wish to avoid the effort of keying and checking the listings*.

The book is divided into two sections to help the reader to get started. Section A gives the necessary background information about how the Extension system works and includes a useful chapter on the various ROM routines used. A listing of a flexible Basic Hexadecimal Loader is also given although most readers will prefer to use a machine code assembler such as ZAPP (Z80 Assembly Programming Package)*.

Section B lists and describes the various component parts of the Extension Program itself. The Fundamental Routines, Tables and Common Subroutines in chapters 1, 2 and 3 of section B should be studied and keyed in **in toto** but thereafter the reader is free to implement as many or as few of the commands as he wishes. Most readers will prefer to start off with simplest commands as listed in Chapter 4.1 so as to get "up and running" as quickly as possible.

When all the commands have been understood and implemented we hope that readers will be encouraged to build their own library of

* For details of how to obtain the Machine Code Extension cassette and ZAPP, the Z80 Assembly Programming Package please turn to the form at the back of this book.

further commands. The possibilities are endless – an integer-only Basic language for special applications such as accounts, a graphics language for constructing static and moving images, an adventure language, etc. We would be pleased to hear from readers of such developments.

Finally, please note that all programs and routines are fully debugged to the best of our knowledge and have been listed direct from the Spectrum so as to ensure complete fidelity. If any errors have slipped through the net please accept our apologies.

Andrew Hewson
November 1984

CONTENTS

INTRODUCTION

Although Sinclair BASIC is widely recognised as one of the easier introductions to computer programming, many more advanced users find its lack of structure and slow speed of execution somewhat frustrating.

In an effort to overcome these shortcomings, the author (himself a long-standing sufferer of the Spectrum's inadequacies) resorted to machine code in order to extend the BASIC Operating System to include the following 36 commands:

PROC – DEFPROC – ENDPROC
REPEAT – UNTIL, WHILE – WEND
IF – ELSE
ON – ONEND
KEYIN, DEFKEY – ENDKEY
ATTR,WIPE
CLA,CLP
SCROLL,SLEFT,SRIGHT
VPRINT
UNDL,DRAW
CAPS,CAPCHK
KYBD
NOISE
UDG
POKE
NEW
SPEED
DEL,REMKILL
RENUM
FREE

The way in which this was achieved is explained fully in this book.

Experience in Assembly Language programming is NOT assumed, but anyone with an elementary knowledge of machine code should be in a position, when they've finished this book, to add their own commands.

SECTION A

BACKGROUND INFORMATION

A.1 ABOUT THIS BOOK

This book is divided into two sections: section A gives background information on the workings of the extension program, and section B details the machine code routines themselves.

This first section explains how the extension program is used from BASIC, and the way in which it works. A list of all the ROM routines used by the program is given at the end of the section, and a versatile BASIC Hex Loader program is described in Chapter 3; this can be used to enter the machine code if an Assembler is not available.

The second part of this book is divided into five chapters. The routines in Chapters 1 and 2 cover the main structure of the BASIC Extension Program, and will need to be entered first. Follow these with the Common Subroutines in Chapter 3, and you will be ready to add your first extended commands from those listed in Chapter 4.

"Fundamental Commands" is perhaps a misleading title for the routines of section 4.1, because the other commands will all function perfectly well without them. However, as a matter of convenience you are strongly recommended to enter these first (although "*SPEED" is only required if you intend using Procedures and/or the "*KEYIN" command).

The remaining 34 command routines are described under the loose headings "Preliminary", "Structured", "Screen Handling" and "Utilities". The first of these covers a variety of functions, and is shown first because they tend to be shorter and easier to understand. Procedures, loops and other structured commands are given next, followed by a number of powerful graphics commands. Structured commands make programs easier to write and debug, whilst advanced screen handling reduces the amount of programming effort required to produce visual effects. Finally, a number of the Utility routines are described. These are intended to make the writing of BASIC programs easier, by carrying out the otherwise boring and time-consuming tasks of block deletion, line renumbering etc.

All the command routines in Chapter 4 are set out in identical format:

1. FUNCTION:	A brief description of what the commands are supposed to do.
2. SYNTAX:	The appearance of the command as it would be in a BASIC program.
3. PARAMETERS:	Numbers, variables etc. used by the command.
4. ERROR REPORTS:	Report messages issued by the command for out-of-range parameters etc.
5. LISTING:	Assembly listing of the routine, followed by a Hex Dump.
6. EXPLANATION:	A description of exactly how the routine works.

Generally speaking, all the commands are self-contained and can therefore be entered or omitted as required.

Ultimately of course, this book is designed to show the average user how to extend his Spectrum's command set to suit his personal requirements. It is therefore hoped that most readers will go on to add their own commands, with the help of Chapter 5.

A.2 USING THE EXTENSION PROGRAM

Just as with the standard instruction set, the extended commands have widely varying syntax requirements (syntax is the computer equivalent of the rules of grammar in human languages). These will therefore be explained with each individual command, but there are some points applicable to all commands:

1. The computer is first taken out of K-mode by typing an asterisk (symbol shift and "B").
2. The command is then typed in full, in upper and/or lower case letters – it doesn't matter which because the extension program will automatically convert any lower case characters to upper case when the command is accepted.
3. If no parameters are required, the command is ready and the ENTER key may be pressed as normal.
4. If anything is to follow the command (be it a name, a number or anything else), a space should now be inserted.
5. The relevant parameters are added, and the ENTER key pressed.

When the parameter(s) of an extended command are specified as numbers, they may be expressed in any of the conventional ways – i.e., a straight-forward number, a variable, a function, etc. If a parameter is specified as a condition, the program will actually accept anything producing a value between 0 and 255 inclusive; although it will only distinguish between zero and non-zero, so everything from one upwards is considered as being one.

When typing an extended command as the first instruction in a program line, a space may be inserted between the line number and the "*" symbol to align it with the rest of the listing (although it will appear indented when the cursor is present). This space may be usefully omitted with the loop instructions to highlight the start and end of the loop.

It is a good idea to type the new instructions in lower case – if the command is then rejected, the last character accepted will be the last one converted to upper case.

If the "syntax error" marker (a flashing question mark) appears immediately after the asterisk, and the first character has not been converted to upper case, then the extension program has been disabled. This would normally happen after NEW or RUN, but can be avoided by using the extended command *NEW, with the first BASIC program line being a call to the enabling routine. If one of the recommended BASIC loaders is used to download the code from cassette or microdrive, then the relevant USR call will automatically be in position at line 0 – and is thus uneditable, as well as being immune to *NEW. Both routines given auto-run, activating the extension program before *NEWing themselves; leaving only line 0 in place. You can then type in your BASIC program and RUN as usual, because the first command re-enables the extension program. Note the extension program must be in and running before either loader can be typed-in, as both use *NEW.

A.3 HEXADECIMAL LOADER

The machine code programs described in this book will be very much easier to enter if a good Assembler is used. However, for those without access to such software this chapter is devoted to a "Hex Loader" program, written in BASIC, which will facilitate the entry of all the routines using the Hex Dump featured at the end of each listing.

Once the program has been entered, it should immediately be SAVEd onto cassette or microdrive by using run 9500. It does not auto-run because it will often be necessary to LOAD sections of code beforehand, for correction and/or modification.

When RUN, the program will initially request a starting address: this should be entered (in hex) from the relevant listing – it's the four character number at the top lefthand corner of the Hex Dump. After a short pause, the screen will be cleared and a tabulation of the next 176 bytes (arranged as 22 rows of 8 bytes) printed. The address of the first byte in each row is shown in the extreme lefthand column; the top one, of course, being the starting address just entered. A flashing cursor will then appear at the top lefthand corner of the table, which can be moved using CAPS SHIFT and the relevant cursor key. It can be rapidly returned to this starting point using CAPS SHIFT and 1.

Entering the code could not be easier – the two digits of the byte are simply typed in hex, whereupon the cursor will move to the next byte. If you mis-type a digit, you will have to re-enter the whole byte as there is no "DELETE" key: just press CAPS SHIFT and 5 to jump back, and then enter the correct value.

The remaining commands are all single-key entries, and are summarised at the end of this chapter. Their functions are as follows:

IDENTIFY – Displays the address of the byte currently pointed to by the cursor.

PAGE COMMANDS – If the routine you are entering uses more memory than is presented on the screen, pressing "K" will tabulate the next block of 176 bytes; you can jump back again using "J". (These keys are easy to remember because "J" is also "–", and "K" is also "+").

QUIT – Control is returned to the start-up routine, allowing you to start entering code at a completely new address.

SAVE – Allows a block of code to be SAVEd onto cassette or microdrive.

CONVERT – Pressing "X" will enter Conversion Mode (to translate Decimal to Hex and vice versa). The cursor will disappear, and you will be prompted for a number: This must be typed-in with either a "D" or an "H" to indicate whether it is Decimal or Hex. For example, to convert the decimal number 1234 into hex, you would type:

1234D (ENTER)

The bottom line of the display would then read:

1234D → 4D2H

COPY – If you have a ZX Printer (or any other printer obeying the standard COPY command) you can make a screen dump of the table to compare with the Hex Dump from the relevant listing.

You are strongly recommended to SAVE each block of code as soon as it has been entered and checked. Although time-consuming, this is far better than losing several hours work from a momentary lapse of your Spectrum's memory.

HEX LOADER COMMANDS:

CAPS SHIFT + 1 : Home Cursor
+ 5 : Cursor Left
+ 6 : Cursor Down
+ 7 : Cursor Up
+ 8 : Cursor Right

0-9; A-F : Hexadecimal Entry Keys
I : Identify
J : Back one "page"
K : Forward one "page"
Q : Quit
S : Save CODE
X : Conversion Mode
Z : Copy

```
 10 REM ********************
 12 REM *                  *
 13 REM *   HEX  LOADER    *
 14 REM *                  *
 15 REM ********************
 16
 20 BORDER 1: PAPER 1: INK 7
 30 CLEAR 59999: POKE 23658,8
 40 GO SUB 9000: REM Initialise
 50 GO TO start
 60
 99 REM ************************
100 REM * HEX to DEC *
110 LET dec=0
120 FOR n=1 TO LEN h$
130 LET i=CODE h$(n): LET dec=dec+(i-7*FN a()-48)*16^(LEN h$-n)
140 NEXT n: RETURN
145
149 REM ************************
150 REM * DEC to HEX *
160 LET h$=""
170 LET rem=INT (16*(dec/16-INT (dec/16))): LET dec=INT (dec/16): LET h$=a$(rem
+1)+h$
180 IF dec<16 THEN  LET h$=a$(dec+1)+h$: RETURN
190 GO TO 170
195
199 REM ************************
200 REM * PRINT DISPLAY *
210 CLS : FOR n=0 TO 21
220 PRINT  FLASH 1;">";
230 LET dec=st+n*8: GO SUB DtoH
235 IF LEN h$<4 THEN  LET h$="0"+h$: GO TO 235
240 PRINT h$;"   ";
```

```
250   FOR m=0 TO 7
260   LET dec=PEEK (st+8*n+m)
270   GO SUB DtoH: PRINT h$;" ";
280   NEXT m
290 PRINT AT n,0;" ": NEXT n
295 RETURN
299 REM **********************
500 REM INPUT & CURSOR CONTROL
502 IF y=y1 AND x=x1 THEN  GO TO 530
505 LET b$(2)=SCREEN$ (y,x)
510 PRINT AT y,x;c$
515 PRINT AT y1,x1;b$(1)
520 LET y1=y: LET x1=x
525 LET b$(1)=b$(2)
530 LET i=CODE INKEY$: IF i=0 THEN  GO TO 530
535 BEEP .01,15: INPUT ""
540 IF FN n() OR FN a() THEN  GO TO poke
560 IF i>7 AND i<12 THEN  LET y=y+(((i=10) OR (i=9 AND x=29)) AND y<21)-(((i=11
) OR (i=8 AND x=8)) AND y>0): LET x=x+3*((i=9 AND x<29)-(i=8 AND x>8)-7*((i=9 AN
D x=29 AND y1<21)-(i=8 AND x=8 AND y1>0))): GO TO input
565 IF i=88 THEN  GO SUB conv
570 IF i=83 THEN  GO SUB save
575 IF i=73 THEN  PRINT £1;AT 1,0;"The cursor is at ";: LET dec=st+8*y+(x-7)/3:
GO SUB DtoH: PRINT £1;h$
576 IF i=90 THEN  PRINT AT y,x;b$(1): COPY : PRINT AT y,x;c$
580 IF i=81 THEN  GO TO start
585 IF i=7 THEN  LET x=8: LET y=0
590 IF i=75 THEN  LET st=st+176: GO TO newsc
591 IF i=74 THEN  LET st=st-176: GO TO newsc
595 GO TO input
599 REM **********************
600 REM    POKE BYTE
605 LET h$=CHR$ i
610 PRINT AT y,x;h$;c$
615 IF INKEY$<>"" THEN  GO TO 615
620 LET i=CODE INKEY$
630 IF i=0 THEN  GO TO 620
640 IF NOT (FN n() OR FN a()) THEN  GO TO 620
650 LET h$=h$+CHR$ i: PRINT AT y,x;h$
660 GO SUB HtoD
670 POKE (st+8*y1+(x1-8)/3),dec
680 IF x=29 AND y<21 THEN  LET x=5: LET x1=29: LET y=y+1
690 LET x=x+3: LET b$(1)=SCREEN$ (y1,x1)
692 BEEP .05,15
695 GO TO input
699 REM **********************
700 REM  * SAVE CODE *
705 PRINT AT y,x;b$(1)
710 INPUT "SAVE from address: "; LINE h$
720 GO SUB HtoD: LET sa1=dec
730 INPUT "...to: "; LINE h$
740 GO SUB HtoD: LET sa2=dec
750 INPUT "Filename: ";f$
760 SAVE f$CODE sa1,sa2-sa1+1
770 PRINT £1;"Rewind tape for Verification -  then press any key": PAUSE 0
780 VERIFY f$CODE
790 RETURN
799 REM **********************
800 REM   * NEW SCREEN *
810 GO SUB print
820 LET x=8: LET x1=7
830 LET y=0: LET y1=y
840 LET b$="  "
850 GO TO input
899 REM **********************
900 REM     * CONVERT *
905 PRINT AT y,x;b$(1)
910 INPUT ">"; LINE h$
920 IF LEN h$<1 OR LEN h$>6 THEN  GO TO 900
925 PRINT £1;AT 1,1;h$;" -> ";
930 LET i$=h$(LEN h$): LET h$=h$( TO LEN h$-1)
940 IF i$="D" THEN  LET dec=VAL h$: GO SUB DtoH: PRINT £1;h$;"H": PRINT AT y,x;
c$: RETURN
950 IF i$="H" THEN  GO SUB HtoD: PRINT £1;dec;"D": PRINT AT y,x;c$: RETURN
960 BEEP .1,5: GO TO 900
999 REM **********************
1000 REM     ** START **
1010 CLS
1020 PRINT AT 2,8; INVERSE 1;"< HEX LOADER >"
1050 INPUT "Start Address: "; LINE h$
1100 GO SUB HtoD: LET st=dec
1110 GO SUB print: GO TO input
1999 REM **********************
9000 REM  * INITIALISE *
9010
```



```
9020 LET a$="0123456789ABCDEF"
9030 LET b$="  "
9040 LET c$=CHR$ 18+CHR$ 1+"_"
9090
9120 DEF FN n()=i>47 AND i<58
9130 DEF FN a()=i>64 AND i<71
9190
9200 LET x=8: LET x1=7
9210 LET y=0: LET y1=y
9290
9300 LET start=1000
9310 LET print=200
9320 LET input=500
9330 LET newsc=800
9340 LET poke=600
9350 LET conv=900
9360 LET save=700
9370 LET HtoD=100
9380 LET DtoH=150
9390
9400 RETURN
9405
9499 REM **********************
9500 REM  * SAVE PROGRAM *                (cassette version)
9510 SAVE "hexload"
9520 PRINT £1;"Rewind tape for Verification -  then press any key": PAUSE 0
9530 VERIFY "hexload"
9540 STOP
9545
9550 REM  * SAVE PROGRAM *                (microdrive version)
9560 SAVE *"m";1;"hexload"
9570 VERIFY *"m";1;"hexload"
9580 STOP
```

```
 1 RANDOMIZE USR 60000
 2
10 REM BASIC EXTENSION PROGRAM              Cassette Loader
20 POKE (PEEK 23635+256*PEEK 23636+1),0
30 CLEAR 59999
40 LOAD "EXT CODE"CODE 60000
50 RANDOMIZE USR 60000
60 *NEW
70
100 SAVE "EXT" LINE 10
110 VERIFY "EXT"
120 STOP
```

```
 1 RANDOMIZE USR 60000
 2
10 REM BASIC EXTENSION PROGRAM              Microdrive Loader
20 POKE (PEEK 23635+256*PEEK 23636+1),0
30 CLEAR 59999
40 LOAD *"m";1;"EXT CODE"CODE 60000
50 RANDOMIZE USR 60000
60 *NEW
70
100 SAVE *"m";1;"EXT" LINE 10
110 VERIFY *"m";1;"EXT"
120 STOP
```

A.4 HOW THE EXTENSION PROGRAM WORKS

The principle behind extending the Spectrum's BASIC lies in the use of the asterisk at the beginning of each command to cause an error. Because the type of error is irrelevant to the computer (the only difference between one error and the next is the message printed in the lower screen area) the subsequent actions will be completely predictable, and it is therefore possible to intercept the error for reconsideration by the extension program. If the error is found to have been produced by one of the new commands, a jump can be made to the relevant machine code routine, so that the designated piece of code can be performed. Once the command has been executed, control can be returned to the ROM program, after first concealing that an error ever occurred.

To fully understand how the interception is effected, it is first necessary to appreciate the normal sequence of events inside the error handler. This may be summarised as follows:

To enter the error routine, a RST 08h is executed, with the following byte holding one less than the required error report code. The contents of the system variable CHADD (23645d,5C5Dh) are then copied to XPTR (23647d,5C5Fh) before proceeding.

CHADD holds the address of the character presently being considered, whilst XPTR indicates the position of the flashing "?" error marker in a syntatically incorrect program line. The effect of the routine at 08h is therefore to position the error marker immediately after the first out-of-place character in the statement.

As there is a limited amount of space between RESTARTS on the Z80, a jump to ERROR2 (83d,53h) is now made to continue the error handling.

When any RESTART is executed, the address of the next instruction is PUSHed onto the stack as the return address. By POPping this number off into the HL register pair, the ERROR2 routine is able to load the error number in the byte after the RST 08h instruction into the system variable ERRNR (23610d,5C3Ah). The Stack Pointer is then reset, such that the next RET instruction will cause a jump to the address specified in the system variable ERRSP (23613d,5C3Dh) and this is where the system can be broken into. Normally the routine would be left via SET-STK at (5829d,16C5h) but by loading ERRSP with the address of the extension program prior to the error-handler being called, it is possible to intercept the error before program execution is stopped.

The extension program itself can be broken down into a number of discrete routines, as illustrated in Fig. 1.

Any error which occurs is initially run through the Main Parser, which essentially checks for an asterisk followed by a character in the range A-Z. Provided these two criteria are met, the error is assumed for the moment to be an extended command and is passed to the rel-

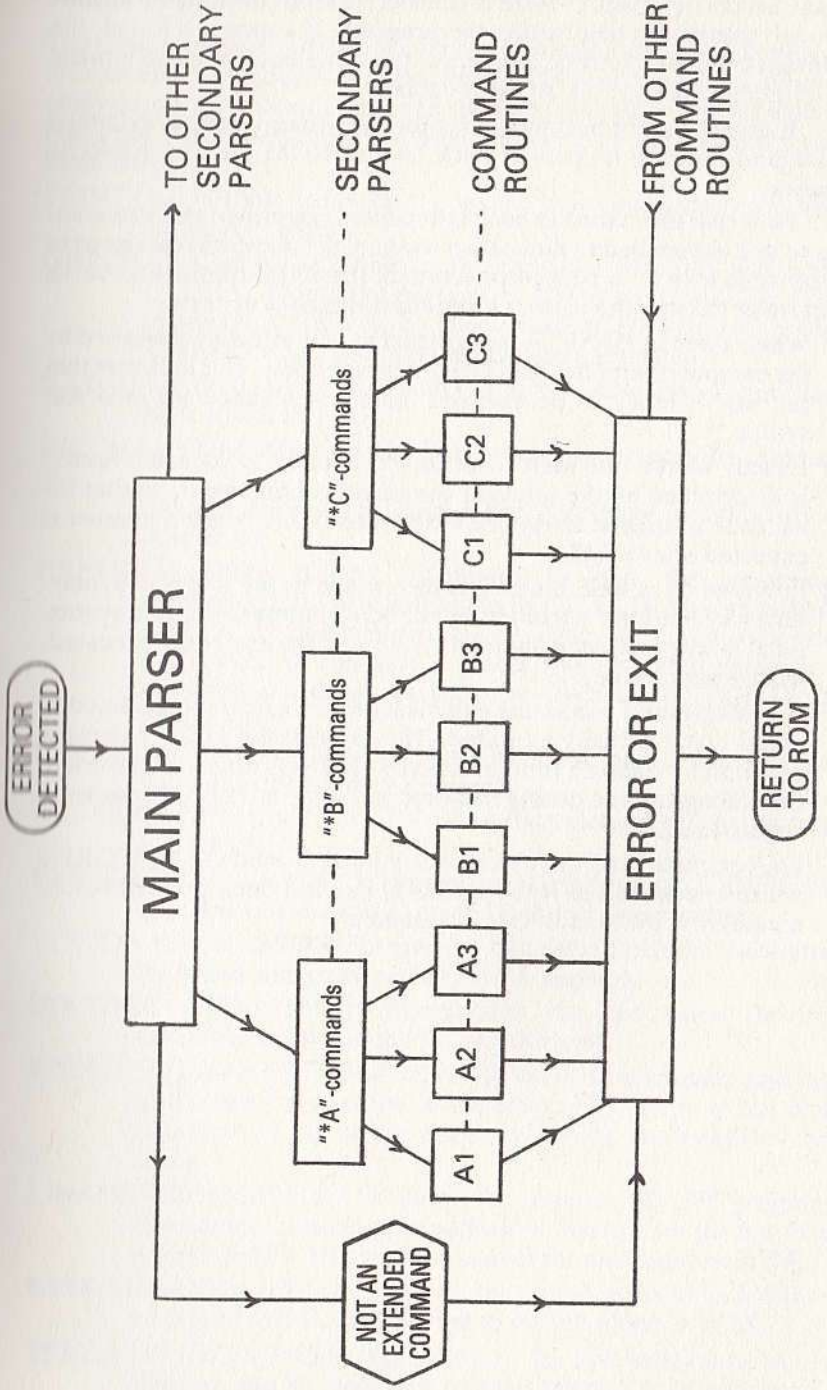

FIG. 1: Outline structure of the BASIC Extension Program.

evant Secondary Parser. Here it is checked more thoroughly against 'model' commands held within the program. If a match is found, the relevant command routine is entered. On completion a return is made to a suitable point on the ROM program.

If at any time it becomes clear that the error is not an extended command, control is passed back to the ROM via the ERROR routine.

Now that the error has been intercepted, identified as a new command and channelled to the correct routine, the question remains as to what to do with it. A closer inspection of the ROM routines reveals a pattern in the way the various standard commands operate:

1) When a line of BASIC is being typed in, it is effectively ignored by the computer until the "ENTER" key is pressed. This indicates that the line is ready to be assessed, and it is scanned for incorrect syntax.
2) Exactly what is and what is not correct in terms of parameters etc., is determined by the relevant command routine itself, so that for instance a string is always expected after VAL, while a number is expected after STR$.
3) Provided all is well, a check is then made to see if this is 'syntax-time' or 'run-time' – a program line being entered will be in syntax-time, whilst a direct command, or a program line being executed, will be in run-time.
4) In syntax-time a premature exit must be performed so that the command is not actually executed. This is achieved in the extension program by calling a routine called SYNEND, which will return to the calling-routine during run-time but jump to the exit routine in syntax-time.
5) On completion of the command, a jump is made to STMT-RET which checks to see if the BREAK key has been pressed before considering the next BASIC statement.

A.5 ROM ROUTINES USED

The compactness of the Extension program can be attributed largely to the extensive use made of the Spectrum ROM routines. These are generally described in context with the routines using them, but for ease of reference there now follows a brief summary of the functions performed by each.

PRINT (16d,0010h): Prints the character whose code is currently held in the A-register.

GETCHAR (24d,0018h): Fetches the character currently addressed by CHADD into the A-register. A return is made only if the character is printable and not a space (CHR$ 32); otherwise CHADD is incremented and the fetch repeated.

NXTCHAR (32d,0020h): CHADD is incremented before jumping back to GETCHAR.

INCCHAD (116d,0074h): CHADD is incremented and the contents of the new address returned in the A-register, whether printable or not.

EXPTNUM (7298d,1C82h): Evaluates in part the numerical expression currently pointed to by CHADD. During syntax-time the routine confirms the presence of a valid numerical expression; in run-time it places the evaluated expression onto the calculator stack, to be fetched by STKTOA or STKTOBC.

EXPT2NM (7290d,1C7Ah): As for EXPTNUM, but searches for two numbers separated by a comma.

NEXT2NM (7289d,1C79h): CHADD is incremented before jumping to EXPT2NM.

EXPTSTG (7308d,1C8Ch): Performs a similar function to EXPTNUM except that a string is expected rather than a number.

STKTOA (7828d,1E94h): Fetches the last value from the calculator stack and compresses it into the A-register.

STKTOBC (7833d,1E99h): Compresses the last value on the calculator stack into the BC register pair.

STKFTCH (11249d,2BF1h): Fetches the various parameters relating to the last string entry on the stack: DE points to the first character of the string whilst BC holds the length of the string.

LINADDR (6510d,196Eh): On entry, HL contains a BASIC program line number. On exit, the address of that line (or the first line after) is held in HL, with the start of the previous line in DE.

RECLAIM (6629d,19E5h): On entry, the first location to be reclaimed is held in DE, and the first to be 'left alone' is in HL.

RECLAIM2 (6632d,19E8h): HL points to the first location to be reclaimed, and BC holds the number of bytes to be reclaimed.

DIFRNCE (6621d,19DDh): Used to find the difference between two addresses, given in the HL and DE register pairs. The result is returned in BC, with HL and DE exchanged.

ALPHA (11405d,2C8Dh): Returns with the carry flag set if the contents of the A register represent an alphabetic character.

NUMERIC (11547d,2D1Bh): Returns with the carry flag reset if the contents of the A register denote a numerical digit.

ALPHNUM (11400d,2C88h): Returns with carry flag set if the contents of the A register denote an alphanumeric character.

NEXTONE (6584d,19B8h): Finds the address of the next line or variable in the program or variables area respectively. The routine is entered with the current line or variable in HL, and exits with the address of the next one in DE. The contents of HL are preserved during this routine.

NXTSTMT (6542d,198Eh): Steps along each statement in a line of BASIC. A return is made after a specified number of statements (specified in the D register) or when a match is found with the character code held in the E register.

SRCHSTM (6539d,198Bh): CHADD is loaded with the contents of the HL register pair before entering NXTSTMT.

LOOKVAR (10418d,28B2h): A search of the variables area is made for the variable whose name is currently pointed to by CHADD. If the variable is found, the carry flag is reset and HL returns pointing to the last letter in the variable name; otherwise the carry flag is set and HL will point to the first letter of the name.

FREEMEM (7962d,1F1Ah): Returns with BC holding the amount of memory including ROM space presently unavailable.

SELDEV (5633d,1601h): The routine is generally entered with the A register holding 02h; this will select the screen as the current channel to which all subsequent printing will be sent.

PRTSTRG (8252d,203Ch): On entry the length of the string to be printed is contained in the BC register pair, with DE pointing to the address of the first character.

INTSTOR (11660d,2D8Ch): On entry HL points to the first of five locations whose contents are to be stored on the calculator stack.

STACKBC (11563d,2D2Bh): The number currently held in BC is stacked in floating-point form.

FPPRINT (11747d,2DE3h): Prints the last value on the calculator stack; calling STACKBC and then FPPRINT therefore provides an easy method of printing any 16-bit number.

CLSCREN (3435d,0D6Bh): Clears the display: "CALL CLSCREN" is effectively the machine code equivalent of the BASIC command "CLS".

BORDER (8852d,2294h): The last value on the calculator stack is fetched and used as the Border colour.

BORDER2 (8859d,229Bh): Entry point to BORDER used when the A register already holds the value to be used as the Border colour.

SCROLL (3584d,0E00h): On entry, the B register holds the number of the top line to be scrolled.

CLRPRB (3807d,0EDFh): Clears the printer buffer.

CPLINES (6528d,1980h): If the line number addressed by HL is less than that given in BC, the carry flag is returned set.

SETMIN (5808d,16B0h): Effectively clears the editing and subsequent areas.

BRKKEY (8020d,1F54h): The carry flag is returned reset if BREAK is being pressed.

COPYBUF (3789d,0ECDh): The contents of the printer buffer are passed to the printer.

CLSLWR (3438d,0D6Eh): The lower part of the screen display is cleared.

OUTCODE (5615d,15EFh): Prints the digit (0-9) held in the A register.

POMSGE (3082d,0C0Ah): On entry, DE holds the base address of the relevant table, and the A register holds the number of the message to be printed in the lower screen area.

OUTNUM (6683d,1A1Bh): Any number in the BC register pair will be printed, provided it is below 10,000d (decimal).

CLEARSP (4247d,1097h): HL signals whether the editing area or the workspace is to be cleared.

STMTRET (7030d,1B76h): The return point after a correctly executed statement. The BREAK key is tested before considering the next line or statement.

CHKEND (7150d,1BEEh): An error report is given if CHADD is not addressing the end of a BASIC statement or line during syntax-time.

DRAWLIN (9402d,24BAh): On entry, B and C hold the Y and X displacements respectively; D and E indicate the sign of each. The required line is then drawn from the present PLOT position.

MAKISPC (5714d,1652h): A space is opened-up immediately before the location addressed by HL.

SECTION B

THE EXTENSION PROGRAM

B.1 FUNDAMENTAL ROUTINES

ASSEMBLER EQUATE TABLE

This table is strictly speaking not part of the extension program because it is used by the assembler program to link labels (e.g. EXPTNUM) to the addresses to which they refer (e.g. & 1C82). The table is a fundamental requirement for those using an assembler and so it is listed here. Those using the hex loader can safely ignore it.

```
BASIC EXTENSION PROGRAM
by Rob Banks

ROM Routines

EXPTNUM   equ  &1C82
EXPT2NM   equ  &1C7A
NEXT2NM   equ  &1C79
EXPTSTG   equ  &1C8C
STKTOA    equ  &1E94
STKTOBC   equ  &1E99
STKFTCH   equ  &2BF1
INCCHAD   equ  &0074
LINADDR   equ  &196E
RECLAIM   equ  &19E5
RECLAM2   equ  &19E8
DIFRNCE   equ  &19DD
MAK1SPC   equ  &1652
ALPHA     equ  &2C8D
ALPHNUM   equ  &2C88
NEXTONE   equ  &1988
NXTSTMT   equ  &1990
SCHSTM    equ  &198B
FREEMEM   equ  &1F1A
SELDEV    equ  &1601
PRTSTRG   equ  &203C
STACKBC   equ  &2D2B
FPPRINT   equ  &2DE3
POMSGE    equ  &0C0A
OUTCODE   equ  &15EF
OUTNUM    equ  &1A1B
DRAWLIN   equ  &24BA
LOOKVAR   equ  &28B2
CLSCREN   equ  &0D6B
CLSLWR    equ  &0D6E
BORDER    equ  &2294
BORDR2    equ  &229B
SCROLL    equ  &0E00
COPYBUF   equ  &0ECD
NUMERIC   equ  &2D1B
INTSTOR   equ  &2D8C
CPLINES   equ  &1980
CLEARSP   equ  &1097
SETMIN    equ  &16B0
BRKKEY    equ  &1F54
STMTRET   equ  &1B76

System Variables

LASTK     equ  23560
REPDEL    equ  23561
REPPER    equ  23562
DEFAD     equ  23563
STRM6     equ  23574
PIP       equ  23609
ERRNR     equ  23610
FLAGS     equ  23611
ERRSP     equ  23613
NEWPPC    equ  23618
NSPPC     equ  23620
PPC       equ  23621
SUBPPC    equ  23623
BORDCR    equ  23624
VARS      equ  23627
PROG      equ  23635
NXTLIN    equ  23637
ELIN      equ  23641
CHADD     equ  23645
```

```
XPTR      equ  23647
FLAGS2    equ  23658
UDGST     equ  23675
COORDS    equ  23677
SPOSN     equ  23688
ATTRP     equ  23693
ATTRT     equ  23695

Other symbols

ATTRST    equ  22528
DISPST    equ  16384
```

ENABLE

This short (13 byte) routine is used to activate all extended commands. A simple USR call from BASIC results in the start address of the Main Parser being inserted into the location addressed by the system variable ERRSP at address 23613 decimal,5C3D hex. Since this is used by the BASIC Operating System to vector error returns, all subsequent errors will initially be sent to the Main Parser for further consideration. Finally CLRTAB is called to reset all stacks etc. used by the extended commands.

```
Activate Commands

ENABLE    ld   hl,(ERRSP)          Make Error Return Address
      2   ld   de,EXTEND           that of the
      3   ld   (hl),e              Main Parser.
      4   inc  hl
      5   ld   (hl),d

      6   call CLRTAB              Clear all stacks.
      7   ret


HEX DUMP:

        EA60   2A   3D   5C   11   6D   EA   73   23
        EA68   72   CD   64   EE   C9
```

MAIN PARSER

The function of the Main Parser is to differentiate extended commands from actual errors. The new commands are directed to the relevant Secondary Parser, whilst 'proper' errors are passed to the Error Handling routine.

In order for an extended command to be accepted, it must first pass some simple tests applied by the Main Parser:

1. The error produced by any extended command must be Report Code C, Nonsence in BASIC;
2. The first character of the command must be an asterisk; and
3. The second character must be alphabetic.

Test number 1 is carried out by the first three lines of the routine: the contents of the system variable ERRNR (23610d,5C3Ah) will have been set to one less than the report code by the calling routine, and will therefore be equal to 11 decimal,0B hexadecimal for Error Report C.

The ensuing five lines perform the second test. On entry to the Main Parser, the system variable CHADD (23645d,5C5Dh) will be

pointing to the character immediately after the one causing the error. It is therefore necessary to decrement the address so obtained to point to the first character, which should be an asterisk (42d,2Ah) for all extended commands.

The third and final test involves getting the second character using the ROM routine GETCHAR, ensuring it is upper case alphabetic and then reducing it to a number between 0 and 25 inclusive.

The number so obtained is used to extract the address of the relevant Secondary Parser from the Vector Table; control them being passed to that routine.

```
Main Parser

EXTEND    ld   a,(ERRNR)           Is Error Report
      2   cp   11                  "Nonsense in BASIC"?

      3   jp   nz,ERROR            Jump if not.

      4   ld   hl,(CHADD)          Fetch the character in
      5   dec  hl                  the BASIC line which caused
      6   ld   a,(hl)              the error.

      7   cp   42                  ("*")
      8   jp   nz,ERROR            Jump if it is not an asterisk.

      9   rst  24                  GETCHAR
     10   or   32                  Make 1st letter of command name
     11   sub  32                  upper case.
     12   ld   (hl),a

     13   sub  65                  Reduce CODE to range 0-25.
     14   jp   c,ERROR

     15   cp   26                  Check it is in range.
     16   jp   nc,ERROR

     17   add  a                   Calculate offset.
     18   ld   h,0
     19   ld   l,a

     20   ld   de,VECTOR           Locate required entry
     21   add  hl,de               in Vector table.
     22   ld   e,(hl)
     23   inc  hl
     24   ld   d,(hl)
     25   ex   de,hl

     26   jp   (hl)                Jump to relevant Secondary Parser.

HEX DUMP:

        EA6D   3A   3A   5C   FE   0B   C2   51   EC
        EA75   2A   5D   5C   2B   7E   FE   2A   C2
        EA7D   51   EC   DF   F6   20   D6   20   77
        EA85   D6   41   DA   51   EC   FE   1A   D2
        EA8D   51   EC   87   26   00   6F   11   FD
        EA95   EC   19   5E   23   56   EB   E9
```

SECONDARY PARSERS

There should be a Secondary Parser for every set of commands beginning with the same letter of the alphabet. Its function is to pass the command to the relevant routine, differentiating between, for example, SLEFT and SRIGHT. Note that the Main Parser only checks the first letter, and would therefore be unable to tell the difference.

All the Secondary Parsers work in the same manner: a pointer is set to the beginning of the 'model' command held within the program, and the command checking routine is called. If the command being

checked matches the model command, then a jump is made to the relevant routine. If not, the pointer is set to to the beginning of the next model and the process repeated. Should the subject command fail to match any of the models, a jump is made to the Error Handling Routine.

Secondary Parsers

```
ACOM     ld   hl,ATTCOM          Point to 1st command  model.
     2   call CKCM1              Check if matches the subject command.
     3   jp   nz,ATTR            Jump to command routine if it does.

     4   jp   ERROR              No more models-Must have been an Error.


CCOM     ld   hl,CHKCOM
     6   call CKCM2
     7   jp   nz,CCHK

     8   ld   hl,CAPCOM
     9   call CKCM2
    10   jp   nz,CAPS

    11   ld   hl,CLPCOM
    12   call CKCM1
    13   jp   nz,CLP

    14   ld   hl,CLACOM
    15   call CKCM2
    16   jp   nz,CLA

    17   jp   ERROR


DCOM     ld   hl,DPCCOM
    19   call CKCM2
    20   jp   nz,DPRC

    21   ld   hl,DKYCOM
    22   call CKCM2
    23   jp   nz,DKEY

    24   ld   hl,DELCOM
    25   call CKCM2
    26   jp   nz,DEL

    27   ld   hl,DRACOM
    28   call CKCM2
    29   jp   nz,DRAW

    30   jp   ERROR


ECOM     ld   hl,EPCCOM
    32   call CKCM1
    33   jp   nz,EPRC

    34   ld   hl,EKYCOM
    35   call CKCM1
    36   jp   nz,EKEY

    37   ld   hl,ELSCOM
    38   call CKCM1
    39   jp   nz,ELSE

    40   jp   ERROR


FCOM     ld   hl,FRECOM
    42   call CKCM1
    43   jp   nz,FREE

    44   jp   ERROR
```



```
ICOM     ld   hl,IFCOM
    46   call CKCM2
    47   jp   nz,IF

    48   jp   ERROR


KCOM     ld   hl,KEYCOM
    50   call CKCM2
    51   jp   nz,KEYS

    52   ld   hl,KBDCOM
    53   call CKCM2
    54   jp   nz,KYBD

    55   jp   ERROR


NCOM     ld   hl,NSECOM
    57   call CKCM2
    58   jp   nz,NOIS

    59   ld   hl,NEWCOM
    60   call CKCM1
    61   jp   nz,NEW

    62   jp   ERROR


OCOM     ld   hl,ONDCOM
    64   call CKCM1
    65   jp   nz,OEND

    66   ld   hl,ONCOM
    67   call CKCM2
    68   jp   nz,ON

    69   jp   ERROR


PCOM     ld   hl,PRCCOM
    71   call CKCM2
    72   jp   nz,PROC

    73   ld   hl,POKCOM
    74   call CKCM2
    75   jp   nz,POKE

    76   jp   ERROR


RCOM     ld   hl,RPTCOM
    78   call CKCM1
    79   jp   nz,REPT

    80   ld   hl,RKLCOM
    81   call CKCM1
    82   jp   nz,RKIL

    83   ld   hl,RNMCOM
    84   call CKCM2
    85   jp   nz,RNUM

    86   jp   ERROR


SCOM     ld   hl,SCLCOM
    88   call CKCM2
    89   jp   nz,SCRL

    90   ld   hl,SLTCOM
    91   call CKCM2
    92   jp   nz,SLEF

    93   ld   hl,SRTCOM
    94   call CKCM2
    95   jp   nz,SRIG

    96   ld   hl,SPDCOM
    97   call CKCM2
    98   jp   nz,SPED

    99   jp   ERROR
```

```
UCOM      ld   hl,UTLCOM
   101    call CKCM2
   102    jp   nz,UNTL

   103    ld   hl,UDGCOM
   104    call CKCM2
   105    jp   nz,UDG

   106    ld   hl,UNDCOM
   107    call CKCM2
   108    jp   nz,UNDL

   109    jp   ERROR

VCOM      ld   hl,VPTCOM
   111    call CKCM2
   112    jp   nz,VPRT

   113    jp   ERROR

WCOM      ld   hl,WHLCOM
   115    call CKCM2
   116    jp   nz,WHLE

   117    ld   hl,WENCOM
   118    call CKCM1
   119    jp   nz,WEND

   120    ld   hl,WIPCOM
   121    call CKCM2
   122    jp   nz,WIPE

   123    jp   ERROR
```

HEX DUMP:

```
EA9C   21  F1  F3  CD  0D  EC  C2  72
EAA4   F3  C3  51  EC  21  24  EF  CD
EAAC   33  EC  C2  06  EF  21  01  EF
EAB4   CD  33  EC  C2  ED  EE  21  4E
EABC   F3  CD  0D  EC  C2  3F  F3  21
EAC4   3B  F3  CD  33  EC  C2  24  F3
EACC   C3  51  EC  21  41  F0  CD  33
EAD4   EC  C2  28  F0  21  74  F2  CD
EADC   33  EC  C2  5A  F2  21  60  F5
EAE4   CD  33  EC  C2  40  F5  21  C0
EAEC   F2  CD  33  EC  C2  95  F2  C3
EAF4   51  EC  21  63  F0  CD  0D  EC
EAFC   C2  49  F0  21  8E  F2  CD  0D
EB04   EC  C2  7B  F2  21  58  F1  CD
EB0C   0D  EC  C2  44  F1  C3  51  EC
EB14   21  28  F5  CD  0D  EC  C2  F9
EB1C   F4  C3  51  EC  21  41  F1  CD
EB24   33  EC  C2  15  F1  C3  51  EC
EB2C   21  54  F2  CD  33  EC  C2  8B
EB34   F1  21  C4  EF  CD  33  EC  C2
EB3C   A1  EF  C3  51  EC  21  66  EF
EB44   CD  33  EC  C2  46  EF  21  C8
EB4C   EE  CD  0D  EC  C2  81  EE  C3
EB54   51  EC  21  85  F1  CD  0D  EC
EB5C   C2  7F  F1  21  7C  F1  CD  33
EB64   EC  C2  5D  F1  C3  51  EC  21
EB6C   23  F0  CD  33  EC  C2  C9  EF
EB74   21  41  EF  CD  33  EC  C2  2B
EB7C   EF  C3  51  EC  21  82  F0  CD
EB84   0D  EC  C2  6B  F0  21  BB  F5
EB8C   CD  0D  EC  C2  64  F5  21  14
EB94   F6  CD  33  EC  C2  C3  F5  C3
EB9C   51  EC  21  22  F4  CD  33  EC
EBA4   C2  F6  F3  21  75  F4  CD  33
EBAC   EC  C2  29  F4  21  C4  F4  CD
EBB4   33  EC  C2  7B  F4  21  E7  EE
EBBC   CD  33  EC  C2  CC  EE  C3  51
EBC4   EC  21  C3  F0  CD  33  EC  C2
EBCC   9D  F0  21  9D  EF  CD  33  EC
EBD4   C2  6C  EF  21  ED  F2  CD  33
EBDC   EC  C2  C5  F2  C3  51  EC  21
EBE4   1D  F3  CD  33  EC  C2  F2  F2
EBEC   C3  51  EC  21  F7  F0  CD  33
EBF4   EC  C2  C9  F0  21  10  F1  CD
EBFC   0D  EC  C2  FD  F0  21  6D  F3
EC04   CD  33  EC  C2  5D  F3  C3  51
EC0C   EC
```


COMMAND CHECKING

The purpose of this routine is to check the command under scrutiny, letter-for-letter against the model command indicated by the Secondary Parser. As a by-product, any lower-case letters are converted to upper-case during execution.

There are two entry-points to the routine: the first, CKCM1, is used when there are no parameters to be checked. CKCM2 is used where there are parameters, and works by first calling CKCM1 and then stepping to the next 'non-space' character – signalling an error if this is a carriage return (ENTER,13d,0Dh) or colon (58d,3Ah); either of which would indicate that no parameters were present.

A quick glance at any of the command routines will show that each is followed by a number, and the name of the command. The number is used by the command checking routine and is always one less than the length of the command; the first character having been stepped over by the Main Parser. The name which follows this number is used as the model against which the subject command is checked.

Note that the initial value of CHADD (i.e. the start of the command being checked) is stored on entry to the routine. Obviously, if the command check fails the pointer must be restored ready for any further candidates from the Secondary Parser. This function is performed by NOTCM, which also signals 'no match' to the Secondary Parser. If the match is positive, however, the routine exits with CHADD pointing to the next character (carriage return or colon for unqualified commands, or the first parameter for those which require them).

```
Command Checking

CKCM1     ld   b,(hl)              b=length of model command.

     2    ld   de,(CHADD)          de=start of subject command.

     3    push de                  Save it in case it doesn't match model.
     4    push hl
     5    pop  ix

     6    inc  ix                  ix=start of model.

LLP1      inc  ix
     8    rst  32                  NXTCHAR
     9    or   32                  Fetch next character,and
    10    sub  32                  ensure it is upper case.

    11    cp   (ix+0)              Compare it with model command.

    12    jr   nz,NOTCM            Jump if it doesn't match.
    13    ld   (hl),a

    14    djnz LLP1                Repeat for all characters.

    15    pop  hl
    16    call INCCHAD             Point to next character.

    17    add  a,b                 Reset zero flag.
    18    ret

NOTCM     pop  hl                  Model command does not match,so
    20    ld   (CHADD),hl          reset CHADD to start of subject.

    21    sub  a                   Set zero flag.
    22    ret
```

```
CKCM2     call CKCM1              Check command.

    24    ret  z                  Return immediately if no match.

    25    rst  32                 NXTCHAR
    26    cp   13                 (ENTER)
    27    ret  z                                Return with Zero flag
    28    cp   58                 (Colon)       set if no parameters
    29    ret                                   present.
```

HEX DUMP:

```
EC0D   46  ED  5B  5D  5C  D5  E5  DD
EC15   E1  DD  23  DD  23  E7  F6  20
EC1D   D6  20  DD  BE  00  20  09  77
EC25   10  F1  E1  CD  74  00  B0  C9
EC2D   E1  22  5D  5C  97  C9  CD  0D
EC35   EC  CB  E7  FE  0D  C8  FE  3A
EC3D   C9
```

SYNEND

This short routine is used by all the command routines to make an early exit during syntax-time. The most significant bit (MSB-bit 7) of the system variable FLAGS (23611d,5C3Bh) will be zero during syntax checking, and 1 during run-time. By first removing the return address from the stack, this routine can therefore either jump back to the calling routine (MSB=1) or straight to EXIT (MSB=0).

EXIT

Provided an extended command has been executed satisfactorily, it is obviously necessary to ensure that the BASIC ROM is not expecting an error when control is returned to it. This can be achieved by loading ERRNR with 255d,FFh (its idle state) and replacing the new error return address on the stack (it was removed when the Main Parser was called). A jump can then be made into the ROM, to the STMT-RET routine.

ERROR

If an error is not recognised as an extended command, then it must be returned to the ROM to be dealt with in the normal manner. Unfortunately, it is not really feasible to make a direct return to the normal ROM Error Handler at 4867d,1303h because this resets ERRSP and thus disables the extended commands. They cannot then be re-enabled by a simple call to the ENABLE routine because this would ultimately produce the "OK" message, which the Spectrum treats as an error report – thus disabling the commands again.

To overcome this problem the ERROR routine has been included in the extension program. On entry to this routine, the contents of FLAGS is inspected to see if the error has occurred during syntax-time. If this is the case, a simple jump can be made into the ROM at 4791d,12B7h (which is where standard syntax errors are dealt with) after first replacing the new error return address on the stack and copying CHADD to XPTR.

If, on the other hand, an error occurs during run-time, control will immediately pass to the RUNER routine. This is actually a copy of the

ROM Error Handler, but with the addition of four bytes at the end to PUSH the new error return address back onto the stack.

Finally, a return is made midway into the ROM routine (4968d,1368h).

EXTENDED ERRORS

Just like the standard Spectrum instructions, extended commands are subject to errors, such as out of range parameters etc. However, before the ERROR routine can be jumped to, the system variable ERRNR must be loaded with a number equivalent to the required report code less one. As this would obviously be inconvenient to carry out in the command routines themselves, a routine is provided with entry-points for all the errors likely to be produced by the extended commands. It is, therefore, only necessary to jump to one of these locations to produce the relevant report message.

```
RUN- or SYNTAX-time

SYNEND    pop  hl                 Remove return address.
     2    bit  7,(iy+1)

     3    jr   z,EXIT             Leave if in Syntax-time,
     4    jp   (hl)               otherwise return to calling routine.

Return to ROM

EXIT      ld   (iy+0),255         Reset ERRNR.

     6    ld   hl,EXTEND          Replace new Error Return Address.
     7    push hl

     8    jp   STMTRET            Back into ROM.

Error Handling

ERROR     bit  7,(iy+1)
    10    jr   nz,RUNER           Jump forward if in run-time.

SYNER     ld   hl,EXTEND
    12    push hl                 Replace Error Return Address.
    13    ld   hl,(CHADD)

    14    ld   (XPTR),hl          Position Error Marker.

    15    jp   4791               Back into ROM.

RUNER     res  5,(iy+1)           Signal ready for next
    17    bit  1,(iy+48)          keypress,and empty printer
    18    call nz,COPYBUF         buffer if used.
    19    ld   a,(ERRNR)
    20    inc  a

    21    push af                 Store actual Report code.
    22    ld   hl,0

    23    ld   (DEFAD),hl         Reset DEFADD,
    24    ld   (iy+38),h          XPTR-hi and
    25    ld   (iy+55),h          FLAGX.

    26    inc  hl                 Make stream 0 point
    27    ld   (STRM6),hl         to channel K.

    28    call SETMIN             Clear calculator stack etc.

    29    call CLSLWR             Clear lower screen area,but signal
    30    set  5,(iy+2)           it will need to be cleared again.

    31    pop  af                 Fetch Report Code.

    32    ld   b,a                Save it temporarily.
```

```
    33    cp   10              Adjust Report codes
    34    jr   c,Numcod        greater than 10 to give
    35    add  a,7             relevant Hex character.

Numcod    call OUTCODE         Print Report code.

    37    ld   a,32            (space)
    38    rst  16              PRINT
    39    ld   a,b             Fetch Report Code again,and use it to
    40    ld   de,5009         address the relevant message.

    41    call POMSGE          Print the message.
    42    xor  a
    43    ld   de,5430

    44    call POMSGE          Print a comma and a space.
    45    ld   bc,(PPC)

    46    call OUTNUM          Print current line no.

    47    ld   a,58            (Colon)
    48    rst  16              PRINT
    49    ld   b,0
    50    ld   c,(iy+13)
    51    call OUTNUM          Print current statement no.
    52    call CLEARSP

    53    ld   a,(ERRNR)       Store Error No. in a,before
    54    ld   (iy+0),255      clearing the system variable.
    55    ld   hl,EXTEND

    56    push hl              Replace Error Return Address.

    57    jp   4968            Back into ROM.

Extended Errors

OKERR     ld   a,&FF
    59    jr   EXERR
VARERR    ld   a,&01
    61    jr   EXERR
SUBERR    ld   a,&02
    63    jr   EXERR
MEMERR    ld   a,&03
    65    jr   EXERR
ARGERR    ld   a,&09
    67    jr   EXERR
INTERR    ld   a,&0A
    69    jr   EXERR
COLERR    ld   a,&13
    71    jr   EXERR
BRKERR    ld   a,&14
    73    jr   EXERR
LOSERR    ld   a,&16
    75    jr   EXERR
PARERR    ld   a,&19
EXERR     ld   (ERRNR),a
    78    jp   ERROR
```

HEX DUMP:

```
EC3E  E1 FD CB 01 7E 28 01 E9
EC46  FD 36 00 FF 21 6D EA E5
EC4E  C3 76 1B FD CB 01 7E 20
EC56  0D 21 6D EA E5 2A 5D 5C
EC5E  22 5F 5C C3 B7 12 FD CB
EC66  01 AE FD CB 30 4E C4 CD
EC6E  0E 3A 3A 5C 3C F5 21 00
EC76  00 22 0B 5C FD 74 26 FD
EC7E  74 37 23 22 16 5C CD B0
EC86  16 FD CB 37 AE CD 6E 0D
EC8E  FD CB 02 EE F1 47 FE 0A
EC96  38 02 C6 07 CD EF 15 3E
EC9E  20 D7 78 11 91 13 CD 0A
ECA6  0C AF 11 36 15 CD 0A 0C
ECAE  ED 4B 45 5C CD 1B 1A 3E
ECB6  3A D7 06 00 FD 4E 0D CD
ECBE  1B 1A CD 97 10 3A 3A 5C
ECC6  FD 36 00 FF 21 6D EA E5
ECCE  C3 68 13 3E FF 18 22 3E
ECD6  01 18 1E 3E 02 18 1A 3E
ECDE  03 18 16 3E 09 18 12 3E
ECE6  0A 18 0E 3E 13 18 0A 3E
ECEE  14 18 06 3E 16 18 02 3E
ECF6  19 32 3A 5C C3 51
```


B.2 TABLES

The Extended Basic program uses a number of tables and stacks, together with a few free bytes of workspace. For neatness, these have all been collected together here.

Vector Table – This is a list of the addresses of each and every Secondary Parser, and is therefore 52 bytes long (2 bytes per address, and a possible 26 different Secondary Parsers). The first address is for the Secondary Parser dealing with extended commands beginning with the letter "A"; the last is that for dealing with extended commands commencing with "Z". The position of the address for "*X"-commands is therefore given by:

PRINT start + 2*(CODE "X" - CODE "A"), where "start" is the address of the first byte of the Vector table.

Of course, there are certain letters of the alphabet which are not used as the initial letter of any extended commands. The relevant entry in the Vector table is therefore assigned the address of the ERROR routine, so that an offending command is dealt with as a normal error.

This approach is very flexible. If you decide to add your own extended command, called perhaps "*ZAP", you have only to write the new Secondary Parser (this book does not describe any commands beginning with "Z") along the lines of the existing ones, and insert its starting address in the Vector table (for *Z-commands, this would obviously be right at the end).

Function Key Table – This is used by the KEYIN command to convert the character codes of a CAPS SHIFT – Numeric key combination into the codes of the digits 0 to 9.

Procedure Stack – Whenever a procedure call is made, the line and statement number of the relevant *PROC command is stored in this area. The first two bytes are used as a pointer to the next free position in the table, and thereafter three bytes are used to hold the information for each procedure called. The table shown allows for up to 10 *nested* procedures in a program, which should prove more than adequate for most purposes.

Repeat-Until Stack – This is essentially similar to the Procedure stack, but as the statement number is not saved, only two bytes are required for each entry in the table.

While-Wend Stack – Exactly the same as the Repeat-Until stack.

Keyin (Function Key) Stack – As it would be bad practice to use the KEYIN function in a Function Key procedure (due to the risk of the routine calling itself) a stack is not strictly necessary for this command – two bytes would be sufficient. However, to maintain flexibility and a degree of standardisation with the other structural commands, it was thought reasonable to allow a few extra bytes.

Workspace – There are a number of occasions when it becomes necessary to temporarily store certain numbers during the execution of an

extended command. Whilst the stack is used as often as practicable, it is not always convenient and some alternative method is required. The Workspace area is 10 bytes long, which is sufficient for all the commands given.

```
Tables & Stacks

VECTOR    defw ACOM
      2   defw ERROR
      3   defw CCOM
      4   defw DCOM
      5   defw ECOM
      6   defw FCOM
      7   defw ERROR
      8   defw ERROR
      9   defw ICOM
     10   defw ERROR
     11   defw KCOM
     12   defw ERROR
     13   defw ERROR
     14   defw NCOM
     15   defw OCOM
     16   defw PCOM
     17   defw ERROR
     18   defw RCOM
     19   defw SCOM
     20   defw ERROR
     21   defw UCOM
     22   defw VCOM
     23   defw WCOM
     24   defw ERROR
     25   defw ERROR
     26   defw ERROR
KEYTAB    defb 51
     28   defb 52
     29   defb 50
     30   defb 49
     31   defb 53
     32   defb 56
     33   defb 54
     34   defb 55
     35   defb 48
     36   defb 00
     37   defb 00
     38   defb 57
PTABST    defw PTAB
PTAB      defm "0000000000
     41   defm "0000000000
     42   defm "0000000000
RTABST    defw RTAB
RTAB      defm "0000000000
     45   defm "0000000000
WTABST    defw WTAB
WTAB      defm "0000000000
     48   defm "0000000000
KTABST    defw KTAB
KTAB      defm "0000000000
WRKSPC    defm "0000000000

HEX DUMP:

        ECFC   EC   9C   EA   51   EC   A8   EA   CF
        ED04   EA   F6   EA   14   EB   51   EC   51
        ED0C   EC   20   EB   51   EC   2C   EB   51
        ED14   EC   51   EC   41   EB   56   EB   6B
        ED1C   EB   51   EC   80   EB   9E   EB   51
        ED24   EC   C5   EB   E3   EB   EF   EB   51
        ED2C   EC   51   EC   51   EC   33   34   32
        ED34   31   35   38   36   37   30   00   00
        ED3C   39   3F   ED   DC   00   02   30   30
        ED44   30   30   30   30   30   30   30   30
        ED4C   30   30   30   30   30   30   30   30
        ED54   30   30   30   30   30   30   30   30
        ED5C   30   5F   ED   DD   00   30   30   30
        ED64   30   30   30   30   30   30   30   30
        ED6C   30   30   30   30   30   30   30   75
        ED74   ED   DC   00   30   30   30   30   30
        ED7C   30   30   30   30   30   30   30   30
        ED84   30   30   30   30   30   BB   ED   DC
        ED8C   00   30   30   30   30   30   30   30
        ED94   30   30   30   30   30   30   30   30
        ED9C   30   30   30
```


B.3 COMMON SUBROUTINES

CRCHK:

This routine is called by *PROC and *DEFPROC during syntax-time, in order to check that all characters in the procedure name are alphanumeric. It can cater for any length of name because it enters a continuous loop, checking one character after another, until a non-alphanumeric character is detected. If this occurs before the end of the statement, then a syntax error will be produced when control returns to the ROM routine STMT-RET, because CHADD will not be pointing to either a colon or a carriage return.

PRTAT:

This routine is entered with the BC register pair pointing to the co-ordinate where printing is to commence: B holds the row number, and C the column number.

PRTSTG:

On entry, the HL register pair points to an address immediately before a string of characters, which holds the length of that string (strings longer than 255 characters are not catered for). Once arranged, the relevant information is passed to the ROM routine at 8252d,203Ch for printing.

INKEY:

Bit 5 of the System Variable FLAGS indicates whether or not the value of the last key entered and stored in LASTK (23560d,5C08h) has been read. The INKEY routine uses this fact to loop round until a new key has been pressed (remember that the ROM's keyboard reading routine is interrupt driven). The code is then returned to the A-register, after first signalling that the value has been taken.

GOTO:

The function of this subroutine is to force a BASIC "GOTO" instruction to a specified line. As this is most frequently done from structured commands (which store the line numbers in their own stacks) a check is first made to ensure that the bottom of the stack has not already been reached (for instance, if the BASIC program interpreted an *ENDPROC before a *PROC call).

GOTO is called with the A-register holding the new statement number, and HL pointing to the location immediately after the last entry in the relevant stack. Assuming the stack is not empty. the new line and statement numbers are inserted into the System Variables NEWPPC (23618d,5C42h) and NSPPC (23620d,5C44h) respectively.

GTLIN:

This routine also causes a BASIC program jump, but unlike "GOTO" it performs the jump itself – the ROM routines will not 'realise' that a

jump has actually been made. This is achieved by loading the address of the new line directly into the System Variable NXTLIN at 23637d,5C55h.

The entry point GTLIN2 is used when the HL register pair already holds the required address – it will normally have to be fetched from WRKSPC where it was placed by the SRPRG routine. This is performed from the main entry point.

SRPRG:

Capable of locating any extended command situated at the very start of a BASIC line, this routine is called by most of the structured commands. The entry point SPFWD is used when the required line number is known to be *after* the calling line in the program.

Use is made of the ROM routine NEXTONE (6584d,19B8h) to step through each line number in turn, and speed is greatly increased by initially searching only for an asterisk, indicating an extended command. Once this is found the entire command can be checked against the model pointed to by the DE register pair upon entry.

On exit, the zero flag will be set only if the required command has been found.

GTCOL:

GTCOL can be used by any routine requiring the three colour parameters BORDER, PAPER and INK. The latter two are combined into a single attribute byte and returned in the B register. The Border colour is returned in the A register.

On entry to the routine, the return address is removed from the stack and stored in the workspace – so allowing error jumps to be made without worrying about surplus entries on the stack.

Parameters are fetched in turn, using COLPM, where they are also checked to be in range.

CLRTAB:

Whenever an extended command forces a program jump (e.g. procedures, REPEAT-UNTIL etc.) it will store the relevant line numbers on the stack, to be retrieved when a return to the calling line is made. In a fully functional program, there will be as many numbers put on the stacks as are taken off, and so the size of each stack at the end of the program will be the same as at the beginning. However, if an error (including "BREAK") should occur during program execution, it is possible that one or more numbers may be left on a stack. If this were to happen too often, the stack would fill up and the program would stop with the report "Out of Memory" whenever a jump was attempted.

To prevent this from happening the CLRTAB subroutine is called from ENABLE and NEW to 'clear' each stack in turn. In fact the stacks are not physically cleared but the pointers are all reset so that the next entry on each stack will be the first.


```
Common Subroutines

CRCHK     rst  24              GETCHAR
      2   call ALPHNUM         Return immediately if first
      3   ret  nc              character is not alphanumeric.

Ch1       rst  32              NXTCHAR
      5   call ALPHNUM         Repeat until non-alphanumeric
      6   jr   c,Ch1           character is found.
      7   ret


PRTAT     push bc              Save row & column pointers.
      9   ld   a,2

     10   call SELDEV          Select screen.

     11   ld   a,22            ("AT")
     12   rst  16              PRINT
     13   pop  bc
     14   ld   a,b             (Column No.)
     15   rst  16              PRINT
     16   ld   a,c             (Row No.)
     17   rst  16              PRINT


PRTSTG    ld   b,0
     19   ld   c,(hl)          bc=length of string.
     20   inc  hl

     21   ex   de,hl           de=1st character.

     22   call PRTSTRG         Use the ROM routine.
     23   ret


INKEY     bit  5,(iy+1)        Wait for
     25   jr   z,INKEY         keypress.

     26   ld   a,(LASTK)       Get CODE of key pressed.

     27   res  5,(iy+1)        Signal it has been taken.
     28   ret


GOTO1     pop  de
     30   sbc  hl,bc           Check for
     31   jp   z,LOSERR        empty stack.
     32   add  hl,bc
     33   push de


GOTO2     ld   de,NSPPC        Store new
     35   ld   (de),a          statement no.
     36   dec  hl
     37   dec  de

     38   ldd                  Unstack new line number
     39   ldd                  into NEWPPC.

     40   inc  hl              Point to 1st empty location on stack.
     41   ret


GTLIN     ld   hl,(WRKSPC)

GTLN2     ld   (NXTLIN),hl     Impose address of new line.

     44   ld   (iy+10),128     Signal "no jump".
     45   ret


SPFWD     ld   hl,(NXTLIN)

SRPRG     ld   (WRKSPC+2),de   Points to model length.

SPnxt     call NEXTONE         Locate address of BASIC line.
     49   ld   (WRKSPC),de
     50   ld   (WRKSPC+6),hl

     51   inc  hl              Step past the four
     52   inc  hl              bytes holding
     53   inc  hl              line number
GetCr     inc  hl              and length.
     55   ld   a,(hl)
```

```
                cp   32          Ignore spaces.
                jr   z,GetCr

                cp   42          ("*")
                jr   z,Astfd     Jump forward with Extended commands.
                ld   hl,(WRKSPC)

                ld   a,(hl)      Test for end
                cp   40          of BASIC program.

                jr   c,SPnxt     Repeat if not reached.

                dec  a           Reset zero flag.
                ret

Astfd           inc  hl          Step past "*".
                ld   de,(WRKSPC+2)

                ld   a,(de)      Fetch length of model.

                inc  a           Allow for 1st letter.

                inc  de          Point to start of model.
                ld   b,a

                ld   c,&FF       cpi decrements bc,so make c large.
SRL             ld   a,(de)

                cpi              Compare subject command with model.

                jr   nz,Negck    Jump if it does not match.
                inc  de

                djnz SRL         Repeat for all characters.
                ret

Negck           ld   hl,(WRKSPC) Check has failed,so
                ld   de,(WRKSPC+2) resume search from
                jr   SPnxt       next statement.


GTCOL           pop  hl          Remove return address,
                ld   (WRKSPC),hl and store in Workspace.
                call EXPTNUM

                cp   44          (Comma)
                jp   nz,ERROR
                call NEXT2NM
                call SYNEND
                call COLPM       Fetch INK parameter.

                call COLPM       Fetch PAPER parameter.
                add  a
                add  a

                add  a           PAPER*8
                add  b           +INK
                call COLPM       Fetch BORDER parameter.

                ld   hl,(WRKSPC) Fetch return address.

                jp   (hl)        Return to calling routine.


COLPM           push af
                call STKTOA
                pop  bc          b = "old" a.
                cp   8

                ret  c           Return with valid colours only.

                pop  bc          Remove return address.
                jp   COLERR


CLRTAB          ld   hl,PTABST
                call Clr
                ld   hl,RTABST
                call Clr
                ld   hl,WTABST
                call Clr
                ld   hl,KTABST
```



```
Clr             ld   d,h         hl = pointer.
                ld   e,l
                inc  de

                inc  de          de = bottom of stack.
                ld   (hl),e
                inc  hl
                ld   (hl),d
                ret
```

HEX DUMP:

```
ED9F  DF CD 88 2C DO E7 CD 88
EDA7  2C 38 FA C9 C5 3E 02 CD
EDAF  01 16 3E 16 D7 C1 78 D7
EDB7  79 D7 C9 06 00 4E 23 EB
EDBF  CD 3C 20 C9 FD CB 01 6E
EDC7  28 FA 3A 08 5C FD CB 01
EDCF  AE C9 D1 ED 42 CA F1 EC
EDD7  09 D5 11 44 5C 12 2B 1B
EDDF  ED A8 ED A8 23 C9 2A 95
EDE7  ED 22 55 5C FD 36 0A 80
EDEF  C9 2A 55 5C ED 53 97 ED
EDF7  CD B8 19 ED 53 95 ED 22
EDFF  9B ED 23 23 23 23 7E FE
EE07  20 28 FA FE 2A 28 0A 2A
EE0F  95 ED 7E FE 28 38 E1 3D
EE17  C9 23 ED 5B 97 ED 1A 3C
EE1F  13 47 0E FF 1A ED A1 20
EE27  04 13 10 F8 C9 2A 95 ED
EE2F  ED 5B 97 ED 18 C2 E1 22
EE37  95 ED CD 82 1C FE 2C C2
EE3F  51 EC CD 79 1C CD 3E EC
EE47  CD 58 EE CD 58 EE B7 B7
EE4F  B7 80 CD 58 EE 2A 95 ED
EE57  E9 F5 CD 94 1E C1 FE 08
EE5F  D8 C1 C3 E9 EC 21 3D ED
EE67  CD 79 EE 21 5D ED CD 79
EE6F  EE 21 73 ED CD 79 EE 21
EE77  89 ED 54 5D 13 13 73 23
EE7F  72 C9
```

B.4 COMMAND ROUTINES

B.4.1 FUNDAMENTAL COMMANDS

NEW

Function:

As "NEW" will deactivate the extension program, there is an extended command *NEW which will erase any BASIC program and variables without disabling the new commands. The first program line, however, (which should be a USR call to ENABLE routine) is not erased. Whilst not as thorough as the standard command, in that it does not physically clear out any RAM, *NEW should prove quite adequate for most purposes.

Syntax:

*NEW

or *NEW b,p,i

Parameters:

b,p,i specify the BORDER, PAPER and INK colours respectively and need only be given if the default colours are to be altered.

The initial default values (i.e. those used until any others are specified) are 1, 1 and 7 (white ink on dark blue background).

Error Reports:

None.

```
Essential Commands - NEW

NEW       rst  24                GETCHAR
      2   call NUMERIC
      3   jr   nc,NEW2           Jump if parameters present.
      4   call SYNEND
Newrun    ld   hl,(PROG)

      6   call NEXTONE           Skip 1st program line.
      7   ld   hl,(VARS)

      8   call RECLAIM           "Clear" program area.
      9   ld   de,(VARS)
     10   ld   hl,(ELIN)
     11   dec  hl

     12   call RECLAIM           "Clear" variables area.

     13   call SETMIN            "Clear" calculator stack etc.

Nwatr     ld   a,15              Default attribute - PAPER 1,INK 7.
     15   ld   (ATTRT),a
     16   ld   (ATTRP),a

Nwbdr     ld   a,1               Default BORDER.
     18   call BORDR2

     19   ld   (iy+0),255        Reset ERRNR.

     20   ld   hl,EXTEND
     21   push hl                Replace Error Return Address.

     22   jp   4720              Jump into ROM program.

NEW2      call GTCOL
     24   ld   (Nwbdr+1),a
     25   ld   a,b
     26   ld   (Nwatr+1),a
     27   jr   Newrun
```



```
NEWCOM    defb 2
     29   defm "NEW

NOTE: Some Assemblers may not accept lines 24 and 26,
      so these can be entered as:
                24  ld (EEAD),a
                26  ld (EEA5),a


HEX DUMP:

        EE81   DF   CD   1B   2D   30   35   CD   3E
        EE89   EC   2A   53   5C   CD   B8   19   2A
        EE91   4B   5C   CD   E5   19   ED   5B   4B
        EE99   5C   2A   59   5C   2B   CD   E5   19
        EEA1   CD   B0   16   3E   0F   32   8F   5C
        EEA9   32   8D   5C   3E   01   CD   9B   22
        EEB1   FD   36   00   FF   21   6D   EA   E5
        EEB9   C3   70   12   CD   35   EE   32   AD
        EEC1   EE   78   32   A5   EE   18   C2   02
        EEC9   4E   45   57
```

Explanation:

Initially, a check is made to see if any parameters are present: if there are, the routine jumps to NEW2. Otherwise the unqualified *NEW is executed immediately, unless in syntax- time.

A number of ROM routines are called during execution:

1. NEXTONE is used to step over the first program line (that way the USR call to ENABLE does not have to be typed in every time *NEW is used).
2. Two calls to RECLAIM are then made to adjust the relevant system variables: the first when the program area is cleared, and the second when the variables are cleared.
3. SETMIN is then called to 'clear' the editing and subsequent areas.
4. The Permanent & Temporary attribute values are loaded into the relevant system variables and the BORDER colour set with a call to BORDR2.

Finally, the EXIT routine is performed, but the jump back into the ROM goes to the end of the standard NEW command routine instead of to STMT-RET.

NEW2 – If colour parameters are specified, a call is made to GTCOL, which returns with the border colour in A and the attribute value in B. These values are then used to alter the default values in the *NEW routine itself.

SPEED

Function:

When a procedure or function key routine is called, the computer will find the line it must jump to by searching through the entire program for the corresponding *DEFPROC or *DEFKEY. If that is toward the end of a long program, the time factor involved can significantly reduce the speed of execution.

To overcome this to some extent, the *SPEED command can be used but only if you are willing to accept a small compromise. That is, the routine to be called must come *after* the calling line in the program. If this is acceptable, *SPEED 1 will cause the computer to start any searches from the line immediately after the calling line, greatly reducing the time taken to find the required position in long programs.

The complementary command, *SPEED 0, will return the program to normal.

Syntax:

*SPEED n

Parameters:

n = 0 or 1

Error Reports:

n>255 – Integer out of range

```
Essential Commands - SPEED

SPEED     call EXPTNUM
      2   call SYNEND
      3   call STKTOA
      4   ld   hl,PROG
      5   and  a
      6   jr   z,EntSpd          Jump forward with SPEED 0.
      7   ld   hl,NXTLIN
EntSpd    ld   (Prcrun+1),hl
      9   ld   (Numky+1),hl
     10   jp   EXIT
SPDCOM    defb 4
     12   defm "SPEED


NOTE: Some Assemblers may not accept lines 8 and 9,
      so these can be entered as:
                EntSpd  ld (EFD4),hl
                     9  ld (F1B6),hl


HEX DUMP:

        EECC   CD   82   1C   CD   3E   EC   CD   94
        EED4   1E   21   53   5C   A7   28   03   21
        EEDC   55   5C   22   D4   EF   22   B6   F1
        EEE4   C3   46   EC   04   53   50   45   45
        EEEC   44
```

Explanation:

The HL register pair is provisionally loaded with the address of the system variable PROG, which holds the address of the first byte in the BASIC program area. The parameter is then tested, and if non-zero the address in HL is changed to that of NXTLIN – another system variable which always holds the address of the next BASIC line to be executed.

In either case, the contents of HL are placed into strategic points in the *PROC and *KEYIN command routines causing both routines to commence searching from the next line instead of the first when finding the relevant places to jump to.

B.4.2 SOME PRELIMINARY COMMANDS

CAPS LOCK

Function:

Many programs require keyboard input in the form of a single key-press, but in decoding these entries it is usually necessary to check for both upper and lower case characters. *CAPS commands allows CAPS LOCK to be set permanently, thus simplifying subsequent decoding.

Syntax:

*CAPS n

Parameters:

n=1 (on) or n=0 (off)

Error Reports:

n>255 – Integer out of range

```
Preliminary Commands - CAPS

CAPS      call EXPTNUM
      2   call SYNEND
      3   call STKTOA
      4   and  a
      5   jr   z,Caps2            Jump forward with CAPS 0.

      6   ld   a,8                Set bit 4.
Caps2     ld   (FLAGS2),a
      8   jp   EXIT
CAPCOM    defb 3
     10   defm "CAPS


HEX DUMP:

        EEED   CD   82   1C   CD   3E   EC   CD   94
        EEF5   1E   A7   28   02   3E   08   32   6A
        EEFD   5C   C3   46   EC   03   43   41   50
        EF05   53
```

Explanation:

EXPTNUM and STKTOA are used to fetch the parameter, which is then tested. If zero, Bit 3 of the system variable FLAGS2 (23658d,5C6Ah) is reset, deactivating CAPS LOCK. Any other number is treated as simply "non-zero", and Bit 3 of FLAGS2 is set to indicate CAPS LOCK is ON

CAPS CHECK

Function:

Whilst *CAPS is very useful for single key entries, its effectiveness is severely reduced when used for complete string inputs – it is quite possible that an inexperienced user could accidentially reset Caps Lock using Caps Shift and 2. To overcome this problem, the *CAPCHK command can be applied to the string input. All lower case characters in the specified string will be converted to upper case, while all other characters will remain unaffected.

Syntax:

*CAPCHK i$

Parameters:

i$ = any string variable

Error Reports:

Variable not found – i$ not defined

```
Preliminary Commands - CAPCHK

CCHK      call EXPTSTG
     2    call SYNEND
     3    call STKFTCH
ChkChr    ld   a,b
     5    or   c
     6    dec  bc
     7    jp   z,EXIT              Leave when whole string checked.
     8    ld   a,(de)

     9    cp   97                  Ignore if < "a".
    10    jr   c,Cend

    11    cp   123                 Ignore if > "z".
    12    jr   nc,Cend

    13    sub  32                  Make it upper case.
    14    ld   (de),a

Cend      inc  de                  Repeat for rest
    16    jr   ChkChr              of string.
CHKCOM    defb 5
    18    defm "CAPCHK    "
```

```
HEX DUMP:

        EF06    CD   8C   1C   CD   3E   EC   CD   F1
        EF0E    2B   78   B1   0B   CA   46   EC   1A
        EF16    FE   61   38   07   FE   7B   30   03
        EF1E    D6   20   12   13   18   EB   05   43
        EF26    41   50   43   48   4B
```

Explanation:

The ROM routine EXPTSTG is utilised to check and stack any given string or string variable, in the same way that EXPTNUM is used for numerical parameters. STKFTCH returns with DE pointing to the first character in the string. By stepping along the string, and subtracting 32 from the code of any character which lies in the range 97-122, the routine is able to convert all lower case letters to upper case.

16-BIT POKE

Function:

When POKEing 16 bit binary numbers (i.e. 0 to 65535 decimal) into RAM it is inconvenient to have to split the number into two 8 bit numbers. E.g. to poke the 16 bit number x into address 32,000 would require the following lines:

POKE 32000,x–256*(INT(x/256))

POKE 32001,INT(x/256)

The equivalent *POKE command would be simply *POKE 32000,x.

Syntax:

*POKE nn,xx

Parameters:

nn = address (0 to 65535)

xx = value to be entered (0 to 65535)

Error Reports:

nn or xx>65535

```
Preliminary Commands - POKE

POKE      call EXPT2NM
     2    call SYNEND
     3    call STKTOBC             Fetch value.
     4    push bc

     5    call STKTOBC             Fetch address.
     6    ld   h,b
     7    ld   l,c
     8    pop  de

     9    ld   (hl),e              Poke value into
    10    inc  hl                  address and
    11    ld   (hl),d              address+1.
    12    jp   EXIT
POKCOM    defb 3
    14    defm "POKE
```

```
HEX DUMP:

        EF2B    CD   7A   1C   CD   3E   EC   CD   99
        EF33    1E   C5   CD   99   1E   60   69   D1
        EF3B    73   23   72   C3   46   EC   03   50
        EF43    4F   4B   45
```

Explanation:

The ROM routine EXPT2NM is used in the same way as EXPTNUM, except that two numbers, separated by a comma, are expected.

During run-time, two successive calls are made to the ROM routine "STKTOBC", which compresses the last number on the calculator stack into the BC register pair.

NOISE

Function:

The standard Spectrum BEEP command is very limited in terms of the sounds it can produce. The new command *NOISE allows the production of white noise, which is useful for a wide range of effects.

Syntax:

*NOISE n

Parameters:

n = length of noise (0-255)

Error Reports:

n>255 – Integer out of range

```
Preliminary Commands - NOISE

NOISE     call EXPTNUM
     2    call SYNEND
     3    call STKTOA
     4    ld   b,a
     5    ld   a,(BORDCR)
     6    rrca                        Derive value to
     7    rrca                        output to
     8    rrca                        Port 254.
NLP1      push bc

    10    ld   b,(hl)                 Output noise
    11    sbc  hl,de                  for random
NLP2      djnz NLP2                   period.
    13    out  (&FE),a

    14    xor  16                     Toggle bit 4.
    15    pop  bc
    16    djnz NLP1
    17    jp   EXIT
NSECOM    defb 4
    19    defm "NOISE

HEX DUMP:

        EF46   CD   82   1C   CD   3E   EC   CD   94
        EF4E   1E   47   3A   48   5C   0F   0F   0F
        EF56   C5   46   ED   52   10   FE   D3   FE
        EF5E   EE   10   C1   10   F3   C3   46   EC
        EF66   04   4E   4F   49   53   45
```

Explanation:

The length of noise to be produced is fetched as a normal numerical parameter.

Since both the Beeper and the Border colour are controlled from I/O Port FEh, the contents of BORDER are now fetched and converted to the relevant output byte. A loop is now entered in which Bit 4 (the Beeper) is toggled on and off at random intervals to produce the noise.

USER DEFINED GRAPHICS

Function:

As standard, the Spectrum allows 21 User Defined Graphics (see pages 92-94 of the Sinclair BASIC manual). In many programs this will not be enough, in which case the extended command *UDG can be used to extend the number of UDG's to 63.

These characters are divided into three blocks numbered 0, 1 and 2. Only block 0 can be printed, and the *UDG command is used to exchange blocks.

For instance to print the first character of each block:

```
100 PRINT "A": REM Graphics character
110 *UDG 1,0
120 PRINT "A": REM Graphics character
130 *UDG 0,1
140 *UDG 2,0
150 PRINT "A": REM Graphics character
160 *UDG 0,2
```

Note lines 130 and 160: these are used to replace the blocks in their current positions. If these lines were omitted and the program run

again, each block would end-up holding a different set of characters to when it started. The following diagram may make this clearer:

BLOCK:	0	1	2	
Start of Program:	0	1	2	
*UDG 1,0	1	0	2	Blocks 0 and 1 exchanged
*UDG 2,0	2	0	1	Blocks 0 and 2 exchanged
End of Program:	2	0	1	

It is obviously very important to keep track of which block contains which set of characters. The order of the numbers after *UDG command will not affect the operation of the program (i.e. *UDG 0,1 is identical to *UDG 1,0) as the blocks are simply exchanged; however, the program will be easier to follow if the convention of treating "," as meaning "to" is followed. For instance, line 110 of the example program is intended to move the characters from block 1 into the display area, block 0. The fact that block 0 is also moved to block 1 is purely academic to the operation of the command. Similarily, in line 130 the intention of the command is to restore the characters back into block 1. In effect, the storing and retrieval of the block 0 characters are completely transparent to the programmer.

The easiest way to enter more than 21 UDG's is to use POKE USR"a"+1 etc. (as per page 94 of the Spectrum manual) and then to use *UDG 0,1 and *UDG 0,2 to store them in their relevant locations.

Note that all blocks occupy consecutive locations in memory, starting at address 65032. SAVEing all three block is therefore achieved by SAVE "udgs" CODE 65032,504 (there are 21 × 8 = 168 bytes to each block).

Syntax:

*UDG x,y

Parameters:

x = UDG Bank number (0 to 2 inclusive)
y = UDG Bank number (0 to 2 inclusive)

Error Reports:

x or y >2 – Integer out of range

```
Preliminary Commands - UDG

UDG       call EXPT2NM
     2    call SYNEND
     3    call GTBLK                  Fetch second block.
     4    push hl

     5    call GTBLK                  Fetch first block.
     6    pop  de
     7    ld   b,c

ULP3      ld   a,(de)                 Exchange
     9    ldi                         one byte from
    10    dec  hl                     each block.
    11    ld   (hl),a
    12    inc  hl
    13    djnz ULP3
    14    jp   EXIT
```

```
GTBLK     call STKTOA
    16    pop  hl
    17    cp   3                    Ensure block is in range.
    18    jp   nc,INTERR
    19    push hl

    20    ld   hl,(UDGST)           hl = block 0.

    21    ld   bc,168               (21*8)bytes/block.

Now enter loop to derive start of block.

Unxt      and  a                    Subtract 000 for block 0
    23    ret  z                              168 for block 1
    24    sbc  hl,bc                          336 for block 2.
    25    dec  a
    26    jr   Unxt
UDGCOM    defb 2
    28    defm "UDG
```

HEX DUMP:

```
EF6C   CD  7A  1C  CD  3E  EC  CD  86
EF74   EF  E5  CD  86  EF  D1  41  1A
EF7C   ED  A0  2B  77  23  10  F8  C3
EF84   46  EC  CD  94  1E  E1  FE  03
EF8C   D2  E5  EC  E5  2A  7B  5C  01
EF94   A8  00  A7  C8  ED  42  3D  18
EF9C   F9  02  55  44  47
```

Explanation:

Once again, EXPT2NM is used to stack two numbers separated by a comma. The subroutine GTBLK is called twice to fetch each parameter and calculate the starting position of the relevant block of UDG's. The loop UL3 does the job of actually exchanging the UDG's.

KEYBOARD

Function:

The standard Spectrum keyboard routines are very much a compromise in terms of the delay which occurs between a key being pressed and it starting to autorepeat, and also in terms of the speed at which it then continues to repeat. The *KYBD command allows the user to set these parameters to personal preference – in addition, the length of the click produced can also be tailored.

Syntax:

*KYBD a,b,c

Parameters:

a = delay ..... (initially 35)
b = speed ..... (initially 5)
c = length of click ..... (initially 0)

Error Reports:

a,b,c >255 – Integer out of range

Preliminary Commands - KYBD

```
KYBD      call EXPTNUM
     2    cp   44                   (Comma)
     3    jp   nz,ERROR
     4    call NEXT2NM
     5    call SYNEND
     6    call STKTOA
     7    ld   (PIP),a              Length of click.
     8    call STKTOA

     9    ld   (REPPER),a           Speed of repeat.
    10    call STKTOA

    11    ld   (REPDEL),a           Delay before repeat.
    12    jp   EXIT
KBDCOM    defb 3
    14    defm "KYBD
```

HEX DUMP:

```
EFA1   CD  82  1C  FE  2C  C2  51  EC
EFA9   CD  79  1C  CD  3E  EC  CD  94
EFB1   1E  32  39  5C  CD  94  1E  32
EFB9   0A  5C  CD  94  1E  32  09  5C
EFC1   C3  46  EC  03  4B  59  42  44
```

Explanation:

The first three lines of the routine check for a numerical expression followed by a comma. The ROM routine NEXT2NM increments CHADD before executing EXPT2NM.

In run-time, three calls to STKTOA are made, with the relevant system variables being set appropriately.

B.4.3 STRUCTURED COMMANDS

PROCEDURES

Function:

Procedures perform a similar function to GOSUB, but allow the relevant routine to be called by name rather than line number.

When the program interprets a *PROC command it will search for a *DEFPROC with the same name – if one is not found the program will halt with error report 'Statement lost'. Assuming all is well however, program execution will branch to the line immediately after the *DEFPROC and continue until an *ENDPROC command is found, whereupon it will revert to the statement immediately after the calling *PROC.

If a *DEFPROC is encountered other than during a *PROC call, the program will jump to the line immediately after the next *ENDPROC.

If an *ENDPROC is read without the relevant *PROC call, the error report 'Statement lost' will be given, as there is nowhere for the program to return to.

Note that whilst *PROC can be used as freely as GOSUB, *DEFPROC and *ENDPROC must have their respective lines all to themselves (although *ENDPROC can be used in other statements, it will not then be found if *DEFPROC is executed).

Syntax:

*PROC name – calls the procedure 'name'
*DEFPROC name – defines the start of procedure name
*ENDPROC – defines the end of a procedure

Parameters:

For *PROC and *DEFPROC – 'name' can be any number(s) and/or character(s). Spaces are also permitted.

Error Reports:

Statement lost – no corresponding *DEFPROC;
Procedure before calling line in Speed 1.
Out of memory – more than 10 procedures nested

```
Structured Commands - PROCEDURES

PROC      bit  7,(iy+1)          Check name in
      2   call z,CHRCHK          syntax-time.
      3   call SYNEND

Prcrun    ld   hl,(PROG)         (NXTLIN) in SPEED 1.
NxtDpc    ld   de,DPCCOM

      6   call SRPRG             Search for DEFPROC.
      7   jp   nz,LOSERR

      8   inc  hl                hl points to start of DEFPROC name.

      9   ld   de,(CHADD)        de points to start of PROC name.
```



```
PLP1      ld   a,(de)
     11   call ALPHNUM
     12   jr   nc,PEND

     13   cpi                    Compare corresponding characters of
     14   jr   nz,NOTFND         PROC and DEFPROC names.
     15   inc  de

     16   jr   PLP1              Repeat for all characters in PROC name.

PEND      ld   a,(hl)            Check DEFPROC name isn't any
     18   call ALPHNUM           longer than PROC name.
     19   jr   c,NOTFND

     20   ld   (CHADD),de        The names match.
     21   ld   de,(PTABST)
     22   ld   hl,RTABST

     23   sbc  hl,de             Check for stack overflow.
     24   jp   z,MEMERR
     25   ld   hl,PPC

     26   ldi                    Stack line no.
     27   ldi
     28   ld   a,(SUBPPC)
     29   inc  a

     30   ld   (de),a            Stack no. of next statement.
     31   inc  de

     32   ld   (PTABST),de       Store new pointer.
     33   call GTLIN

     34   jp   ELSRUN            See IF - ELSE.

NOTFND    ld   hl,(WRKSPC)       Names did not match,so
     36   jr   NxtDpc            continue searching.
PRCCOM    defb 3
     38   defm "PROC


DPRC      bit  7,(iy+1)
     40   call z,CHRCHK          Check name in syntax-time.
     41   call SYNEND
     42   ld   de,EPCCOM

     43   call SPFWD             Search for next ENDPROC.
     44   jp   nz,LOSERR
     45   call GTLIN
     46   jp   ELSRUN
DPCCOM    defb 6
     48   defm "DEFPROC


EPRC      call SYNEND
     50   ld   hl,(PTABST)
     51   ld   bc,PTAB
     52   sbc  hl,bc             Check stack not empty.
     53   jp   z,LOSERR
     54   add  hl,bc
     55   dec  hl

     56   ld   a,(hl)            a = statement no.

     57   call GOTO2             Stack line & statement nos.

     58   ld   (PTABST),hl       Update pointer.
     59   jp   EXIT
EPCCOM    defb 6
     61   defm "ENDPROC
```

HEX DUMP:

```
EFC9   FD  CB  01  7E  CC  9F  ED  CD
EFD1   3E  EC  2A  53  5C  11  41  F0
EFD9   CD  F3  ED  C2  F1  EC  23  ED
EFE1   5B  5D  5C  1A  CD  B8  2C  30
EFE9   07  ED  A1  20  30  13  18  F3
EFF1   7E  CD  B8  2C  38  27  ED  53
EFF9   5D  5C  ED  5B  3D  ED  21  5D
F001   ED  ED  52  CA  DD  EC  21  45
F009   5C  ED  A0  ED  A0  3A  47  5C
```

```
F011   3C   12   13   ED   53   3D   ED   CD
F019   E5   ED   C3   4D   F1   2A   95   ED
F021   1B   B3   03   50   52   4F   43   FD
F029   CB   01   7E   CC   9F   ED   CD   3E
F031   EC   11   63   F0   CD   F0   ED   C2
F039   F1   EC   CD   E5   ED   C3   4D   F1
F041   06   44   45   46   50   52   4F   43
F049   CD   3E   EC   2A   3D   ED   01   3F
F051   ED   ED   42   CA   F1   EC   09   2B
F059   7E   CD   D9   ED   22   3D   ED   C3
F061   46   EC   06   45   4E   44   50   52
F069   4F   43
```

Explanation:

PROC – During syntax-time, a call is made to the 'common' routine CRCHK, which checks that the procedure name given is legal, before exiting via SYNEND.

In run-time the check is circumvented to save time – the name must in any case be correct for it to have passed syntax. A call is then made to SRPRG to locate the first DEFPROC. Assuming one is found, the name is checked against that in the PROC statement by the loop PL1. If it does not match then the program loops back via NOTFND to consider the next DEFPROC.

When a match is found, a jump is made to PEND which first checks there are no more characters in the DEFPROC name – thus a procedure called 'firstname' can be differentiated from one called simply 'first'.

A check is now made to ensure the procedure stack is not already full (in which case the error report "Out of memory" is given) before storing the current line number and the incremented statement number.

Finally, GTLIN is called followed by a jump to ELSRUN – the former prepares the relevant system variables, whilst the latter updates CHADD to point to the end of the current BASIC line. (This is necessary to prevent any further statements in the line being executed before the procedure is called.)

When control returns to the ROM, the combined effect of these two routines is to force program execution to re-commence from the line immediately after the recognised DEFPROC, without the ROM having to make a GOTO or GOSUB jump.

DEFPROC – As with *PROC, the name-checking function is called only during syntax-time. In run-time, a search is made for the next *ENDPROC – as this must come after the *DEFPROC, the faster SPFWD routine can be used. GTLIN is called prior to leaving via ELSRUN.

ENDPROC – The size of the stack is now checked, and a 'statement lost' error report given if it is empty. Otherwise, the number of the next statement in the line containing the calling *PROC is removed from the stack and a call made to GOTO2, which unstacks the line number and loads both parameters into the required system variables. In consequence a standard GOTO will be performed when control is passed back to the ROM.

REPEAT-UNTIL LOOPS

Function:

All program lines between *REPEAT and *UNTIL will be executed in a continuous loop until the specified condition is true, whereupon program execution will continue from the line after *UNTIL.

Note that both *REPEAT and *UNTIL must have their own individual line numbers with no other statements on those lines.

Syntax:

*REPEAT
*UNTIL condition

Parameters:

*UNTIL – condition is True (1) or False (0)

Error Reports:

Statement lost – *UNTIL without *REPEAT
Out of memory – more than 10 nested loops

```
Structured Commands - REPEAT-UNTIL

REPT      call SYNEND
     2    ld   hl,WTABST
     3    ld   de,(RTABST)
     4    call FCHLIN              Get current line no.

     5    inc  bc                  Point to next line.

     6    call STKLIN              Stack it.

     7    ld   (RTABST),hl         Revise pointer.
     8    jp   EXIT
REPCOM    defb 5
    10    defm "REPEAT


FCHLIN    pop  bc                  Remove return address
    12    sbc  hl,de               while checking
    13    jp   z,MEMERR            stack not already full.
    14    push bc

    15    ld   hl,PPC              Put current
    16    ld   c,(hl)              line number
    17    inc  hl                  into bc.
    18    ld   b,(hl)
    19    ret


STKLIN    ex   de,hl
    21    ld   (hl),c              Stack line number
    22    inc  hl                  and increment hl
    23    ld   (hl),b              to point to
    24    inc  hl                  top of stack.
    25    ret


UNTIL     call EXPTNUM
    27    call SYNEND
    28    call STKTOA
    29    ld   hl,(RTABST)
    30    ld   bc,RTAB
    31    sbc  hl,bc               Check stack not empty.
    32    jp   z,LOSERR
    33    add  hl,bc
    34    and  a

    35    jr   nz,Rtrue            Jump forward if condition true.

    36    call GOTO2               Loop back to last REPEAT.
    37    jp   EXIT
```

```
Rtrue     dec  hl                  Condition was true,
    39    dec  hl                  so reset pointer before
    40    ld   (RTABST),hl         continuing to next BASIC line.
    41    jp   EXIT
UTLCOM    defb 4
    43    defm "UNTIL
```

HEX DUMP:

```
F06B   CD  3E  EC  21  73  ED  ED  5B
F073   5D  ED  CD  89  F0  03  CD  97
F07B   F0  22  5D  ED  C3  46  EC  05
F083   52  45  50  45  41  54  C1  ED
F08B   52  CA  DD  EC  C5  21  45  5C
F093   4E  23  46  C9  EB  71  23  70
F09B   23  C9  CD  82  1C  CD  3E  EC
F0A3   CD  94  1E  2A  5D  ED  01  5F
F0AB   ED  ED  42  CA  F1  EC  09  A7
F0B3   20  06  CD  D9  ED  C3  46  EC
F0BB   2B  2B  22  5D  ED  C3  46  EC
F0C3   04  55  4E  54  49  4C
```

Explanation:

REPEAT – This routine stores the number of the next line (or rather, the present line +1) on the Repeat stacks, after first checking that it is not already full. FCHLIN and STKLIN are specified as subroutines as they are also called by *WHILE.

UNTIL – The ROM routines EXPTNUM and STKTOA are used to evaluate the expression following the *UNTIL command. During syntax-time, the first of these routines is used to confirm the presence of a valid numerical expression. In run-time, it places the evaluated expression onto the calculator stack, from where it is later collected by STKTOA, provided that it can be compressed into the A register (i.e., it is an integer in the range 0-255).

After the stack has been confirmed not empty, the value returned is tested. If it is zero, (i.e., false) the GOTO2 routine is called to force a program jump back to the line after the last *REPEAT. If it is non-zero (i.e., true) the last entry on the stack is cleared and the routine exited as normal – program execution will continue at the statement after the *UNTIL command.

WHILE-WEND LOOPS

Function:

Basically similar to a *REPEAT-UNTIL loop, with the major difference that the condition is evaluated *before* entry to the loop: if it is true, program execution will continue until the next *WEND command loops it back to the *WHILE instruction. If the condition is false, program execution will continue from after the next *WEND.

As with *REPEAT-UNTIL, both *WHILE and *WEND must have their own line numbers.

Syntax:

*WHILE condition
*WEND

Parameters:

*WHIILE – condition may be True (1) or False (0)

Error Reports:

Statement lost – *WHILE without *WEND, and vice versa

Out of memory – as for *REPEAT-UNTIL

Structured Commands - WHILE-WEND

```
WHLE      call EXPTNUM
     2    call SYNEND
     3    call STKTOA
     4    and  a
     5    jr   z,Wfalse            Jump forward if condition false.

Wtrue     ld   hl,KTABST
     7    ld   de,(WTABST)
     8    call FCHLIN              Stack un-incremented
     9    call STKLIN              line number.

    10    ld   (WTABST),hl         Update pointer.
    11    jp   EXIT

Wfalse    ld   de,WENCOM
    13    call SPFWD               Search for next WEND.
    14    jp   nz,LOSERR

    15    call GTLIN               Jump forward in BASIC.
    16    jp   EXIT
WILCOM    defb 4
    18    defm "WHILE

WEND      call SYNEND
    20    ld   hl,(WTABST)
    21    ld   bc,WTAB
    22    sub  a                   Specify 1st statement
    23    call GOTO1               and jump to line no. on top of stack.

    24    ld   (WTABST),hl         Update pointer.
    25    jp   EXIT
WENCOM    defb 3
    27    defm "WEND
```

HEX DUMP:

```
F0C9   CD  82  1C  CD  3E  EC  CD  94
F0D1   1E  A7  28  13  21  89  ED  ED
F0D9   5B  73  ED  CD  89  F0  CD  97
F0E1   F0  22  73  ED  C3  46  EC  11
F0E9   10  F1  CD  F0  ED  C2  F1  EC
F0F1   CD  E5  ED  C3  46  EC  04  57
F0F9   48  49  4C  45  CD  3E  EC  2A
F101   73  ED  01  75  ED  97  CD  D1
F109   ED  22  73  ED  C3  46  EC  03
F111   57  45  4E  44
```

Explanation:

WHILE – The expression following *WHILE is evaluated and one of the two possible routes taken accordingly: i) Expression TRUE: Calls are made to FCHLIN and STKLIN to save the contents of PPC in the WHILE stack. Note that this is not incremented before storage because *WEND must loop the program back to the *WHILE instruction itself, so that the condition may be re-evaluated. ii) Expression FALSE: Program execution is forced to recommence after the next *WEND command by calling SPFWD and GTLIN.

WEND – The last entry on the WHILE stack is passed to the relevant system variables using GOTO1 – note that the statement number is set to zero so that the program will continue from the first statement in the new line.

IF-ELSE

Function:

The extended commands *IF-*ELSE are direct equivalents to the IF-THEN-ELSE structure found in many BASIC dialects, THEN having been replaced by a colon. The actions involved in execution are as follows:

First, the condition "cond" is evaluated. If it is false, the program will jump to the statement immediately after the corresponding *ELSE command (which must be on the same program line). If "cond" is true, execution will continue from the statement immediately after the "*IF cond:" until the *ELSE command is reached, whereupon a jump is made to the next line.

Syntax:

*IF cond: statement(s): *ELSE: statement(s)

Parameters:

Cond – True (1) or False (0)

Error Reports:

Integer out of range – "cond">255

```
Structured Commands - IF-ELSE

IF        call EXPTNUM
      2   rst  24                 GETCHAR   Ensure colon
      3   cp   58                 (Colon)   is present.
      4   jp   nz,ERROR
      5   call SYNEND
      6   call STKTOA
      7   and  a

      8   jp   nz,EXIT            Leave now if condition true.

FdELSE    ld   de,32042           d=125,e=42("*").

     10   call NXTSTMT            Find next Extended command.

     11   jp   c,EXIT             Leave if not present.

     12   rst  32                 NXTCHAR
     13   cp   69                 ("E")
     14   jr   nz,FdELSE          Jump back if 1st letter not "E".

     15   ld   hl,ELSCOM          Make more thorough
     16   call CKCM1              check of command.

     17   jr   z,FdELSE           Jump back if not "ELSE".
     18   jp   EXIT
IFCOM     defb 1
     20   defm "IF


ELSE      rst  24                 GETCHAR
     22   cp   58                 (Colon)
     23   jp   nz,ERROR
     24   call SYNEND
```



```
ELSRUN    ld   de,31744           d=124,e=0
     26   call NXTSTMT            Find end of BASIC line.
     27   jr   nc,ELSRUN
     28   jp   EXIT
ELSCOM    defb 3
     30   defm "ELSE
```

```
HEX DUMP:

        F115    CD   82   1C   DF   FE   3A   C2   51
        F11D    EC   CD   3E   EC   CD   94   1E   A7
        F125    C2   46   EC   11   2A   7D   CD   90
        F12D    19   DA   46   EC   E7   FE   45   20
        F135    F2   21   58   F1   CD   0D   EC   28
        F13D    EA   C3   46   EC   01   49   46   DF
        F145    FE   3A   C2   51   EC   CD   3E   EC
        F14D    11   00   7C   CD   90   19   30   F8
        F155    C3   46   EC   03   45   4C   53   45
```

Explanation:

IF – During syntax-time, the presence of a valid numerical expression is confirmed by a call to EXPTNUM. In addition a simple check is made to ensure that the statement ends with a colon. This is intended to signal the user if he tries to use the command in an "IF-THEN" – type construct. During run-time, the evaluated expression is fetched, and a simple exit made if it is TRUE – thus the BASIC program will continue from the next statement.

If the expression is FALSE, a search is made along the line for the corresponding *ELSE command, using the ROM routine NXTSTMT. Before entering this routine, the D register is loaded with the maximum number of statements to check (125) and the E register contains the code of the character to be searched for – 42d ("*") in this case. If an asterisk is not found (as indicated by the carry flag being set), the routine is left.

Provided one is found, a more thorough check is made to ensure that it is indeed the required *ELSE command, using the command checking routine. Note that the intial letter "E" must be checked separately.

ELSE – Once again, a colon must be present after the command. In run-time, repeated calls to NXTSTMT update the CHADD pointer to indicate the end of the BASIC line, before exiting.

ON-ONEND

Function:

Essentially similar to *IF-*ELSE, but with any number of program lines in between.

Action is as follows:

If "cond" is true, *ON is effectively ignored.

If "cond" is false, program execution will jump to either:

1. the next *ON command, or
2. the next *ONEND command

*ONEND is always ignored by the program – it just acts as a marker for when "cond" is false.

Syntax:

*ON cond:
*ONEND

Parameters:

cond – True (1) or False (0)

Error Reports:

Integer out of range – "cond">255

```
Structured Commands - ON-ONEND

ON        call EXPTNUM
     2    call SYNEND
     3    call STKTOA
     4    and  a                    Leave now if
     5    jp   nz,EXIT              condition true.
     6    ld   de,ONCOM

     7    call SPFWD                Search for next "ON" or "ONEND".

     8    jp   nz,OKERR             Give "OK" report if not found.
     9    ld   hl,(WRKSPC+6)

    10    call GTLIN2               Make BASIC program jump.
    11    jp   EXIT
ONCOM     defb 1
    13    defm "ON


OEND      call SYNEND               Leave immediately as
    15    jp   EXIT                 only an end-marker.
OENCOM    defb 4
    17    defm "ONEND


HEX DUMP:

        F15D    CD   82   1C   CD   3E   EC   CD   94
        F165    1E   A7   C2   46   EC   11   7C   F1
        F16D    CD   F0   ED   C2   D1   EC   2A   9B
        F175    ED   CD   E8   ED   C3   46   EC   01
        F17D    4F   4E   CD   3E   EC   C3   46   EC
        F185    04   4F   4E   45   4E   44
```

Explanation:

ON – The following expression is evaluated as before, and the routine left if it is found to be TRUE. If FALSE, the program is searched for "*ON" using SPFWD. Note that "*ONEND" will also pass this test, because the check is only for the first two letters after the asterisk.

If neither of these commands is found, the "OK" 'error' report is given and the program ends. Otherwise GTLIN2 is used to cause program execution to continue from the next *ON or *ONEND command.

ONEND – As this is used purely as a marker to end a series of one or more *ON commands, nothing need be done except to leave via either SYNEND or EXIT.

KEYIN-DEFKEY-ENDKEY

Function:

Waits for a numerical key (0-9) to be pressed, and enters the value into the specified numeric variable or waits for any key(s) to be pressed, then adds the CODE to the end of the specified string. If the keys pressed were CAPSHIFT and any numerical key, a check is made to see if the key has been defined as a function key – if it has, the relevant routine is executed and program execution will recommence *at* the line with the *KEYIN command. Thus if *KEYIN is not the first statement in a line, all preceeding statements will be executed each time a function key is used.

Function keys are defined by using *DEFKEY n, where n is a number between 0 and 9 inclusive. All program lines between *DEFKEY and *ENDKEY will be executed whenever CAPSHIFT and the relevant numerical key are pressed together during a *KEYIN i$ command. If a function key has not been defined, the keypress will be ignored.

As with *DEFPROC-*ENDPROC, the execution of the actual *DEFKEY command will cause a jump to the line after the next *ENDKEY – both must have their own line numbers.

Syntax:

*KEYIN i
*KEYIN i$, *DEFKEY n; *ENDKEY

Parameters:

i = any existing numeric variable
i$ = any existing, undimensioned string variable
(i.e. a$-z$)
n = 0 to 9 inclusive

Error Reports:

Statement lost – *DEFKEY without *ENDKEY, or vice versa

```
Structured Commands - KEYIN,DEFKEY-ENDKEY

KEYS      rst  24                   GETCHAR
     2    call ALPHA                Check parameter is a valid variable.
     3    jp   nc,ERROR

     4    rst  32                   NXTCHAR
     5    cp   36                   ("$")  Jump forward with
     6    jp   nz,NONLY                    numeric variables.

     7    rst  32                   NXTCHAR
     8    call SYNEND

KEYIN     call INKEY                Wait for keypress.
    10    cp   4
    11    jr   c,KEYIN

    12    ld   (WRKSPC+8),a         Store CODEs greater than 3.

    13    cp   15                   Jump if CAPS+9.
    14    jr   z,Numky
    15    cp   14
    16    jr   z,KEYIN
```

```
        cp   13                 Leave if ENTER.
        jp   z,EXIT

        jr   nc,Stoky           Jump forward with valid characters.

Numky   ld   hl,(PROG)          (NXTLIN) in SPEED 1.
Numky2  ld   de,DKYCOM

        call SRPRG              Find DEFKEY.

        jr   nz,KEYIN           Loop back if not found.
        ld   a,(WRKSPC+8)

        inc  hl                 Store address of
        push hl                 DEFKEY parameter.

        sub  4                  Reduce CODE to range 0-11.
        ld   c,a
        ld   b,0

        ld   hl,KEYTAB          Fetch relevant
        add  hl,bc              entry from
        ld   a,(hl)             table.
        pop  hl

        cp   (hl)               Compare with DEFKEY parameter.
        jr   z,DkyFnd

        ld   hl,(WRKSPC)        Continue searching if
        jr   Numky2             they don't match.

DkyFnd  ld   de,(KTABST)        DEFKEY found,so
        ld   hl,WRKSPC          first check
        sbc  hl,de              size of stack.
        jp   z,MEMERR
        ld   hl,PPC

        ldi
        ldi                     Store current
        ld   (KTABST),de        line number,
        call GTLIN              update pointer
        jp   EXIT               and make jump.

Stoky   ld   hl,CHADD
        dec  (hl)               Reduce CHADD to point
        dec  (hl)               to variable name.

        call LOOKVAR            Search variables area.
        jp   c,VARERR
        inc  hl

        ld   c,(hl)             Fetch current
        inc  hl                 length of string
        ld   b,(hl)             into bc,and
        inc  bc                 allow for extra byte to be added.
        push hl
        push bc
        add  hl,bc

        call MAK1SPC            Make room.
        ex   de,hl
        inc  de
        pop  bc
        pop  hl

        ld   (hl),b             Store new length.
        dec  hl
        ld   (hl),c
        ld   a,(WRKSPC+8)

        ld   (de),a             Store CODE.
        jp   EXIT

NONLY   ld   hl,(CHADD)         Put CHADD
        dec  hl                 back to start of
        ld   (CHADD),hl         variable name.
        ld   (WRKSPC),hl
        call EXPTNUM
        call SYNEND
        call STKTOA
NxtNum  call BRKKEY
        jp   nc,BRKERR

        call INKEY              Wait for number
        call NUMERIC            key to be pressed.
        jr   c,NxtNum
```



```
        sub  48                 Reduce CODE to range 0-9.
        push af

        ld   hl,(WRKSPC)        Reduce CHADD again
        ld   (CHADD),hl         to point to variable name.

        call LOOKVAR            Find variable.
        jp   c,VARERR
        pop  af
        inc  hl
        ld   e,a
        ld   d,0

        call INTSTOR            Store value of key pressed.
        jp   EXIT
KEYCOM  defb 4
        defm "KEYIN

DKEY    rst  24                 GETCHAR
        call NUMERIC            Ensure parameter is a digit.
        jp   c,ERROR

        rst  32                 NXTCHAR
        call SYNEND
        ld   de,EKYCOM
        call SPFWD              Find next ENDKEY.
        jp   nz,LOSERR

        call GTLIN              Jump.
        jp   EXIT
DKYCOM  defb 5
        defm "DEFKEY

EKEY    call SYNEND
        ld   hl,(KTABST)
        ld   bc,KTAB
        sub  a                  Specify 1st statement.

        call GOTO1              Make jump.

        ld   (KTABST),hl        Update pointer.
        jp   EXIT
EKYCOM  defb 5
        defm "ENDKEY
```

HEX DUMP:

```
F18B  DF CD 8D 2C D2 51 EC E7
F193  FE 24 C2 19 F2 E7 CD 3E
F19B  EC CD C3 ED FE 04 38 F9
F1A3  32 9D ED FE 0F 28 0B FE
F1AB  0E 28 EE FE 0D CA 46 EC
F1B3  30 40 2A 53 5C 11 74 F2
F1BB  CD F3 ED 20 DC 3A 9D ED
F1C3  23 E5 D6 04 4F 06 00 21
F1CB  31 ED 09 7E E1 BE 28 05
F1D3  2A 95 ED 18 E0 ED 5B 89
F1DB  ED 21 95 ED ED 52 CA DD
F1E3  EC 21 45 5C ED A0 ED A0
F1EB  ED 53 89 ED CD E5 ED C3
F1F3  46 EC 21 5D 5C 35 35 CD
F1FB  B2 28 DA D5 EC 23 4E 23
F203  46 03 E5 C5 09 CD 52 16
F20B  EB 13 C1 E1 70 2B 71 3A
F213  9D ED 12 C3 46 EC 2A 5D
F21B  5C 2B 22 5D 5C 22 95 ED
F223  CD B2 1C CD 3E EC CD 94
F22B  1E CD 54 1F D2 ED EC CD
F233  C3 ED CD 1B 2D 38 F2 D6
F23B  30 F5 2A 95 ED 22 5D 5C
F243  CD B2 28 DA D5 EC F1 23
F24B  5F 16 00 CD 8C 2D C3 46
F253  EC 04 4B 45 59 49 4E DF
F25B  CD 1B 2D DA 51 EC E7 CD
F263  3E EC 11 8E F2 CD F0 ED
F26B  C2 F1 EC CD E5 ED C3 46
F273  EC 05 44 45 46 4B 45 59
F27B  CD 3E EC 2A 89 ED 01 8B
F283  ED 97 CD D1 ED 22 89 ED
F28B  C3 46 EC 05 45 4E 44 4B
F293  45 59
```

Explanation:

***KEYIN** – The *KEYIN command is entered at KEYS, which differentiates between numerical and string inputs; the former are dealt with by the routine NONLY and the latter by KEYIN.

S-Input – The common routine INKEY is first used to load the CODE of the last key pressed into the accumulator, where it is inspected. If it represents a caps-shifted numerical key, it is passed to Numky. If it is "ENTER" the routine is left, and if it is any other key having a code greater the 15d,0Fh it is passed to Stoky.

Numky – The BASIC program area is searched for a *DEFKEY using SRPRG; if one is not found then the routine loops back to await a further key press.

Assuming one is found, the number following it is compared with the required one using the Function Key Table to translate between the code of the key pressed and that of the digit following the *DEFKEY command.

Once a match is achieved, the code at Dkyfnd is used to check the size of the KEYIN stack, store the present line number and make the necessary program jump.

Stoky – The designated string is located in the variables area using the ROM routine LOOKVAR, and the code of the key pressed added to the end using MAK1SPC to make the extra byte available.

NONLY – This is basically similar in operation to S-input, but only numerical keys are allowed. The value is stored in the relevant numeric variable using the ROM routine INTSTOR.

DEFKEY – The parameter is checked to ensure it is a single numerical digit in syntax time. In run-time, the BASIC program is searched for the next *ENDKEY command, and the relevant jump made in the normal manner.

ENDKEY – GOTO is called to unstack the last *KEYIN entry, and to signal the necessary program jump.

B.4.4 SCREEN HANDLING

ABSOLUTE DRAW

Function:

The DRAW command in Spectrum BASIC is relative – i.e. DRAW x,y will draw a line to a point x pixels horizontally and y pixels vertically from the present PLOT position. The extended command *DRAW is absolute, i.e., the line will be drawn from the present position to a point x,y. There is thus no need to calculate displacement values when trying to draw to a known point.

Syntax:

*DRAW x,y

Parameters:

x = horizontal co-ordinate of print to be drawn to

y = vertical co-ordinate of print to be drawn to

Error Reports:

Integer out of range – x>255 or y>175

```
Screen Handling Commands - DRAW

DRAW      call EXPT2NM
      2   call SYNEND
      3   call STKTOA                 Get y-coordinate.
      4   push af

      5   call STKTOA                 Get x-coordinate.

      6   ld   de,&0101               Set both signs to +ve.

      7   ld   hl,COORDS              Get current PLOT position.

      8   sub  (hl)                   Calculate x-displacement.
      9   jr   nc,Draw1

     10   neg                         Adjust displacement &
     11   ld   e,&FF                  sign if -ve.
Draw1     ld   c,a
     13   pop  af
     14   inc  hl

     15   sub  (hl)                   Calculate y-displacement.
     16   jr   nc,Draw2

     17   neg                         Adjust displacement &
     18   ld   d,&FF                  sign if -ve.
Draw2     ld   b,a

     20   call DRAWLIN                Draw the line.
     21   jp   EXIT
DRACOM    defb 3
     23   defm "DRAW


HEX DUMP:

        F295   CD   7A   1C   CD   3E   EC   CD   94
        F29D   1E   F5   CD   94   1E   11   01   01
        F2A5   21   7D   5C   96   30   04   ED   44
        F2AD   1E   FF   4F   F1   23   96   30   04
        F2B5   ED   44   16   FF   47   CD   BA   24
        F2BD   C3   46   EC   03   44   52   41   57
```

Explanation:

As the ROM routine DRAWLIN will ultimately be used to draw the actual line, and this has its own error handling capabilities, there is no need to check the given parameters. All that need be done is to subtract the present PLOT position (as held in COORDS) from the required destination to arrive at the horizontal and vertical displacements required. These are passed to the ROM routine in the B and C registers respectively – D and E represent their sign. Initially they are both set to 1, indicating positive.

UNDERLINE

Function:

Provides an exceptionally easy method of underlining characters without repeated use of PRINT AT, CHR$ 8 and OVER.

A semi-colon is required after the line to be underlined. The print position is left unaltered by the command.

Syntax:

*UNDL n

Parameters:

n = Number of character positions back to underline (0 to 255)

Error Reports:

n>255 – Integer out of range

```
Screen Handling Commands - UNDL

UNDL      call EXPTNUM
     2    call SYNEND
     3    call STKTOA
     4    and  a                    Disqualify "UNDL 0".
     5    jp   z,INTERR
     6    ld   b,a
     7    push bc
     8    ld   a,2

     9    call SELDEV               Select screen.

ULP1      ld   a,8                        Send CHR$ 8 to screen
    11    rst  16                   PRINT  relevant no. times.
    12    djnz ULP1

    13    ld   a,21                 ("OVER")
    14    rst  16                   PRINT
    15    ld   a,1
    16    rst  16                   PRINT
    17    pop  bc
ULP2      ld   a,95                       Send "_" to screen
    19    rst  16                   PRINT  specified no. times.
    20    djnz ULP2
    21    jp   EXIT
UNDCOM    defb 3
    23    defm "UNDL

HEX DUMP:

        F2C5   CD   82   1C   CD   3E   EC   CD   94
        F2CD   1E   A7   CA   E5   EC   47   C5   3E
        F2D5   02   CD   01   16   3E   08   D7   10
        F2DD   FB   3E   15   D7   3E   01   D7   C1
        F2E5   3E   5F   D7   10   FB   C3   46   EC
        F2ED   03   55   4E   44   4C
```


Explanation:

Two loops are used: the first, UL1 repeatedly sends the control code CHR$8 to the screen, in order to backspace the required number of times. OVER is then temporarily set to 1 before entering UL2, which sends the required underline character to the screen.

VERTICAL PRINTING

Function:

*VPRINT will print the specified string downward from the present print position. Its principle use is likely to be in printing spaces in conjunction with *SLEFT and *SRIGHT, but it will no doubt be found useful elsewhere.

Note that positioning and colour items cannot normally be included with the command, and will therefore need to be set up beforehand.

Syntax:

*VPRINT string

Parameters:

String = quoted string (e.g. "hello"),
simple string (e.g. a$) or
string array (e.g. a$(n))

Error Reports:

Out of screen – VPRINT does not cause screen scrolling.
Variable not found – string variable not defined.

```
Screen Handling Commands - VPRINT

VPRNT     call EXPTSTG
     2    call SYNEND
     3    ld   a,2
     4    call SELDEV               Select screen.

     5    call STKFTCH              Fetch parameters of string.
VLP1      ld   a,b
     7    or   c
     8    dec  bc

     9    jp   z,EXIT               Leave when all printed.
    10    push bc
    11    ld   a,(de)
    12    inc  de
    13    push de

    14    rst  16                   PRINT
    15    call GTPOS                Get new print position.
V1        and  a
    17    jr   nz,V2

    18    ld   c,32                 Adjust if printing down
    19    dec  b                    righthand of screen.

V2        inc  b                    Next row.

    21    dec  c                    Back one column.

    22    call PRTAT                Set new position.
    23    pop  de
    24    pop  bc
```

```
    25    jr   VLP1              Repeat for whole string.
VPTCOM    defb 5
    27    defm "VPRINT
```

```
HEX DUMP:

        F2F2   CD   8C   1C   CD   3E   EC   3E   02
        F2FA   CD   01   16   CD   F1   2B   7B   B1
        F302   0B   CA   46   EC   C5   1A   13   D5
        F30A   D7   CD   2D   F5   A7   20   03   0E
        F312   20   05   04   0D   CD   AB   ED   D1
        F31A   C1   18   E3   05   56   50   52   49
        F322   4E   54
```

Explanation:

STKFTCH is used to load the length into BC and the position of the first character in the string into DE. After printing the first character, the present print position (as obtained from GTPOS) will be one place to the right and one above the required position for the next character. The column counter is therefore decremented, whilst the row counter is incremented.

Note that the above does not apply when the string is being printed down the extreme right of the screen – the print position after printing a character will be on the correct row, but on the left of the screen. This anomaly is dealt with by the code V1/V2.

CLEAR ATTRIBUTES & CLEAR PRINT

Functions:

CLEAR ATTRIBUTES – It would often be helpful to alter the PAPER and INK colours of a display without erasing any printing. PAPER p: INK i only alter subsequent printing, not any that already exists.

The extended command *CLA can be used to alter the colour attributes of the entire screen, without affecting any printing – although the latter can be "hidden" by using identical values for the PAPER and INK components.

CLEAR PRINT – *CLP provides the complementary command to *CLA, that is, it will erase all printing from the screen without upsetting any attribute settings.

Syntax:

*CLA n

*CLP

Parameters:

For *CLA n = Attribute value (0 to 255) – see page 116 of the Spectrum BASIC manual for details on deriving Attribute values.

For *CLP None

Error Reports:

For *CLA n>255 – Integer out of range

For *CLP None


```
Screen Handling Commands - CLA,CLP

CLA       call EXPTNUM
      2   call SYNEND
      3   call STKTOA
      4   ld   hl,ATTRST          First byte to be altered.

      5   ld   b,4                Alter 4*192 bytes.

      6   call CLOOP              Enter values.

      7   ld   (ATTRP),a          Make it permanent.
      8   jp   EXIT
CLACOM    defb 2
     10   defm "CLA


CLP       call SYNEND
     12   sub  a                  All bytes will be 0.

     13   ld   hl,DISPST          First byte of Display File.

     14   ld   b,32               Alter 32*192 bytes.

     15   call CLOOP              Reset bytes.
     16   jp   EXIT
CLPCOM    defb 2
     18   defm "CLP


CLOOP     push bc
     20   ld   b,192
CLP2      ld   (hl),a             POKE value.
     22   inc  hl
     23   djnz CLP2
     24   pop  bc
     25   djnz CLOOP
     26   ret
```

```
HEX DUMP:

        F324   CD   82   1C   CD   3E   EC   CD   94
        F32C   1E   21   00   5B   06   04   CD   52
        F334   F3   32   BD   5C   C3   46   EC   02
        F33C   43   4C   41   CD   3E   EC   97   21
        F344   00   40   06   20   CD   52   F3   C3
        F34C   46   EC   02   43   4C   50
```

Explanation:

Both routines call the common subroutine "CLOOP" to load one block of either the display or the attributes file with the code in the A register: a parameter for *CLA but always 0 for *CLP.

*CLA calls CLOOP four times, as there are 4 × 192 (= 22 rows × 32 columns) = 768 bytes in the Attributes file.

*CLP calls the loop 32 times because there are 32 × 192 (= 32 bytes/line × 8 lines/row × 24 rows) = 6144 bytes in the display file.

SCREEN WIPE

Function:

Most programs require the screen display to be cleared with different colours, for instance the extension program colours might be obtained by the line:

100 BORDER 1 : PAPER 1 : INL 7 : CLS

A more compact alternative is to use *WIPE 1,1,7. Note the order of the parameters: BORDER followed by PAPER then INK. FLASH and BRIGHT are automatically reset to zero (i.e. off)

Syntax:

*WIPE b,p,i

Parameters:

b = BORDER colour (0 to 7)
p = PAPER colour (0 to 7)
i = INK colour (0 to 7)

Error Reports:

b,p or i >7 – Invalid colour.

```
Screen Handling Commands - WIPE

WIPE      call GTCOL              Fetch colours.

     2    call BORDR2             Change the Border colour.
     3    ld   a,b

     4    ld   (ATTRP),a          Permanently change
     5    call CLSCREN            screen colours.
     6    jp   EXIT
WIPCOM    defb 3
     8    defm "WIPE


HEX DUMP:

        F352    C5   06   C0   77   23   10   FC   C1
        F35A    10   F6   C9   CD   35   EE   CD   9B
        F362    22   78   32   8D   5C   CD   6B   0D
        F36A    C3   46   EC   03   57   49   50   45
```

Explanation:

The subroutine GTCOL is called to extract the three parameters and to combine the PAPER and INK elements into a single attribute byte. This is loaded into ATTRP, whilst the border colour is passed to BORDR2.

ATTRIBUTE SETTING

Function:

This command temporarily sets the attributes within the defined "window" area to a specified value. Note that print is unaffected by this command, but that printing anywhere in the area *after* the *ATTR command will alter the relevant attributes to the existing permanent values.

Syntax:

*ATTR (a,b to p,q)n

Parameters:

a,b = co-ordinate of top left hand corner of 'window'
p,q = co-ordinate of bottom right hand corner of 'window'
n = attribute value to be set (0 to 255)

Error Reports:

a or p>23; b or q>31; n>255 – Integer out of range
p>a; q>b – parameter error


```
Screen Handling Commands - ATTR

ATTR      cp   40                 ("(")
     2    jp   nz,ERROR
     3    call NEXT2NM
     4    cp   204                ("TO")
     5    jp   nz,ERROR
     6    call NEXT2NM
     7    cp   41                 (")")
     8    jp   nz,ERROR
     9    rst  32                 NXTCHAR
    10    call EXPTNUM
    11    call SYNEND
    12    ld   ix,WRKSPC-5
    13    ld   b,5                Five nos. to fetch.
ALP1      push bc
    15    inc  ix

    16    call STKTOA             Fetch parameter.

    17    ld   (ix+4),a           Store in workspace.
    18    pop  bc
    19    djnz ALP1

    20    cp   24                 Check first row no.
    21    jp   nc,INTERR

    22    ld   hl,ATTRST          hl points to attribute for (0,0).

    23    ld   de,32              No. attributes per row.
    24    ld   b,a

    25    and  a                  Jump forward if
    26    jr   z,Row1             starting at (0,0).

ALP2      add  hl,de              Add 32 for each row down.
    28    djnz ALP2

Row1      ld   a,(ix+3)           Now hl is start of 1st row.

    30    cp   32                 Check 1st column no. in range.
    31    jp   nc,INTERR
    32    ld   c,a

    33    add  hl,bc              Now hl is 1st coordinate.
    34    ld   a,(ix+1)

    35    cp   32                 Check 2nd column no. is in range.
    36    jp   nc,INTERR

    37    sub  c                  Calculate length of each line.
    38    jp   c,PARERR
    39    ld   c,a
    40    ld   a,(ix+2)

    41    cp   24                 Check 2nd row no. is in range.
    42    jp   nc,INTERR

    43    sub  (ix+4)             Calculate no. of lines.
    44    jp   c,PARERR
    45    ld   b,a

    46    ld   a,(ix+0)           Fetch specified attribute value.
    47    inc  b
    48    inc  c

ALP3      push bc                 Loop once for each line.
    50    push hl

    51    ld   de,32              Calculate start of
    52    add  hl,de              next line and
    53    ex   de,hl              store in de.
    54    pop  hl
    55    ld   b,c

ALP4      ld   (hl),a             Loop once for each byte in line.
    57    inc  hl
    58    djnz ALP4
    59    ex   de,hl
    60    pop  bc
    61    djnz ALP3
    62    jp   EXIT
ATTCOM    defb 3
    64    defm "ATTR
```

```
HEX DUMP:

      F372   FE   28   C2   51   EC   CD   79   1C
      F37A   FE   CC   C2   51   EC   CD   79   1C
      F382   FE   29   C2   51   EC   E7   CD   82
      F38A   1C   CD   3E   EC   DD   21   90   ED
      F392   06   05   C5   DD   23   CD   94   1E
      F39A   DD   77   04   C1   10   F4   FE   18
      F3A2   D2   E5   EC   21   00   58   11   20
      F3AA   00   47   A7   28   03   19   10   FD
      F3B2   DD   7E   03   FE   20   D2   E5   EC
      F3BA   4F   09   DD   7E   01   FE   20   D2
      F3C2   E5   EC   91   DA   F5   EC   4F   DD
      F3CA   7E   02   FE   18   D2   E5   EC   DD
      F3D2   96   04   DA   F5   EC   47   DD   7E
      F3DA   00   04   0C   C5   E5   11   20   00
      F3E2   19   EB   E1   41   77   23   10   FC
      F3EA   EB   C1   10   EF   C3   46   EC   03
      F3F2   41   54   54   52
```

Explanation:

The syntax checking is obviously a little more involved than usual, requiring two calls to NEXT2NM and one to EXPTNUM. Note how both these routines conveniently return with the A register holding the code of the first character after the number.

In run-time, the first thing to be done is to store all five numbers in the workspace, where they can be readily accessed. This done, the routine locates the relevant attribute address for the character position (a,b), and calculates the length of the line by subtracting the first x-co-ordinate from the 2nd. Similarly the number of lines is calculated from (p-a).

Two (nested) loops are then entered. The outer sets the starting point of each line by adding 32 to the start of the previous line; this loops back (p-a) times. The inner loop enters the attribute value (n) into each location in the line, stepping along from the starting position (q-b) times.

SCREEN SCROLLING

Function:

This command will move each row below and including row x up n lines – row x disappears, while the bottom row (line 23) becomes blank.

Syntax:

SCROLL x,n

Parameters:

x = top row number to scroll up to (0 to 22)
n = number of rows to scroll

Error Reports:

22>x>256 – subscript wrong
24>n>256 – subscript wrong
x or n>255 – Integer out of range.


```
Screen Handling Commands - SCROLL

SCROL     call EXPT2NM
     2    call SYNEND
     3    call STKTOA
     4    cp   25                  Pointless to scroll more
     5    jp   nc,SUBERR           than 24 times,
     6    and  a                   or no times at all.
     7    jp   z,SUBERR
     8    ld   b,a
     9    push bc
    10    call STKTOA
    11    pop  bc
    12    ld   c,a

    13    ld   a,22                Calculate no. of lines
    14    sbc  c                   to move.
    15    jp   c,SUBERR
    16    inc  a
    17    ld   c,a
SLP1      push bc
    19    ld   b,c
    20    call SCROLL
    21    pop  bc

    22    djnz SLP1                Loop back specified no. times.
    23    jp   EXIT
SCLCOM    defb 5
    25    defm "SCROLL

HEX DUMP:

      F3F6   CD   7A   1C   CD   3E   EC   CD   94
      F3FE   1E   FE   19   D2   D9   EC   A7   CA
      F406   D9   EC   47   C5   CD   94   1E   C1
      F40E   4F   3E   16   99   DA   D9   EC   3C
      F416   4F   C5   41   CD   00   0E   C1   10
      F41E   F8   C3   46   EC   05   53   43   52
      F426   4F   4C   4C
```

Explanation:

The ROM routine "SCROLL" is called the specified number of times, with the B register always containing the line number to scroll up to.

SIDEWAYS SCROLLING

Function:

***SLEFT**

The *SLEFT command is used to shift the screen display leftward by one character position. It is not necessary to move the whole display. Page 164 of the Spectrum BASIC manual explains how the screen can be thought of as being split into three sections: lines 0 to 7, lines 8 to 15 and lines 16 to 23. Any or all of these sections can be moved independently of the others, by selecting suitable parameter values. In addition, it is possible to specify the point at which shifting stops/starts: i.e. all characters to the left of and including the specified column will be shifted left by one character position. The character at the leftmost position will then reappear in this column. The parameter value is determined as follows:

The column number (0 to 31) at which shifting is to commence
+
32 if the bottom third is to be moved
+
64 if the middle third is to be moved
+
128 if the upper third is to be moved

Note that multiples of 32 will have no effect as the column number would be zero, and there are no characters to the left of column 0.

***SRIGHT**

Operation of this command is exactly the same as for *SLEFT except:

1. Movement is in the opposite direction (!)
2. The note regarding multiples of 32 no longer applies, since there are columns to the right of column 0. Note however that using 31 as the column number will produce no effect, for similar reasons.

Syntax:

*SLEFT n

*SRIGHT n

Parameters:

n (32 to 255) – see above.

Error Reports:

n<32 – Parameter error

n>255 – Integer out of range.

```
Screen Handling Commands - SLEFT,SRIGHT

SLFT      call EXPTNUM
     2    call SYNEND
     3    call STKTOA
     4    cp   32                 Signal error if bits 5-7 all zero.
     5    jp   c,PARERR
     6    ld   c,a

     7    and  31                 Leave if bits 0-4 all zero.
     8    jp   z,EXIT
     9    ld   a,c

    10    ld   bc,LMov            Specify routine to be used.
    11    bit  7,a
    12    jr   z,Lmid

    13    ld   hl,16384           Move upper third
    14    ld   de,22528           if required.
    15    call MOVE
Lmid      bit  6,a
    17    jr   z,Lbot

    18    ld   hl,18432           Move middle third
    19    ld   de,22784           if required.
    20    call MOVE
Lbot      bit  5,a
    22    jr   z,Lend

    23    ld   hl,20480           Move lower third
    24    ld   de,23040           if required.
    25    call MOVE
Lend      jp   EXIT

LMov      and  31                 Ignore bits 5-7.
    28    ld   c,a

    29    ld   a,(hl)             Store "end" byte.
    30    ld   d,h
    31    ld   e,l
    32    inc  hl

    33    ldir                    Shift row.
    34    ret
SLTCOM    defb 4
    36    defm "SLEFT
```



```
SRGT      call EXPTNUM
    38    call SYNEND
    39    call STKTOA
    40    cp   32                 Error if bits 5-7 all zero.
    41    jp   c,PARERR

    42    ld   bc,RMov            Specify routine to be called.
    43    bit  7,a
    44    jr   z,Rmid

    45    ld   hl,16415           Move upper third
    46    ld   de,22559           if required.
    47    call MOVE
Rmid      bit  6,a
    49    jr   z,Rbot

    50    ld   hl,18463           Move middle third
    51    ld   de,22815           if required.
    52    call MOVE
Rbot      bit  5,a
    54    jr   z,Rend

    55    ld   hl,20511           Move lower third
    56    ld   de,23071           if required.
    57    call MOVE
Rend      jp   EXIT

RMov      and  30
    60    ld   c,a
    61    ld   a,31               Calculate no. of
    62    sub  c                  columns to move.
    63    ld   c,a

    64    ld   a,(hl)             Store "end" byte.
    65    ld   d,h
    66    ld   e,l
    67    dec  hl

    68    lddr                    Shift line.
    69    ret
SRTCOM    defb 5
    71    defm "SRIGHT

MOVE      ld   (MV1+1),bc         Set routine to be used.
    73    push bc

    74    ld   b,8                No. rows per third.
MLP1      push hl
    76    ex   de,hl

    77    call MOV2               Move relevant attributes.
    78    ex   de,hl
    79    pop  hl
    80    push de
    81    push bc

    82    ld   b,8                No. pixel lines per row.

MLP2      call MOV2               Move lines.
    84    djnz MLP2
    85    pop  bc
    86    pop  de
    87    djnz MLP1
    88    pop  bc
    89    ret

MOV2      push bc
    91    push hl
    92    push af
    93    ld   b,0
MV1       call RMov               or LMov.

    95    ld   (de),a             Put 1st byte into other end of line.
    96    pop  af
    97    pop  hl
    98    ld   bc,32

    99    add  hl,bc              Point to next line.
   100    pop  bc
   101    ret


NOTE: Some Assemblers may not accept line 72,
      so this may be entered as:
                MOVE  ld (F4EE),bc
```

```
HEX DUMP:

      F429   CD   82   1C   CD   3E   EC   CD   94
      F431   1E   FE   20   DA   F5   EC   4F   E6
      F439   1F   CA   46   EC   79   01   6B   F4
      F441   CB   7F   28   09   21   00   40   11
      F449   00   58   CD   CB   F4   CB   77   28
      F451   09   21   00   48   11   00   59   CD
      F459   CB   F4   CB   6F   28   09   21   00
      F461   50   11   00   5A   CD   CB   F4   C3
      F469   46   EC   E6   1F   4F   7E   54   5D
      F471   23   ED   B0   C9   04   53   4C   45
      F479   46   54   CD   82   1C   CD   3E   EC
      F481   CD   94   1E   FE   20   DA   F5   EC
      F489   01   B6   F4   CB   7F   28   09   21
      F491   1F   40   11   1F   58   CD   CB   F4
      F499   CB   77   28   09   21   1F   48   11
      F4A1   1F   59   CD   CB   F4   CB   6F   28
      F4A9   09   21   1F   50   11   1F   5A   CD
      F4B1   CB   F4   C3   46   EC   E6   1E   4F
      F4B9   3E   1F   91   4F   7E   54   5D   28
      F4C1   ED   B8   C9   05   53   52   49   47
      F4C9   48   54   ED   43   EE   F4   C5   06
      F4D1   08   E5   EB   CD   E8   F4   EB   E1
      F4D9   D5   C5   06   08   CD   E8   F4   10
      F4E1   FB   C1   D1   10   EC   C1   C9   C5
      F4E9   E5   F5   06   00   CD   B6   F4   12
      F4F1   F1   E1   01   20   00   09   C1   C9
```

Explanation:

Both *SLEFT and *SRIGHT operate in a similar manner:

1. The relevant parameters are obtained and checked.
2. The upper three bits of the parameter are checked in sequence, each bit indicating whether the relevant section of the display should be moved (Bit 'ON') or not (Bit 'OFF'):

   Bit 7 – Upper third
   Bit 6 – Middle third
   Bit 5 – Lower third

   Each time a move is to be made, the HL and DE register pairs are loaded with the starting address of the relevant third of the Display and Attribute files respectively, and a call is made to MOVE.
3. A loop is then entered to repeat steps 4 and 5 eight times, once for each character row of the block to be moved.
4. The subroutine MOV2 is now called to shift the attribute bytes of each row.
5. Another loop is now entered to move all 8 pixel lines of each character row; again by calling MOV2.

Note that MOV2 itself will call either Rmov or Lmov, depending on which is supplied in the BC register pair on entry to MOVE. Both these subroutines are responsible for moving either one row of attributes, or one pixel line of display, in the relevant direction.

B.4.5 UTILITIES

FREE MEMORY

Function:

The command *FREE provides a quick estimate of the amount of unused memory available for BASIC. Note that all stacks and workspace used by the extension program are contained within the module itself. The size of the stacks will therefore not alter the value returned by *FREE.

The BYTES FREE message is printed from the last print position.

Syntax:

*FREE

Parameters:

None

Error Reports:

None

```
Utility Commands - FREE

FREE      call SYNEND
     2    call FREEMEM          bc=no.bytes used.

     3    ld   hl,0             (= 65,536)
     4    sbc  hl,bc            Calculate no.bytes free.
     5    ld   b,h
     6    ld   c,l

     7    call STACKBC          Stack in floating-point form.

     8    call GTPOS            Print from current
     9    call PRTAT            print position.

    10    call FPPRINT          Print number on stack.
    11    ld   hl,BYTFRE

    12    call PRTST            Print "BYTES FREE" message.
    13    jp   EXIT
BYTFRE    defb 11
    15    defm " BYTES FREE
FRECOM    defb 3
    17    defm "FREE

GTPOS     ld   bc,(SPOSN)       Get current print position.

    19    ld   hl,6177          h=24,l=33.

    20    sbc  hl,bc            Convert to normal row/column format.
    21    ld   b,h
    22    ld   a,l

    23    cp   32               Jump forward with columns 0-31.
    24    jr   c,ClmOK

    25    sub  a                Adjust for
    26    inc  b                column 32.
ClmOK     ld   c,a
    28    ret

HEX DUMP:

      F4F9   CD   3E   EC   CD   1A   1F   21   00
      F501   00   ED   42   44   4D   CD   2B   2D
      F509   CD   2D   F5   CD   AB   ED   CD   E3
      F511   2D   21   1B   F5   CD   BA   ED   C3
      F519   46   EC   0B   20   42   59   54   45
      F521   53   20   46   52   45   45   22   03
      F529   46   52   45   45
```

Explanation:

Although there is no "FREE" command in Spectrum BASIC, there is actually a ROM routine (called FREEMEM) which calculates how much memory has been used. After calling the routine, the number so obtained is subtracted from 65,536 to give the amount of free space available, and then stored on the calculator stack by STACKBC. Next the current print position is obtained using the common routine GTPOS from where the complete message is printed by calling first FPPRINT and then PRTSTG.

BLOCK DELETE

Function:

Deletes all lines between x and y (inclusive) from a BASIC program.

Syntax:

*DEL x,y

Parameters:

x = First line to be deleted (1 - 9999)
y = Last line to be deleted (1 - 9999)

Error Reports:

Integer out of range – x,y>9999

```
Utility Commands - DEL

DEL       call EXPT2NM
     2    call SYNEND
     3    call STKTOBC            Fetch last line into
     4    ld   h,b                hl and increment,so
     5    ld   l,c                last line specified
     6    inc  hl                 is also deleted.

     7    call LINADDR            Fetch address of specified line.
     8    push hl
     9    call STKTOBC

    10    ld   h,b                Fetch first line to
    11    ld   l,c                be deleted,and
    12    call LINADDR            obtain relevant address.
    13    ex   de,hl
    14    pop  hl

    15    call RECLAIM            Reclaim area inbetween.
    16    jp   EXIT
DELCOM    defb 2
    18    defm "DEL

HEX DUMP:

        F52D   ED   4B   88   5C   21   21   18   ED
        F535   42   44   7D   FE   20   38   02   97
        F53D   04   4F   C9   CD   7A   1C   CD   3E
        F545   EC   CD   99   1E   60   69   23   CD
        F54D   6E   19   E5   CD   99   1E   60   69
        F555   CD   6E   19   EB   E1   CD   E5   19
        F55D   C3   46   EC   02   44   45   4C
```

Explanation:

The two parameters are checked and fetched using EXPT2NM and STKTOBC. Both are converted to an address using the ROM routine LINADDR, and the space in between is then recovered using RECLAIM.

REM DELETE

Function:

REM statements can add considerably to program documentation, and are very helpful in debugging exercises. However, in some applications they may occupy memory space needed for programs or data. As a last resort in such situations the command *REMKILL can be used to delete all REM statements from a program.

Syntax:

*REMKILL

Parameters:

None

Error Reports:

None

```
Utility Commands - REMKILL

RKIL      call SYNEND
     2    ld   hl,(PROG)
Gtlin     ld   a,(hl)
     4    cp   40                 Leave if end of BASIC program
     5    jp   nc,EXIT            area has been reached.
     6    inc  hl

     7    inc  hl                 Point to length of line,
     8    ld   (WRKSPC),hl        and store for later use.
     9    inc  hl

    10    inc  hl                 Point to 1st character in line.
    11    ld   a,(hl)

    12    ld   de,33002           d=128,e=234("REM").

    13    cp   e                  Is 1st CODE a REM?

    14    jr   z,KILLN            Jump forward if it is.

    15    call SCHSTMT            Check remaining statements in line.

    16    jr   nc,KILST           Jump forward if REM found.

    17    inc  hl                 Move on to consider
    18    jr   Gtlin              next line.

KILST     ld   d,h
    20    ld   e,l
    21    dec  de
    22    ld   a,13               ("ENTER")
KLP1      inc  hl                 Enter loop to
    24    cp   (hl)               locate end of line.
    25    jr   nz,KLP1

    26    call DIFRNCE            Obtain length from REM.
    27    push bc

    28    call RECLAM2            Reclaim this area.
    29    pop  bc
    30    ld   hl,(WRKSPC)

    31    ld   e,(hl)             Fetch length of
    32    inc  hl                 line BEFORE deletion
    33    ld   d,(hl)             took place.
    34    ex   de,hl

    35    sbc  hl,bc              Subtract amount reclaimed.
    36    ex   de,hl

    37    ld   (hl),d             Store new length of line.
    38    dec  hl
    39    ld   (hl),e
    40    dec  hl
    41    dec  hl
```

```
     42    call NEXTONE              Point to next line.
     43    ex   de,hl

     44    jr   Gtlin                Loop back and repeat process.

KILLN     dec  hl
     46    ld   b,(hl)               Fetch length of
     47    dec  hl                   text + newline.
     48    ld   c,(hl)

     49    inc  bc                   Add 2 for bytes
     50    inc  bc                   holding line no.

     51    inc  bc                   And 2 more for bytes
     52    inc  bc                   holding length.
     53    dec  hl

     54    dec  hl                   Point to start of line.

     55    call RECLAM2              Delete whole line.

     56    jr   Gtlin                Loop back to consider next line.
RKLCOM    defb 6
     58    defm "REMKILL
```

HEX DUMP:

```
F564   CD  3E  EC  2A  53  5C  7E  FE
F56C   28  D2  46  EC  23  23  22  95
F574   ED  23  23  7E  11  EA  80  BB
F57C   28  2E  CD  8B  19  30  03  23
F584   18  E4  54  5D  1B  3E  0D  23
F58C   BE  20  FC  CD  DD  19  C5  CD
F594   E8  19  C1  2A  95  ED  5E  23
F59C   56  EB  ED  42  EB  72  2B  73
F5A4   2B  2B  CD  B8  19  EB  18  BE
F5AC   2B  46  2B  4E  03  03  03  03
F5B4   2B  2B  CD  E8  19  18  AF  06
F5BC   52  45  4D  4B  49  4C  4C
```

Explanation:

A loop is entered to deal with each BASIC line in turn.

If the first character is a REM instruction, then the whole line must be deleted using KILLN. Otherwise the line is checked, statement by statement, for any further REM's – if one is found, then the rest of the line is deleted.

Note that RECLAIM cannot be used because the amount of memory reclaimed is required in subsequent calculations.The action is therefore performed by calling DIFRNCE, saving the register and calling RECLAM2.

LINE RENUMBER

Function:

It can be useful on occasions to renumber parts of a BASIC program, either to 'tidy-up' the listing or to make room for more lines. This is easily achieved with the *RENUM command. Such a short routine cannot, of course, renumber GOTO's and GOSUB's, but with all the structured commands now provided these instructions are rendered virtually obsolete anyway.

Note that this command only alters the line numbers, it does not affect their position in memory.

Syntax:

*RENUM x1,y1,x2,n

Parameters:

x1 = first line number of block to be renumbered
y1 = last line number of block to be renumbered
x2 = first new line number
n = increments between lines in new block.

Error Reports:

Integer out of range x1,y1,x2>9999
n>255

```
Utility Commands - RENUM

RNUM      call EXPT2NM
     2    cp   44                   (Comma)
     3    jp   nz,ERROR
     4    call NEXT2NM
     5    call SYNEND
     6    call STKTOA
     7    ld   (WRKSPC+4),a         Store increment.
     8    call STKTOBC

     9    ld   (WRKSPC+2),bc        Store 1st new line no.
    10    call STKTOBC
    11    inc  bc

    12    ld   (WRKSPC),bc          Store last old line no. +1.
    13    call STKTOBC
    14    ld   h,b
    15    ld   l,c

    16    call LINADDR              Fetch address of 1st old line no.
Rnext     ld   de,(WRKSPC)
    18    ld   b,d
    19    ld   c,e

    20    call CPLINES              Leave when last
    21    jp   nc,EXIT              old line no. reached.
    22    ld   de,(WRKSPC+2)

    23    ld   (hl),d               Enter new line no.
    24    inc  hl
    25    ld   (hl),e
    26    ex   de,hl
    27    ld   a,(WRKSPC+4)
    28    ld   c,a
    29    ld   b,0

    30    add  hl,bc                Derive next line no. to be entered.
    31    ld   (WRKSPC+2),hl
    32    ex   de,hl
    33    dec  hl

    34    call NEXTONE              Point to next line.
    35    ex   de,hl
    36    jr   Rnext
RNMCOM    defb 4
    38    defm "RENUM
```

HEX DUMP:

```
F5C3   CD  7A  1C  FE  2C  C2  51  EC
F5CB   CD  79  1C  CD  3E  EC  CD  94
F5D3   1E  32  99  ED  CD  99  1E  ED
F5DB   43  97  ED  CD  99  1E  03  ED
F5E3   43  95  ED  CD  99  1E  60  69
F5EB   CD  6E  19  ED  5B  95  ED  42
F5F3   4B  CD  80  19  D2  46  EC  ED
F5FB   5B  97  ED  72  23  73  EB  3A
F603   99  ED  4F  06  00  09  22  97
F60B   ED  EB  2B  CD  B8  19  EB  18
F613   DA  04  52  45  4E  55
```

Explanation:

EXPT2NM and NEXT2NM are used to load-in the four parameters during syntax-time. These are then stored in the workspace before using LINADDR to calculate the address of the first line to be renumbered. The loop at Rnext is then entered to renumber each line in turn, until the last line specified, or the end of the BASIC program area, is reached.

B.5 ADDING YOUR OWN COMMANDS

You should by now have a thorough understanding of the way in which the extension program works, and be in a position to start thinking about adding your own commands. A good Assembler will be an invaluable asset here, and a disassembly of the Spectrum ROM would also be useful.

Remember the basic format of any command routine is always the same:

1. Check syntax of any parameters using EXPTNUM etc. as applicable.
2. Exit during syntax by calling SYNEND.
3. Fetch any parameters, and check them for errors.
4. Execute the routine accordingly.
5. Make a jump to EXIT.

Remember that errors detected in syntax-time go straight to ERROR. Run-time errors should cause a jump to the relevant "Extended Error" entry point.

The Spectrum ROM contains a wealth of useful routines, and these should be used wherever practical to reduce both the size and effort of your new command routines.

Unless you are already fluent with machine code you would do well to keep your routines simple, and similar in form to those already described. Try something like *CAPS to start with, perhaps a command to suppress the "scroll?" message (see page 175 of the Sinclair BASIC manual for details of the relevant System Variable).

Once your command routine has been written, it must be 'patched-in" to the rest of the extension program:

Firstly check through the Vector Table to establish whether or not a Secondary Parser already exists for the initial letter of your new command. If there is not one listed, then you will need to write a new one, not forgetting to enter the start address into the relevant position in the Vector Table.

Assuming you are using an Assembler, extending an existing Secondary Parser to include your new command is simply a matter of adding three extra lines immediately before the "JP ERROR" instruction at the end. The entire program can then be re-assembled, taking no more than a few seconds on most assemblers.

If you do not have access to an assembler however, all is not lost. What you must do is to set up a second Secondary Parser, and alter the relevant entry in the Vector Table to point to this. Then, instead of using "JP ERROR" as the last instruction, you should jump back to the start of the original Secondary Parser. The relevant address can be obtained from the Vector Table before it is overwritten by the address of the second Secondary Parser.

It is hoped that the above advice, together with the structural design of the extension program itself, will enable the reader to progress further into this fascinating subject than the scope of this book allows.

Other products for the ZX Spectrum/ZX Spectrum+ from Hewson Consultants:

CASSETTES

	Price incl. VAT and p & p
Utilities	
Machine Code Extensions for Spectrum Basic All the Basic programs and machine code routines in this book.	£3.95
ZAPP (Z80 Assembly Programming Package) A comprehensive machine code assembler, disassembler, editor, monitor, hex dump and file handler. Assembles from memory or file. Available on cassette for cassette usage, transfer to Sinclair microdrive or transfer to Rotronics Wafer (please specify).	£19.95
Adventure Movies	
Avalon The original 3D adventure movie featuring Maroc the Mage.	£7.95
The Dragontorc of Avalon Maroc leaves Avalon to retrieve the Dragontorc.	price to be announced
Platform Games	
Technician Ted Explore the microchip factory.	£5.95
The Seiddab Triology	
3D Lunattack	£7.95
3D Seiddab Attack	£5.95
3D Space Wars	£5.95
Graphic Adventures	
Fantasia Diamond	£7.95
Quest	£5.95
Aviation Simulators	
Heathrow International	£7.95
Nightflite II	£7.95

BOOKS

20 Best Programs for the ZX Spectrum	£5.95
40 Best Machine Code Routines for the ZX Spectrum	£5.95

Machine Code Extensions

for

Spectrum Basic

ORDER FORM

Please supply the following items (tick box as required)

	£
☐ Machine Code Extensions for Spectrum Basic	.....
☐ ZAPP – Cassette version	.....
☐ ZAPP – Microdrive version	.....
☐ ZAPP – Waferdrive version	.....
☐ Avalon	.....
☐ The Dragontorc of Avalon	.....
☐ Technician Ted	.....
☐ 3D Lunattack	.....
☐ 3D Seiddab Attack	.....
☐ 3D Space Wars	.....
☐ Fantasia Diamond	.....
☐ Quest	.....
☐ Heathrow International	.....
☐ Nightflite II	.....
☐ 20 Best Programs for the ZX Spectrum (Book)	.....
☐ 40 Best Machine Code Routines for the ZX Spectrum (Book)	.....
	____
Total	____

☐ I enclose a cheque for the total amount **or**
☐ Please debit my Access/Barclaycard account

Number ..... Signed..... Date .....

Name .....

Address .....

.....

.....

Please complete the questionnaire overleaf and despatch to: Hewson Consultants Ltd., 56B Milton Trading Estate, Milton, Abingdon, Oxon OX14 4RX, England. Tel: 0235 832939.

Machine Code Extensions

for

Spectrum Basic

QUESTIONNAIRE

Where did you learn about this book?

Saw an advertisement ☐
Read a magazine article ☐
Recommended by a friend ☐
Saw it in a shop ☐
Other (please specify) .....

Which computer magazines do you read?

Regularly .....

Occasionally .....

How good is this book?

Excellent ☐
Reasonable ☐
Poor (please give reasons) ☐

.....

Is the price right?

Good value for money ☐
About right ☐
Over priced ☐

Any other comments?

Thank you for helping us in this survey.

Hewson Consultants Ltd.

Machine Code Extensions for Spectrum Basic

Now you can find out from the experts how to add new commands to Spectrum Basic to make your programs faster, more versatile, more concise and better structured. No extra hardware "add-ons" are required.

New commands include

***VPRINT**

Now you can PRINT vertically down the screen – ideal for tables, columns of figures and building better pictures.

***PROC, *DEFPROC, *ENDPROC**

Structure your Basic programs by defining separate PROCEDURES as used in the most sophisticated programming languages.

***IF, *ELSE**

Streamline your program alternatives and avoid those hard-to-follow GOTO statements.

***SLEFT, *SRIGHT**

Create a display window on your screen and scroll it independently of the rest of the display.

– and many more.

The book features step-by-step instructions on how to implement new commands from the simplest to the most sophisticated. Extensive use is made of the routines in the Spectrum ROM thus ensuring that the new commands are compatible with the ordinary commands and that the machine code involved is very compact so that plenty of RAM remains for your Basic programs.

With this book you can learn all you need to create your own commands – you could even invent your own programming language.

Machine Code Extensions for Spectrum Basic was especially written by Rob Banks for Hewson Consultants who are leading experts on the Sinclair ZX Spectrum and publishers of many home computer programs and computer books.

This book is the third in a series on the ZX Spectrum including 40 Best Machine Code Routines for the ZX Spectrum which was voted 1983 Computer Book of the Year.

£6.95