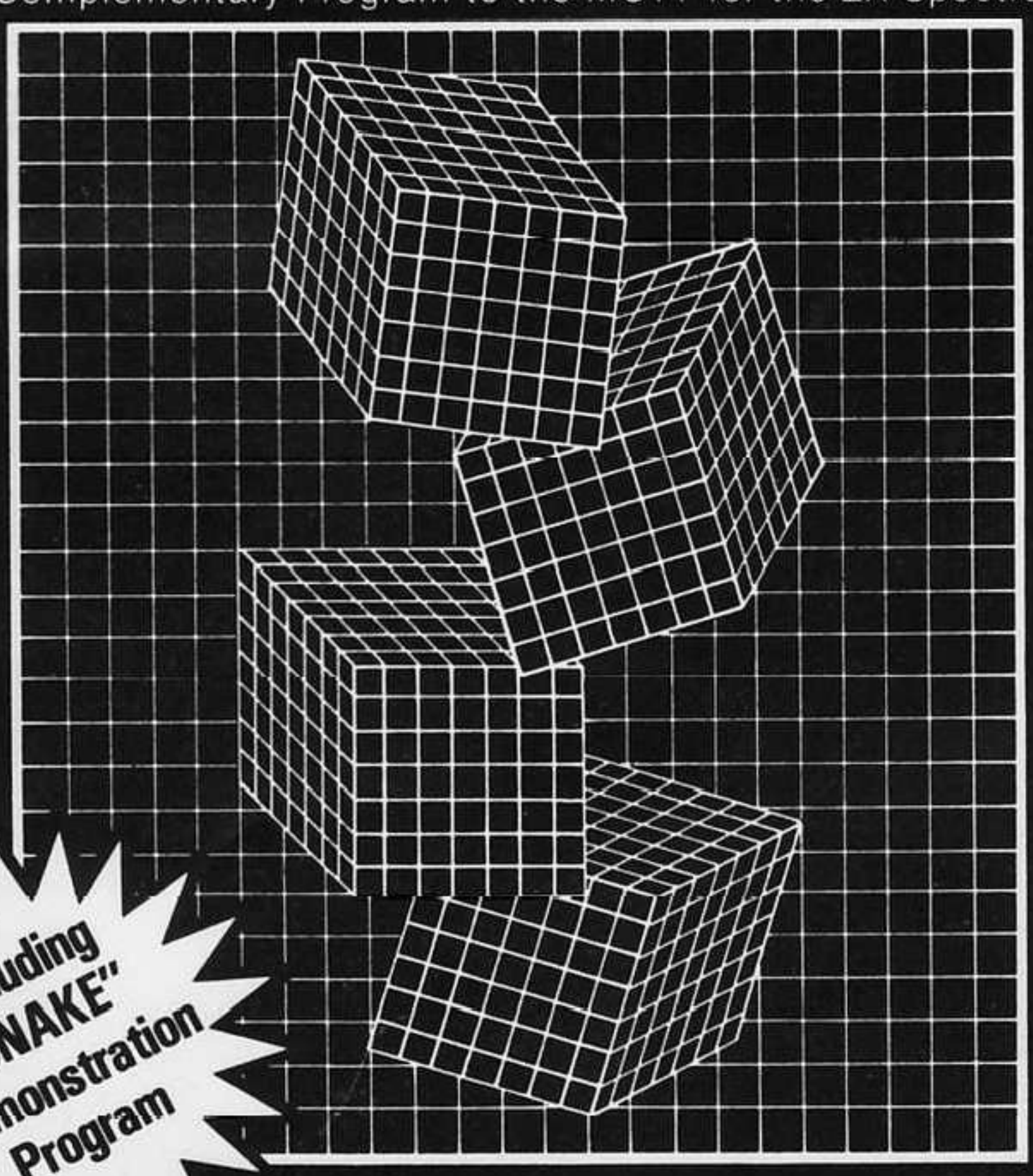
FULL SCREEN EDITOR/ASSEMBLER

(A Complementary Program to the MCTT for the ZX Spectrum)



Including "SNAKE" Demonstration Program

ZX Spectrum 16k & 48k Edition

FULL SCREEN EDITOR/ASSEMBLER

FOR THE SINCLAIR ZX SPECTRUM 16K AND 48K

OXFORD COMPUTER PUBLISHING LIMITED 1983
P.O. BOX 99, OXFORD, ENGLAND

WRITTEN BY JAMES HUTCHBY

TABLE OF CONTENTS

1. INTRODUCTION

Congratulations on your purchase of Oxford Computer Publishing Ltd.'s Editor/Assembler program for your Sinclair Spectrum. This is one of the most powerful machine code programming tools available for the ZX range of computers. With our Machine Code Test Tool (MCTT) program (available separately) you have a complete machine code development environment.

The Editor/Assembler is a package that operates in two distinct parts. The first part, the editor, allows you to enter and edit your machine coded programs (source code) in mnemonic form with data, branch addresses etc. represented by symbols. The second part, the assembler, converts this source code into object code which is the form understood by the Z-80 microprocessor. Object code can be displayed on the screen, saved to cassette tape, stored in memory and a hardcopy taken on the ZX printer, all under your control.

MCTT allows you to debug programs prepared with the Editor/Assembler. Corrections can be made to the source code, the source code re-assembled and the object code checked out until it works properly. The time taken to switch between the Editor/Assembler and MCTT is only a few seconds so this is a very effective combination of tools for program development.

2. LOADING

Connect your Spectrum to a television, a power supply, a cassette player and a ZX printer (if you own one) and switch on. Disconnect the MIC lead from the cassette player and insert the Editor/Assembler tape. Type LOAD "", press ENTER and start the cassette player running. The program will load in three parts and when the loading is complete it will auto-run.

There are two versions of the Editor/Assembler on the supplied cassette tape. One is for 48K Spectrums and the other for 16K Spectrums. The 48K version is complete — both the editor and assembler parts of the package are resident in memory at the same time. The 16K version, however, is divided into two — the editor and the assembler are separate programs. This is necessary because the complete Editor/Assembler is too large a program to be entirely contained within the limited memory of a 16K Spectrum. Source code is entered with the editor program, and the assembler program is then loaded so as to overlay the editor but leave the text buffer untouched. The source code can then be assembled into object code and, if necessary, the editor re-loaded to make corrections to the source (again preserving the contents of the text buffer) and the process repeated. (See APPENDIX 2 for further details).

Note that the 16K version can be loaded into a 48K Spectrum if you wish to assemble very large programs and need a few extra 'K' of memory.

If you own a copy of Machine Code Test Tool it must be loaded into your computer BEFORE the Editor/Assembler. Follow the instructions in the former's manual and then load the latter as detailed above. The Editor/Assembler will recognise the presence of MCTT and protect it from being overwritten. This only applies to owners of 48K Spectrums.

3. FULL SCREEN TEXT EDITOR

3.1 Introduction

The editor half of O.C.P.'s Editor/Assembler package is a powerful tool that allows you to enter, examine and modify lines of source code. At your disposal are commands to change, insert and delete individual characters within a line; to move, copy and delete single lines or blocks of lines as a whole; to locate, change and delete specified strings of characters as well as commands to output text to the ZX printer and to cassette tape — indeed many of the facilities previously only offered by sophisticated word processors.

3.2 The keyboard

If you are used to programming in BASIC you will find the organisation of the keyboard somewhat simplified. There are no 'keywords', neither are there 'graphics' nor 'extended' modes. A 'caps lock' however is included. The lock can be turned on at any time by pressing the CAPS LOCK key (CAPS SHIFT'ed 2) and an inverse C will immediately appear in the top right hand corner of the screen to remind you of this. The lock can be turned off by pressing the same key again and the inverse C will disappear.

If the caps lock is off:—
unshifted keys give lower case letters and numbers; CAPS SHIFT'ed key upper case (capital) letters and 'cursor movement and control functions' (more about these later).

If the caps lock is on:—
unshifted keys give upper case letters and 'cursor movement and control functions';
CAPS SHIFT'ed keys give lower case letters and numbers.

Use of the SYMBOL SHIFT with a key gives the subsidiary red SINGLE-CHARACTER legend on the key (irrespective of caps lock status). For example SYMBOL SHIFT'ed N gives a comma and SYMBOL SHIFT'ed D gives a backslash. Note the keys which aren't marked with a single red character (specifically Q,W,E,Y,U,I,A,S,D,F and G) all give the copyright symbol since this character isn't available from the P key.

All keys automatically repeat if held down.

3.3 Upper and lower case

As a general rule all editor commands can be entered in either upper or lower case. In fact, this equivalence of the two cases has been adopted as a convention throughout the entire Editor/Assembler package — line commands, extended commands, opcode mnemonics, operand mnemonics, symbol names etc. can all be in either upper or lower case or, for that matter, a mixture of upper and lower case.

The few exceptions to this rule are (i) the string specified in a (C)hange string extended command (see below) remains as entered, and (ii) the names given to cassette files retain their case specific identity.

3.4 The screen display

The area of RAM memory which contains your entered lines of source code is known as the 'text buffer' and the television screen acts as a 'window' onto this

buffer. Obviously only 24 lines can be displayed at a time but certain keys enable you to alter the relative position of the window in the buffer so as to bring undisplayed lines into view. This process is known as 'scrolling'.

When the Editor/Assembler is first loaded into your computer the text buffer contains a single empty line with line number 00000. At this stage in your familiarisation with the program it is recommended that you load into your machine the demonstration file also contained on the supplied cassette tape. This will provide you with ample material on which to practise the various editor commands and so enable you to become quickly proficient in their use. To do this, complete the following procedure (don't worry about what's happening — all will be explained in due course).

(i) Press both the CAPS SHIFT and the SYMBOL SHIFT keys simultaneously.
(ii) Press the SPACE key.
(iii) Type in a single L and press ENTER.
(iv) Start the cassette player. The demonstration file is recorded after copies of the Editor/Assembler itself on both sides of the tape but the editor will read past these until it finds what it's looking for. When the loading is complete the first page (24 lines) of the demonstration program will be displayed and you can proceed with this manual.

Individual source code lines are divided into a number of segments or 'fields' — specifically a label, an opcode and an operand field. The use of fields leads to a neat, ordered and highly readable display. (Incidently the division into fields will be handled conveniently and automatically for you by the editor as you type in new text.)

Note however that comment lines remain exactly as you enter them. A comment line is any line which contains a semicolon (;) in the first column of its label field. They are included purely as an aid to documentation (like BASIC REM statements) and are ignored during assembly. A comment line can be up to 32 characters in length (including the semicolon).

Associated with each line in the text buffer is a 5-digit line number (in the range 00000 to 99999) which appears to the left of the label field. It is important to realise that these line numbers hold none of the significance of the line numbers in a BASIC program. They can occur in any order and need not even be unique. Their principal use is in tracing errors that occur at assembly time.

As a point of interest the Editor/Assembler writes to the television screen using special techniques to display 42 characters per line as opposed to the 32 per line normally available from BASIC. The attributes of white ink on blue paper have been chosen as being especially appealing to the eye but they can be altered if so desired (see BASIC extended command below).

3.5 The cursor

Find on the screen a single blinking character (it will normally be the first character of the first line). This is known as the 'cursor' and its blinking tells you that you're in, what is referred to as, 'normal edit mode'. In this mode the cursor can be moved about the screen by pressing the keys with arrows above them (CAPS SHIFT'ed 5 to 8). As you would expect the key with an arrow pointing to the left moves the cursor to the left, that pointing upwards moves the cursor up and so on. Try these now. The cursor's motion is said to be non-destructive in

the sense that as it passes over existing characters on the screen it doesn't change them.

Experiment with moving the cursor to the borders of the display. Notice how it stops short of the line number at the left hand margin. Try now to move the cursor off the bottom of the screen. See it marching downwards until it reaches the last line and finds it can go no further. Note how the whole display is then scrolled up to bring the 'next' line into view (and the previous top line is lost from sight). Hold down the key and watch it repeating. The scrolling is continued until the last line in the text buffer is reached.

Naturally you'll scroll the display in the other direction if you attempt to move the cursor off the top line of the screen. Again scrolling is continued until the first line in the buffer is reached when it ceases. Try this now.

It's also possible to scroll the display in either direction by a whole 24 lines (called a 'page') at a time. This allows you to rapidly scan through a large amount of text to find a particular line. Press the TRUE VIDEO key (CAPS SHIFT'ed 3) to page upwards and INV VIDEO (CAPS SHIFT'ed 4) to page downwards. Again the functions repeat (as indeed do all of the above) if the keys are held down.

The ENTER key also has some control over the cursor. Unshifted it advances it to the next tab stop (i.e. the start of the next field). CAPS SHIFT'ed it moves to the beginning of the next line.

Now that you've learnt how to move the blinking cursor to any character in any line in the text buffer you can be taught how to actually alter displayed text.

3.6 Deleting character(s)

Move the blinking cursor to the character you wish to delete. Press the DELETE key (CAPS SHIFT'ed 0) and the selected character is removed. Notice how the rest of the line is shifted one space to the left to fill the gap (and a space appears in at the right). Hold down the DELETE key and watch it repeating.

3.7 Changing character(s)

This is simplicity itself. Move the blinking cursor to the character you wish to change. Just type the new character and the display immediately reflects the alteration. Notice how the cursor is advanced one space to the right of the changed character (unless of course it is already hard up against the right hand margin). So you can easily type in whole strings of corrections.

3.8 Inserting character(s)

It's also possible to insert new characters into existing lines. To do this move the blinking cursor to the desired point of insertion and press the GRAPHICS key (CAPS SHIFT'ed 9). The editor goes into insert mode and the cursor stops blinking. Space is opened up in the line immediately preceding the character under the cursor whilst still blinking. As you type in new text the rest of the line is shifted to the right (the rightmost character being lost in the process).

You can use the ◄ and DELETE keys to backspace and correct mistakes in this mode and the ► key to insert spaces up to the next tab position (note that this is different from its function in normal edit mode where it moves the blinking cursor just one space).

When you've completed your insertion of new characters press ENTER. The line is closed up and the cursor starts blinking again indicating you're back in normal edit mode.

3.9 Division into fields

You may have noticed how the editor automatically tabulates your text into neat columns when you move the blinking cursor away from a line. Fields need only be separated by a space when you type them in — the editor will do the rest. Note that you are allowed up to 6 characters in the label field, up to 4 characters in the opcode field and up to 20 characters in the operand field.

None of this applies to comment lines (a semicolon in the first column) which remain exactly as you enter them.

The EDIT key (CAPS SHIFT'ed 1) can be used to cancel any changes made to a line and to place the blinking cursor back on the first character. You are not committed to any alterations you may make to a line until you move the blinking cursor off the line (or more strictly attempt to move the cursor off) or enter a line command or extended command. Up till then you can press the EDIT key at any time to restore the line to its original state.

This now completes the description of the most basic of the editor's capabilities. Before proceeding with the sections on line commands and extended commands, it is suggested that you practise the above techniques until they become entirely familiar.

3.10 LINE COMMANDS

Line commands act upon a single line or block of lines as a whole rather than upon specific characters within a line. To select a line command move the blinking cursor to (anywhere on) the line that the command is to apply to and press both shift keys simultaneously. The cursor will disappear and the chosen line will be highlighted by the appearance of a small square to the right of the line number. This will indicate that you're in line command mode and remind you of the line you've selected. The next key pressed will determine the function to be performed.

Abort command — CAPS SHIFT + SYMBOL SHIFT

Press both shift keys again to exit immediately from line command mode (i.e. if you change your mind). The line will be un-highlighted and the blinking cursor restored to its original position.

Delete line — D

Press D and the highlighted line will be deleted from the text buffer. Lines below will be scrolled upwards to fill the gap and you'll be returned to normal edit mode.

Insert line(s) — I

Press I and a new line (with line number 00000) will be inserted into the text buffer immediately following the highlighted line. This new line will initially be empty but a non-blinking cursor will be placed in the first column. Thereafter you can type in text as you desire with the ▶ key acting as a tab to the start of the next field and the ◀ and DELETE keys acting as backspaces to correct mistakes.

If you press ENTER at this stage the above process will be repeated; another new line will be created following the one you've just entered and again a non-blinking cursor will be placed in its first column.

When you've typed in the last line you wish to insert press both shift keys simultaneously (rather that ENTER). The cursor will start blinking again showing you are back in normal edit mode.

Copy line — C

Press C and the chosen line will be marked for copying. The blinking cursor will return to its original position but the highlighting will remain while the copy is pending.

Move line — M

Press M and the chosen line will be marked for moving. The blinking cursor will return to its original position but the highlighting will remain while the move is pending.

Move/copy to here — H

After you've marked a line for moving or copying you'll be back in normal edit mode. You can move the blinking cursor anywhere in the text buffer and perform all the usual edit functions except mark another line. To complete the move or copy operation press H when in line command mode. The marked line will be moved or copied to the point immediately following the line just selected.

Whenever a move or copy is pending a small square will appear in the top left hand corner of the screen once the display is scrolled. This will serve as a reminder. The pending operation can, however, be cancelled at any time by pressing ENTER when in line command mode (see below).

Note that a line cannot be moved or copied directly to the beginning of the text buffer. It must instead be moved or copied to immediately after the first line and then the first line moved around it.

Set block marker — B

A block of lines can be deleted, moved or copied as easily as a single line. The procedures are analogous to those just described except that a block of lines is selected instead of just one.

To select a block move the blinking cursor to either the first or the last line in the block and enter line command mode by pressing both shift keys simultaneously. Then press B and the line will be marked as a block terminator. The next line command (delete, move or copy) you enter will apply to the entire block rather that just the line you actually enter the line command on. Note that a block is defined as all the lines between and including the line that you marked as a block terminator and the line that you subsequently select for delete, move or copy.

If you select a block move or copy you must complete the operation by moving the blinking cursor to some other point in the text buffer and pressing H when in line command mode. The entire block will then be moved or copied to the point following the line just selected.

Naturally this line must be outside the range of the original block or an "INVALID REQUEST" error will be signalled. Note also that although the (B)lock line command can be used to mark either the first or the last line in a block it must be applied before the complementary (D)elete, (M)ove or (C)opy line

command. Finally note that lines or blocks of lines that are moved or copied always retain their original line numbers.

Remove block markers — ENTER

Press ENTER and any pending move or copy operation will be cancelled, any block marker deleted from the text buffer and all highlighting removed from the screen.

Display from top — T

Press T and the first page (24 lines) of the text buffer will be displayed with the blinking cursor positioned on the first character of the first line.

Display end — E

Press E and just the last line in the text buffer will be displayed.

Repeat find — F

Press F and the previously entered find string command will be repeated (see (F)ind extended command below).

Invoke extended command processor — SPACE

Press SPACE and extended command mode will be entered (see below).

The last five line commands are slightly different in that they don't act upon the line the blinking cursor is on when the command is selected. Consequently they can be used at any time.

3.11 EXTENDED COMMAND PROCESSOR

As mentioned briefly above the key sequence CAPS SHIFT + SYMBOL SHIFT followed by SPACE invokes what is known as the 'extended command processor'. Its function will now be discussed in detail.

In extended command mode the bottom two lines of the screen are cleared and a short prompt message followed by a non-blinking cursor is displayed. The editor waits for you to type in an extended command line, which consists of a command verb (in most cases just a single letter) followed by optional arguments. The line can be up to 23 characters in length and, as usual, the ◄ and DELETE keys backspace to correct mistakes. Pressing ENTER then instructs the computer to execute your chosen command. Note that pressing ENTER on an empty line (no spaces entered) aborts the extended command processor and returns you immediately to normal edit mode.

Renumber — Nstart, inc

Renumbers all the lines in the text buffer. The first line is given the line number 'start', the second line is given the number 'start + inc', the third 'start + inc + inc' and so on. If either of the two numbers following the N is omitted the editor uses the most recently specified value as a default. Both these values are set to 10 when the Editor/Assembler is first loaded into your computer.

e.g.	N	renumbers the lines 10,20,30,40,...
	N100	renumbers the lines 100,110,120,130,...
	N,50	renumbers the lines 100,150,200,250,...
	N10,100	renumbers the lines 10,110,210,310,...

It is good practice to renumber the text buffer prior to an assembly. Any errors that occur can then be easily traced and corrected by noting the relevant line numbers.

Go to line — Gnumber

Searches for a line with the specified line number. The search begins with the line currently at the top of the display (NOT the first line in the buffer) and continues to the end of the text buffer. If the line is found the window into the buffer is scrolled so as to bring that line into view at the top of the screen. If the line isn't found then the message "NOT FOUND" is displayed briefly and no scrolling takes place.

Find character string — Fstring

Searches the text buffer for occurences of the specified string. The search begins with the line currently at the top of the display (NOT the first line in the buffer) and continues to the end of the text buffer. If the character string is found then the display is scrolled so as to bring the line containing the string to the top of the screen. If not the message "NOT FOUND" is displayed briefly and no scrolling takes place. Note that no quotes (") are necessary around the string and that all characters following the F (up to 20) are significant.

The repeat (F)ind line command is a variant on the above. It uses the last string specified by a (F)ind extended command but begins its search from the SECOND line on the screen. Use of the two commands in conjunction will find all the occurences of a specified string in a piece of text.

The (F)ind commands are not case specific. That is to say that if you type in, for instance, Fnop (or FNOP or FnOp for that matter) occurences of NOP, nop, NoP etc. will all be located.

Change character string — Cstring

This command searches the text buffer for ALL occurences of the string specified in a previously entered (F)ind command and changes EACH of them to the new character string specified in this (C)hange command. The search begins with the line currently at the top of the display (NOT the first line in the buffer) and continues to the end of the text buffer. When the command is completed you are returned to normal edit mode and the display is not scrolled.

Again no quotes (") are necessary around the change string and all characters following the C (up to 20) are significant.

If the change string is a null string (i.e. C on its own is entered) then all occurences of the find string will be deleted from the text buffer.

Display free memory — MEM

Displays on the bottom line of the screen the amount of free memory available to the editor IN DECIMAL followed by (in brackets) the start address of the free memory area IN HEXADECIMAL. This second value is useful as it is effectively the transfer address for 'into memory' assembled programs. The free memory includes any memory currently occupied by the symbol table since this is available to the text buffer. Press any key to return to normal edit mode.

Return to BASIC — BASIC

Returns the user to the Spectrum's BASIC interpreter. All the usual BASIC statements can then be executed, programs run etc. The screen colours can be changed (via BORDER, INK AND PAPER commands) if so desired.

To re-start the Editor/Assembler type in either 'RANDOMIZE USR cold ENTER' or 'RANDOMIZE USR warm ENTER'. The 'cold start' will clear the text buffer, but the 'warm start' will preserve its original contents.

Clear buffer — CLEAR

Clears the text buffer and inserts a single empty line with line number 00000 into it (equivalent to a cold start). Also marks any symbols in the symbol table as 'delete protected' (see 'old symbol table' switch below).

Call monitor — MCTT

This command only applies to the 48k version of the Editor/Assembler. It calls O.C.P. 's Machine Code Test Tool program if resident in memory. It signals an "INVALID REQUEST" error however if MCTT isn't present. Note that MCTT must be loaded into your computer BEFORE the Editor/Assembler. When the latter is loaded it recognises the presence of the monitor and protects it from being overwritten. The MCTT S command (see MCTT Manual) returns control to the Editor/Assembler preserving the contents of the text buffer.

This combination of tools provides the programmer with a sophisticated machine code development environment. If you don't yet possess a copy of 'Machine Code Test Tool' it is highly recommended that you make the investment.

Note for MCTT users

Instead of setting breakpoints with the MCTT B command, the instruction CALL 0F8C6H may be assembled directly into your source program. To test your program just use the Editor/Assembler RUN command instead of the MCTT command, and the MCTT breakpoint routine will be activated as soon as the above call 0F8C6H instruction is executed.

Of course, MCTT must first of all have been loaded and called at least once explicitly using the MCTT command from the Editor/Assembler for this to be effective.

Call user program — RUN

Executes a program in the object code buffer (see assembler chapter). In effect the editor branches to the first byte in the buffer. No check is made that sensible information exists there, so care must be exercised in using this command. Control is returned to the editor by a RET instruction in the user program.

It is suggested that this command is only used IMMEDIATELY following 'into memory' assemblies as other editing operations could corrupt the object code buffer.

Print — Pnumber

Prints a number of lines in the text buffer on the ZX printer. Printing begins with the line currently at the top of the display (NOT the first line in the buffer) and continues downwards. If 'number' is specified then that number of lines is printed. If 'number' is omitted or is zero then the rest of the buffer is printed. The

BREAK key (CAPS SHIFT'ed SPACE) will return you to normal edit mode at any time.

Save to tape — Sfilename

Saves the contents of the text buffer to cassette tape with the name 'filename'. Note that no quotes ('') are necessary around the name and that every character following the S is significant. That is to say that if you type in a space between the S and the first character it will be included in the filename. Note also that the filename is limited by the normal Spectrum conventions to 10 characters in length — any further characters will bc ignored. If no filename is specified, the default name 'SourceCode' is used.

The usual "Start tape and press any key" message is displayed. Proceed as you would for an ordinary BASIC save. The BREAK key (CAPS SHIFT'ed SPACE) will abort the operation at any time. On completion the bottom two lines of the screen are restored and you're returned to normal edit mode. Stop the tape.

Load from tape — Lfilename

Loads into the text buffer files from cassette tape previously created by the (S)ave extended command and in the process overwrites the existing buffer contents. Only files made by the (S)ave command are able to be loaded as they're specially 'watermarked' when being recorded to distinguish them from other file types. If a filename is specified (using the same format as above) the editor searches until it finds a file with a name that matches and then loads it. If no filename is specified the editor loads the first file if finds.

If the file is larger than the available memory the message "TOO BIG" is displayed on the bottom line of the screen and no loading takes place (the original contents of the buffer are preserved). In this event press any key to return to normal edit mode.

If an error occurs during a load (i.e. the checksum fails) the message "TAPE ERROR" is displayed on the bottom line of the screen. Again press any key to return to normal edit mode. Note that if a tape error occurs or the BREAK key is pressed to abort the operation the text buffer is left in whatever state it happens to be in. However the CLEAR extended command will always enable you to recover from a severely corrupted buffer.

After a successful load the first page of the new buffer contents is displayed and you're returned to normal edit mode.

Append from tape — Xfilename

Very similar in operation to the (L)oad command described above but with the distinction that files read from tape are appended (i.e. added) to existing text in the text buffer rather than replacing it. The same error messages and BREAK procedures apply as in the (L)oad command but, of course, the "TOO BIG" error is far more likely to occur since sufficient memory must be available to hold both the old and the new text.

It is important to realise that the editor does not 'interlace' the lines by line number but simply 'tacks on' each successive tape file to the existing contents of the text buffer. The re(N)umber command will automatically renumber the entire buffer properly following the append.

Verify from tape — Vfilename

Verifies cassette files with the contents of the text buffer OR with the contents of the object code buffer produced by the assembler (see later). The editor can distinguish between the types of files and knows what to verify. Note that object code files cannot be loaded or appended but only verified.

As usual no quotes (") are necessary around the filename and spaces after the V are significant. If a filename is specified it will be searched for but if no filename is specified the editor will verify the next file it finds.

During a verification data is not loaded into memory but compared to what is already there. If the verification fails the message "TAPE ERROR" is displayed on the bottom line of the screen. Press any key to return to normal edit mode if this happens. If the verification succeeds however no special message is displayed and the editor returns automatically to normal edit mode.

The BREAK key (CAPS SHIFT'ed SPACE) will abort the operation at any time.

It is recommended that every (S)ave is followed immediately by (V)erify.

Assemble text buffer — Afilename/s1/s2/s3 . . .

Assembles the text buffer with switches s1,s2,s3 . . . active. If an object code tape is produced, it will be saved with the name 'filename'. This filename must be specified using the same format as all other cassette commands. If no filename is specified, the default name 'ObjectCode' is used.

The assemble command will be covered in much greater detail in the next chapter.

Assembler switches

/NO — Suppress output of object code to tape.
/NS — Suppress symbol table listing or printout.
/NL — Suppress assembly listing or printout.
/LP — Output symbol table and assembly listing to ZX printer.
/WE — Wait on errors until key pressed (C cancels).
/IM — Assemble object code directly into memory.
/AO — Assemble with absolute origin.
/OS — Assemble using old symbol table.

3.12 EDITOR ERROR MESSAGES

The editor will signal an "INVALID REQUEST" error under the following circumstances:—
if you enter a non-existent line or extended command,
if the syntax of an extended command is incorrect,
if you attempt to call MCTT before loading it,
if you attempt to set multiple move, copy or block markers,
if you attempt to move or copy a block to within itself,
if you attempt a (C)hange string command without having specified a (F)ind string.
In each case the error message will be briefly displayed on the screen before you're returned to normal edit mode.

If your text buffer grows very large it is possible that you may run out of memory. Should this happen the message "OUT OF MEMORY" is briefly displayed before you're returned to normal edit mode. If at this point you must add further statements to your program you will need to delete some comment lines first.

If you encounter an "OUT OF MEMORY" error when attempting to move a large block of lines to some other point in the text buffer then move the block in smaller pieces one at a time. This is because sufficient memory must be available to hold both the original block and a copy of the block at its new location (i.e. a block move is effectively a block copy followed by a block delete).

4. ASSEMBLER

4.1 Introduction

The function of the assembler is to convert opcode mnemonics and operands entered via the editor (source code) into machine code form directly executable by the Spectrum's Z-80 microprocessor (object code) and to output this information to tape, memory, screen and/or printer.

4.2 Invoking the assembler

The assembler is invoked from normal edit mode (cursor blinking) by pressing both shift keys simultaneously followed by the SPACE key. An extended command line of the form Afilename/s1/s2/s3 . . . is then typed in at the base of the screen and ENTER pressed. This instructs the computer to assemble the text buffer with various options or 'switches' selected. Switches are two letter commands separated by slashes (/). There are eight switches in all and a command line may contain any number of these switches in any order. Switches and their functions will be discussed shortly.

The characters immediately following the 'A' in an assembler command line specify a filename to be used if object code is saved to tape at the end of the assembly. As usual, no quotes ('') are necessary around the filename and every character (up to 10 of them) following the 'A' is significant.

4.3 Source line format

The assembler takes as input lines of source code entered via the editor. As explained earlier these lines are divided into three fields — a label, an opcode and an operand field.

The opcode is the heart of the assembly-language line. It is a mnemonic name that represents one of the individual instructions in the instruction set of the Z-80 microprocessor. Appendix 5 at the end of this manual contains a complete list of these mnemonics in the form understood by the assembler. Note that mnemonics can be entered in either upper or lower case.

Each of the instructions in the Z-80's repertoire has a pre-defined format, and a certain number of operands associated with its operation. Some instructions require no operands, while others require one or two or even more. Appendix 5 gives the number of operands and the correct format for each instruction. Operands may be register names, data values, symbols, expressions, or combinations of these things.

The label field is optional. Labels are used in lieu of BASIC line numbers so that one instruction may reference another. The conventions for naming labels are detailed below.

Remarks (i.e. comments) are allowed after operands. Simply prefix the remark with a semicolon (;). Remark lines are allowed if the first character in the line is a semicolon. Blank lines are also allowed to space out a listing. Both comments and blank lines are ignored by the assembler.

4.4 Two pass assembler

The assembler makes two passes through the text buffer. The first pass checks the syntax of the source code lines and creates a table storing all the symbols used in the program alongside their values. The screen is clear during the first pass. The second pass converts the source code mnemonics into their object code equivalents and calculates fully all the numeric and symbolic operands. This information is displayed on the screen during the second pass.

The BREAK key (CAPS SHIFT'ed SPACE) can be used at any time during the two passes to abort the assembly. During the second pass, any key other than BREAK can be used to pause the assembly and allow you to examine the assembly listing. Press any key again to continue.

4.5 The assembly listing

The assembly listing consists of three parts. The righthand section is simply a copy of the source code from the text buffer. The two columns on the left contain the location at which the object code will reside followed by the object code itself. Both are in hexadecimal.

Error messages are always displayed AFTER the lines that caused them. Error messages are very comprehensive and a complete list of errors, their causes and how to correct them is to be found at the end of this chapter. If possible a small arrow will be used to highlight the point in the line which actually caused the error.

At the end of the assembly a count is given of the total number of errors found.

4.6 The object code buffer

The assembler stores the object code it produces (remember that this is the binary data that the Z-80 chip actually understands) in an area of memory called the object code buffer. This is normally the free memory immediately following the text buffer. In most cases this is NOT the location that the code is designed to run at. When the buffer is saved to tape, however, the information in the file header is such that when the tape is re-loaded into the computer (from BASIC by a LOAD ""CODE command) it will load to the point at which it can be executed.

Object code is written to tape at the end of the second pass of the assembler. The usual "Start tape and press any key" message is displayed. Proceed as you would for a normal BASIC save. On completion of the save you're automatically returned to normal edit mode. The recording can then be verified by a (V)erify extended command and it is recommended that you do so before entering any further commands.

4.7 Symbols and the Symbol Table

Symbol names are strings of one to six characters. The first character MUST be a letter, subsequent characters can be letters (A to Z), numbers (0 to 9) or one of seven special characters (# . ? @ % $ _). The mnemonics for registers and

register pairs CANNOT be used as symbol names. Symbol names are stored internally in upper case form so there is no difference between, for example, LABEL1, label1, LABel1, lAbEl1 etc.

There are three types of symbol —ordinary labels used to identify particular points in a program, EQU symbols used to store the values of constants, and DEFL symbols which are re-definable EQU symbols (see section on pseudo-ops below). All symbols have an internal 16 bit value.

The following are examples of valid symbol names:—
loop1 LABEL2 not_eq lbl101 END start a#%$b

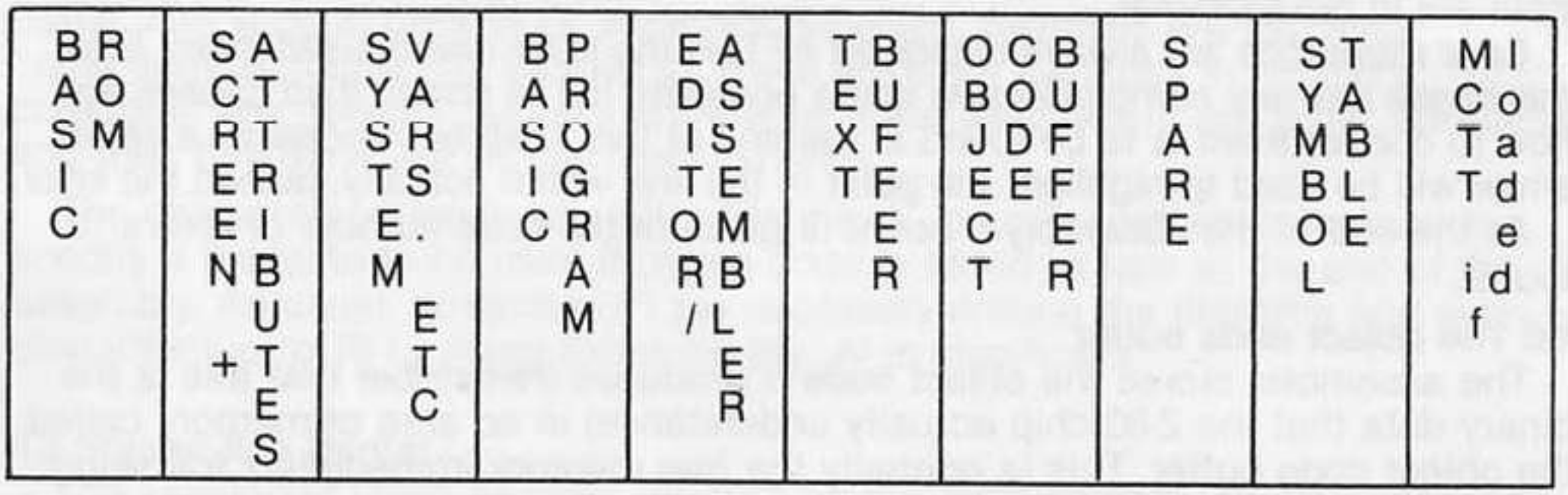
The following are examples of invalid symbol names:—
3label @start bc toolong A #lbl1 ix

At the end of an assembly listing, a sorted list of all the symbols used in a program (the symbol table) is printed. Each symbol name is printed in capitals alongside its 16-bit value in hexadecimal. A number of single letter designators are used to indicate the type of the symbol:—

an L signifies a DEFL symbol,
an M signifies a multiply defined symbol (see assembler error messages),
a U signifies a symbol whose value is unknown.

These appear to the right of the symbol name.

4.8 Memory map



Editor/Assembler entry points:—
Warm = 24304
Cold = 24301

Stand-alone editor
Warm = 24204
Cold = 24201

Stand-alone assembler
24201

4.9 Assembler Switches

/NO — No Object tape switch. Suppresses the output of object code to cassette tape at the end of an assembly. It's often convenient to assemble a program on a trial basis before committing it to tape. Any assembly errors can then be corrected, and an object tape only produced when all such errors have been dealt with.

/NS — No Symbol table listing or printout switch. Suppresses the output of the symbol table to both the screen and the ZX printer.

/NL — No assembly Listing or printout switch. Suppresses the output of the assembly listing to both the screen and the ZX printer. Note that lines which contain errors are always displayed even if this switch is active.

If the /NS and the /NL switches are both active, only the total error count will be displayed at the end of an assembly. This is by far the fastest means of assembling the text buffer.

/LP — Output assembly listing and symbol table to Line Printer switch. If a hardcopy of the assembly is required, this switch causes both the assembly listing and the symbol table to be sent to the ZX printer (if connected).

Oxford Computer Publishing will shortly be releasing a version of the Editor/Assembler program capable of driving a Centronics printer interface and producing 80-column assembler listings. Contact us at PO Box 99, Oxford if you'd like further details.

/WE — Wait on Errors switch. This switch instructs the assembler to stop if an error is found during the assembly process. Pressing any key (except C or BREAK) after the error wait will restart the assembler. Pressing C will turn off the /WE switch and the assembly will continue without waiting on further errors. Pressing BREAK (CAPS SHIFT'ed SPACE) will abort the assembly.

/IM — Assemble Into Memory switch. This very useful switch instructs the assembler to store object code at the first free address in the system such that it can be run at that address. No ORiGin instruction (see pseudo-ops below) is necessary — the assembler automatically relocates your program to this convenient point in memory. The /IM switch implies a /NO switch and so no object tape is produced. Note that the address at which object code is stored can be obtained from the MEM extended command (see above).

The RUN extended command can be used to execute programs assembled into memory. Control is returned to the editor by a RET instruction at the end of the program.

The usefulness of this command is fully realised if you possess a copy of O.C.P.'s monitor program "Machine Code Test Tool". A program can be assembled into memory and de-bugged in situ. When completely working, the program can be assembled in final form to tape.

ORG statements can be included if you wish to store object code at specific places in memory (this technique is not recommended). ORG values specified must be between the end of the text buffer and the top of free memory.

/AO — Assemble with Absolute Origin switch. An absolute assembly is one in which object code is written to cassette tape. Such assemblies must have origins defined by ORG statements (see pseudo-ops below). On completion of the second pass of the assembler an object code tape is prepared with the correct header information such that it can be loaded back into memory by a LOAD ""CODE command from BASIC and will load to the address specified by the original ORG statement.

A filename for the object code tape can be specified when the assembler is invoked (see above) — if a filename isn't specified the default filename 'ObjectCode' is used. The default condition for an assembly is an absolute assembly i.e. A/AO is equivalent to A.

Note that, although object code from an absolute assembly is also stored in an object code buffer immediately above the text buffer, it CANNOT, in general, be executed at this location.

/OS — Assemble with Old Symbol table switch. This switch works in conjunction with the CLEAR extended command (see the section on extended commands). Its inclusion is primarily for the benefit of owners of 16K Spectrums although owners of 48K Spectrums with very large programs may find it useful.

Unless this switch is active all assemblies begin with a clear symbol table. However, if the /OS switch is selected then any symbols which have been previously marked as 'delete protected' (the CLEAR command does this) are retained in the symbol table. By this means the values of symbols can be transferred from one program to another (or rather one section of a program to another — where labels defined in the one are referenced in the other).

A long program is broken down into small sections, and each section is separately assembled and the object code saved to tape. The final address of the first section is noted (the easiest way of doing this is to attach a label to an END statement) and passed onto the next section in an ORG statement, and the process repeated until the whole program has been dealt with (using the CLEAR command between each section). From BASIC the separate object code tapes can be loaded into memory. Each section will join up with the previous one, and then the whole can be saved via a SAVE "filename"CODE command to make one complete program.

Note that only symbols marked as 'delete protected' are retained in the symbol table — current symbols (symbols added to the table by the current program) are cleared out before each assembly. Symbols carried over from program to program retain their original status i.e. a DEFL symbol remains a DEFL symbol. If a DEFL symbol is re-defined by the current program, its original value is lost the first time the current program is assembled.

4.10 Value Operands

Operands with numeric values are coded as constants or expressions. The range of values for an operand is determined by the instruction it is associated with. For example, in the instruction LD HL,nn 'nn' can be any 16-bit value, whereas in the instruction LD A,n 'n' can only be an 8-bit value. Certain instructions are even more restrictive — for example, in the instruction IM i 'i' is only valid as 0, 1 or 2.

Constants

A constant can be expressed as a symbolic label, the location counter symbol $, a character string or a numeric value.

a) Symbolic labels

Every symbol in a program has a value set to it during the assembly (set to zero if there is an error in its definition line). The syntax of a symbol name has already been discussed.

b) Location counter — $

The $ character may be used to represent the current value of the assembler location counter — that is the address at which the next byte of object code will be (logically) stored. e.g. JR $ will put the C.P.U. in an infinite loop.

c) Character strings

Character string constants may be used if enclosed by single quotes ('). The string is right justified (that is to say that all but the rightmost two characters are ignored) and is evaluated as a 16-bit constant based upon the ASCII codes of its characters. Embedded single quotes are represented by two in succession.

Examples:
LD A,'a' loads the accumulator with 61H.
LD A,'''' loads the accumulator with 27H.
LD HL,'ab' loads the HL register pair with 6162H.

d) Numeric constants

The assembler recognises four types of numeric constant, namely binary (base 2), octal (base 8), decimal (base 10) and hexadecimal (base 16). A numeric constant consists of a string of numbers (0 to 9) or hexadecimal digits (0 to 9, A to F) followed by a base determiner. This suffix specifies the type of the constant as follows:—

B = binary
O or Q = octal
D = decimal (default base)
H = hexadecimal

If no base determiner is present the assembler assumes the constant to be decimal. The following numeric constants are all equivalent — 10010B, 22O, 22Q, 18, 18D, 12H.

Note that hex numbers which start with a letter MUST be prefixed by a zero or the assembler will assume you are using a symbol e.g. FFH is invalid, write as 0FFH.

All constants are stored internally using 16 bits. Numbers greater than 65535 are accepted but any overflow from them is lost.

4.11 EXPRESSION EVALUATION

The assembler features a very powerful expression evalutor. Symbols, constants and operators can be combined in any order to produce complex expressions. These greatly simplify the programmer's task by automatically generating such things as table lengths, string lengths, constants, addresses and other data. As a general rule, an expression can be substituted in any instruction that would otherwise take just a simple numeric operand.

The operators that may be used in expressions are shown below. Associated with each operator is a precedence level — this determines the order in which operands are combined within an expression to produce a result. Operators of high precedence evaluate before operators of lower precedence — operators of equal precedence evaluate from left to right. The order of evaluation can, however, be controlled by the inclusion of parentheses (brackets).

OPERATOR	FUNCTION	PRECEDENCE	
=	equals	1	lowest
<>	not equals	1	precedence
>	greater than	1	
<	less than	1	
>=	greater than or equal	1	
<=	less than or equal	1	
+	addition	2	
—	subtraction	2	
OR	logical or	3	
!	logical or	3	
XOR	logical exclusive or	3	
AND	logical and	4	
&	logical and	4	
*	multiplication	5	
/	division	5	
MOD	modulo	5	
NOT	1's complement	6	highest
( )	parentheses	7	precedence

Notes:
(i) Comparison operators (the first six of the above) compare two terms and give —1 if the result is true and 0 if it's false. e.g. 7 > 4 gives —1, 9 < = 1 gives 0, 7 <> 6 gives —1.
(ii) Logical operators are bitwise in operation, that is they act upon each of the 16 bits in each of their two terms in succession to give a 16 bit result. e.g. 1100110B AND 1001011B equals 1000010B, 1110110B OR 0110001B equals 1110111B, 1101101B XOR 1001100B equals 0100001B.
(iii) Division is integral, the fractional part being lost. Multiplication is 16 bit, overflows are not detected. e.g. 7/2 equals 3, 0FFFFH * 2 equals 0FFFEH.
(iv) The modulo operator gives the remainder result of a division. e.g. 23 MOD 7 equals 2.
(v) The NOT operator is a unary operator, it gives the one's complement of a single number (1's = > 0's = > 1's). e.g. NOT 10011011B equals 1111111101100100B.
(vi) Parentheses may be used to group parts of an expression together to force those parts to be evaluated first. In effect they override the fixed precedence levels. Operations of the same precedence are applied from left to right across an expression.
(vii) Parentheses may be nested to any depth. e.g. (1*(2+(3*(4+5)))) equals 29.
(viii) The first operator in an expression cannot be a left parenthesis e.g. LD A, (4*8)+3 is invalid and will cause an error. Rewrite as LD A,3+(4*8).
(ix) An expression such as 1010AND0111 will generate an error message because the assembler assumes the 'A' in 'AND' to be part of the first constant. To avoid this problem, rewrite the expression as 1010 AND 0111.

4.12 Pseudo-Opcodes

Recall that the opcode field of a line of source code contains a mnemonic for one of the Z-80's instructions. In addition the assembler recognises certain other

mnemonics in this field called pseudo-opcodes or pseudo-ops. These are not machine executable instructions but rather directives to the assembler to perform specific operations, set up data structures etc. Labels are compulsory on DEFL and EQU statements but optional on other pseudo-op statements.

ORG — ORiGin. This pseudo-op takes a single number or expression as an argument and sets the assembler location counter to the value of this operand. If a label is present on an ORG statement, it takes on the value of the counter before its modification. Although ORG statements can appear anywhere in a program listing, it is considered good practice to define an origin in the first line.

For example the statement ORG 1234H sets the assembler location counter to the value 1234 hexadecimal and subsequent instructions produce object code designed to run at this address. The statement ORG $+100 reserves a hundred bytes of storage (equivalent to DEFS 100 — see below).

Note that if you assemble your program with the 'into memory' switch active no ORG statements are necessary. The assembler automatically produces object code at the first free address in the system.

END — This pseudo-op simply marks the end of the assembly source code. It takes no arguments and its inclusion in a program is optional but if present then all lines following the END statement are ignored by the asssembler.

The next three pseudo-ops generate byte(s) of data at the current assembly location rather than machine code instructions. Data structures produced by these directives can be used as general constants, messages, look-up tables etc.

DEFB or DB — DEFine Byte. This pseudo-op takes 8-bit numbers or expressions as arguments and stores the values of the operands at the current assembly location (and increments the location counter by one). Multiple arguments are accepted if separated by commas. For example the statement DEFB 10,0FFH,'a' generates three bytes of object code — 10, 255 and 97 decimal. If the value of an operand is greater than 255 or less than — 256, a "FIELD OVERFLOW" error is signaled and only the low order 8-bits are stored.

DEFW or DW — DEFine Word. This pseudo-op takes 16-bit numbers or expressions as arguments and stores the values of the operands at the current assembly location (and increments the location counter by two). Standard Zilog format is employed with the low-order byte being stored before the high-order byte. Again multiple arguments are accepted if separated by commas. For example the statement DEFW —1,1000H,10 generates six bytes of object code — 255, 255, 0, 16, 10 and 0 decimal.

DEFM or DM — DEFine Message. This pseudo-op takes a string as an argument and generates object code bytes corresponding to the ASCII codes for the characters in the string. The string must be enclosed by single quotes (') and, as usual, embedded single quotes are represented by two in succession. For example the statement DEFM 'Hello' generates five bytes of object code — 72, 101, 108, 108 and 111 decimal; the statement DEFM '''a''' generates three bytes of object code — 39, 97 and 39 decimal.

DEFS or DS — DEFine Storage. This pseudo-op takes a single number or expression as an argument and reserves a number of bytes for data storage. In effect the location counter is incremented by the value of the operand. For

example DEFS 5 sets aside five bytes. If a label is used on this statement, it takes on the value of the location counter prior to the incrementing operation. No object code is generated by this pseudo-op.

EQU and DEFL (or DL) — These last two pseudo-ops do not generate any object code but instead assign values to symbols. Both require labels in their statements and both take a single number or expression as an argument. The symbolic label specified is set to the value of the operand specified. The distinction between EQU and DEFL pseudo-ops is that the same label may be DEFL'ed many times to different values in the same program while an EQU'ated label may only be set once. Note that in the assembly listing the value assigned to the symbol appears in the first column rather than the current value of the location counter.

For example COMMA EQU 44 defines a symbol with the name 'COMMA' and gives it the value 44 (the ASCII code for a comma). When referring to this value in a program use the symbolic form LD A,COMMA rather than the absolute form LD A,44 to aid clarity and understanding. LABEL EQU $ assigns to LABEL the current value of the assembler location counter.

A symbol defined initially by a DEFL directive can be subsequently re-defined by either a DEFL or an EQU directive but cannot be re-defined as an ordinary label. Only the last value of a DEFL symbol will appear in the symbol table listing.

4.13 Assembler Error Messages

Assembler errors can be broadly divided into three categories — namely catastrophic errors, fatal errors and non-fatal errors.

Two events are catastrophic — the overflowing of the object code buffer and the overflowing of the symbol table. Both result in the assembly being aborted and a message to that effect being printed on the screen. Press any key to return to normal edit mode if this happens. (The assembly can also be aborted at any time by pressing the BREAK key (CAPS SHIFT'ed SPACE)).

Both these events are caused by insufficient memory being available. To assemble the text buffer successfully you will have to provide enough work space for the assembler to use. The easiest way to do this is to delete some comment lines. Failing this use shorter label names, and as a final resort assemble your program in sections (transfering the symbol table from one section to the next).

Fatal and non-fatal errors lead to a modification in the object code produced by a line of source code yet allow the assembly to continue to other lines. A fatal error is one in which the assembler cannot understand your instructions to it or too much information is missing from a statement for any object code to be generated. Non-fatal errors are less extreme — object code is generated but 'unresolved' parts of it are set to zero or, in the case of a "FIELD OVERFLOW" error, set to the least significant bits of the offending expression.

If possible the error will be 'highlighted' by a small arrow which points to the invalid character(s) in the line.

Note that if a line contains several errors only the 'worst' one will be displayed. This avoids the situation where, say, a jump relative instruction to an undefined label generates an "UNDEFINED SYMBOL" error as well as an unnecessary "BRANCH OUT OF RANGE" error because zero was substituted for the unknown value.

"BAD LABEL" — Non-fatal. An illegal label name was used. Use 1 to 6 characters in a label starting with a letter — don't use register names. Check the label used and alter accordingly.

"MULTIPLE DEFINITION" — Non-fatal. The same label name was used more than once. Note that each definition is flagged as an error but the first value assigned to the symbol is used in expressions. Make each label name unique.

"BAD OPCODE" — Fatal. The assembler didn't recognise the opcode or pseudo-op mnemonic used. Check spelling.

"UNDEFINED SYMBOL" — Non-fatal. A symbol was encountered that had not been defined. Define the symbol as label, EQU or DEFL.

"MULTIPLY DEFINED SYMBOL" — Non-fatal. An expression contained a reference to a symbol defined more than once (see "MULTIPLE DEFINITION" above).

"DIVISION BY ZERO" — Non-fatal. An expression used 0 as a divisor. Change to a non-zero value.

"FIELD OVERFLOW" — Non-fatal. The result of an expression was too large to be contained within the relevant bits of the object code field for the instruction. For instance a 16-bit number was used when an 8-bit operand was expected. Change the value to fit.

"BRANCH OUT OF RANGE" — Non-fatal. The relative address displacement in a JR or DJNZ instruction was greater than +127 or less than —128. Shorten the code or use a JP instruction.

"MISSING INFORMATION" — Fatal. Operand(s) absent or incomplete in source code line. Check the format of the instruction or pseudo-op.

"BAD ADDRESSING MODE" — Fatal. An invalid addressing mode was used. e.g. 'ld a,hl','jr m,label','add hl,ix' etc. Use the correct operands for the instruction.

"BAD EXPRESSION" — Fatal or non-fatal. The syntax of an operand or expression was incorrect. Redefine.

APPENDIX 1
FULL SCREEN TEXT EDITOR — SUMMARY OF COMMANDS AND MODES

The keyboard — Unshifted keys give lower case letters and numbers; CAPS SHIFT'ed keys give upper case letters and cursor movement & control functions. Toggle caps lock status with CAPS LOCK key. SYMBOL SHIFT'ed keys give subsidiary red single-character legend. All keys auto-repeat if held down.

NORMAL EDIT MODE (blinking cursor)

Moving the cursor:—

◄	— left one column
▼	— down one line
▲	— up one line
►	— right one column
TRUE VIDEO	— up one page (24 lines)
INV. VIDEO	— down one page (24 lines)
ENTER	— right to next tab stop (next field)
CAPS SHIFT'ed ENTER	— left margin of next line down

Changing character(s) :— type new text over old text.
Deleting character(s) :— use DELETE key.
Inserting character(s) :— press GRAPHICS key and cursor stops blinking. Type in new text. Use ► key to insert spaces up to next tab stop and ◄ & DELETE keys to backspace. Press ENTER to exit insert mode.
Cancel changes :— Press EDIT key and all changes made to a line are cancelled. Cursor placed on first character.

LINE COMMANDS

Press both shift keys on desired line. Line is highlighted. Next key pressed determines function to be performed:—

CAPS SHIFT + SYMBOL SHIFT — Aborts line command.
D —Deletes line or block of lines.
I —Enters line insert mode and cursor stops blinking.
Type in text. Use ► key to insert spaces up to next tab stop and ◄ & DELETE keys to backspace.
Press ENTER to insert another new line.
Press both shift keys to exit line insert mode.
C —Marks line or block of lines for copying.
M —Marks line or block of lines for moving.
H —Moves or copies line or block of lines to here.
B —Marks line as block terminator.
ENTER — Cancels pending move/copy & deletes block markers.
T —Displays from top of text buffer.
E —Displays last line in text buffer.
F —Repeats last entered (F)ind extended command.
SPACE — Invokes extended command processor.

EXTENDED COMMANDS

Press both shift keys and then the SPACE key to enter extended command mode. Type in command at base of screen and press ENTER to execute it. Press ENTER on an empty line to abort extended command.

Nstart,inc — Renumbers every line in the text buffer.
Gnumber — Searches for the specified line from the top of the display onwards.
Fstring — Searches for the specified string from the top of the display onwards.

Cstring — Changes all occurences of the previously specified (F)ind string to this string.
C — Deletes all occurences of the previously specified (F)ind string.
MEM — Displays free memory length and start address.
BASIC — Returns to Spectrum's BASIC interpreter.
CLEAR — Clears the text buffer and 'delete protects' symbols.
MCTT — Calls Machine Code Test Tool.
RUN — Branches to the first address in the object code buffer.
Pnumber — Prints a number of lines starting with the line at the top of the display.
P — Prints the rest of the text buffer.
Sfilename — Saves the text buffer to tape.
S — Saves the buffer with default filename 'SourceCode'.
Lfilename — Loads a file from tape into the text buffer.
L — Loads the next file on tape.
Xfilename — Appends a file to the text buffer.
X — Appends the next file on tape.
Vfilename — Verifies a file with the text buffer.
V — Verifies the next file on tape.
Afilename/s1/s2/s3... — Invokes the assembler with switches s1,s2,s3... active.
A/s1/s2/s3... — Invokes the assembler with default filename 'ObjectCode' and switches s1,s2,s3... active.

Assembler switches

/NO — Suppress output of object code to tape.
/NS — Suppress symbol table listing or printout.
/NL — Suppress assembly listing or printout.
/LP — Output symbol table and assembly listing to ZX printer.
/WE — Wait on errors until key pressed (C cancels).
/IM — Assemble object code directly into memory.
/AO — Assemble with absolute origin.
/OS — Assemble using old symbol table.

APPENDIX 2 — 16k version

Stand-alone Editor — type LOAD "Editor" ENTER

All the commands available in the combined Editor/Assembler program are available in the stand-alone Editor except for:—

(i) the (A)ssemble command
(ii) the 'call MCTT' command

There are two entry points to the stand-alone Editor — a cold and a warm start. The cold start is used at the beginning of an edit session and clears the text buffer. The warm start is used thereafter and preserves the contents of the text buffer.

Stand-alone Assembler — type LOAD "Assembler" ENTER

The ONLY commands available in the stand-alone Assembler are:—

(i) (A)ssemble the text buffer
(ii) RUN object code
(iii) (V)erify object tapes
(iv) Return to BASIC

The function and syntax of these commands has already been discussed. Note that only the first page of the text buffer is displayed and that you're automatically entered into extended command mode.

The complete scenario for program development in a 16k Spectrum is therefore:—

(a) Load the stand-alone Editor, select a cold start and type in the source code.
(b) Load the stand-alone Assembler, so as to overlay the Editor but leave the text buffer intact.
(c) Assemble the source code and check the object code.
(d) If the object code is incorrect then rewind the tape and load the stand-alone Editor again. Make the necessary changes to the source code and repeat step (b).

APPENDIX 3 — 'Snake' demonstration program

Included on the 48K side of the supplied cassette tape is the source code for a simple game involving the guidance of a 'snake' around the screen. This file is primarily intended as practice material for the editor functions, but can be played in its own right. 'Snake' is recorded on the tape immediately after the Editor/Assembler.

To play 'snake', you must first assemble it. This is most easily done by assembling directly into memory (type A/IM/NL/NS ENTER when in extended command mode). The program can then be executed by the RUN command.

The 'snake' must be guided around the screen so as to avoid both the border and its own tail. This gets progressively harder as the 'snake' grows longer. Use the '1' and 'Q' keys to move it up and down and the 'O' and 'P' keys to move it to the left and to the right. Pressing 'X' returns you to the assembler.

A smaller demonstration program is included on the 16K side of the tape.

The contents of the cassette tape are as follows:

Side 1 Editor/Assembler 48K
Snake (Assembler source code)
udg

Side 2 Editor
Assembler
Snake-Inst (Assembler source code)
Snake-mc

Because the text buffer in a 16K Spectrum is too small to hold the complete source code of the "Snake" program, this program has been separately assembled and the object code saved on side 2 of the cassette under the name "Snake-mc". To run this program just type LOAD "Snake-mc" ENTER and the program will start automatically when loaded.

APPENDIX 4 — Character Generator

To load the Character Generator program type LOAD "udg" ENTER and the program will start automatically, full instructions will be displayed on the screen.

APPENDIX 5
THE COMPLETE Z-80 INSTRUCTION SET IN THE FORM ACCEPTABLE TO THE ASSEMBLER

The following notation is used:—

n represents 8 bits of immediate data in the range 0 to 255 (00 to 0FFH),
nn represents 16 bits of immediate data in the range 0 to 65535 (0000 to 0FFFFH),
d represents an 8-bit displacement in the range —128 to +127,
e represents an 8-bit offset in a jump relative instruction,
b represents a bit number in the range 0 to 7.

Each of the above can be replaced by a symbol or expression in a source code statement.

As a convenience the form (IX + 0) can be replaced by (IX) and (IY + 0) replaced by (IY).

ADC A, (HL)
ADC A, (IX + d)
ADC A, (IY + d)
ADC A,A
ADC A,B
ADC A,C
ADC A,D
ADC A,E
ADC A,H
ADC A,L
ADC A,n
ADC HL,BC
ADC HL,DE
ADC HL,HL
ADC HL,SP
ADD A, (HL)
ADD A, (IX + d)
ADD A, (IY + d)
ADD A,A
ADD A,B
ADD A,C
ADD A,D
ADD A,E
ADD A,H
ADD A,L
ADD A,n
ADD HL,BC
ADD HL,DE
ADD HL,HL
ADD HL,SP
ADD IX,BC
ADD IX,DE
ADD IX,IX
ADD IX,SP
ADD IY,BC
ADD IY,DE
ADD IY,IY
ADD IY,SP
AND (HL)
AND (IX + d)
AND (IY + d)
AND A
AND B
AND C
AND D
AND E
AND H
AND L
AND n
BIT b, (HL)
BIT b, (IX + d)
BIT b, (IY + d)
BIT b,A
BIT b,B
BIT b,C
BIT b,D
BIT b,E
BIT b,H
BIT b,L
CALL nn
CALL C,nn
CALL M,nn
CALL NC,nn
CALL NZ,nn
CALL P,nn
CALL PE,nn
CALL PO,nn
CALL Z,nn
CCF
CP (HL)
CP (IX + d)
CP (IY + d)
CP A
CP B
CP C
CP D
CP E
CP H
CP L
CP n
CPD
CPDR
CPI
CPIR
CPL
DAA
DEC (HL)
DEC (IX + d)
DEC IY + d)
DEC A
DEC B
DEC C
DEC D
DEC E
DEC H
DEC L
DEC BC
DEC DE
DEC HL
DEC IX
DEC IY
DEC SP

DI
DJNZ e
EI
EX (SP),HL
EX (SP),IX
EX (SP),IY
EX AF,AF'
EX DE,HL
EXX
HALT
IM 0
IM 1
IM 2
IN A,(n) or IN A,n
IN A,(C)
IN B,(C)
IN C,(C)
IN D,(C)
IN E,(C)
IN F,(C)
IN H,(C)
IN L,(C)
INC (HL)
INC (IX + d)
INC (IY + d)
INC A
INC B
INC C
INC D
INC E
INC H
INC L
INC BC
INC DE
INC HL
INC IX
INC IY
INC SP
IND
INDR
INI
INIR
JP (HL)
JP (IX)
JP (IY)
JP nn
JP C,nn
JP M,nn
JP NC,nn
JP NZ,nn
JP P,nn
JP PE,nn
JP PO,nn
JP Z,nn
JR e
JR C,e
JR NC,e
JR NZ,e
JR Z,e
LD (nn),A
LD (nn),BC
LD (nn),DE
LD (nn),HL
LD (nn),IX
LD (nn),IY
LD (nn),SP
LD (BC),A
LD (DE),A
LD (HL),A
LD (HL),B
LD (HL),C
LD (HL),D
LD (HL),E
LD (HL),H
LD (HL),L
LD (HL),n
LD (IX + d),A
LD (IX + d),B
LD (IX + d),C
LD (IX + d),D
LD (IX + d),E
LD (IX + d),H
LD (IX + d),L
LD (IX + d),n
LD (IY + d),A
LD (IY + d),B
LD (IY + d),C
LD (IY + d),D
LD (IY + d),E
LD (IY + d),H
LD (IY + d),L
LD (IY + d),n
LD A,(nn)
LD A,(BC)
LD A,(DE)
LD A,(HL)
LD A,(IX + d)
LD A,(IY + d)
LD A,A
LD A,B
LD A,C
LD A,D
LD A,E
LD A,H
LD A,L
LD A,n
LD A,I
LD A,R
LD B,(HL)
LD B,(IX + d)
LD B,(IY + d)
LD B,A
LD B,B
LD B,C
LD B,D
LD B,E
LD B,H
LD B,L
LD B,n
LD BC,(nn)
LD BC,nn
LD C,(HL)
LD C,(IX + d)
LD C,(IY + d)
LD C,A
LD C,B
LD C,C
LD C,D
LD C,E
LD C,H
LD C,L
LD C,n
LD D,(HL)
LD D,(IX + d)
LD D,(IY + d)
LD D,A
LD D,B
LD D,C
LD D,D
LD D,E
LD D,H
LD D,L
LD D,n
LD DE,(nn)
LD DE,nn
LD E,(HL)
LD E,(IX + d)

LD E,(IY + d)
LD E,A
LD E,B
LD E,C
LD E,D
LD E,E
LD E,H
LD E,L
LD E,n
LD H,(HL)
LD H,(IX + d)
LD H,(IY + d)
LD H,A
LD H,B
LD H,C
LH H,D
LD H,E
LD H,H
LD H,L
LD H,n
LD HL,(nn)
LD HL,nn
LD I,A
LD IX,(nn)
LD IX,nn
LD IY,(nn)
LD IY,nn
LD L,(HL)
LD L,(IX + d)
LD L,(IY + d)
LD L,A
LD L,B
LD L,C
LD L,D
LD L,E
LD L,H
LD L,L
LD L,n
LD R,A
LD SP,(nn)
LD SP,nn
LD SP,HL
LD SP,IX
LD SP,IY
LDD
LDDR
LDI
LDIR
NEG

NOP
OR (HL)
OR (IX + d)
OR (IY + d)
OR A
OR B
OR C
OR D
OR E
OR H
OR L
OR n
OTDR
OTIR
OUT (C),A
OUT (C),B
OUT (C),C
OUT (C),D
OUT (C),E
OUT (C),H
OUT (C),L
OUT (n),A or OUT n,A
OUTD
OUTI
POP AF
POP BC
POP DE
POP HL
POP IX
POP IY
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY
RES b,(HL)
RES b,(IX + d)
RES b,(IY + d)
RES b,A
RES b,B
RES b,C
RES b,D
RES b,E
RES b,H
RES b,L
RET
RET C
RET M

RET NC
RET NZ
RET P
RET PE
RET PO
RET Z
RETI
RETN
RL (HL)
RL (IX + d)
RL (IY + d)
RL A
RL B
RL C
RL D
RL E
RL H
RL L
RLA
RLC (HL)
RLC (IX + d)
RLC (IY + d)
RLC A
RLC B
RLC C
RLC D
RLC E
RLC H
RLC L
RLCA
RLD
RR (HL)
RR (IX + d)
RR (IY + d)
RR A
RR B
RR C
RR D
RR E
RR H
RR L
RRA
RRC (HL)
RRC (IX + d)
RRC (IY + d)
RRC A
RRC B
RRC C
RRC D

RRC E
RRC H
RRC L
RRCA
RRD
RST 00H
RST 08H
RST 10H
RST 18H
RST 20H
RST 28H
RST 30H
RST 38H
SBC A,(HL)
SBC A,(IX + d)
SBC A,(IY + d)
SBC A,A
SBC A,B
SBC A,C
SBC A,D
SBC A,E
SBC A,H
SBC A,L
SBC A,n
SBC HL,BC
SBC HL,DE
SBC HL,HL
SBC HL,SP
SCF
SET b,(HL)
SET b,(IX + d)
SET b,(IY + d)
SET b,A
SET b,B
SET b,C
SET b,D
SET b,E
SET b,H
SET b,L
SLA (HL)
SLA (IX + d)
SLA (IY + d)
SLA A
SLA B
SLA C
SLA D
SLA E
SLA H
SLA L
SRA (HL)
SRA (IX + d)
SRA (IY + d)
SRA A
SRA B
SRA C
SRA D
SRA E
SRA H
SRA L
SRL (HL)
SRL (IX + d)
SRL (IY + d)
SRL A
SRL B
SRL C
SRL D
SRL E
SRL H
SRL L
SUB (HL) or SUB A,(HL)
SUB (IX + d) or SUB A,(IX + d)
SUB (IY + d) or SUB A,(IY + d)
SUB A or SUB A,A
SUB B or SUB A,B
SUB C or SUB A,C
SUB D or SUB A,D
SUB E or SUB A,E
SUB H or SUB A,H
SUB L or SUB A,L
SUB n or SUB A,n
XOR (HL)
XOR (IX + d)
XOR (IY + d)
XOR A
XOR B
XOR C
XOR D
XOR E
XOR H
XOR L
XOR n

Notes

Acknowledgement

OCP would like to thank Derek and Nualla Smith, Elizabeth Oliver and Julie Perkins for the many hours spent de-bugging the FINANCE MANAGER.

Whilst we try very hard to produce a totally bug-free program it is always conceivable that there is one bug that we have missed. Users who feel that they have identified such a bug or who would like to find out more about our expanding range of super friendly programs please contact us at the address below.

OXFORD COMPUTER PUBLISHING COMPANY LTD.
P.O. BOX 99, OXFORD, ENGLAND

Produced by Artset Graphics, The Old British School House, East Street, Chesham, Buckinghamshire. Tel. (0494) 771938.

OTHER ZX SPECTRUM PROGRAMS FROM OXFORD COMPUTER PUBLISHING INCLUDE:

1. MACHINE CODE TEST TOOL

a complementary monitor and de-bug program to the full screen Editor Assembler (also available for the ZX81) £9.95

2. MASTER TOOL KIT (16 & 48K)

add many new commands and facilities to your ZX Spectrum with this remarkable program, Re-number, Auto-Number, Delete/Copy and Move Block, String Search and Substitute, Variable Dump, Cross Reference, Trace Function, Real Time Clock, and Alarm, Operating Program Line Display plus many more features. Programming will never be the same after you have purchased MASTER TOOL KIT. £9.95

3. CHESS — THE TURK

a very comprehensive chess program for the 48K Spectrum £8.95

4. ADDRESS MANAGER

a fast machine-coded application program that offers Spectrum owners a professional standard address/data filing, indexing and retrieval system. Will store up to 400 full name and addresses in 48K, full screen entry is standard £8.95

5. FINANCE MANAGER

a powerful and flexible MENU DRIVEN program for practically all domestic and business accounting applications. Features include AUTOMATIC DOUBLE ENTRY, ANALYSIS, STANDING ORDERS, RECONCILIATION and FULL SCREEN ENTRY. Available for 48K only £8.95

An 80 column centronics printer version of this program is available on request.

These programs will be followed by high quality Games and Business programs including VAT Manager, Sales Ledger, Purchase Ledger, Stock Control etc.

Made in England
Oxford Computer Publishing Ltd.
P.O. Box 99, Oxford, England