# Laser Genius by Ocean

# Introduction

Z80 Assemblers and Monitors have been available, not surprisingly, since the introduction of the Z80 microprocessor itself. Looking back to the original software available from ZILOG it is clear that the nature of these programs has changed very little. There is no doubt that the software currently available for home micros has come a long way but there have been no radical departures from the standard format and the ZILOG programs are still ‘state of the art’.

What we have tried to do with the Genius system is to incorporate as much as possible from previous designs and go a few steps further. The assembler now supports a small high level language called Phoenix which we hope will speed up program development and add clarity to listings without incurring the speed and size penalties associated with some high level compiled languages. The traditional text and line number based editors have also come under scrutiny and we believe that the hybrid utilised by Genius makes the best use of both worlds. As well as conditional assembly, a mode of assembly has been added which allows selective assembly of a source library based on the current symbol table.

Perhaps the best opportunity for innovation, however, lies in machine code debugging. The title ‘Monitor’ suggests that the program being debugged is continuously ‘Monitored’ by the controlling software. In practice it is the user who does all the work and most of us have spent many a midnight hour single stepping a program by hand to find when and how a particular bit is set in a particular location. Often the instruction being single stepped will take microseconds to execute whilst the operator takes a million times longer to acquaint themselves with the outcome. The Analyser part of Genius takes care of these problems with a speed and vigilance that human intervention cannot match.

When you first open this manual and look at the number of commands and options available, you may be tempted to think that it’s going to be a system for very advanced users only. In fact you will find that it does all the things that conventional assemblers and monitors do in more or less the same way and so if you’ve already used one, you should have no difficulty with the conventional parts. There are a lot of extra facilities to become acquainted with and so you won’t know the system backwards in one week, but you should find that none of the extended features of the system are any more difficult to master than the basic features of any other assembler or monitor. You don’t have to be a Genius to use it. In fact the ‘Genius’ system was not named with the programmers who wrote it in mind, nor with the program itself in mind, but with the user in mind. It’s our belief that by automating all the drudgery that’s been put up with for years, users will have time to use just a little bit more of their own creative Genius. Those of us who have had an opportunity to use this system have developed a reluctance to return to conventional means. We hope you will share our view.

## *Assembler Loading Instructions*

TAPE: On Tape 1 of the Genius package you will find the following files:

Side A:

| | |
|---|---|
| GENASM | A BASIC loader/relocator program. |
| REL | A binary file used by the BASIC loader/relocator. |
| ASM | Three binary files comprising the assembler itself. |
| HASH | |
| TOOLS | |

Side B:

| | |
|---|---|
| TRANS | Transfer loader (Spectrum only). |
| TRANSBIN | Transfer utility (Spectrum only). |
| SIEVE.ASM | Sieve of Eratosthenes example program in assembly language. |
| SIEVE.PHX | Sieve of Eratosthenes example program in Phoenix. |
| ELLIPSE.ASM | Ellipse drawing program in assembly language. |
| ELLIPSE.PHX | Ellipse drawing program in Phoenix. |
| MPAFNCS.PHX | Multiple precision arithmetic routines in Phoenix. |

For AMSTRAD disc users all the above files are on Side A of the disc.

## *Loading the Assembler*

Before loading on the AMSTRAD, reset your machine by pressing CTRL-SHIFT-ESC together.

To load the assembler just type RUN “GENASM” (on the AMSTRAD) or LOAD “GENASM” (on the Spectrum, the loader program will auto-run).

GENASM is a BASIC loader/relocator and will give you a menu of options to choose from. You can load the assembler with, optionally, the assembler tool-kit (see 2. “Editor Immediate Commands”), and the Hash Extensions (see 6. “The Hash Extensions”). There is also a facility for Spectrum users to produce a microdrive version of the assembler.

The loader program also allows you to set the ink and paper colours to the values you want to use within the assembler, and alter the value of the line-feed character sent to the printer during listing (setting this to zero can cure problems of double line spacing encountered with some printers).

# 1. The Screen Editor

The editor divides the screen into blocks of text which will be referred to as sentences. Any text typed onto the screen will be entered into the current sentence. When the screen is clear, each screen line represents a single (blank) sentence. Sentences may, however, be longer than one screen line - if you type text over the end of a screen line you will see that a new line is inserted beneath the current line, a 'hyphen' character is printed at the end of the current line showing you that this sentence is continued on the next line, and the character you have just typed actually appears at the start of the newly inserted line.

## *1.1 Key Usage*

### a) Amstrad

The cursor may be positioned at any location on the screen by using the cursor keys. Text already on the screen may be altered by moving the cursor to the required position and editing the characters there.

It is possible to move the cursor to the start (end) of the current screen line by holding the control (CTRL) key down and then pressing the left (right) arrow key. Similarly, the cursor may be moved to the start (end) of the current sentence by pressing the control and up (down) arrow keys simultaneously.

The editor can function in either overwrite (the default) or insert modes. These are switched (or toggled) from one to the other by pressing CTRL and TAB together. Notice that in insert mode the current cursor character and all those to its right in the current sentence are shifted right one position to make room for the new one. Similarly, the CLR and DEL keys, which delete respectively the current cursor character and the one to its left, shift all the text to the right of the deleted character (in the current sentence) to fill the space left by this character.

The current sentence may be deleted by pressing SHIFT and DEL together. The part of the screen below the current sentence will be scrolled up to fill the gap left by the deleted sentence. It is also possible to delete to the end of the current sentence from the current cursor position by typing CTRL and DEL together. Any screen lines made blank by doing this will be filled by the screen scrolling up as for deleting a whole sentence. A new (blank) sentence can be inserted above the current sentence by typing SHIFT and TAB together. The part of the screen consisting of the current sentence and everything below it will be scrolled down to make room for the new sentence.

Pressing the TAB key moves the cursor to the next tab stop. These are set at every eighth character position. Typing SHIFT right arrow also performs a TAB. Typing SHIFT left arrow performs a "back tab", moving the cursor to the previous tab stop.

The screen can be scrolled up or down by half the screen height automatically by typing SHIFT down arrow (to scroll down) of SHIFT up arrow (to scroll up). In both cases the cursor will be left in the same, relative, screen position.

Typing CTRL "L" will clear the screen and CAPS LOCK will toggle the upper lower case mode as in Amstrad BASIC.

### b) Spectrum

The cursor may be positioned at any position on the screen by using the usual cursor keys (CAPS SHIFT and one of the numeric keys 5,6,7,8). Text already on the screen may be altered by moving the cursor to the required position and typing over the text already there.

The editor can function in either overwrite (the default) or insert modes. These are switched (or toggled) from one to the other by pressing CAPS SHIFT and "1" together. Notice that in insert mode the current cursor character and all those to its right in the current sentence are shifted right one position to make room for the new character. CAPS SHIFT "0" deletes the character to the left of the cursor and SYMBOL SHIFT "0" deletes the character under the cursor. Both of these actions will shift all the text to the right of the deleted character (in the current sentence) to fill the space left by the deleted character.

Note that typing SYMBOL SHIFT "W" will insert a single space at the current cursor position (leaving the cursor at this position) even when you are in overwrite mode. This can be useful for occasional insertions in overwrite mode.

The current sentence may be deleted by pressing CAPS SHIFT and "3" together. The part of the screen below the current sentence will be scrolled up to fill the gap left by the deleted sentence. It is also possible to delete to the end of the current sentence from the cursor position by typing SYMBOL SHIFT “F”. Any screen lines made blank by doing this will be filled by the screen scrolling up as for deleting a whole sentence. A new (blank) sentence can be inserted above the current sentence by typing SYMBOL SHIFT “I”. The part of the screen consisting of the current sentence and everything below it will be scrolled down to make room for the new sentence.

Pressing SYMBOL SHIFT “E” moves the cursor to the next tab stop. These are set at every eighth character position. Typing SYMBOL SHIFT “Q” performs a “back tab”, moving the cursor to the previous tab stop.

Typing SYMBOL SHIFT “G” will clear the screen.

Typing CAPS SHIFT “2” will toggle the upper/lower case mode.

On the Spectrum some keys have been programmed to give characters which they normally give only in extended mode. These are given below.

```
SYMBOL SHIFT “Y” gives “[”
SYMBOL SHIFT “U” gives “]”
SYMBOL SHIFT “S” gives “|”
SYMBOL SHIFT “D” gives “\”
```

Also, the combination SYMBOL SHIFT “A” is used as an escape key. To stop any operation you can press this combination of keys. The mnemonic “ESC” or the term “escape key” will be used throughout to refer to these keys.

## *1.2 Key Usage in Brief*

### Editor Key Usage Summary

**Note:**

**AMSTRAD:** In the following, “up”, “right”, “down” and “left” denote the cursor or arrow keys. SHIFT and CTRL stand for the shift and control keys respectively; these keys should be held down with the other key indicated to give the required effect.

**SPECTRUM:** In the following CS and SS represent the CAPS SHIFT and SYMBOL SHIFT keys respectively; these keys should be held down with the other key indicated to give the required effect. “Up”, “Right”, “Down” and “Left” denote the usual Spectrum cursor keys (CAPS SHIFT 5 to 8).

| AMSTRAD | SPECTRUM | EFFECT |
|---|---|---|
| `Up` | CS “7” | Move the cursor to the previous line. |
| `Down` | CS “6” | Move the cursor to the next line. |
| `Left` | CS “5” | Move the cursor left one character. |
| `Right` | CS “8” | Move the cursor right one character. |
| `SHIFT Up` | | Scroll the screen up by half the screen height, leaving the cursor in the same screen position. |
| `SHIFT Down` | | Scroll the screen down by half the screen height, leaving the cursor in the same screen position. |
| `SHIFT Left` | CS “4” | Move the cursor to the first tab stop before the current position. |
| `SHIFT Right` | SS “E” | Move the cursor to the next tab stop. The stops are set at every eighth character position. |
| `CTRL Up` | | Move the cursor to the start of the current sentence. |
| `CTRL Down` | | Move the cursor to the end of the current sentence. |
| `CTRL Left` | | Move the cursor to the start of the current line. |
| `CTRL Right` | | Move the cursor to the end of the current line. |
| `CLR` | SS “0” | Delete the current cursor character. |
| `DEL` | CS “0” | Delete the character to the left of the cursor. |
| `SHIFT DEL` | CS “3” | Delete the current sentence. |
| `CTRL DEL` | SS “F” | Delete to the end of the current sentence |
| `TAB` | SS “E” | Move the cursor to the next tab stop. Tab stops are set at every eighth |

| | | |
|---|---|---|
| | | character position. |
| SHIFT TAB | SS “I” | Insert a new (blank) sentence at the current cursor line. |
| COPY | SS “W” | Insert a single space at the current cursor position, leaving the cursor at this position. |
| CTRL “L” | SS “G” | Clear the screen. |
| CTRL TAB | CS “1” | Toggles the insert/overwrite mode. |
| CAPS LOCK | CS “2” | Toggles the caps lock. |
| CTRL CAPS LOCK | | Toggles the shift lock. |
| | SS “Y” | Gives”[” |
| | SS “U” | Gives”]” |
| | SS “S” | Gives “\|” |
| | SS “D” | Gives “\” |
| | SS “A” | Escape key. |

## *1.3 Syntax Checking and Error Messages*

The editor expects each sentence to contain either a legal Z80 assembly language instruction or an editor immediate command (see 2. Editor Immediate Commands). It checks to see if this is so whenever the cursor is moved off a sentence. That is, the editor syntax-checks a sentence whenever the cursor is moved off that sentence. If the sentence contains a legal instruction no action is taken. If, however, the line contains a syntactical error, an error message will be inserted, as a new sentence, above the illegal one. The cursor will be placed at the position in the sentence where the syntax-checker detected the error.

You will find that you cannot type on or delete characters from the error message. You can delete the error message by using SHIFT up arrow, but if you do so the sentence referred to by the message will also go. Similarly, if you delete the bad-syntax line, the error message will be removed. It is not possible to insert a new sentence between the error message and the illegal sentence.

An example error message:

```
        ** illegal second operand **
        ld a,de
```

The cursor would be placed just after the “e” in the line “ld a,de”. See Appendix B: “Immediate Error Messages” for a list of possible messages and their meanings.

Once an error message has been given for a sentence (and the sentence not altered since) the cursor may be moved on and off that sentence without hindrance (i.e. it will not be placed at the error position again). Once the sentence is altered, however, it again becomes liable to syntax-checking. If the error is corrected, the error message will be removed when the sentence is next checked.

### 1.3.1 Legal Assembly Language Statements

A legal Z80 assembly language statement has the following form:

[<label> :] [<Z80 instruction>] [; <comment text>]

All of the three fields making up the instruction are optional. The content of each field is described below.

<label> This may contain upper and lower case alphabetic characters, the digits 0 to 9, and the characters “$”, “.”, “_”. It may not begin with a digit. Labels may be up to 240 characters long.

The assembler distinguishes between upper and lower case in labels. So “SYNCH1”, “Synch1”, “synch1” would all represent different names.

A label must be followed by a colon. (Spaces are allowed between the end of the label and the colon).

<Z80 instruction> This can be a standard Z80 opcode, all of which are listed in Appendix A; a standard pseudo-op, which are described in 4.3; a Genius assembler directive, as given in 4.2.

<comment text> This is any string of characters. It must be separated from the rest of the line by a semicolon.

## 1.3.2 Expressions in Assembly Language Statements

Arithmetic expressions can be used anywhere, in an assembly language statement, that a number is required. An expression may contain any of the following:

<decimal number> A string of decimal digits (0 to 9) e.g. 175.

<binary number> A string of binary digits (0, 1) immediately preceded by a "%" character, e.g. %11010.

<octal number> A string of octal digits (0 to 7) immediately preceded be an "@" character, e.g. @7041.

<hexadecimal number> A string of hexadecimal digits (0 to 9, and A to F in upper or lower case) immediately preceded by a "#" character.

OR A string of hexadecimal digits, starting with a (decimal) digit and terminated by an "H" (upper case). For a number such as #FF = 255 you would need to write 0FFH to satisfy the requirement that the number start with a decimal digit.

```
e. g. #FFFF=65535=-1=0FFFFH
      #3412=3412H
```

**NOTE:** Number constants must be in the range 0 to 65535 or the syntax checker will not accept them. You can use the unary minus operator to enter negative integers, i.e. you can write -1 instead of 65535 or #FFFF.

<labels and names> Strings of up to 240 characters in length. They may contain any alphabetic characters (upper or lower case), decimal digits, the characters "$", "_". A name should not begin with a digit.

<character constant> A single character enclosed in double quotes, e.g. "A". This will be treated as the ASCII code of the character given. Character constants can also be used to generate control codes, e.g. "\13" will represent the number 13 (a carriage return). This can be used to emphasise the fact that this is intended as a character. (See also "4.1.2 A Note on Assembler Strings").

<location counter, $> The character "$" represents the current location (program counter). This will be equal to the start address of the instruction being assembled, e.g.

```
ORG #1000
LD HL,$
```

would produce

```
LD HL,#1000
```

in object code.

<storage counter, . > The character "." represents the current storage location of the object code being produced, i.e. where it is being put in memory.

<arithmetic operators> One of the symbols below.

Binary operators:

| | |
|---|---|
| * | multiply |
| / | divide |
| % | mod |
| + | add |
| - | subtract |
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| ?= | equal to |
| != | not equal to |

| | |
|---|---|
| << | shift left |
| >> | shift right |
| @< | rotate left |
| @> | rotate right |
| & | bit wise AND |
| \| | bit wise OR |
| ^ | bit wise exclusive OR |
| && | logical AND |
| \|\| | logical OR |

Unary operators:

| | |
|---|---|
| – | unary minus |
| ! | logical NOT |
| ^ | bit wise complement |
| * | “contents of” |

Parenthesis:

| | |
|---|---|
| [ | |
| ] | square brackets are used to indicate parenthesis |

**NOTE:** An operator precedence technique is used for evaluating expressions. For a description of this and of all the available operators, see Appendix D.

## *1.4 The Screen Buffer and Text Scrolling*

The editor keeps a copy in memory of the text on the screen. In fact, the physical (monitor) screen may be considered as a window on this screen buffer. Thus if a line of text is scrolled off the top of the screen by the cursor moving off the bottom, if the screen is made to scroll down (by moving the cursor off the top of the screen) this line will be re-printed on the top line. Similarly, lines scrolled off the bottom of the screen may be recovered by scrolling the screen up. (When the top of the screen buffer is reached the cursor will move to the left hand side of the screen, if it is not already there, and the screen will scroll down no further. When the bottom of the buffer is reached, the cursor will move to the left hand side of the screen but the editor will then extend the buffer by one (blank) sentence).

Of course, the screen buffer is not of unlimited size and some text which is not on the physical screen may be deleted to make room for other operations (for instance, new text being added to the screen by typing). Only as much room as is required for the operation will be generated in this way, so only as much text as will release sufficient space will be deleted. The text at the top of the buffer will be the first to go; and if this will not generate sufficient room, the text at the bottom of the screen copy will be removed. If it is not possible to generate enough space by this method, a message to this effect will be displayed. There are two possible such messages:

(i) message:

```
** no space **
** (press ESC) **
```

This message will appear if there is no more space in the screen buffer for the text being typed in from the keyboard. After it has appeared and you have pressed the escape key, you will find that you can no longer type on the screen. You can only use keys which delete screen text and thus release some space for the screen buffer (including CTRL “L” which clears the screen and its buffer). After doing this you can use the “SET SPACE” command to give yourself more room for the screen buffer and tokenised text (see 2. Editor Immediate Commands).

(ii) message:

```
** clear screen? (Y/N) **
```

This message implies that insufficient space is available for the current operation to be performed. Answering Y (for “yes”) will cause the screen to be cleared, releasing some space and the operation will be attempted again. Answering N (for “no”) will abort the operation. You can give yourself more space to work with by using the “SET SPACE” command (see 2. Editor Immediate Commands).

## *1.5 Text Entry and Multiple Statement Lines*

The source text of assembly language programs is stored by the editor/assembler in tokenised form, i.e. the key words recognised by the assembler are stored in memory as single byte integers. This is done to reduce the size of the source file and to speed the assembly process.

The contents of the source file are ordered by line number. That is, the source text is stored in blocks (called "paragraphs"), each of which starts with a (decimal) integer in the range 0 to 65534. Each block or paragraph may, however, contain more than one assembly language statement.

To enter a paragraph containing just one assembly language statement just type the number of the paragraph and the instruction in one sentence and press ENTER. For example:

```
    10 ld a,"p" <ENTER>
```

Note that the cursor is moved to the start of the next screen line.

To continue this paragraph just type the next instruction and press ENTER again. So now you will have typed something like:

```
    10 ld a,"p" <ENTER>
    call #BB5A <ENTER>
```

When the ENTER key is pressed the current sentence is first syntax-checked. If the sentence is found to be illegal then an error message is produced (see 1.3 Syntax Checking and Error Messages). If the sentence is alright then it is either an editor immediate command, in which case it is obeyed (see 2. "Editor Immediate Commands"), or it is a Z80 assembly language instruction. In this latter case the statement will be tokenised and the editor will try to enter it into the current source file. Now starting at the current sentence and moving

up the screen copy, the editor searches for a sentence that holds a legal assembly language statement and has a line number. This search is terminated if the editor comes across a sentence which does not hold a legal assembly language statement before it finds a line-numbered sentence. In this case the message:

```
    ** no line number **
    ** (press ESC) **
```

will be issued, and the current sentence will not be inserted into the source file.

If the editor finds a line-numbered sentence then it searches down the screen copy from this sentence (tokenising the text as it goes) until it finds a sentence which is not a legal Z80 assembly language instruction. This it treats as a delimiter and inserts all the sentences from the line-numbered one to this one into the source file under the line number that it found. These sentences will include the current sentence. For example, typing the following on a blank screen:

```
    call #BB5A
```

will give the " ** no line number **" message. While if the line above the current one has:

```
    10 ld a,"p"
```

now typing

```
    call #BB5A
```

gives a single paragraph containing both the statements

```
    10 ld a,"P"
```

and

```
    call #BB5A
```

A paragraph may be continued in this way until its tokenised version becomes more than 1K in length. If this happens, the message:

```
    ** line too long **
    ** (press ESC) **
```

will be displayed and the line will not be entered into the source file. It is, however, intended that each paragraph contain a fairly small number of statements, say up to 20. This allows each paragraph to be visible in its entirety on the monitor screen and keeps the amount of workspace required by the LIST command (see 2. Editor Immediate Commands) down.

To see the paragraph which you have entered, type (on a blank sentence):

```
    list 10
```

You will see the two statements above listed as:

```
    10   LD A,"p"
    CALL #BB5A
```

See “2. Editor Immediate Commands” for further description of the “LIST” command.

Note also that if a paragraph of statements is half on the physical screen and half off, the editor will not remove those statements which are off the screen when it needs to generate space. (see “1.4 The Screen Buffer and Screen Scrolling”). The off-screen statements can always be scrolled back onto the screen for editing. If the ENTER key is pressed when the cursor is on one of the on-screen sentences the editor will still be able to find the line number for that sentence, even though it is “off-screen “.

# 2. Editor Immediate Commands

As described in “1.1 Text Entry and Multiple Statement Lines”, the syntax checker expects each sentence to contain either a legal Z80 assembly language statement or an editor immediate command. In the latter case the command will be obeyed and the sentence will not be stored in the text filer The commands available are given below.

Glossary of Terms

[...] Items within square brackets are optional.

<line range> A range of line numbers in the form nl-n2 (n1 must be less than n2).

e.g. `10-20` means line numbers from 10 to 20 inclusive.
`-30` means all lines from the start of the file to line 30 inclusive.
`50-` means all lines from line 50 to the end of the file inclusive.

A line range is always optional and defaults to 0-65534, i.e. the whole of the source text.

<string> A string of characters. This will always be enclosed in quotes.

<expression> An arithmetic expression consisting of:
integer constants
symbol names
allowable operators

See Appendix D: “Arithmetic Expressions and Operator Precedence”.

<parameter list> A list of strings (enclosed in quotes) and expressions separated by commas.

<base> One of the integers: 2 (for binary)
8 (for octal)
10 (for decimal)
16 (for hexadecimal)

<decimal integer> Usually this should be in the range 0 to 65535.

<microdrive number> An integer in the range 1 to 8.

### *2.0 The Available Commands*

### *2.1 Screen Commands*

`CLS` Clear the visible screen and the editor’s copy in memory.

Amstrad only:

`MODE` <0, 1 or 2> Change screen mode (as in Amstrad BASIC).

`SET SPACE` <decimal integer> The editor has a fixed amount of (memory) space in which to store both the screen buffer and the tokenised source text. The command SET SPACE is used to set the size of the editor’s space. It may be used at any time. The integer parameter is the amount of space you want to give the editor; it must be at least 10241 bytes. When the editor is first entered it will have 10K of space available to it.

If you reduce the amount of space available to the editor it may be necessary for the editor to delete the current source file; before doing this it will ask:

```
        ** delete the whole file? (Y/N) **
```

Answering N will cause the command to be abandoned. Similarly, if you wish to increase the amount of space available, it may be necessary to delete the current symbol table. The editor will ask your permission before so doing.

See also Appendix E : “The Memory Map”.

## *2.2 Text Editing Commands*

`LIST` [<line range>]

List the given line (paragraph) range of the source to the screen. Listing can be temporarily halted by pressing any key. Pressing ESC while listing is halted will terminate the command, pressing any other key will restart it.

Note that the listing can only be halted after a whole paragraph has been displayed on the screen. This prevents you from listing a portion of a paragraph re-entering it and thus losing some text inadvertently.

`LLIST` [<line range>]

As LIST but output is sent to the printer. LLIST can be halted at any time during printing, not just at paragraph boundaries.

`DELETE` [<line range>]

Remove those paragraphs with numbers within the given range from the source text. If the line range given represents the whole of the source text the message “** delete the whole file? (Y/N) **” will be displayed. Answering Y (for “yes”) will cause the source text to be deleted, answering N (for “no”) will abandon the command.

Note that single paragraphs can be deleted from the source file by typing the line number on a blank sentence and pressing ENTER, e.g. Line 10 may be deleted by typing:

```
10 <ENTER>
```

`COPY` <line range 1>, <line range 2>

Replace line range 2 with line range 1, retaining line range 1. The source text will be renumbered from one in steps of one.

`MOVE` <line range 1>, <line range 2>

Replace line range 2 with line range 1, removing line range 1. The source will be renumbered from 1 in steps of 1.

`RENUM` [<new start> [,<step> [,<old start>]]]

Renumber the source text paragraphs from <old start>, in steps of size <step>, starting at <new start>. If this is not possible, i.e. the step size is so large that the new last line number will be greater than 65534, or <new start> is less than <old start>, the source line numbers will be unaltered.

All the parameters of RENUM are optional. Both <new start> and <step> default to 10. <old start> defaults to 0, so that the whole file will be renumbered e.g. RENUM <ENTER> is equivalent to RENUM 10,10,0 <ENTER>.

## *2.3 The Search and Replace Facility*

`FIND` “<string>” [,<line range>]

Searches for occurrences of <string> in the source text within the specified line range. When FIND reaches an occurrence of the search string it lists the whole of the paragraph which contains it to the screen, and places the cursor at the start of the found string. Pressing any key will cause FIND to look through the rest of the listed paragraph for further occurrences, again placing the cursor at the start of the string if it finds one, and waiting for a key press.

When the end of the paragraph is reached (i.e. there are no more instances of the search string in the line) FIND again waits for a key to be pressed. This time pressing escape will terminate the search, while pressing any other key makes FIND continue its search with the next paragraph.

A single space in the search string can be used to represent any number of spaces in the source file. The search routine differentiates between upper and lower case. Opcode mnemonics and Z80 operands should be entered in upper case. The search string cannot contain the double quote character.

REPLACE “<string l>”, “<string 2>” [,<line range>]

Searches for occurrences of <string l> in the source text (within the specified line range) and gives you the opportunity to replace each instance found with <string 2>. REPLACE does this by first finding an occurrence of <string 1> and listing the paragraph containing it to the screen (just as in FIND). Then the message

```
        ** replace? (Y/N) **
```

will be inserted above the statement in which <string l> has been found and the cursor placed at the start of the string. Answering “Y” (for “yes”) will cause the string to be exchanged for <string 2>. Answering “N” (for “no”) will make REPLACE search the rest of the listed paragraph for further occurrences of <string 1>.

Replacement as described above can create syntax errors in the altered statement. Should this occur, the usual syntax error message for the error will be inserted above the bad statement, REPLACE will move onto the next statement, and the new version of the line will NOT be inserted into the source text. (The old version will remain).

REPLACE can be halted at the end of a paragraph by pressing escape (as for FIND).

NEXT

This command may be used to continue a FIND or REPLACE which has been halted by pressing the ESC key. The search is continued from the paragraph after the last one listed. NEXT will continue in FIND or REPLACE mode according to which was the last executed. The search and replace strings will be the last ones specified. and the line range bounding the search will be that given in the last FIND or REPLACE command.

## *2.4 The Calculator*

PRINT <expression> [,<base>]

The PR INT command gives you a calculator facility from within the editor. It prints the value of the expression given in the base specified (if no base is specified then the default value set by the BASE command is used). PRINT uses twos complement signed arithmetic to evaluate the expression. The result is given in the range -32768 to 32767.

NOTE: Expressions are evaluated using an operator precedence technique, for instance:

```
2+3*4
```

is evaluated as:

```
2+(3*4)
```

For a full description of the available operators and their precedences see Appendix D.

UPRINT <expression> [,<base>]

As for PRINT but uses unsigned arithmetic and gives the result in the range 0 to 65535.

NOTE: Both the commands PRINT and UPRINT have access to the assembler symbol table (see 3. The Symbol Table and 4. The Assembler) and can be used to find the values of any of these symbols or expressions involving them.

BASE <2, 8, 10 or 16>

Set the default base used by PRINT and UPRINT. This will be 16, hexadecimal, if the BASE command has not been used. Only the integers 2, 8. 10 and 16 will be accepted as legal bases.

## *2.5 Printer Commands*

FORM

Issue a form feed to the printer.

WIDTH [<decimal

Set the width of the printer page in characters. This defaults to 65536.

integer>]

LENGTH [<decimal integer>] Set the length, in lines, of the printer page. This defaults to 65536. A form feed is issued after the given number of lines have been printed.

MARGIN I<decimal integer>] This sets a left hand margin so that when a line reaches the width of the page it is continued on the next line but starting at the margin position given.

## *2.6 Tape. Disc and Microdrive Commands*

All the commands in this section require a filename as one of the parameters. This is represented in the text by "<string>". The commands refer to tape, disc and microdrive. Input and output devices are selected by the commands given at the end of this section or by filename specifiers included in the filename string. These take the following form:

**a) Amstrad Users**

| | |
|---|---|
| "A:filename" | Use drive A |
| "B:filename" | Use drive B |

**b) Spectrum Users**

| | |
|---|---|
| "1:filename" | Use microdrive number 1 |
| "2:filename" | Use microdrive number 2 |
| … | |
| … | |
| "8:filename" | Use microdrive number 8 |

**NOTE:** Using the above file specifiers overrides the default input/output selections but does not affect them. There should be no space between the drive specifying letter/number and the colon.

For Spectrum users: Should an error occur during tape or microdrive operations, control will return to BASIC. To re-enter the assembler, type:

```
RANDOMIZE USR 65533
```

All tape/microdrive commands can be halted by pressing the SPACE key.

The following commands refer to tape, disc and microdrive.

SAVE ["<string>" [,<line range>]] Saves the given range of source text to a file named <string> on tape, disc or microdrive.

If no file name is given then the name used will be that specified in the last LOAD command executed.

If the LOAD command has not been used then the message:

```
** no file loaded **
** (press ESC) **
```

will be issued.

For Amstrad users: Should an error occur during saving an error message of the form:

```
** tape/disc message **
drive A: discfull
** (press ESC) **
```

will be issued. The message will be deleted when you press the escape key.

LOAD "<string>" [,<line range>] Attempts to load a file named <string> from tape/disc or microdrive and replaces the text in <line range> with the contents of this file. If there is insufficient space for this an error message will be issued. You can allot more space for source text using the SET SPACE command if necessary.

If the line range is not given then it defaults (as usual) to 0-65534, thus causing the whole of the current text file to be replaced.

If the line range given does not represent the whole of the current source text, then after loading the source will be renumbered from 1 in steps of 1 to prevent inconsistencies in the line numbering.

VERIFY “<string>”
[,<line range>]

Attempts to verify a source file named <string> or a section of a source file named <string> against the source file in memory or part of the source file in memory. If verification fails then an error message is displayed:

```
        ** verification failed **
        ** (press esc) **
```

CODE “<string>”,
<expression l>,
<expression2>

Saves a block of memory to tape/disc or microdrive as a binary file named <string>. Expression 1 is the start address of the block, expression 2 is the end address. These expressions may contain label references from the last piece of code assembled. Thus if you have a label “start” at the start of your code, and a label “finish” at the end, you can write:

```
        CODE “file”, start, finish
```

to save it. However, please note that if you have used the PUT assembler directive (see 4. The Assembler) then you will have to adjust the label values to point to the position of the generated code in memory, rather than the execution position of the code.

LOAD ASCII
“<string>”
[,<options>
[,<decimal integer>lj

**NOTE:** This is a tool-kit command: it is only available if you loaded the tool-kit with the assembler/editor.

This command is provided to allow you to load source files which are not in the editor’s tokenised form. In particular it allows you to convert source files written for other assemblers to Genius files.

The command attempts to open a file named <string> for input from tape, disc or microdrive. The <options> parameter is a decimal integer telling LOAD ASCII the format of the file to be loaded. This integer is constructed as follows: Add together the values describing your source file from those below:

1 File is “pure ASCII”, i.e. do not add 1 into your options if your file contains line numbers in hex/ binary. Note that if you select this option LOAD ASCII will treat #1A, the CP/M end of file marker, as the end of file.

2. Add 2 to your options if the lines in your file are terminated by a carriage return (ASCII 13) and a line feed. Otherwise LOAD ASCII assumes that each line is terminated by a carriage return alone.

4 Add 4 to your option integer if your file does not have colons after its labels. This will allow-the syntax checker to accept such statements.

8 Add 8 to your options to make LOAD ASCII skip the first four bytes of the input file.

Example option integers:

7 will load pure ASCII files with lines terminated by both a carriage return and line feed, such as those produced by ARNOR’s MAXAM for the Amstrad.

8 will load files with hex line numbers and lines terminated by just a carriage return such as those produced by HI-SOFT’s DEVPAC for the Amstrad.

4 will load files with hex line numbers and lines terminated by first a carriage return. Colons will not be expected after labels. This option will cover files produced by the Picturesque assembler for the Spectrum. The file being loaded is listed to the screen as it loads. If any of the statements found in the file is illegal then the usual message for the error will be issued and loading will be halted. Pressing ESC will terminate LOAD ASCII. Pressing any

other key will cause a semicolon and the message “** bad line **” to be printed at the start of the incorrect statement and it will then be entered as a comment. Loading will then continue. The “** bad line **” message allows you to find such lines later using the FIND command.

CAT — Display the current tape/disc/microdrive directory in alphabetical order (not Spectrum tape).

The following commands are available only to disc/microdrive users. For more information AMSTRAD users should refer to the disc operating manual.

ERA “<string>”
Erase files whose names match <string> (Amstrad users may incorporate wildcards. see your AMSTRAD disc operating manual).

TAPE.IN — Direct the firmware to take input from tape.

TAPE.OUT — Direct the firmware to write to tape.

TAPE — Direct the firmware to read from, and write to, tape.

The following commands apply to Amstrad disc only:

ERA <string l>”, ”<string 2>”
Rename a file named <string2>, <string1>. Neither string may contain wildcards (Amstrad disc only).

DRIVE A — Set the current drive to “A”.

DRIVE B — Set the current drive to “B”.

DISC.IN — Direct the firmware to take input from disc.

DISC.OUT — Direct the firmware to write to disc.

DISC — Direct the firmware to read from, and write to, disc.

The following commands refer to Spectrum microdrive only:

MDRV [<microdrive number>]
Direct the firmware to read from, and write to, the selected microdrive. If no microdrive number is specified, input/output defaults to drive 1.

MDRV.IN — Direct the firmware to take input from microdrive only.

MDRV.OUT — Direct the firmware to send output to microdrive only.

## *2.7 Assembly Commands*

ASSEM — Command to assemble the current source text. Assembly is controlled by “assembler directives” embedded in the source text. See “4. The Assembler” for a full description of this command.

The current symbol table is cleared (i.e. made empty) before assembly starts.

ASSEMC — This command is the same as ASSEM except that the assembler’s symbol table is not cleared before assembly starts. This allows you to continue a previous assembly or to “link” pre-written and assembled routines into the current source. (See “2.8 Symbol Table Commands”, especially EXPORT, IMPORT, REDUCE, CLEAR).

ASSEML — This command assembles selected routines from a subroutine library. The procedure for using ASSEML and the circumstances under which it should be used are now described in detail.

**NOTE:** This is a tool-kit command: it is only available if you loaded the tool-kit with the assembler/editor.

### 2.7.1 Selective Assembly

Most assembly language programmers find that when starting a new project there are a lot of routines required that have been written for use with previous programs. It is quite common to build up a subroutine library and then just pull out those routines that are required. Sometimes the subroutines are split into distinct chunks but more often than not there is a hierarchical structure to the program which requires a painstaking and time consuming search through the code followed by an equally painful session with the editor. This facility is provided to automate the whole process. The graphics library used by Laser BASIC will shortly be available for use with the Genius system.

ASSEML selectively assembles the current source file according to the contents of the current symbol table. It is probably best to illustrate its use by example. Suppose you are embarking on a project that uses the GTBL, PTBL, INVV and MIRV Laser BASIC routines.

The procedure is as follows:

(i) Load the main program containing the references to GTBL. PTBL etc.

(ii) Assemble in the normal way using ASSEM or ASSEMC. In fact pass 2 should throw up symbol undefined errors for each of the labels GTBL, PTBL etc. so it's worth putting a +report off directive at the start of your file before assembling (see Section 4.2).

(iii) Load the subroutine library source file in the normal way.

(iv) Assemble using ASSEML (assemble library). Note that the library file should contain a valid ORG and that object code output is controlled by the usual directives. The library file may contain +includes and any other directives required (see 4. The Assembler).

(v) When assembly is complete an object file will have been produced in the normal way but the symbol table will contain only those labels that were passed as "missing" by the main program. This symbol table should be saved off using the EXPORT command (see Section 2.8).

(vi) The main program (containing references to the library) can now be assembled by loading the symbol table (which was EXPORTed after the ASSEML), unless it is already in memory, and then assembling with the ASSEMC command. Any further assemblies of the main program will not require steps (i) to (v) but instead, only the symbol table need be loaded.

**NOTE:**

(i) After assembly, the main program cannot be executed unless the object code produced by the assembly of the library routines is resident in memory.

(ii) If you wish the two assembled files to run contiguously then you will need to know the end address of one of the assembled files. It is easier to configure the final program so that the library code precedes the main program code. If you wish to do this then type: PRINT $ after assembling the library routines and ORG your main program to this value. This will mean that you can edit and assemble the main program without re-assembling the library.

(iii) Assembling a subroutine library may take a considerable number of passes. In fact the number of passes required depends on the nesting depth of backward references in the subroutine library. For this reason subroutine libraries should be designed to minimise backward references. If your subroutine library does not contain any *include directives then assembly can take place entirely in memory which will greatly speed up the process. Tape users assembling files which contain +include directives will need to rewind the tape after each pass so particular note should be taken of these points. The end of each pass is indicated in the normal way.

(iv) This process can be used to check if a program contains any redundant code. Most programs have only one entry point and so if the main program is merely a call to this entry point and the program itself is treated as a subroutine library then the assembled code will be at variance with a straight assembly if any redundant code were contained within it.

## *2.8 Symbol Table Commands*

The following commands allow you to view and alter the contents of the symbol table produced by the last assembly of source code. For more information on the symbol table and how it is produced see "4. The Assembler" and "3. The Symbol Table".

TABLE [<decimal integer>]

Lists the current symbol table in ASCII order (i.e. ordered lexically by ASCII code of the characters in the names). Each symbol has an entry of the form:

```
<symbol> 5FFE
```

The number on the right of the entry is the value of the symbol in hexadecimal. Only those symbols which the programmer has attempted to define in the source code will be listed by TABLE i.e. symbols which are referenced in the source but not defined anywhere will not appear (see the command MISSING). Symbols which the programmer has unsuccessfully attempted to define will appear with an asterisk before their value. For instance, if the statement:

```
name3:EQU name1 / name2
```

appears and "name2" has value zero, then name3 will be undefined and will be listed by TABLE as:

```
name3     *0000
```

The integer which may be given as part of the TABLE command sets the width of the field in which each symbol is printed. If a symbol is longer than the field given it will be truncated to fit. The field width defaults to 16. TABLE will list as many symbols on one line as will fit with the current field.

`LTABLE` [<decimal integer>]
As for TABLE but sending it's output to the printer.

`TABLEN` [<decimal integer>]
Lists the current symbol table in numerical order (i.e. ordered by the values of the symbols in the table).

The optional integer sets the field width in which each symbol will be listed (see TABLE).

`LTABLEN` [<decimal integer>]
As for TABLEN but sending its output to the printer.

`MISSING` [<decimal integer>]
MISSING lists to the screen all those symbols which the programmer has referenced but not defined in the last piece of source code assembled. The optional integer sets the field width in which each symbol will be listed (see TABLE).

`LMISSING` [<decimal integer>]
As for MISSING but sending its output to the printer.

`UNUSED` [<decimal integer>]
UNUSED lists to the screen all those symbols which the programmer defined (or attempted to define) in the last piece of source code assembled, but did not actually reference. The optional integer sets the field width in which each symbol will be listed (see TABLE).

`LUNUSED` [<decimal integer>]
As for UNUSED but sending its output to the printer.

`CLEAR`
Clears the current symbol table, i.e. makes the current symbol table "empty".

**NOTE:** The following three commands are tool-kit commands: they will not be available unless you loaded the tool-kit with the assembler.

`EXPORT` "<string>"
Saves the current symbol table to a tape, disc or microdrive file named <string>. This can be reloaded using the IMPORT command. For more information and possible uses of this command see "3. The Symbol Table".

`IMPORT` "<string>"
Merges a previously EXPORTed symbol table with the current table. If in doing this a name is multiply defined a message of the form:

```
** multiple name definition: <name> **
```

will be issued and the current value of the symbol will remain in the table. For more information on this command, see "3. The Symbol Table".

`REDUCE`
This command is used in conjunction with the CARGO assembler directive to select certain symbols in the current table for EXPORTing.

REDUCE removes all symbols which were not specified as CARGO during the last assembly. This allows you to save principal entry points (and macro definitions) only, instead of the whole table which may contain many labels only of use within the program and not as entry points.

For more information on this command see “3. The Symbol Table”.

## *2.9 Miscellaneous Commands*

EXECUTE
<expression>
[,<parameter list>]

Call the code at the address to which the expression evaluates. As usual the expression may contain label references, so that you can call a particular routine in your code easily.

If a parameter list is given then the values of the parameters are made available to the code being EXECUTEd. This is done in the same way as AMSTRAD BASIC’s CALL command passes parameters.

Each parameter is passed as a 16 bit integer. For an expression parameter this is the twos complement value of the expression. For a string parameter it is the address of a descriptor for the string.

A string descriptor is three bytes long. The first byte contains the length of the string. The following two give the address of the start of the string.

On entry to your code the A register will hold the number of parameters given to the EXECUTE command. The IX index register will contain the address of the parameters. The parameters are stored in reverse order; so that, if n parameters are given, the i$^{th}$ one will be at IX+2*(n-i) (low byte) and IX+2+(n-i)+I (high byte). (Where i runs from 1 to n).

As an example we will pass one string parameter and one number to a piece of code.

The command typed might be

```
EXECUTE#1000, “Call me”, 39
```

Then the code at #1000 could be as below:

```
CP 2        ; Make sure that two
JP NZ,error ; parameters have been given.
LD L,(IX+2) ; HL = address of string
LD H,(IX+3) ; descriptor
LD B,(HL)   ; B = length of string.
INC HL
LD A,(HL)
INC HL
LD H,(HL)
LD L,A      ; HL points to start of string.
LD E,(IX+0)
LD D,(IX+1) ; DE = value of second parameter.
```

The address given is executed using a Z80 “CALL” instruction and control should therefore be returned to the editor by a “RET” instruction. If this is done successfully the editor will print the message “** (press ESC) **” in the bottom left corner of the screen. When you press the escape key the screen will be cleared and the editor will reprint the contents of the screen as it was before EXECUTE was used.

STATS

This command lists some information on Genius’ memory map to the screen. The data given is as follows:

```
** buffer <addr>
** screen <addr>
** file <addr>
** table <addr>
** low <addr>
```

buffer: The address of the start of Genius’ tape/disc buffer above which the program starts.
screen: The address of Genius’ screen buffer.
file: The address of the current source file.

table: The address of the current symbol table.
low: The low limit, below which the assembler will not let you PUT object code.

**NOTE:** Memory between low and table is available for assembling into memory (see ‘PUT - Section 4.3.5’).

`EXIT` Returns control to the program which called Genius (usually BASIC) OR exits from the housekeeping facility and returns to the main editor (see 2.10).

## *2.10 The Housekeeping Facility*

`HOUSEWORK` Enters the housekeeping facility. Use the command EXIT to return to the main editor.

This facility is provided to allow you to look after the contents of your discs/microdrives. When you enter the HOUSEWORK command, the screen will be split into two windows. The bottom four lines of the screen will now be used for command entry; the usual full-screen editor runs in this foul line window. The top part of the screen is now used for output, initially the current disc/microdrive (or tape on the Amstrad) is CATed into this

window.

The following commands from the main editor’s immediate mode are still available to you in the house keeping facility. Their syntax of use is the same as in the main editor (see 2.6).

AMSTRAD:

```
CAT
ERA
REN
TAPE
TAPE.IN
TAPE.OUT
DISC
DISC.IN
DISC.OUT
DRIVE.A
DRIVE.B
```

SPECTRUM:

```
CAT
ERA
TAPE
TAPE.IN
TAPE.OUT
MDRV
MDRV.IN
MDRV.OUT
```

The following new commands are available to you in the housekeeping facility:

`COPY` “<filename 1>” , “<filename2>” Copies <filename2> to a new file named <filename1>. You will be prompted for source and destination disc/tape or microdrive as necessary.

On the AMSTRAD: ASCII files will be copied character by character. Other files will be loaded into memory and saved under the new name. The file will be loaded from #40 upwards and will therefore overwrite anything in memory from this address to the bottom of the symbol table. If there is not enough room to load the file here you will be asked if you want to delete the symbol table to release some space for the load. If this does not make enough room, the file cannot be copied.

On the SPECTRUM: On tape COPY will only copy files produced by the Genius assembler (source, object or name table).

On microdrive the file will be loaded from RAMTOP upwards and will therefore over write anything in memory from this address to the bottom of the symbol table. If there is not enough room to load the file here you will be asked if you want to delete the symbol table to release some space for the load. If this does not make enough room, the file cannot be copied.

**AMSTRAD only:**

`<RSX name>` [,<parameter list>] This command gives you access to any RSXs which are currently logged on, including tape and disc commands. See your Amstrad manuals for the syntax of these RSXs.

CPC 464 users should note that string parameters must be passed directly as in the following example.

```
|REN,“PEQUOD”, “ACUSHNET”
```

In BASIC this would have to be done as below:

```
A$=”PEQUOD”
B$=”ACUSHNET”
|REN,@A$,@B$
```

# 3. The Symbol Table

**NOTE:** In the following the terms “symbol”, “name” and “label” are used interchangeably. Names can consist of alphabetic characters (upper and lower case); (decimal) digits; the characters “$”, “.”, “”. A name may not start with a digit. Names may be up to 240 characters in length.

The Genius assembler makes two passes over source code which it is assembling. On the first pass it constructs a table of all the names which you have defined or referenced in the source. As far as possible the assembler will give each of these symbols a value. (In the case of a label on an ordinary 280 instruction this value is the current location, i.e. the address at which object code is being assembled; in the case of an EQU pseudo-op it is whatever the expression operand of EQU evaluates to).

On the second pass the assembler will fill in the values of any symbols which it could not calculate on pass 1 and use the final values to generate the object code.

The Genius assembler actually creates more than one symbol table in certain circumstances. Usually all names are inserted into a “global” table; however, during macro expansion and (hash extension) function definition a “local” table is used. A new local table is created for each use of a macro, and for each function. This prevents different expansions of a macro in which a symbol is defined from producing multiple definition errors. It also means that different functions can use the same names for local variables.

Macro definitions are also stored in a special type of local table.

The symbol table remains in memory even after assembly has finished. You can access the table and the values of symbols in it using various immediate commands. For example, you can PRINT the value of a label or an expression involving symbols (see 2.4). You can view the whole table by use of the TABLE or TABLEN commands (see 2.8).

These facilities are useful for debugging purposes when you may need to know the address of a particular piece of your code for single stepping or setting a breakpoint.

## *3.1 Saving and Loading Symbol Tables*

Suppose you have written a library of useful machine code routines which you want to use in a program you are writing. You could do this by merging the source of the library with that of your program (using LOAD, see 2.6); or you could use the *include directive (see 4.2.3). With both these methods the assembler has to cope with the extra source at each assembly.

An alternative to these methods is given by the immediate commands EXPORT and IMPORT. The EXPORT command will save the current symbol table to backing store, and IMPORT will merge a previously EXPORTed table with the current one. So after assembling your library routines you can save the symbol table generated using EXPORT. Then before assembling your program which uses the library you can IMPORT the library’s symbol table. Assemble your program’s source using ASSEMC instead of ASSEM to prevent the symbol table being cleared. Your source can refer to symbols in the library source and since they now appear in the symbol table this will be acceptable to the assembler.

Many routines contain labels only of use within the routine itself (for instance, loop start markers). Your library of routines may contain only a small number of labels which you want to be able to refer to in other programs (the main entry points to the routines in the library). EXPORT, however, saves the whole symbol table including these unwanted labels. It is possible to save just the labels you want as entry points to your routines by using the CARGO assembler pseudo-op and the REDUCE immediate command.

For every symbol in the library that you wish to save an entry for in an EXPORTed table you should include a statement of the form:

```
        <label>: CARGO
```

in the source. (Note that this will not cause a multiple definition error when the symbol is actually defined elsewhere).

The REDUCE command will remove from the symbol table any name that has not been declared as CARGO during assembly. EXPORT can now be used to save off the table consisting of just CARGO labels.

EXPORT also saves information recording those Phoenix library routines which have already been referenced. Thus you can EXPORT a table generated by the assembly of some Phoenix routines and use them in later programs without the compiler including extra copies of its run-time library routines.

(See “6. Example Programs” for an example of using CARGO, REDUCE, EXPORT and IMPORT).

(See 2.7.1 for an alternative method of dealing with a library of routines).

**NOTE:** This section is included for interest only. It gives a little more detail on the actual format of the symbol table.

Each name has an entry in the table of the following form. A string of characters representing the name itself; two bytes of flags giving the type of the name and its status (well-defined, unused etc.); two bytes containing the value of the name. So each name requires its own length plus 4 bytes of symbol table space.

Each table is kept in ASCII order. A name is inserted by finding the position at which it should be inserted (this is done by the symbol table search routine). Then inserting enough room for it by block moving the part of the table below the insertion position down in memory (see Appendix E). The new name is then copied into the space made for it.

Since each table is kept in ASCII order it is possible to use a fast binary search technique for looking up names in a table.

## *4.1 Using the Assembler*

The assembler is invoked using the ASSEM, ASSEMC or ASSEML commands from within the editor. It assembles the current source in two passes, writing object code to memory, tape, or disc (microdrive for the Spectrum versions) as specified by assembler directives embedded in the source text. Source files on tape, disc (or microdrive) can also be assembled by use of the "*include" directive. (See "4.2 Assembler Directives").

Assembly can be temporarily halted at any time by pressing any key. It can then be stopped by pressing ESC or continued by pressing any other key.

### 4.1.1 Assembler Errors

The assembler informs you of any errors it has found by displaying a relevant message; a line giving the type of error, the line number and the number of the statement within that line in which the error was detected; the statement at that line.

For example:

```
** multiple name definition
** error in line 20: 1
            20 label:equ#1234
```

After the error message has been issued the assembler rings the bell (beeps). It will now either wait for you to press a key or just continue assembly. The default behaviour being to wait for a key press. Pressing ESC will abandon assembly, pressing any other key will make the assembler continue. The "*report" directive can be used to stop the assembler waiting after a message has been written. (See "4.2 Assembler Directives").

The possible error types and their meanings are given below.

| | |
|---|---|
| warning: | Indicates that the code produced may not be what was intended. Assembly will go through to the end. |
| error: | Indicates a more serious fault than a "warning". The resulting code would probably not be usable if assembly continued, so if any errors have been detected on pass 1 assembly will be halted at the end of this pass. |
| fatal error: | The assembler cannot continue, assembly is stopped immediately. This can occur, for instance, if the disc becomes full whilst writing object to a file, or the assembler runs out of space for the symbol table. |

For a complete list of assembler errors and their meanings see Appendix C.

At the end of each pass the assembler issues a message telling you the pass number; how many warnings were given during the pass; how many errors were detected during the pass; as below.

```
** pass <pass number>
** warnings <warning count>
** errors <error count>
```

A small number of errors can be issued at the very end of pass 1. In this case the error message is written just before the end-of-pass message above. For instance:

```
** code in name table **
** pass         1
** warnings     0
** errors       0
```

The possible errors at this time are:

```
** code too high **
** code in name table **
** code in program **
** bad nesting **
```

See Appendix C for the meanings of these messages.

### 4.1.2 A Note on Assembler Strings

Strings occurring in assembly language statements may contain characters defined by their ASCII codes rather than the character itself. This is done by preceding the required code with a “\” character as in the following example:

```
DEFB “Hello, world. \13 \10”
```

The string given here is terminated by a carriage return (ASCII 13) and a linefeed (ASCII 10).

Because of this use of the “\” character to include one of these it is necessary to put two in a row into the string. For example:

```
DEFB “A string with a\\”
```

will generate

```
A string with a \
```

in object code.

Since the double quote character (") is used as a string terminator, to include one of these in a string you must also precede it with a “\”. For example:

```
DEFB "A character constant is like \“A\””
```

will generate

```
A character constant is like “A”
```

in object code.

These comments also apply to character constants. So

```
“\13”
```

is a legal character constant. This can be preferable to just using the number 13 as it suggests to anyone reading the code that you intended this as a character (a carriage return).

## *4.2 Assembler Directives*

All assembler directives start with an asterisk (“*”).

Those directives which act as a switch. i.e. turn a facility on or off, have a default setting which is given in parentheses at the end of the directive’s description.

### 4.2.1 Output Directing

```
*screen on
*screen off
```

Allow or disallow output of any kind to the screen (including error messages, listing, end-of-pass messages).

(Default is “on”).

```
*printer on
*printer off
```

Allow or disallow output to the printer. Turning this facility on will direct error messages and end-of-pass messages to the printer. This does not affect output to the screen, i.e. you can have output to the screen and the printer together or you can turn the screen output off.

(Default is “off”).

### 4.2.2 Listing

```
*list on
*list off
```

Turn listing to the screen of the source code during assembly on or off.

This option is only effective if output to the screen is enabled. This is the default case.

(Default is “off”).

So to get a listing during assembly of your code (to the screen) you will just need to have the following:

```
*list on
...
... (your code here)
```

```
...
*list off
```

The “*list off” is not necessary if you want all the code to the end of the source listed.

The listing produced during assembly by use of a “*list on” directive contains more information than just your source code. Each statement is listed in three fields. These are illustrated below.

For statements which generate code:

<current location> <code generated> <the statement>

The current location is given as a hexadecimal word and the code generated as hex bytes.

**Note:** Hash extension will have the “code generated” field as many of them can create a lot of code, e.g. the #DUE pseudo-op (see 5. The Hash Extensions).

Example:

```
0100 DD360A47 LD (IX+71),10
```

If the statement has a paragraph number attached to it this will be listed between the code generated and statement fields.

For statements which generate no code:

<value of the statement> = <the statement>

Example:

```
BB5A= TXT_OUTPUT :EQU #BB5A
```

As exceptions to this rule *list, *llist and *include will both be listed with the current location in the left-hand field. This allows you to determine particular addresses in your code easily.

```
*llist on
*llist off
```

Turn listing (to the printer) of the source code during assembly on or off. This does not affect output or listing to the screen.

This option is only effective if output to the printer has been allowed by a “*printer on”.

(Default is “off”).

So to get a listing to the printer of some of your code, during assembly, you will need the following:

```
*printer on
*llist on
...
... (your code here)
...
*llist off
*printer off
```

The “*llist off” and “*printer off” are both unnecessary if you want these options to be “on” to the end of the assembly.

```
*form
```

Sends a form feed character to the printer if the printer has been enabled by a “*printer on”.

`*title` “<string>” [,<expression>]

Sets a heading to be printed at the top of each (printer) page to be the string given. The expression, if given, will be used to set the current page number.

```
*maclist on
*maclist off
```

If listing is on then using “*maclist on” turns on the listing of source by macro expansions. If this facility is off then just the statement containing the macro use will be listed.

When macro listing is turned on the statement containing the macro use will be listed AFTER the source generated by the macro expansion.

(Default is “off”).

### 4.2.2 The Tape/Disc/Microdrive Facility

`*include`
“<filename>”

Causes the assembler to open the specified tape, disc or microdrive file and assemble its contents at the current location. The file must be a Genius source file. *includes may not be nested, i.e. a source file which is *included should not contain a *include directive. The file will be loaded from the current input device unless a specifier is provided in the filename to direct otherwise.

Assembly is continued from the statement after the +include when the file is exhausted.

Any errors detected during a “*include” will be reported in the usual way but with an extra line giving the name of the file in which the error occurred. An example error message is given below.

```
** multiple name definition
** error in line  50:  2
** in file: “phoenix2”
```

The state of the options already chosen by you (such as *list, *llist) is not affected by the “*include” command. Thus it is not necessary to put a “*list” or any other option switch in the included file itself, they can be put in the main source file.

For example, if you are assembling the source:

```
10 *include “phoenix1”
   *include “phoenix2”
```

and you want to list (to the screen) all the source in the two files you could put the following:

```
10 *list on
20 *include “phoenix1”
   *include “phoenix2”
```

If, however, you only want to list the contents of “phoenix1” then you should put directives in as below.

```
10*list on
20 *include “phoenix1”
   *list off
   *include “phoenix2”
```

Any changes to the switch states made in an included file will remain in effect after the “*include” is complete. Thus if “phoenix1” contains a “*list on”, the source

```
10 *include “phoenix1”
   *include “phoenix2”
```

will give a listing of both files.

If listing to the screen or to the printer is turned on while a file is “*included the statement containing the “*include” will be listed AFTER the source text in the file.

**Using *include with Tape**

The file to be included must be read on both passes of the assembly process. Thus the tape will need to be rewound at the start of pass two. Alternatively you can record two copies of the source file(s) on your tape one after the other, one for each pass.

`*openout`
“<filename>”

Causes the assembler to open an output file and write subsequently generated object code to that file. Output must be terminated by a “*closeout”. Output will be sent to the correctly selected output device unless the filename contains a specifier to direct otherwise.

Care should be taken when writing object code to file storage if the source contains more than one ORG statement (see "4.3 Assembler Pseudo-ops"). The object code will be written to the file as a contiguous block. In this case it may be better to write blocks of code with different ORG addresses to different files. If you intended only to separate two blocks of code by some space then you could use a DEFS pseudo-op instead of an ORG, and still write all the code to one file.

A warning will be issued on pass one by the assembler if it finds a PUT directive while sending output to a file. The PUT will be ignored.

`*closeout`

Terminates output of object code to a tape or disc file. This directive must be specified if an *openout has been employed.

**Amstrad Tape Users:**

The source file is loaded in 2K sections and between blocks you will be prompted to "press PLAY then any key". Filenames which are preceded by an exclamation mark will load without prompts.

If you are also outputting the assembled file to tape then the object code will also be written out in 2K blocks. Before the assembler writes out object code you will be prompted with "press REC and PLAY then any key". You should now insert the tape you are saving the object code onto and press REC and PLAY followed by any key. Once this is complete you will be prompted again for more source code (unless the file is exhausted) i.e. "press PLAY then any key". This cycle is repeated until assembly is complete but you will probably find that you are prompted more frequently to load source, than save object.

**Spectrum Tape Users:**

The source file is loaded in 2K sections and between blocks you will be prompted to "** insert source tape, press a key **". You should insert the source tape, press PLAY on your tape recorder and then press any key. If you delay too long before pressing a key, the tape may run over the start of the block. If this happens the block will not load. You will need to rewind the tape a little and attempt to load the block again. Once the block has loaded you will be prompted to "** stop tape **", again you should respond as soon as possible. After a delay, the block will be assembled and you will be prompted for the next block with "** insert source tape, press any key **", and so on.

If you are also outputting the assembled file to tape then the object code will also be written out in 2K blocks. Before the assembler writes out object code you will be prompted with "** insert object tape, press any key **". You should now insert the tape you are storing the object code onto and press REC and PLAY followed by any key. Once the block has been written out you will be prompted with "** stop tape **". You should now remove the object tape and re-insert the source tape ready for the next block of source code (unless the source file is exhausted). This cycle is repeated until assembly is complete but you will probably notice that you are prompted more frequently to load source, than save object.

`*prompts on`
`*prompts off`

(Spectrum only). This option can be used to enable or disable prompts for source and object microdrives. If you want to read source from and write object to the same microdrive(s) throughout assembly AND these microdrive(s) both have the correct cartridges in them when you type your assembly command: then you will not need to be prompted to insert the source/ object microdrive(s). In this case you should put a "*prompts off" in your source.

Otherwise whenever the assembler needs to open a file it will prompt you to insert the correct cartridge. If you are writing to and reading from the same microdrive then you will be prompted for the correct cartridge every time the assembler needs to read/write a 2K block of source/object.

(Default is "on").

### 4.2.4 Miscellaneous

`*count on`
`*count off`

If this option is specified and the listing is off, a continuous display of the number of the line currently being assembled will be given. If the listing is on, this directive is not effective.

(Default is "off").

`*report on`

If this option is specified the assembler will wait for a key to be pressed `*report off`

after it has generated an error message to give you time to take note of the content of the message. If this option is off the assembler will just continue. The assembler always rings the bell when an error is detected.

(Note that assembly can be paused at any time by pressing a key, and terminated by pressing the ESC key while the assembler is pausing).

(Default is "on").

`*code on`
`*code off`

This directive causes the assembler to switch its production of object code on and off. A "*code off" directive will halt the production of object code when it is encountered.

This facility can be used to see if your source assembles correctly without actually generating any object code. Doing this can speed the assembly process when you would normally be writing the object code to tape, disc or microdrive; as these devices will not need to be accessed for output.

(Default is "on" i.e. object code will be produced).

`*print` "<string>"

This directive will print the string given to the screen and printer (if they are enabled, see 4.2.1) on both pass 1 and pass 2.

You can use this directive to keep track of where you are in an assembly, or, with the "pause directive, to allow you to change source or object tapes, discs or microdrives at appropriate points.

`*pause`

This directive causes the assembler to stop and wait for a key to be pressed (on both passes).

### 4.2.2 Looping

`*while` <expression>

Marks the start of an assembler "while" loop, which must be terminated with a "*endw". The assembler will evaluate the expression and if it is TRUE (i.e. non-zero) the statements up to the *endw will be assembled. Otherwise assembly will be turned off for the duration of the loop and resumed after the endw. while loops may be nested. (Labels should not be defined inside a loop which will be executed more than once as this will cause a multiple definition error. See, however, the pseudo-op DL.).

See also the pseudo-ops COND, ELSE, ENDC.

`*endw`

Causes control to be passed back to the "*while" matching this *endw.

`*repeat`

Marks the start of an assembler repeat loop, which must be terminated by a "*until".

`*until` <expression>

The assembler evaluates the expression and, if it is FALSE (i.e. zero) control is passed back to the *repeat matching this *until. Otherwise assembly is continued from the statement after the *until.

## *4.3 Assembler Pseudo-Ops*

### 4.3.1 Initialising Memory

`DB` <byte definition list>
`DEFB`
`DEFM`

These pseudo-ops are used to initialise bytes at assembly time. They are all the same and are given to increase Genius' compatibility with other assemblers.

e.g. `DB 1,2,3,4`

causes 4 bytes at the current location to be initialised to 1,2,3,4 respectively.

e.g. `DB "Hello, world."`

causes the string "Hello, world." to be placed in memory at the current location, one character per byte. (See 4.2.2 "A Note on Assembler

Strings".)

DW <word definition list>
DEFW

These directives are used to initialise words (pairs of bytes) at assembly time.

e.g. `DW 1,2,3,4`

causes 8 bytes starting at the current location to be initialised to 1,0,2,0,3,0,4,0 respectively.

e.g. `DW #1234`

causes two bytes at the current location to be initialised to #34, #12 respectively, i.e. the bytes are installed in low-byte, high-byte order as is usual with Z80 machine code.

Strings cannot be given as operands to these pseudo-ops.

### 4.3.2 Allocating Space

DS <expression>
DEFS

These pseudo-ops are used to leave a block of space in the object code generated by the assembler. The assembler evaluates the expression given and increments the current location counter by that amount.

e.g. `DS 4`

causes the assembler to leave 4 bytes of space.

The space left is initialised to zero by the assembler.

Note that if the expression given contains symbol references they must be well defined on pass 1 else the program counter cannot be correctly updated by the assembler. If this is not so a fatal error results.

### 4.3.3 Assigning Values to Symbols

<label>: EQU <expression>

This pseudo-op is used to define and initialise a new symbol. No code is generated by this instruction.

e.g. `SPACE: EQU 32`

defines a symbol "SPACE" whose value is 32. This can be used extensively to make your code more readable.

An attempt to redefine a label using EQU will result in a multiple definition error. (See also pseudo-ops DL and DEFL).

<label>: DL <expression>
DEFL

These pseudo-ops have the same function as EQU but may be used to redefine the value of an existing symbol without causing a multiple definition error. In particular they may be used within an assembler 1oop.

e.g.
```
cnt   :EQU 1
      *WHILE cnt<=10
       DB cnt*cnt
cnt   *DL cnt+1
      *ENDW
```

will generate a table of squares of the numbers from 1 to 10 in memory.

No code is generated by these pseudo-ops.

### 4.3.4 Conditional Assembly

COND <expression>

The assembler evaluates the expression and, if it is TRUE (i.e. non-zero) assembly continues with the statement after the COND. If it is false assembly is turned off until an ELSE or an ENDC matching this COND is encountered. There must be a terminating ENDC for every COND. Conditional assembly statements may be nested.

ELSE

If assembly was turned off by the COND matching this ELSE then the ELSE will turn it back on. Otherwise it will turn the assembly off.

ENDC

This pseudo-op terminates a COND statement. If the matching COND turned assembly off then the ENDC will turn it back on.

An example of the use of COND, ELSE and ENDC is given below.

The values of assembler symbols “amstrad” and “microdrive” would need to be set before the assembler encounters this section of source.

```
COND amstrad
   *
   *
   * (AMSTRAD code here)
   *
   *
ELSE
COND microdrive
   *
   *
   * (SPECTRUM microdrive code here)
   *
   *
ELSE
   *
   *
   * (SPECTRUM tape code here)
   *
   *
ENDC
ENDC
```

This piece of source code is used to assemble one of three different versions of the same program. The possible versions are for AMSTRAD and Spectrum micros. The Spectrum version is again split between tape and microdrive versions (on the AMSTRAD it is possible to treat tape and disc in exactly the same way).

If “amstrad” is non-zero (i.e. TRYE) then the AMSTRAD section of the code would be assembled. If “amstrad” is zero then the value of “microdrive” is used to distinguish between tape and microdrive versions on the Spectrum. In any of these cases only one of the three pieces of code will be assembled, and assembly will continue after the last ENDC.

Note that an ENDC is need for each COND.

### 4.3.5 The Location Counter and Storing Object Code

ORG <expression>

Causes the assembler to set its location counter to the value of the expression. (The default start value of the location counter is #100 on the Amstrad and RAMTOP on the Spectrum). So code generated after an ORG will be assembled ready to run at the ORG address.

Note that object code assembled into memory will NOT be stored at the ORG address unless a PUT pseudo-op is used to set the storage location.

PUT <expression>

Causes the assembler to store object code generated at the address given (the value of the expression). Thus code which would overwrite the assembler or some other program in memory can be ORGed at the correct position but actual stored elsewhere.

An attempt to PUT object code in an area of memory occupied by Genius will result in an error. If object code starts off at a legal position but grows to overwrite some part of Genius at message such as:

```
** code in name table **
```

will be issued at the end of pass 1 and assembly stopped. (Note that since code is not written to memory on pass 1 no harm will have been done). (See Appendix C for descriptions of the messages which can be issued when this error occurs). In order to ascertain where you can PUT object code, use the LOW value given by the STATS command. Object code can be assembled anywhere between LOW and TABLE (both given by STATS - see Section 2.9).

The AMSTRAD jump block is protected in the same way.

### 4.3.5 Macro Definition and Use

`<label>: MACRO`
`<parameter list>`

Causes <label> to be defined as a macro name. The statements following this one up to an ENDM pseudo-op will be treated as macro definition statements and will not generate any code.

The parameter list is a list of labels separated by commas, each one starting with a "\" character. These may be used in the statements comprising the macro definition.

e.g. A macro to swap two sixteen bit registers could be written as:

```
exg :MACRO \p1, \p2
     PUSH ip1
     PUSH ip2
     POP ip1
     POP ip2
     ENDM
```

To use this macro you would write something like:

```
\exg BC,DE
```

which would generate the following code when assembled:

```
PUSH BC
PUSH DE
POP BC
POP DE
```

Macro parameters may be used to replace whole operands only. For instance, it is alright to put:

```
CP \param
```

and then to give, say, "Z" #1 as the replacement for this parameter; but you may not put:

```
CP \param+1
```

and then use "Z" as the actual parameter.

(See also the directive *maclist).

At assembly time, the occurrences of macro parameter use (\<parameter name>) are replaced with the corresponding actual parameter in the macro use statement. This can create syntax errors which will be reported to you with the usual error message for that syntax fault. The statement containing the error will not be assembled.

`ENDM`

This marks the end of a macro definition and returns the assembler to its code generating mode.

Some examples of the use of macros are given below.

On the AMSTRAD a RST 3 (RST #18, a FAR CALL) takes three bytes as in-line parameters. The first two of these represent the address of a routine to call. The third is the required ROM select when the call is made. You could define a macro as below to aid you in using these restarts.

```
FAR CALL: MACRO \p1, \p2
          RST 018
          DEFW \p1; CALL address
          DEFB\p2; ROM select
          ENDM
```

This may then be used by writing the following:

```
      \FAR CALL <address>,<ROM select>
```

The other restart instructions on the AMSTRAD take similar parameters and could have macros defined for them in a like manner.

On the Spectrum, RST #8 is used to access channel handling routines. You could define macros to help you use these like the one below:

```
      OPEN FILE: MACRO
                 RST 08
                 DB #22; Hook code for “open file”.
                 PUSH IX
                 POP HL
                 LD (CURCHL),HL
                 ENDM
```

This will open a file and a channel for it. (The address of the filename needs to have been previously stored in one of the Spectrum’s system variables). CURCHL is a Spectrum system variable which is set, in this example, to the address of the channel variables. This allows you to use the input and output character routines in the Spectrum ROM.

One further example, using nested macros, defines a macro “PRINT” which takes a string as a parameter and prints it to the screen, and a macro “ERRMES” which uses PRINT to issue a “Syntax error” message. (“prtstr” is a subroutine taking a null-terminated string as an in-line parameter, “PRINT CHAR” is a - machine-specific - routine to actually send a character to the screen).

```
      prtstr:  POP HL
      psloop:  LD A (HL)
               INC HL
               OR A
               JR Z,prtdone
               CALL PRINT_CHAR
               JR psloop
      prtdone: JP (HL)

      PRINT:   MACRO \p1
               CALL prtstr
               DEFB \p1
               ENDM

      ERRMES:  MACRO \p1
\PRINT “Syntax error: \0”
               \PRINT \p1
               ENDM
```

To use ERRMES you could put something like:

```
      \ERRMES “mismatched parentheses. \13\10\0”
```

# 5. The Hash Extensions

**NOTE:** The Hash Extensions are optional when loading the Genius Editor/Assembler. See the loading instructions for details.

The Genius assembler has a set of extensions available to it. These consist of a number of extra pseudo-ops each beginning with a "#" character (hence the name "hash extensions"). Their purpose is to provide an integer-based compiled language, Phoenix, for you to test your ideas and algorithms before writing them in machine code. It is also quite possible to write programs in Phoenix and leave it at that.

The compiler allows you to define integer and character variables, functions and one-dimensional arrays. These can all be used in expressions which the compiler will turn into run-time code, i.e. executable machine code, to evaluate these expressions.

You can also use conditional, #IF, statements; #WHILE-AENDW, and #REPEAT – #UNTIL loops to control the flow of your programs.

Phoenix statements may be freely mixed with assembly language and the results of expressions are always available to you in the HL register pair.

Below is a list of all the available pseudo-ops and their uses. Chapter 6 contains some examples of the use of the compiler language, it is a good idea to study these examples to become familiar with the extensions.

### *Glossary of Terms*

[...] Items enclosed in square brackets are optional.

< Type> One of the mnemonics INT, CHAR, PINT, PCHAR. See 5.2.1 for the meanings of these.

<Assembler expression> An ordinary expression involving numbers, symbols and operators but no compiler-only operators. See Appendix D for descriptions of the allowable operators and those which are of restricted use.

<Assembler expression list> A list of <Assembler expression>'s separated by commas.

<Compiler expression> This is an expression which is to be translated into machine code to evaluate it. It will not be evaluated at assembly time. These expressions may contain the compiler-only operators (see Appendix D and "5.1 Phoenix expressions").

<label> A label as in the assembler. See "1.3 Syntax Checking and Error Messages" or "3. The Symbol Table".

### *5.1 Phoenix Expressions*

There are two pseudo-ops available for compiling expressions. One compiles an expression to be evaluated using signed arithmetic and the other unsigned arithmetic. These are #DSE for "Define Signed Expression" and #DUE for "Define Unsigned Expression". They are used in the following way:

```
        #DSE <Compiler expression>
        #DUE <Compiler expression>
```

These pseudo-ops take the <Compiler expression> given as operand and translate it into machine code to evaluate the expression at run-time.

The run-time evaluation is performed using a stack to store temporary results and operands to arithmetic operations. An arithmetic operation is carried out by a CALL to one of a library of arithmetic routines. Only those arithmetic routines corresponding to operations which you actually use in you: program will be relocated into your code by the assembler (see 5.2.7 "The #LIB Pseudo-Op").

You can use more operators and constructs in compiler expressions than you can in assembler expressions. For details on all the operators available to Genius expressions see Appendix D. The extra constructs allowed are described below.

### 5.1.1 Assignment

Assignments can be made to variables in compiler expressions, so if you have defined a variable “state”, you could write:

```
#DUE state=1
```

This would set the value “state” to be 1 (at run-time). In Phoenix expressions, the assignment operator can be used more than once. It can be used just like the ordinary operators “+”, “*” etc. except that the left hand side must be a single variable name or something equivalent to it (for instance an array reference).

An assignment has a value just as “2+3” has a value (5). The value of “state=1” is 1; in general the value of an assignment is the value of its right hand side. So we could write:

```
#DSE x=y=0
```

(x and y are both names of variables). This would assign the value 0 to both x and y,

A more complex example is:

```
#DSE y=(z=x*x)+x-20
```

This would give “z” the value “x*x” (x squared) and give “y” the value “x*x#x–20” (x squared plus x minus 20).

For more on assignments see “5.1.4 Pointers”.

### 5.1.2 Arrays

In section 5.2.1 you will see how to define (and initialise) space for compiler variables. Once you have created a variable you can use it as a base for indexing into memory. This gives you (one-dimensional) arrays or vectors of compiler variables. You do not actually declare a compiler variable to be an array but you name a block of memory space and then use that name as an array name.

For example, the statement:

```
xcoord: #DS INT,10
```

allocates space for 10 integers at the current location (this is 20 bytes). The first integer in, this space will have name “xcoord”. The other integers in the space can be accessed by writing (in your compiler expressions):

```
xcoord [<compiler expression>]
```

So writing

```
xcoord [7]
```

would return the value of the 7th integer in the array. The value of “xcoord” itself can be accessed by writing either just “xcoord” or “xcoord [0]”. Since array indices start at 0, in the case above the last array element can be accessed as “xcoord [9] “. No check is made on the value given as the index. So referring to “xcoord [10]” is legal but will return the value held in the two bytes following the space allotted to the array; this is unlikely to be meaningful.

Assignment can be made to array elements. You might write

```
#DSE xcoord [1] = 100
```

in an expression. Letting the array index go out of bounds can be dangerous in this case as you may write into other variables or into program space.

### 5.1.3 Functions

As you will see in 5.2.3 it is possible to define “functions” in Phoenix. These can then be “called” during expression evaluation. Each function returns a value which you can use or choose to ignore: using the function may have other effects, i.e. you can use the function like a procedure. Some functions may be intended to be used as procedures and may therefore return undefined values anyway.

You can pass parameters to a function. The definition of the function determines how many parameters a function requires. The compiler does not check that you have given the correct number when you write a function call, so care must be taken here.

Suppose you have defined a function “square” which takes one parameter and returns the square of the parameter’s value. To use the function you would write:

```
#DSE y=square(x+1)
```

Setting the value of "y" to the square of "x+1". Notice that round brackets are used to section off the function's parameter list.

If a function takes more than one parameter you should separate the individual parameter expressions by commas. Suppose you have defined a function to print an integer to the screen in a base given by a parameter, then this could be called as below:

```
#DSE based_print(scale*xcoord [count],10)
```

Notice that this time we are ignoring the value returned by the function.

### 5.1.4 Pointers

Two of the types of variable which you can define in Phoenix are "pointers". This means that they hold the addresses of other variables rather than containing a piece of data. You can define pointers to integers and pointers to characters.

You can find the address of a variable's storage space by using the "&" (unary prefix) operator.

You can access the piece of data which a pointer points to by using the " " operator.

As an example suppose we have defined a (null terminated) string with the statement:

```
begin: #DI CHAR, "There was no possibility of... \0"
```

and we want to copy this to a new location for editing. At this location we need space for the string:

```
rewrite: #DS CHAR,80; One Line of characters.
```

To copy the string we need two pointers to characters, one pointing to the original string and one pointing to the space.

```
P_str: #DS PCHAR,1; For pointing to the string.
P_spc: #DS PCHAR,1; For pointing to the space.
```

We need to initialise the pointers using the "&" ("address of") operator.

```
#DUE p_str=&begin
#DUE p_spc=&rewrite
```

To do the copying we use the "&" ("contents of") operator. We could write:

```
#WHILE *p_str!=0    ; Have we reached the end of the string?
#DUE *p_spc=*p_str  ; Contents of p_spc equals contents of p_str
#DUE p_spc=p_spc+1  ; Increment p_spc
#DUE p_str=p_str+1  ; Increment p_str
#ENDW               ; End of the Loop
#DUE *p_spc=0       ; Null terminate the copy
```

You can do the job in rather less code using some of the other Phoenix operators. The special incrementation operators can be used instead of the lines adding 1 to the pointers. That is, you could write:

```
#DUE p_spc++
#DUE p_str++
```

We can now compress the three lines inside the loop into one by putting these incrementation operators into the expression which does the copying.

```
#WHILE *p_str!=0
#DUE *p_spc++=*p_str++
#ENDW
#DUE *p_spc=0
```

Now recall that an assignment has a value; the value of the right hand side of the assignment (5.1.1). So we can test for the end of the string having been reached in the *WHILE statement as well, since the string is terminated by a 0 which has truth value FALSE and will therefore stop the loop. The null terminator will also be copied into the new space so we won't need the last line any more. So the final version of the string copying program is:

```
#WHILE *p_spc++=*p_str++
#ENDW
```

Note that the “++” operator will increment a pointer to character by 1 (to step over one character) but will increment a pointer to integer by 2 (to step over one integer). Similarly, the “-“ operator decrements pointers to characters and integers by 1 and 2 respectively.

## *5.2 The Hash Extension Pseudo-Ops*

### 5.2.1 Declaring Variables

Phoenix has four types of variable. These are integer, character, pointer-to-integer and pointer-to-character. These types are represented by the mnemonics INT, CHAR, PINT, PCHAR respectively when a distinction is necessary in a particular pseudo-op.

The pseudo-ops for declaring any of these types are given below.

[<label>:] `#DS` <Type>, <Assembler expression>

This pseudo-op allocates enough space for the number of variables of type <Type> given by the value of the expression. It is similar to the DEFS or DS assembler pseudo-ops in that the space is allocated at the current location and is initialised to zero (but see “5.2.3 Defining Functions” as PDS performs differently inside a function). If a label is given then it will be entered into the symbol table and marked as being a compiler variable of type <Type>. This allows you to reference one of the variables using an array construction or just by the use of <label>.

For example:

```
var1: #DS INT,1
```

defines space for 1 integer called “var1”,

```
string: #DS CHAR,40
```

defines space for 40 characters. These could be accessed using a construction of the form:

```
string [39]=0
```

(This would “null terminate” the string).

<label>: `#DI` <Type>, <assembler expression list>

This pseudo-op defines and initialises variables. It will define as many variables of type <Type> as there are expressions in the <Assembler expression list>. Each one will have space allocated for it and this space will be initialised to the value of the expression associated with it.

For example:

```
One_int: #DI INT,#1234
```

will define an integer called “one_int” and give it an initial value of #1234.

```
greeting: #DI CHAR,“Good morrow. \13 \10”
```

will define space for 14 characters and initialise this space to the string given. Now, for instance, a use of the construction:

```
greeting [5]
```

would return the (character) value “m”.

You may only use strings for initialisation if you have specified type CHAR.

**NOTE:** #DI may not be used inside a function. (See 5.2.3 Defining Functions).

### 5.2.2 Compiling Expressions

`#DSE` <Compiler expression>
`#DUE` <Compiler expression>

These two pseudo-ops generate machine code to evaluate the compiler expressions they are given as operands. For instance the statements:

```
x: #DS INT 1
y: #DS INT 1
```

```
#DUE x=y=0
```

would generate the code:

```
LD HL,0
LD (<address of y>),HL
LD (<address of x>),HL
```

The statement:

```
#DUE x+y
```

will generate:

```
LD HL,(<address of y>)
PUSH HL
LD HL,(<address of x>)
PUSH HL
CALL <addition routine>
POP HL
; Final result in HL
```

The #DSE pseudo-op compiles the expression to be evaluated using signed arithmetic. The #DUE pseudo-op compiles the expression to be evaluated using unsigned arithmetic. The arithmetic routines will detect some errors (overflow and division by zero), when this happens a bit in the byte (IX-1) is set to signal that an error has occurred during evaluation of the expression. You can test these bits in your own code but the compiled code will take no action itself i.e. the program will just continue. (The IX register is used by the compiled code and you should not alter its contents while running a compiled program (see the CSTACK pseudo-op).

The bits which are set are:

| | |
|---|---|
| overflow: | BIT 0 (corresponding to the Z80 C flag) |
| division by 0: | BIT 2 (corresponding to the Z80 P/V flag) |

The result of an expression is left in the HL register so that you can access it in machine code if you wish. (See the example programs in Chapter 6, especially the multiple precision routines.)

### 5.2.3 Defining Functions

You can define functions for use in compiler expressions using the following pseudo-ops. Function definitions may not be nested but a function may use another one at run-time. A function returns a value which may be used in expressions or just ignored. (This value is left in the HL register pair for access from pure machine code). A function is invoked or called from within a compiler expression by writing the function name followed by list of parameter values enclosed in parentheses. This is illustrated below.

```
   <name> (<parameter list>)
or<name>()
```

Where <parameter list> is a list of compiler expressions separated by commas. The second case should be used when the function in question takes no parameters.

#FNC <Type>

Declares the start of a function definition. There must be a #BEGIN - #END pair in the code following the #FNC.

<label>: #PRM <Type>

This pseudo-op is used to declare <label> as a parameter to the function being defined. A function may have up to 60 parameters. In a call to the function the actual parameters must be given in the same order that the dummy parameters have been defined. Parameter names are entered into a local table associated with the function name, so you can use the same names in different functions (or re-use global names) without multiple definition errors.

Parameter declarations must appear after the #FNC statement and before the #BEGIN.

#BEGIN

Declares the start of the definition of the body of a function. There should be no code generating statements between the #FNC statement and the #BEGIN.

`#END`

Declares the end of a function definition. Parameter and variable declarations should not appear between the #BEGIN and the #END. At run-time the result of a function is left in the HL register pair.

`#RETURN`

This pseudo-op generates code to perform a return from the function. It can appear anywhere within the function (even in positions where you have pushed some data onto the stack yourself. If you use the #RETURN pseudo-op the Value returned by the function will be the contents of the HL register pair before the #RETURN is executed.

It is not necessary to put a #RETURN immediately before a #END directive (if you do the code to execute a return from a function will be generated twice: one of them will be impossible to reach at run-time).

### An Example of a Function Definition

The following piece of code defines a Phoenix function to return the (unsigned) sum of the squares of its two parameters.

```
sumsq : #FNC INT
x     : #PRM INT
y     : #PRM INT
        #BEGIN
        #DUE x*x+y*y
        #END
```

You would use this by writing something like:

```
#DUE z=sumsq(3,4)
```

This would set z to the value 25.

**NOTE:** The #DS pseudo-op may be used between a #FNC and a #BEGIN. In this case it does not allocate any space in the object code but causes space to be allocated on the stack when the function is called, i.e. in this case the storage is “automatic” rather than “static”. When the function returns the storage space will be de-allocated and returned to the stack.

#DI may not be used between a #FNC and a #BEGIN.

### 5.2.4 Conditional Statements

Phoenix provides a conditional statement in the form of the following pseudo-ops.

`#IF` <compiler expression>

Marks the start of a section of code which is to be evaluated conditionally depending on the value of the compiler expression given (at run-time). A #IF statement must have matching #ENDIF or #ELSE statements. If the compiler expression evaluates to TRUE (non-zero) then the (compiled version of the) code following the #IF statement will be executed otherwise control is passed to the statement following the #ENDIF statement matching the #IF if there is one.

Note that #IF uses signed evaluation on its operand expression.

`#ELSE`

This pseudo-op should only be used after a matching #IF statement and must have a matching #ENDIF later in the source. The section of code between the #ELSE and #ENDIF statements will be executed if and only if the compiler expression operand of the matching #IF evaluated to FALSE (zero).

`#ENDIF`

Marks the end of a section of code which is to be conditionally executed. A #ENDIF statement must have a matching #IF or #IF – #ELSE pair.

As an example of the use of the conditional we define a function to return the maximum of its two parameters.

```
Max : #FNC INT
x   : #PRM INT
Y   : #PRM INT
      #BEGIN
      #IF x>y
      #DSE x
      #ELSE
      #DSE y
      #ENDIF
      #END
```

If the value of "x" is larger than that of "y" we get the value of "x" into the HL register pair, else we get the value of "y". Since we do nothing else in this function this value will remain until the function returns when it will be used as the result of the function.

### *5.2.5 Loop Constructs*

Phoenix provides two types of looping construct. The first (the "while" loop) tests the loop's termination condition at the start of the loop. The second (the "repeat" loop) tests the condition at the end of the loop. In the "repeat" loop case the loop will therefore always be executed at least once: with a "while" loop the condition may be FALSE when the start of the loop is first encountered and the body of the loop never executed.

`#WHILE` <compiler expression>
Marks the start of a #WHILE - #ENDW loop. This pseudo-op generates code to evaluate the compiler expression given as operand and to jump to the statement following the #ENDW matching this #WHILE if the result of the expression is FALSE (zero). If the expression is TRUE (non-zero) then execution continues with the statement immediately after the #WHILE instruction.

A #WHILE statement must be used to terminate a #WHILE loop. The #ENDW returns control to the code compiled by the #WHILE statement causing the termination condition to be tested again.

`#ENDW`
Marks the end of a #WHILE loop. This pseudo-op compiles a jump back to its matching #WHILE.

`#REPEAT`
Marks the start of a #REPEAT – #UNTIL loop. This pseudo-op does not generate any code.

`#UNTIL` <compiler expression>
Marks the end of a #REPEAT – #UNTIL loop. This pseudo-op generates code to evaluate its compiler expression operand, test the result, and possibly pass control back to the #REPEAT matching this #UNTIL. If the result is false then control will be passed back to the start of the loop, else execution continues from the statement after this #UNTIL.

### 5.2.6 Setting the Execution Stack

**NOTE:** The #STACK pseudo-op should appear somewhere in any compiler source. At the start of the source is the best place to put it.

The code compiled to evaluate expressions uses the IX register to keep note of the stack pointer and to index parameter values in functions. The value of IX must therefore be initialised to the value required by the compiled code. This should be done using the following pseudo-op.

`#STACK` [<assembler expression>]
This pseudo-op generates code to set IX to a value related to the stack pointer's value. The optional (assembler) expression given as Operand to #STACK can be used to set the stack pointer to a new value i.e. it will generate a "LD SP,word" instruction. For some applications of the compiled language it may be wise to move the stack from the position set by the AMSTRAD firmware as you have only 256 bytes of space available in this position.

**PLEASE NOTE:**
This pseudo-op also generates a DEC SP instruction (to make one byte of room for use as PHOENIX flag space). So to return from a routine

containing a #STACK pseudo-op you should use either:

```
LD SP,IX
RET
```

or

```
INC SP
RET
```

rather than just a RET instruction.

### 5.2.7 The #LIB pseudo-op

The code generated by the compiler to evaluate expressions contains CALLs to a number of arithmetic routines to actually perform the evaluation. These routines must appear somewhere in the object code produced by the assembly of your source. The PLIB pseudo-op is used to relocate the arithmetic routines down into your object code. It should appear somewhere in your source after you have used all the Phoenix operators that you are going to use in this piece of code. This is because #LIB only relocates the routines associated with operators that you have actually referenced in your code.

`#LIB` This pseudo-op allows your compiled code to run independently of the Genius assembler by relocating any arithmetic routines required for the evaluation of your expressions down into your object code. You should place a #LIB statement in your source after you have used all those operators that you will need in the whole program being assembled. (The statement containing a given operator must actually have been reached by the assembler: a statement that is not assembled owing to conditional assembly cannot cause any operator routines to become liable to relocation by #LIB.)

**NOTE:** The tool-kit command EXPORT will save data recording those library routines which have already been relocated. So that when you IMPORT a symbol table into a Phoenix program the #LIB will not create extra copies of library routines that have been included in previously assembled code. You must include a #LIB pseudo-op in the source of the pre-compiled routines.

# 6. Example Programs

This chapter contains listings of some programs and routines in BASIC, the PHOENIX compiled language, and in machine code. These are intended as examples of the use of the assembler and the compiled language. (The BASIC versions are included for comparison and to help illustrate the algorithms used).

## *6.1 Example 1 - The Sieve of Eratosthenes*

This example is the famous “Sieve of Eratosthenes” written in BASIC and in PHOENIX. The sieve is a method of extracting the prime numbers from the integers. The algorithm embodied in the programs is as follows.

List the integers in the range from which you want to extract the primes. (In the programs this is 2 to 5000). Then for each integer, n, in this range strike out the integers 2*n, 3*n, 4*n, 5*n etc., i.e. we remove the multiples of n. (You must start with n=2, not 1, why?). When you have finished, any numbers which you have not struck out will be prime.

This is translated into a computer program by defining an array, “primeflags”, with one entry for each of the integers in the range we are considering. The elements of the array are all initialised to zero. Then for each integer, n, in the range we set primeflags(2*n), primeflags(3*n), etc. to 1 to signal that each of the multiples of n is not a prime. When we are done any entry in the array “primeflags” which is not set to 1 corresponds to a prime number, i.e. if primeflags(m) is 0 for a number m, then m is prime. The implementation given is not the best possible. You might like to try and improve on it.

Sieve of Eratosthenes written in BASIC and PHOENIX.

```
10  REM
20  REM Sieve of Eratosthenes.
30  REM
40  DIM primeflags%(5000)
50  count%=2
60  WHILE count%<=500
70  IF primeflags%(count%)=1 THEN 130
80  count1%=count%+count%
90  WHILE count1%<=5000
100 primeflags%(count1%)=1
110 count1%=countl%+count%
120 WEND
130 count%=count%+1
140 WEND
150 FOR count%=2 T0 5000
160 IF primeflags%(count%)=0 THEN PRINT count%,
170 NEXT count%
180 FND
```

```
10  ;
    ; Sieve of Eratosthenes.
    ;
20  ;
    ; AMSTRAD (1) or SPECTRUM (0) ?
    ;
    AMSSPEC:
           EQU 1
    ;
30  ;
    ;Function to write a character
    ; to the screen.
    ;
    ; This must be before “print.int”
    ; as it is used by “print.int”.
40  print.char:
           #FNC   INT
    chr :  #PRM   CHAR
           #BEGIN
           #DSE   chr
           COND AMSSPEC
           LD   A,L
           CALL #BB5A
           ELSE
           LD A,L
           CALL #9F4
           ENDC
           #END
    ;
    ;
50  ;
    ; Inner integer print function.
    ;
    ; This must be before "print.int"
    ; as it is used by "print.int".
    ;
60  print.int1:
           #FNC    INT
    pval : #PRM    INT
    chr  : #DS     CHAR,1
           #BEGIN
           #IF     pval!=0
           #DSE    chr=pval!=0
           #DSE    print.int1(pval/10)
           #DSE    print.char(chr+#30)
           #ENDIF
           #END
    ;
    ;
70  ;
    ; Integer print function.
    ;
80  print.int:
           #FNC    INT
    pval : #PRM    INT
    chr  : #DS     CHAR,1
           #BEBIN
           #IF     pval?=0
           #DSE    print.char("0")
           #ELSE
           #DSE    print.int1(pval)
           #ENDIF
    ;
           #DSE print.char(" ")
           #END
    ;
    ;
90  ;
    ; Workspace.
    ;
    count: #DS     INT,1
    count1:
           #DS     INT,1
    ;
    primeflags:
           #DS     INT,5001
    ;
100 ;
    ; Main calculation loop.
    ;
    ; This loop marks non-primes
    ; in the array "primeflags".
    ;
110 ;
    ; Enter here (EXECUTE start).
    ;
    start: #STACK  ; Set IX for PHOENIX use.
    ;
    ; On the SPECTRUM set the channel to the main screen.
    ;
           COND AMSSPEC
    ;
           LD A,2
           CALL #1601 ; ROM's CHAN OPEN
    ;
           ENDC
    ;
120        #DSE   count=2
    ;
           #WHILE count<=2500
    ;
           #IF    primeflsgs[count]?=0
    ;
           #DUE   count1=count
    ;
           #WHILE [count1=count1+count]<=5000
           #DUE   primeflags[count1]=1
           #ENDW
    ;
           #ENDIF
    ;
           #DSE ++count ; Increment count.
    ;
           #ENDW
    ;
130 ;
    ; Loop to print the primes found.
    ;
           #DUE   count=2
    ;
           #WHILE count<=5000
    ;
           #IF    primeflags[count]?=0
           #DUE   print.int(count)
           #ENDIF
    ;
           #DUE ++count
    ;
           #ENDW
    ;
140 ;
    ; Done, replace strak pointer and return.
    ;
           LD SP,IX
           RET
    ;
150 ;
    ; Include library routines for arithmetic.
    ;
           #LIB
    ;
```

### *6.2 Example 2 - Drawing Ellipses*

This example consists of an algorithm to draw ellipses implemented in BASIC, PHOENIX and hand-crafted machine code.

The algorithm used is an adaptation of Bresenham's incremental circle generating algorithm. Unless you are familiar with this it will not be at all clear how it works. For a full explanation of the method you are referred to "Fundamentals of Computer Graphics" by J.D. Foley and A. Van Dam, and "Procedural Elements for Computer Graphics" by D.F. Rogers.

The implementation used requires numbers larger than 65535, i.e. too large to fit into the two-byte integer variables of BASIC or PHOENIX. In BASIC this problem is solved by using floating point variables. In PHOENIX a set of multiple precision (integer) arithmetic routines is used. These routines can be assembled to use any length (in bytes) of integer. The precision required (i.e. the length in bytes of the integers to be handled by the routines) is set by the value of a symbol "precision". You could use these routines in your own programs: the method of use is illustrated in lines 50 and 60 of the PHOENIX ellipse drawing program; the calling conventions for the multiple precision routines are given as comments in the routines themselves.

Ellipse drawing program written in BASIC, PHOENIX and machine code.

```
100 REM
110 REM BASIC ELLIPSE DRAWING PROGRAM.
120 REM
121 REM Although written in AMSTRAD BASIC this program can
122 REN easily be converted to run on the spectrum: the only
123 REM significant difference is the use of WHILE loops.
124 REM
130 INPUT "Axes ";a,b
140 xi=0:yi=a
150 a2=a*a:b2=b+b
1bO a2s=a2:b2s=b2*(2*yi+1)
170 DELTAi=2*(1-b)
180 WHILE yi>0
190 PLOT xi,yi:PLOT -xi,yi:PLOT xi,-yi:PLOT -xi,-yi
200 IF DELTAi>0 THEN GOSUB 250 ELSE IF DELTAi=0 THEN GOSUB 310 ELSE GOSUB
    340
210 WEND
220 END
230 REM
240 REM
250 delta=DELTAi+DELTAi-a2s
260 IF delta<=0 THEN GOSUB 310 ELSE GOSUB 400
270 RETURN
280 REM
290 REM Make a diagonal step.
300 REM
310 xi=xi+1:a2s=a2s+a2+a2:yi=yi-1:b2s=b2s-b2-b2:DELTAi=DELTAi+a2s-
    b2s:RETURN
320 REM
330 REM
340 delta=DELTAi+DELTAi+b2s
350 IF delta<=0 THEN xi=xi+1:a2s=a2s+a2+a2:DELTAi=DELTAi+a2s:RETURN ELSE
    GOSUB 310
36O RETURN
370 REM
380 REM Make a vertical step.
390 REM
400 yi=yi-1:b2s=b2s-b2-b2:DELTAi=DELTAi-b2a:RETURN
```

```
10  ;
    ; Set AMSTRAD to 0 for use on the SPECTRUM.
    ;
    AMSTRAD:
           EQU 1
    ;
20  ;
    ; PHOENIX ellipse drawing routines.
    ;
    ; On entry:
    ;
    ; major = x semi-axis of ellipse.
    ; minor = y semi-axis of ellipse.
    ; xs    = x co-ordinate of centre.
    ; ys    = y co-ordinate of centre.
    ;
30  minor: #DI    INT,70
    major: #DI    INT,100
    ;
    xs   : #DS    INT,1
    ys   : #DS    INT,1
    ;
    xi   : #DS    INT,1
    yi   : #DS    INT,1
    ;
40  ;
```

```
; Set the precision for multiple precision routines.
precision:
       EQU 4
;
;
; Include the multiple precision source.
 *INCLUDE "MPA-FNCS.PHX"
;
;
; Multiple precision workspace.
;
a2   : DEFS precision
b2   : DEFS precision
a2s  : DEFS precision
b2s  : DEFS precision
;
DELTAi:
       DEFS precision
delta: DEFS precision
;
long.zero:
       DEFS precision
;
long.one:
       DEFB 1
       DEFS precision-1
;
; Ellipse work functions.
;
;
; Make a vertical (downwards) step.
;
vertical:
       #FNC   INT
       #BEGIN
       #DSE   --yi
       #DSE   mpa.sub(b2s,b2s,b2) ;        b2s=b2s-b2-b2
       #DSE   mpa.sub(b2s,b2s,b2)
       #DSE   mpa.sub(DELTAi,DELTAi,b2s) ; DELTAi=DELTAi-b2s
       #END
;
;
; Make a horizontal (right) step.
horizontal:
       #FNC   INT
       #BEGIN
       #DBE   ++xi
       #DSE   mpa.add(a2s,a2s,a2) ;        a2s=a2s+a2+a2
       #DSE   mpa.add(a2s,a2s,a2)
       #DSE   mpa.add(DELTAi,DELTAi,a2s) ; DELTAi=DELTAi+a2s
       #END
;
;
; Make a diagonal step.
;
diaqona1:
       #FNC   INT
       #BEGIN
       #DSE   horizontal()
       #DSE   vertical()
       #END
;
;
;
; Decision function; go diagonally or horizontally?
;
test_horizontal:
       #FNC   INT
       #BEGIN
       #DSE   mpa.add(delta,DELTAi,DELTAi> ; delta=DELTAi+DELTAi+b2s
       #DSE   mpa.add(delta,delta,b2s)
;
       #IF    mpa.sign (delta)>0
```

```
           #DSE   diagonal ()
           #ELSE
           #DSE   horizontal ()
           #ENDIF
           #END
    ;
120 ;
    ; ROM entries for plotting points.
    ;
           COND AMSTRAD
    ;
    GRA_PLOT_ABSOLUTE:
           EQU #BBEA
    GRA_SET_ORIGIN:
           EQU #BBC9
    ;
           ELSE
    ;
    ; Origin set at (127,100) on the SPECTRUM.
    ;
    GRA_PLOT_ABSOLUTE:
           LD   A,E
           ADD  A,127
           LD   C,A
           LD   A,L
           ADD  A,100
           LD   B,A
           JP   #22E5 ; ROM plot routine.
    ;
           ENDC
    ;
130 ;
    ; Plot a single point.
    ;
    plot : #FNC   INT
    xp   : #PRM INT
    yp   : #PRM INT
           #BEGIN
           #DSE   xp
           PUSH HL
           #DSE   yp
           POP  DE
           CALL GRA_PLOT_ABSOLUTE
           #END
    ;
```

```
;
; Plot four points of the ellipse (by symmetry).
plot4: #FNC   INT
xp   : #PRM   INT
yp   : #PRM   INT
       #BEGIN
       #DSE   plot(xp,yp)
       #DSE   plot(-xp,yp)
       #DSE   plot(-xp,-yp)
       #DSE   plot(xp,-yp)
       #END
;
;
; Initialise lonq variables.
;
;***************
; Entry point.
;***************
start: #ST4CK ; Set PHOENIX stack and flag byte.
;
       COND AMSTRAD
       LD   HL,200
       LD   DE,200
       CALL GRA_SET_ORIGIN
       ENDC
;
;
       #DSE    xi=0
       #DSE    yi=major
;
       #DSE    mpa.extend(a2,&major) ; a2=a*a
       #DSE    mpa.mul(a2,a2,a2)
;
       #DSE    mpa.extend(b2,&minor) ; b2=b*b
       #DSE    mpa.mul(b2,b2,b2)
;
;
       #DSE    mpa.add(a2s,a2,long.zero) ; a2s=a2
       #DSE    mpa.extend(b2s,&yi) ; b2s=b2*(2*yi+1)
       #DSE    mpa.add(b2s,b2s,b2s)
;
       #DSE    mpa.add(b2s,b2s,long.one)
       #DSE    mpa.mul(b2s,b2,b2s)
;
;
       #DSE    mpa.extend(DELTAi,&minor) ; DELTAi=2*(1-b)
       #DSE    mpa.sub(DELTAi,long.one,DELTAi)
       #DSE    mpa.add(DELTAi,DELTAi,DELTAi)
;
;
;*************************
; Main loop.
;*************************
;
       #WHILE yi>0
;
       #DSE    plot4(xi,yi)
;
;
       #IF     mpa.siqn(DELTAi)>0
;
       #DSE    mpa.add(delta,DELTAi,DELTAi) ; delta=DELTAi+DELTAi-a2
       #DSE    mpa.sub(delta,delta,a2)
;
       #IF     mpa.sign(delta)>0
       #DSE    vertical()
       #ELSE
       #DSE    diaganal()
       #ENDIF
;
       #ELSE
;
       #IF     mpa.sign(DELTAi)?=0
```

```
;
       #DSE    diagonal()
       #ELSE
       #DSE    test_horizontal()
;
       #ENDIF
;
       #ENDIF
;
; End of main loop.
;
       #ENDW
;
; Replace stack pointer.
;
       LD SP,IX
       RET
;
;
; Include any PHOENIX library routines used.
;
; #LIB
```

```
;
; Set AMSTRAD to 0 if using a spectrum.
AMSTRAD:
       EQU 1
;
;
; Machine code ellipse drawing routines.
;
; Enter at "start"
;
; (xo),(yo) = centre of ellipse.
;
; (major),(minor) = the axes of the ellipse.
;
;
; Multiplication routine.
;
; Multiplies BC by HL-DE giving 32 bit result in HL-IX
;
mul32: LD   IX,0
       LD   A,32
mult : ADD  IX,IX
       ADC  HL,HL
       RL   E
       RL   D
       JR   NC,noadd
       ADD  IX,BC
       JR   NC,noadd
       INC  HL
noadd: DEC A
       JR NZ,mult
       RET
;
;
;
; Workspace.
;
; Ellipse axis.
;
minor: DEFW 100
major: DEFW 60
;
;Ellipse centre.
;
xo   : DEFW 127
Yo   : DEFW 100
;
```

```
; current plot point.
;
x1   : DEFS 2
y1   : DEFS 2
;
       COND AMSTRAD
;
; Multiplier for screen mode on the AMSTRAD.
;
x_factor:
      DEFS 1
;
      ENDC
;
;
; Work variables. (See BASIC listing).
;
a2   : DEFS 4
a2s  : DEFS 4
b2   : DEFS 4
b2s  : DEFS 4
;
DELTAi:
       DEFS 4
;
;
; Entry point.
; Type "EXECUTE start" from GENIUS editor.
;
start:
;
; Set the origin on the AMSTRAD.
;
       COND AMSTRAD
       LD   DE,(yo) ; Note : this is in standard co-ordinates.
       LD   HL,(x0)
       CALL GRA_SET_ORIGIN
;
;
; Calculate the screen eode expansion factar (AMSTRAD) .
       CALL SCR_GET_MODE
       SUB  3
       NEG
       LD   (x_factor),A
;
;
; Adjust ellipse x-axis.
;
       LD   B,A
       LD   HL,(minpr)
       JR   stexplop
explop:
       SRL  H
       RR   L
stexplop:
       DJNZ explop
;
; HL = corrected (pixel co-ordinate) axis.
;
       LD   (minor),HL
;
       ENDC
;
       LD   HL,0
       LD   (xi),HL
;
       LD   HL,(major)
;
; Reduce the y-axis (AMSTRAD) to give pixel co-ardinates.
;
       COND AMSTRAD
;
       SRL  H
       RR   L
       LD   (major),HL
;
       ENDC
;
       LD   (yi),HL
;
;
; Initialise work variables.
;
       LD   B,H ; a2=a*a
       LD   C,L
       LD   DE,0
       CALL mul32
       LD   (a2),IX
       LD   (a2+2),HL
;
       LD   (a2s),IX ; a2s=a2
       LD   (a2s+2),HL
;
       LD   HL,(minor) ; b2=b*b
       LD   B,H
       LD   C,L
       LD   DE,0
       CALL mul32
       LD   (b2),IX
       LD   (b2s),HL
;
       LD   HL,(yi) ; b2s=b2*(2*yi+1)
       ADD  HL,HL
       INC  HL
       LD   B,H
       LD   C,L
       LD   HL,(b2)
       LD   DE,(b2+2)
       CALL mul32
       LD   (b2s),IX
       LD   (b2s+2),HL
;
       LD   HL,1 ; DELTAi=2*(1-b)
       LD   DE,(minor)
       OR   A
       SBC  HL,DE
       ADD  HL,HL
       LD   (DELTAi),HL
       LD   HL,#FFFF ; Sign extend to 32 bits (requires b>0)/
       LD   (DELTAi+2),HL
;
; ************************************
; Main calculation loop.
; ************************************
;
;
;
;
main_loop:
       LD   HL,(yi) ; While yi>0
       BIT  7,H
       RET  NZ
;
       LD   DE,(xi)
;
; HL already has (yi).
;
; Plot four points of the ellipse (by symmetry).
; We actually plot (xi,yi), (-xi,yi), (-xi,-yi), (xi,-yi).
;
; On the AMSTRAD expand for the particular screen made.
;
       COND AMSTRAD
;
;
;
       ADD  HL,HL
;
```

```
           LD   A,(x_factor)
           LD   B,A
           EX   DE,HL
           JR   stxexp
    xexp : ADD   HL,HL
    stxexp:
           DJNZ xexp
           EX   DE,HL
    ;
           ENDC
    ;
200        PUSH DE
           PUSH DE
           PUSH HL
           CALL GRA_PLOT_ABSOLUTE ; (xi,yi)
210        POP  HL
           POP  DE
           PUSH HL
           LD   A,D ; Negate xi
           CPL
           LD   D,A
           LD   A,E
           CPL
           LD   E,A
           INC  DE
           PUSH DE
           CALL GRA_PLOT_ABSOLUTE ; (-xi,yi)
220        POP  DE
           POP  HL
           LD   A,H ; Negate yi
           CPL
           LD   H,A
           LD   A,L
           CPL
           LD   L,A
           INC  HL
           PUSH HL
           CALL GRA_PLOT_ABSOLUTE ; (-xi,-yi)
230        POP  HL
           POP  DE
           CALL GRA_PLOT_ABSOLUTE ; (xi,-yi)
    ;
    ;
240 ;
           LD   DE,(DELTAi) ; IF DELTAi>0
           LD   HL,(DELTAi+2)
           CALL testdeh1
           JR   Z,godiag
           JR   NC,gohoriz
250        LD   HL,(DELTAi) ; delta=DELTAi+DELTAi-a2s
           ADD  HL,HL
           EX   DE,HL
           LD   HL,(DELTAi+2)
           ADC  HL,HL
           EX   DE,HL
           LD   BC,(a2s)
           OR   A
           SBC  HL,BC
           EX   DE,HL
           LD   BC,(a2s+2)
           SBC  HL,BC
           CALL testdeh1 ; IF delta<=0
           JR   Z,godiag
           JR   NC,godiag
           CALL vertical
           JR   main_loop
    ;
260 ;
    ;

    gohoriz:
           LD   HL,(DELTAi) ; delta=DELTAi+DELTAi+b2s
           ADD  HL,HL
```

```
       EX   DE,HL
       LD   HL,(DELTAi+2)
       ADC  HL,HL
       EX   DE,HL
       LD   BC,(b2s)
       ADD  HL,BC
       EX   DE,HL
       LD   BC,(b2s+2)
       ADC  HL,BC
       CALL testdeh1 ; IF delta<=0
       JR   Z,horiz
       JR   NC,horiz
;
;
godiag:
       CALL vertical ; Make a diagonal step. (Go up then along).
horiz: LD   HL,(xi) ; Make a horizontal step.
       INC  HL
       LD   (xi),HL
;
;
;
; a2s=a2s+a2+a2
;
       LD   B,2
inca2s1:
       LD   HL,(a2s)
       LD   DE,(a2)
       ADD  HL,DE
       LD   (a2s),HL
       LD   HL,(a2s+2)
       LD   DE,(a2+2)
       ADC  HL,DE
       LD   (a2s+2),HL
       DJNZ inca2s1
;
;
; DELTAi=DELTAi+a2s
;
       PUSH HL
       LD   HL,(DELTAi)
       LD   DE,(a2s)
       ADD  HL,DE
       LD   (DELTAi),HL
       LD   HL,(DELTAi+2)
       POP  DE
       ADC  HL,DE
       LD   (DELTAi+2),HL
       JP   main_loop
;
;
;
;
vertical:
       LD   HL,(yi) ; Make a vertical step.
       DEC  HL
       LD   (yi),HL
;
;
; b2s=b2s-b2-b2
;
       LD   B,2
decb2s1:
       LD   HL,(b2s)
       LD   DE,(b2)
       OR   A
       SBC  HL,DE
       LD   (b2s),HL
       LD   HL,(b2s+2)
       LD   DE,(b2+2)
       SBC  HL,DE
       LD   (b2s+2),HL
       DJNZ decb2s1
```

```
    ;
320 ;
    ; DELTAi=DELTAi-b2s
    ;
           PUSH HL
           LD   HL,(DELTAi)
           LD   DE,(b2s)
           OR   A
           SBC  HL,DE
           LD   (DELTAi),HL
           LD   HL,(DELTAi+2)
           POP  DE
           SBC  HL,DE
           LD  (DELTAi+2),HL
           RET
    ;
    ;
330 ;
    ; Test the 32-bit number in HL-DE for its sign.
    ;
    testdeh1:
           OR   A
           BIT  7,H
           RET  NZ ; NC, NZ means less than zero.
           LD   A,D
           OR   E
           SCF
           RET  NZ ; C, NZ means greater then zero.
           LD   A,H
           OR   L
           SCF
           RET
    ;
340 ;
    ; AMSTRAD firmware routine calls.
    ;
           COND AMSTRAD
    ;
    GRA_SET_ ORIGIN:
           EQU #BBC9
    GRA_PLOT_ABSOLUTE:
           EQU #BBEA
    SCR_GET_MODE:
           EQU #BC11
    ;
           ELSE
    ;
    ; Spectrum plot routine.
    ;
    GRA_PLOT_ABSOLUTE:
           LD   A,(xo)
           ADD  A,E
           LD   C,A
           LD   A,(yo)
           ADD  A,L
           LD   B,A
           JP   #22E5 ; ROM plot routine
    ;
    ;
           ENDC
```

# 7. Spectrum File Transfer Utility

This section is only applicable to Spectrum tape users who have created object files using the *openout assembler directive (see 4.2.3). The utility itself consists of the first two files of Tape 1 Side B. To execute the utility, rewind the tape and type LOAD "TRANS". The program will load and auto-run.

You will be given a menu with two choices:

```
1. Load for execution
2. Load for saving
```

Press "1" to choose option 1 and "2" for option 2.

Option 1: This option allows you to execute binary files created by the Genius assembler. You will be prompted for a filename and a load address. The Utility will then load the file specified at the address given and display the address of the byte after the last byte of code loaded. It will then ask you if you wish to load another file (you may want to load a library of routines for use by your program). If you answer yes you will again be prompted for a filename and a load address. This cycle repeats until you answer no to the "load another file? (Y/N)" question, when you will be prompted for an execution address. Your code is then executed by a JP instruction to the address specified.

Option 2: This option allows you to convert Genius format binary files to normal Spectrum CODE format. You will be prompted for a filename, the utility will then load that file from tape. You will then be asked:

```
Load another file? (Y/N)
```

Answering "Y" (for yes) will bring the filename prompt back. The new file will be loaded at the end of the last one. This allows you to concatenate different Genius binary files to form a single Spectrum CODE file.

This cycle repeats until you answer "N" (for no) to the "load another file" question, when you will be prompted for a load address. This is the default load address to go into the CODE file's header for use by BASIC.

You will then be prompted to press play and record and a key to start the saving of the CODE file. When this is done you will be returned to the main menu.

**NOTE:** Binary files saved from the assembler by the CODE immediate command are directly executable by BASIC. You do not need to use this utility on these files.

# Appendix A - Z80 Instruction Codes

Below is a list of the standard Z80 instructions in alphabetical order. The Genius assembler will accept non-standard mnemonics for a small number of these, the alternative version being given below the standard one in the table.

Immediate data in the instructions and object codes are represented as follows:

| Object | Source | Meaning |
|---|---|---|
| `llhh` | `addr` | A16 bit address. "`ll`" represents the low (least significant) and "`hh`" the high byte. |
| `llhh` | `word` | A 16 bit integer. (As "`addr`") |
| bb | `byte` | An 8 bit integer. |
| dd | `disp` | An 8 bit displacement. |

Note that the DJNZ and JR instructions are given as taking an 8 bit displacement as the operand. The assembler, however, treats this operand as a 16 bit address and calculates the correct displacement to put into the object code. (If the address given is too far away from the current instruction an error is given).

The table also gives the execution time of each instruction in T states (clock cycles). This can be used to calculate the execution time in microseconds by the following formula:

$$\text{Execution time in T states} = \frac{\text{Execution time in microseconds}}{\text{Clock rate in MHz}}$$

Some instructions have two times given (e.g. "JR C,disp" and "CPIR"). For conditional instructions these are condition true/condition false times. For the automatic repeat instructions these are execution times for one repetition, counter 0/counter <> 0.

| OBJECT CODE | SOURCE STATEMENT | | T-STATES |
|---|---|---|---|
| 8E | ADC | A,(HL) | 7 |
| | ADC | (HL) | |
| DD8Edd | ADC | A,(IX+disp) | 19 |
| | ADC | (IX+disp) | |
| FD8Edd | ADC | A,(IY+disp) | 19 |
| | ADC | (IX+disp) | |
| 8F | ADC | A,A | 4 |
| | ADC | A | |
| 88 | ADC | A,B | 4 |
| | ADC | B | |
| 89 | ADC | A,C | 4 |
| | ADC | C | |
| 8A | ADC | A,D | 4 |
| | ADC | D | |
| 8B | ADC | A,E | 4 |
| | ADC | E | |
| 8C | ADC | A,H | 4 |
| | ADC | H | |
| 8D | ADC | A,L | 4 |
| | ADC | L | |
| CEbb | ADC | A,byte | 7 |
| | ADC | byte | |
| ED4A | ADC | HL,BC | 15 |
| ED5A | ADC | HL,DE | 15 |
| ED6A | ADC | HL,HL | 15 |
| ED7A | ADC | HL,SP | 15 |
| 86 | ADD | A,(HL) | 7 |
| | ADD | (HL) | |
| DD86dd | ADD | A,(IX+disp) | 19 |
| | ADD | (IX+disp) | |
| FD86dd | ADD | A,(IY+disp) | 19 |
| | ADD | (IY+disp) | |
| 87 | ADD | A,A | 4 |
| | ADD | A | |
| 80 | ADD | A,B | 4 |

| OBJECT CODE | SOURCE STATEMENT | T-STATES |
|---|---|---|
| | ADD B | |
| 81 | ADD A,C | 4 |
| | ADD C | |
| 82 | ADD A,D | 4 |
| | ADD D | |
| 83 | ADD A,E | 4 |
| | ADD E | |
| 84 | ADD A,H | 4 |
| | ADD H | |
| 85 | ADD A,L | 4 |
| | ADD L | |
| C6bb | ADD A,byte | 7 |
| | ADD byte | |
| 09 | ADD HL,BC | 11 |
| 19 | ADD HL,DE | 11 |
| 29 | ADD HL,HL | 11 |
| 39 | ADD HL,SP | 11 |
| DD09 | ADD IX,BC | 15 |
| DD19 | ADD IX,DE | 15 |
| DD29 | ADD IX,IX | 15 |
| DD39 | ADD IX,SP | 15 |
| FD09 | ADD IY,BC | 15 |
| FD19 | ADD IY,DE | 15 |
| FD29 | ADD IY,IY | 15 |
| FD39 | ADD IY,SP | 15 |
| A6 | AND (HL) | 7 |
| DDA6dd | AND (IX+disp) | 19 |
| FDA6dd | AND (IY+disp) | 19 |
| A7 | AND A | 4 |
| A0 | AND B | 4 |
| A1 | AND C | 4 |
| A2 | AND D | 4 |
| A3 | AND E | 4 |
| A4 | AND H | 4 |
| A5 | AND L | 4 |
| E6bb | AND byte | 7 |
| CB46 | BIT 0,(HL) | 12 |
| DDCBdd46 | BIT 0,(IX+disp) | 20 |
| FDCBdd46 | BIT 0,(IY+disp) | 20 |
| CB47 | BIT 0,A | 8 |
| CB40 | BIT 0,B | 8 |
| CB41 | BIT 0,C | 8 |
| CB42 | BIT 0,D | 8 |
| CB43 | BIT 0,E | 8 |
| CB44 | BIT 0,H | 8 |
| CB45 | BIT 0,L | 8 |
| CB4E | BIT 1,(HL) | 12 |
| DDCBdd4E | BIT 1,(IX+disp) | 20 |
| FDCBdd4E | BIT 1,(IY+disp) | 20 |
| CB4F | BIT 1,A | 8 |
| CB48 | BIT 1,B | 8 |
| CB49 | BIT 1,C | 8 |
| CB4A | BIT 1,D | 8 |
| CB4B | BIT 1,E | 8 |
| CB4C | BIT 1,H | 8 |
| CB4D | BIT 1,L | 8 |
| CB56 | BIT 2,(HL) | 12 |
| DDCBdd56 | BIT 2,(IX+disp) | 20 |
| FDCBdd56 | BIT 2,(IY+disp) | 20 |
| CB57 | BIT 2,A | 8 |
| CB50 | BIT 2,B | 8 |
| CB51 | BIT 2,C | 8 |
| CB52 | BIT 2,D | 8 |
| Cb53 | BIT 2,E | 8 |
| CB54 | BIT 2,H | 8 |
| CB55 | BIT 2,L | 8 |
| CB5E | BIT 3,(HL) | 12 |
| DDCBDD5E | BIT 3,(IX+disp) | 20 |
| FDCBDD5E | BIT 3,(IY+disp) | 20 |
| CB5F | BIT 3,A | 8 |
| CB58 | BIT 3,B | 8 |

| OBJECT CODE | SOURCE STATEMENT | T-STATES |
|---|---|---|
| CB59 | BIT 3,C | 8 |
| CB5A | BIT 3,D | 8 |
| CB5B | BIT 3,E | 8 |
| CB5C | BIT 3,H | 8 |
| CB5D | BIT 3,L | 8 |
| CB66 | BIT 4,(HL) | 12 |
| DDCBdd66 | BIT 4,(IX+disp) | 20 |
| FDCBdd66 | BIT 4,(IY+disp) | 20 |
| CB67 | BIT 4,A | 8 |
| CB60 | BIT 4,B | 8 |
| CB61 | BIT 4,C | 8 |
| CB62 | BIT 4,D | 8 |
| Cb63 | BIT 4,E | 8 |
| CB64 | BIT 4,H | 8 |
| CB65 | BIT 4,L | 8 |
| CB6E | BIT 5,(HL) | 12 |
| DDCBDD6E | BIT 5,(IX+disp) | 20 |
| FDCBDD6E | BIT 5,(IY+disp) | 20 |
| CB6F | BIT 5,A | 8 |
| CB68 | BIT 5,B | 8 |
| CB69 | BIT 5,C | 8 |
| CB6A | BIT 5,D | 8 |
| CB6B | BIT 5,E | 8 |
| CB6C | BIT 5,H | 8 |
| CB6D | BIT 5,L | 8 |
| CB76 | BIT 6,(HL) | 12 |
| DDCBdd76 | BIT 6,(IX+disp) | 20 |
| FDCBdd76 | BIT 6,(IY+disp) | 20 |
| CB77 | BIT 6,A | 8 |
| CB70 | BIT 6,B | 8 |
| CB71 | BIT 6,C | 8 |
| CB72 | BIT 6,D | 8 |
| Cb73 | BIT 6,E | 8 |
| CB74 | BIT 6,H | 8 |
| CB75 | BIT 6,L | 8 |
| CB7E | BIT 7,(HL) | 12 |
| DDCBDD7E | BIT 7,(IX+disp) | 20 |
| FDCBDD7E | BIT 7,(IY+disp) | 20 |
| CB7F | BIT 7,A | 8 |
| CB78 | BIT 7,B | 8 |
| CB79 | BIT 7,C | 8 |
| CB7A | BIT 7,D | 8 |
| CB7B | BIT 7,E | 8 |
| CB7C | BIT 7,H | 8 |
| CB7D | BIT 7,L | 8 |
| DCllhh | CALL C,addr | 17/10 |
| FCllhh | CALL M,addr | 17/10 |
| D4llhh | CALL NC,addr | 17/10 |
| C4llhh | CALL NZ,addr | 17/10 |
| F4llhh | CALL P,addr | 17/10 |
| ECllhh | CALL PE,addr | 17/10 |
| E4llhh | CALL PO,addr | 17/10 |
| CCllhh | CALL Z,addr | 17/10 |
| CDllhh | CALL addr | 17 |
| 3F | CCF | 4 |
| BE | CP (HL) | 7 |
| DDBEdd | CP (IX+disp) | 19 |
| FDBEdd | CP (IY+disp) | 19 |
| BF | CP A | 4 |
| B8 | CP C | 4 |
| B9 | CP B | 4 |
| BA | CP D | 4 |
| BB | CP E | 4 |
| BC | CP H | 4 |
| BD | CP L | 4 |
| FEBB | CP byte | 7 |
| EDA9 | CPD | 16 |
| EDB9 | CPDR | 16/21 |
| EDA1 | CPI | 16 |
| EDB1 | CPIR | 16/21 |
| 2F | CPL | 4 |

| OBJECT CODE | SOURCE STATEMENT | | T-STATES |
|---|---|---|---|
| 27 | DAA | | 4 |
| 35 | DEC | (HL) | 11 |
| DD35dd | DEC | (IX+disp) | 23 |
| FD35dd | DEC | (IY+disp) | 23 |
| 3D | DEC | A | 4 |
| 05 | DEC | B | 4 |
| 0B | DEC | BC | 6 |
| 0D | DEC | C | 4 |
| 15 | DEC | D | 4 |
| 1B | DEC | DE | 6 |
| 1D | DEC | E | 4 |
| 25 | DEC | H | 4 |
| 2B | DEC | HL | 6 |
| DD2B | DEC | IX | 10 |
| FD2B | DEC | IY | 10 |
| 2D | DEC | L | 4 |
| 3B | DEC | SP | 6 |
| F3 | DI | | 4 |
| 10dd | DJNZ | disp | 8/13 |
| FB | EI | | 4 |
| E3 | EX | (SP),HL | 19 |
| DDE3 | EX | (SP),IX | 23 |
| FDE3 | EX | (SP),IY | 23 |
| 08 | EX | AF,AF’ | 4 |
| EB | EX | DE.HL | 4 |
| D9 | EXX | | 4 |
| 76 | HALT | | 4 |
| ED46 | IM | 0 | 8 |
| ED56 | IM | 1 | 8 |
| ED5E | IM | 2 | 8 |
| ED78 | IN | A,(C) | 12 |
| DBbb | IN | A,(byte) | 11 |
| ED40 | IN | B,(C) | 12 |
| ED48 | IN | C,(C) | 12 |
| ED50 | IN | D,(C) | 12 |
| ED58 | IN | E,(C) | 12 |
| ED60 | IN | H,(C) | 12 |
| ED68 | IN | L,(C) | 12 |
| 34 | INC | (HL) | 11 |
| DD34dd | INC | (IX+disp) | 23 |
| FD34dd | INC | (IY+disp) | 23 |
| 3C | INC | A | 4 |
| 04 | INC | B | 4 |
| 03 | INC | BC | 6 |
| 0C | INC | C | 4 |
| 14 | INC | D | 4 |
| 13 | INC | DE | 6 |
| 1C | INC | E | 4 |
| 24 | INC | H | 4 |
| 23 | INC | HL | 6 |
| DD23 | INC | IX | 10 |
| FD23 | INC | IY | 10 |
| 2C | INC | L | 4 |
| 33 | INC | SP | 6 |
| EDAA | IND | | 16 |
| EDBA | INDR | | 16/21 |
| EDA2 | INI | | 16 |
| EDB2 | INIR | | 16/21 |
| E9 | JP | (HL) | 4 |
| DDE9 | JP | (IX) | 8 |
| FDE9 | JP | (IY) | 8 |
| DAllhh | JP | C,addr | 10 |
| FAllhh | JP | M,addr | 10 |
| D2llhh | JP | NC,addr | 10 |
| C2llhh | JP | NZ,addr | 10 |
| F2llhh | JP | P,addr | 10 |
| EAllhh | JP | PE,addr | 10 |
| E2llhh | JP | PO,addr | 10 |
| CAllhh | JP | Z,addr | 10 |
| C3llhh | JP | addr | 10 |
| 38dd | JR | C,disp | 12/7 |

| OBJECT CODE | SOURCE STATEMENT | | T-STATES |
|---|---|---|---|
| 30dd | JR | NC,disp | 12/7 |
| 20dd | JR | NZ,disp | 12/7 |
| 28dd | JR | Z,disp | 12/7 |
| 18dd | JR | disp | 12 |
| 02 | LD | (BC),A | 7 |
| 12 | LD | (DE),A | 7 |
| 77 | LD | (HL),A | 7 |
| 70 | LD | (HL),B | 7 |
| 71 | LD | (HL),C | 7 |
| 72 | LD | (HL),D | 7 |
| 73 | LD | (HL),E | 7 |
| 74 | LD | (HL),H | 7 |
| 75 | LD | (HL),L | 7 |
| 36bb | LD | (HL),byte | 10 |
| DD77dd | LD | (IX+disp),A | 19 |
| DD70dd | LD | (IX+disp),B | 19 |
| DD71dd | LD | (IX+disp),C | 19 |
| DD72dd | LD | (IX+disp),D | 19 |
| DD73dd | LD | (IX+disp),E | 19 |
| DD74dd | LD | (IX+disp),H | 19 |
| DD75dd | LD | (IX+disp),L | 19 |
| DD36ddbb | LD | (IX+disp),byte | 19 |
| FD77dd | LD | (IY+disp),A | 19 |
| FD70dd | LD | (IY+disp),B | 19 |
| FD71dd | LD | (IY+disp),C | 19 |
| FD72dd | LD | (IY+disp),D | |
| FD73dd | LD | (IY+disp),E | 19 |
| FD74dd | LD | (IY+disp),H | 19 |
| FD75dd | LD | (IY+disp),L | 19 |
| FD36ddbb | LD | (IY+disp),byte | 19 |
| 32llhh | LD | (addr),A | 13 |
| ED43llhh | LD | (addr),BC | 20 |
| ED53llhh | LD | (addr),DE | 20 |
| 22llhh | LD | (addr),HL | 16 |
| DD22llhh | LD | (addr),IX | 20 |
| FD22llhh | LD | (addr),IY | 20 |
| ED73llhh | LD | (addr),SP | 20 |
| 0A | LD | A,(BC) | 7 |
| 1A | LD | A,(DE) | 7 |
| 7E | LD | A,(HL) | 7 |
| DD7Edd | LD | A,(IX+disp) | 19 |
| FD7Edd | LD | A,(IY+disp) | 19 |
| 3Allhh | LD | A,(addr) | 13 |
| 7F | LD | A,A | 4 |
| 78 | LD | A,B | 4 |
| 79 | LD | A,C | 4 |
| 7A | LD | A,D | 4 |
| 7B | LD | A,E | 4 |
| 7C | LD | A,H | 4 |
| ED57 | LD | A,I | 9 |
| 7D | LD | A,L | 4 |
| 3Ebb | LD | A,byte | 7 |
| ED5F | LD | A,R | 9 |
| 46 | LD | B,(HL) | 7 |
| DD46dd | LD | B,(IX+disp) | 19 |
| FD46dd | LD | B,(IY+disp) | 19 |
| 47 | LD | B,A | 4 |
| 40 | LD | B,B | 4 |
| 41 | LD | B,C | 4 |
| 42 | LD | B,D | 4 |
| 43 | LD | B,E | 4 |
| 44 | LD | B,H | 4 |
| 45 | LD | B,L | 4 |
| 06bb | LD | B,byte | 7 |
| ED4Bllhh | LD | BC,(addr) | 20 |
| 01llhh | LD | BC,word | 10 |
| 4E | LD | C,(HL) | 7 |
| DD4Edd | LD | C,(IX+disp) | 19 |
| FD4Edd | LD | C,(IY+disp) | 19 |
| 4F | LD | C,A | 4 |
| 48 | LD | C,B | 4 |

| OBJECT CODE | SOURCE STATEMENT | | T-STATES |
|---|---|---|---|
| 49 | LD | C,C | 4 |
| 4A | LD | C,D | 4 |
| 4B | LD | C,E | 4 |
| 4C | LD | C,H | 4 |
| 4D | LD | C,L | 4 |
| 0Ebb | LD | C,byte | 7 |
| 56 | LD | D,(HL) | 7 |
| DD56dd | LD | D,(IX+disp) | 19 |
| Fd56dd | LD | D,(IY+disp) | 19 |
| 57 | LD | D,A | 4 |
| 50 | LD | D,B | 4 |
| 51 | LD | D,C | 4 |
| 52 | LD | D,D | 4 |
| 53 | LD | D,E | 4 |
| 54 | LD | D,H | 4 |
| 55 | LD | D,L | 4 |
| 16bb | LD | D,byte | 7 |
| ED5Bllhh | LD | DE,(addr) | 20 |
| 11llhh | LD | DE,word | 10 |
| 5E | LD | E,(HL) | 7 |
| DD5Edd | LD | E,(IX+disp) | 19 |
| FD5Edd | LD | E,(IY+disp) | 19 |
| 5F | LD | E,A | 4 |
| 58 | LD | E,B | 4 |
| 59 | LD | E,C | 4 |
| 5A | LD | E,D | 4 |
| 5B | LD | E,E | 4 |
| 5C | LD | E,H | 4 |
| 5D | LD | E,L | 4 |
| 1Ebb | LD | E,byte | 7 |
| 66 | LD | H,(HL) | 7 |
| DD66dd | LD | H,(IX+byte) | 19 |
| FD66dd | LD | H,(IY+byte) | 19 |
| 67 | LD | H,A | 4 |
| 60 | LD | H,B | 4 |
| 61 | LD | H,C | 4 |
| 62 | LD | H,D | 4 |
| 63 | LD | H,E | 4 |
| 64 | LD | H,H | 4 |
| 65 | LD | H,L | 4 |
| 26bb | LD | H,byte | 7 |
| 2Allhh | LD | HL,(addr) | 16 |
| 21llhh | LD | HL,word | 10 |
| ED47 | LD | I,A | 9 |
| FD2Allhh | LD | IX,(addr) | 20 |
| FD21llhh | LD | IX,word | 14 |
| FD2Allhh | LD | IY,(addr) | 20 |
| FD21llhh | LD | IY,word | 14 |
| 6E | LD | L,(HL) | 7 |
| DD6Edd | LD | L,(IX+disp) | 19 |
| FD6Edd | LD | L,(IY+disp) | 19 |
| 6F | LD | L,A | 4 |
| 68 | LD | L,B | 4 |
| 69 | LD | L,C | 4 |
| 6A | LD | L,D | 4 |
| 6B | LD | L,E | 4 |
| 6C | LD | L,H | 4 |
| 6D | LD | L,L | 4 |
| 2Ebb | LD | L,byte | 7 |
| ED4F | LD | R,A | 9 |
| ED7Bllhh | LD | SP,(addr) | 20 |
| F9 | LD | SP,HL | 6 |
| DDF9 | LD | SP,IX | 10 |
| FDF9 | LD | SP,IY | 10 |
| 31llhh | LD | SP,word | 10 |
| EDA8 | LDDD | | 16 |
| EDB8 | LDDR | | 16/21 |
| EDA0 | LDI | | 16 |
| EDB0 | LDIR | | 16/21 |
| ED44 | NEG | | 8 |
| 00 | NOP | | 4 |

| OBJECT CODE | SOURCE STATEMENT | | T-STATES |
|---|---|---|---|
| B6 | OR | (HL) | 7 |
| DDB6dd | OR | (IX+disp) | 19 |
| FDB6dd | OR | (IY+disp) | 19 |
| B7 | OR | A | 4 |
| B0 | OR | B | 4 |
| B1 | OR | C | 4 |
| B2 | OR | D | 4 |
| B3 | OR | E | 4 |
| B4 | OR | H | 4 |
| B5 | OR | L | 4 |
| F6bb | OR | byte | 7 |
| EDBB | OTDR | | 16/21 |
| EDB3 | OTIR | | 16/21 |
| ED79 | OUT | (C),A | 12 |
| ED51 | OUT | (C),B | 12 |
| ED49 | OUT | (C),C | 12 |
| ED51 | OUT | (C),D | 12 |
| ED59 | OUT | (C),E | 12 |
| ED61 | OUT | (C),H | 12 |
| ED69 | OUT | (C),L | 12 |
| D320 | OUT | (byte),A | 11 |
| EDAB | OUTD | | 16 |
| EDA3 | OUTI | | 16 |
| F1 | POP | AF | 10 |
| C1 | POP | BC | 10 |
| D1 | POP | DE | 10 |
| E1 | POP | HL | 10 |
| DDE1 | POP | IX | 14 |
| FDE1 | POP | IY | 14 |
| F5 | PUSH | AF | 11 |
| C5 | PUSH | BC | 11 |
| D5 | PUSH | DE | 11 |
| E5 | PUSH | HL | 11 |
| DDE5 | PUSH | IX | 15 |
| FDE5 | PUSH | IY | 15 |
| CB86 | SET | 0,(HL) | 12 |
| DDCBdd86 | SET | 0,(IX+disp) | 20 |
| FDCBdd86 | SET | 0,(IY+disp) | 20 |
| CB87 | SET | 0,A | 8 |
| CB80 | SET | 0,B | 8 |
| CB81 | SET | 0,C | 8 |
| CB82 | SET | 0,D | 8 |
| CB83 | SET | 0,E | 8 |
| CB84 | SET | 0,H | 8 |
| CB85 | SET | 0,L | 8 |
| CB8E | SET | 1,(HL) | 12 |
| DDCBdd8E | SET | 1,(IX+disp) | 20 |
| FDCBdd8E | SET | 1,(IY+disp) | 20 |
| CB8F | SET | 1,A | 8 |
| CB88 | SET | 1,B | 8 |
| CB89 | SET | 1,C | 8 |
| CB8A | SET | 1,D | 8 |
| CB8B | SET | 1,E | 8 |
| CB8C | SET | 1,H | 8 |
| CB8D | SET | 1,L | 8 |
| CB96 | SET | 2,(HL) | 12 |
| DDCBdd96 | SET | 2,(IX+disp) | 20 |
| FDCBdd96 | SET | 2,(IY+disp) | 20 |
| CB97 | SET | 2,A | 8 |
| CB90 | SET | 2,B | 8 |
| CB91 | SET | 2,C | 8 |
| CB92 | SET | 2,D | 8 |
| Cb93 | SET | 2,E | 8 |
| CB94 | SET | 2,H | 8 |
| CB95 | SET | 2,L | 8 |
| CB9E | SET | 3,(HL) | 12 |
| DDCBDD9E | SET | 3,(IX+disp) | 20 |
| FDCBDD9E | SET | 3,(IY+disp) | 20 |
| CB9F | SET | 3,A | 8 |
| CB98 | SET | 3,B | 8 |
| CB99 | SET | 3,C | 8 |

| OBJECT CODE | SOURCE STATEMENT | T-STATES |
|---|---|---|
| CB9A | SET 3,D | 8 |
| CB9B | SET 3,E | 8 |
| CB9C | SET 3,H | 8 |
| CB9D | SET 3,L | 8 |
| CBA6 | SET 4,(HL) | 12 |
| DDCBddA6 | SET 4,(IX+disp) | 20 |
| FDCBddA6 | SET 4,(IY+disp) | 20 |
| CBA7 | SET 4,A | 8 |
| CBA0 | SET 4,B | 8 |
| CBA1 | SET 4,C | 8 |
| CBA2 | SET 4,D | 8 |
| CbA3 | SET 4,E | 8 |
| CBA4 | SET 4,H | 8 |
| CBA5 | SET 4,L | 8 |
| CBAE | SET 5,(HL) | 12 |
| DDCBDDAE | SET 5,(IX+disp) | 20 |
| FDCBDDAE | SET 5,(IY+disp) | 20 |
| CBAF | SET 5,A | 8 |
| CBA8 | SET 5,B | 8 |
| CBA9 | SET 5,C | 8 |
| CBAA | SET 5,D | 8 |
| CBAB | SET 5,E | 8 |
| CBAC | SET 5,H | 8 |
| CBAD | SET 5,L | 8 |
| CBB6 | SET 6,(HL) | 12 |
| DDCBddB6 | SET 6,(IX+disp) | 20 |
| FDCBddB6 | SET 6,(IY+disp) | 20 |
| CBB7 | SET 6,A | 8 |
| CBB0 | SET 6,B | 8 |
| CBB1 | SET 6,C | 8 |
| CBB2 | SET 6,D | 8 |
| CbB3 | SET 6,E | 8 |
| CBB4 | SET 6,H | 8 |
| CBB5 | SET 6,L | 8 |
| CBBE | SET 7,(HL) | 12 |
| DDCBDDBE | SET 7,(IX+disp) | 20 |
| FDCBDDBE | SET 7,(IY+disp) | 20 |
| CBBF | SET 7,A | 8 |
| CBB8 | SET 7,B | 8 |
| CBB9 | SET 7,C | 8 |
| CBBA | SET 7,D | 8 |
| CBBB | SET 7,E | 8 |
| CBBC | SET 7,H | 8 |
| CBBD | SET 7,L | 8 |
| C9 | RET | 10 |
| D8 | RET C | 11/5 |
| F8 | RET M | 11/5 |
| D0 | RET NC | 11/5 |
| C0 | RET NZ | 11/5 |
| F0 | RET P | 11/5 |
| E8 | RET PE | 11/5 |
| E0 | RET PO | 11/5 |
| C8 | RET Z | 11/5 |
| ED4D | RETI | 14 |
| ED45 | RETN | 14 |
| CB16 | RL (HL) | 15 |
| DDCBdd16 | RL (IX+disp) | 23 |
| FDCBdd16 | RL (IY+disp) | 23 |
| CB17 | RL A | 8 |
| CB10 | RL B | 8 |
| CB11 | RL C | 8 |
| CB12 | RL D | 8 |
| CB13 | RL E | 8 |
| CB14 | RL H | 8 |
| CB15 | RL L | 8 |
| 17 | RLA | 4 |
| CB06 | RLC (HL) | 15 |
| DDCBdd06 | RLC (IX+disp) | 23 |
| FDCBdd06 | RLC (IY+disp) | 23 |
| CB07 | RLC A | 8 |
| CB00 | RLC B | 8 |

| OBJECT CODE | SOURCE STATEMENT | T-STATES |
|---|---|---|
| CB01 | RLC C | 8 |
| CB02 | RLC D | 8 |
| CB03 | RLC E | 8 |
| CB04 | RLC H | 8 |
| CB05 | RLC L | 8 |
| 07 | RLCA | 4 |
| ED6F | RLD | 18 |
| CB1E | RR (HL) | 15 |
| DDCBdd1E | RR (IX+disp) | 23 |
| FDCBdd1E | RR (IY+disp) | 23 |
| CB1F | RR A | 8 |
| CB18 | RR B | 8 |
| CB19 | RR C | 8 |
| CB1A | RR D | 8 |
| CB1B | RR E | 8 |
| CB1C | RR H | 8 |
| CB1D | RR L | 8 |
| 1F | RRA | 4 |
| CB0E | RRC (HL) | 15 |
| DDCBdd0E | RRC (IX+disp) | 23 |
| FDCBdd0E | RRC (IY+disp) | 23 |
| CB0F | RRC A | 8 |
| CB08 | RRC B | 8 |
| CB09 | RRC C | 8 |
| CB0A | RRC D | 8 |
| CB0B | RRC E | 8 |
| CB0C | RRC H | 8 |
| CB0D | RRC L | 8 |
| 0F | RRCA | 4 |
| ED67 | RRD | 18 |
| C7 | RST #00 | 11 |
| CF | RST #08 | 11 |
| D7 | RST #10 | 11 |
| DF | RST #18 | 11 |
| E7 | RST #20 | 11 |
| EF | RST #28 | 11 |
| F7 | RST #30 | 11 |
| FF | RST #38 | 11 |
| 9E | SBC A,(HL) | 7 |
| | SBC (HL) | |
| DD9Edd | SBC A,(IX+disp) | 19 |
| | SBC (IX+disp) | |
| FD9Edd | SBC A,(IY+disp) | 19 |
| | SBC (IY+disp) | |
| 9F | SBC A,A | 4 |
| | SBC A | |
| 98 | SBC A,B | 4 |
| | SBC B | |
| 99 | SBC A,C | 4 |
| | SBC C | |
| 9A | SBC A,D | 4 |
| | SBC D | |
| 9B | SBC A,E | 4 |
| | SBC E | |
| 9C | SBC A,H | 4 |
| | SBC H | |
| 9D | SBC A,L | 4 |
| | SBC L | |
| DEbb | SBC A,byte | 7 |
| | SBC byte | |
| ED42 | SBC HL,BC | 15 |
| ED52 | SBC HL,DE | 15 |
| ED62 | SBC HL,HL | 15 |
| ED72 | SBC HL,SP | 15 |
| 37 | SCF | 4 |
| CBC6 | RES 0,(HL) | 12 |
| DDCBddC6 | RES 0,(IX+disp) | 20 |
| FDCBddC6 | RES 0,(IY+disp) | 20 |
| CBC7 | RES 0,A | 8 |
| CBC0 | RES 0,B | 8 |
| CBC1 | RES 0,C | 8 |

| OBJECT CODE | SOURCE STATEMENT | T-STATES |
|---|---|---|
| CBC2 | RES 0,D | 8 |
| CBC3 | RES 0,E | 8 |
| CBC4 | RES 0,H | 8 |
| CBC5 | RES 0,L | 8 |
| CBCE | RES 1,(HL) | 12 |
| DDCBddCE | RES 1,(IX+disp) | 20 |
| FDCBddCE | RES 1,(IY+disp) | 20 |
| CBCF | RES 1,A | 8 |
| CBC8 | RES 1,B | 8 |
| CBC9 | RES 1,C | 8 |
| CBCA | RES 1,D | 8 |
| CBCB | RES 1,E | 8 |
| CBCC | RES 1,H | 8 |
| CBCD | RES 1,L | 8 |
| CBD6 | RES 2,(HL) | 12 |
| DDCBddD6 | RES 2,(IX+disp) | 20 |
| FDCBddD6 | RES 2,(IY+disp) | 20 |
| CBD7 | RES 2,A | 8 |
| CBD0 | RES 2,B | 8 |
| CBD1 | RES 2,C | 8 |
| CBD2 | RES 2,D | 8 |
| CbD3 | RES 2,E | 8 |
| CBD4 | RES 2,H | 8 |
| CBD5 | RES 2,L | 8 |
| CBDE | RES 3,(HL) | 12 |
| DDCBDDDE | RES 3,(IX+disp) | 20 |
| FDCBDDDE | RES 3,(IY+disp) | 20 |
| CBDF | RES 3,A | 8 |
| CBD8 | RES 3,B | 8 |
| CBD9 | RES 3,C | 8 |
| CBDA | RES 3,D | 8 |
| CBDB | RES 3,E | 8 |
| CBDC | RES 3,H | 8 |
| CBDD | RES 3,L | 8 |
| CBE6 | RES 4,(HL) | 12 |
| DDCBddE6 | RES 4,(IX+disp) | 20 |
| FDCBddE6 | RES 4,(IY+disp) | 20 |
| CBE7 | RES 4,A | 8 |
| CBE0 | RES 4,B | 8 |
| CBE1 | RES 4,C | 8 |
| CBE2 | RES 4,D | 8 |
| CbE3 | RES 4,E | 8 |
| CBE4 | RES 4,H | 8 |
| CBE5 | RES 4,L | 8 |
| CBEE | RES 5,(HL) | 12 |
| DDCBDDEE | RES 5,(IX+disp) | 20 |
| FDCBDDEE | RES 5,(IY+disp) | 20 |
| CBEF | RES 5,A | 8 |
| CBE8 | RES 5,B | 8 |
| CBE9 | RES 5,C | 8 |
| CBEA | RES 5,D | 8 |
| CBEB | RES 5,E | 8 |
| CBEC | RES 5,H | 8 |
| CBED | RES 5,L | 8 |
| CBF6 | RES 6,(HL) | 12 |
| DDCBddF6 | RES 6,(IX+disp) | 20 |
| FDCBddF6 | RES 6,(IY+disp) | 20 |
| CBF7 | RES 6,A | 8 |
| CBF0 | RES 6,B | 8 |
| CBF1 | RES 6,C | 8 |
| CBF2 | RES 6,D | 8 |
| CbF3 | RES 6,E | 8 |
| CBF4 | RES 6,H | 8 |
| CBF5 | RES 6,L | 8 |
| CBFE | RES 7,(HL) | 12 |
| DDCBDDFE | RES 7,(IX+disp) | 20 |
| FDCBDDFE | RES 7,(IY+disp) | 20 |
| CBFF | RES 7,A | 8 |
| CBF8 | RES 7,B | 8 |
| CBF9 | RES 7,C | 8 |
| CBFA | RES 7,D | 8 |

| OBJECT CODE | SOURCE STATEMENT | T-STATES |
|---|---|---|
| CBFB | RES 7,E | 8 |
| CBFC | RES 7,H | 8 |
| CBFD | RES 7,L | 8 |
| CB26 | SLA (HL) | 15 |
| DDCDdd26 | SLA (IX+disp) | 23 |
| FDCDdd26 | SLA (IY+disp) | 23 |
| CB27 | SLA A | 8 |
| CB20 | SLA B | 8 |
| CB21 | SLA C | 8 |
| CB22 | SLA D | 8 |
| CB23 | SLA E | 8 |
| CB24 | SLA H | 8 |
| CB25 | SLA L | 8 |
| CB2E | SRA (HL) | 15 |
| DDCDdd2E | SRA (IX+disp) | 23 |
| FDCDdd2E | SRA (IY+disp) | 23 |
| CB2F | SRA A | 8 |
| CB28 | SRA B | 8 |
| CB29 | SRA C | 8 |
| CB2A | SRA D | 8 |
| CB2B | SRA E | 8 |
| CB2C | SRA H | 8 |
| CB2D | SRA L | 8 |
| CB3E | SRL (HL) | 15 |
| DDCDdd3E | SRL (IX+disp) | 23 |
| FDCDdd3E | SRL (IY+disp) | 23 |
| CB3F | SRL A | 8 |
| CB38 | SRL B | 8 |
| CB39 | SRL C | 8 |
| CB3A | SRL D | 8 |
| CB3B | SRL E | 8 |
| CB3C | SRL H | 8 |
| CB3D | SRL L | 8 |
| 96 | SUB (HL) | 7 |
| DD96dd | SUB (IX+disp) | 19 |
| DD96dd | SUB (IY+disp) | 19 |
| 97 | SUB A | 4 |
| 90 | SUB B | 4 |
| 91 | SUB C | 4 |
| 92 | SUB D | 4 |
| 93 | SUB E | 4 |
| 94 | SUB H | 4 |
| 95 | SUB L | 4 |
| D6bb | SUB byte | 7 |
| AE | XOR (HL) | 7 |
| DDAEdd | XOR (IX+disp) | 19 |
| FDAEdd | XOR (IY+disp) | 19 |
| AF | XOR A | 4 |
| A8 | XOR B | 4 |
| A9 | XOR C | 4 |
| AA | XOR D | 4 |
| AB | XOR E | 4 |
| AC | XOR H | 4 |
| AD | XOR L | 4 |
| EEbb | XOR byte | 7 |

## Appendix B - Immmiediate Error Messages And Their Meanings

NOTE: The PRINT immediate command can give the following error messages after evaluating its expression operand. These messages are described in appendix C.

```
** unknown name **
** undefined name **
** division by 0 **
** overflow **
```

`** “expected **`

A string has not been properly terminated with a double quote.

`** bad code range **`

The start address given to the CODE command is greater than the end.

`** bad line range **`

A line range given to an immediate command has its start line number greater than its end.

`** can’t wipe **`

An attempt has been made to wipe the current source file (Spectrum 128k only).

`** comment or end of line expected **`

There is some text following a legal statement or immediate command which is not necessary.

`** digit expected **`

The expression parser is expecting a number in a particular base and the first character it has found is not a digit in this base.

`** expression expected **`

You have not given an expression operand to an immediate command which requires one.

`** expression too complex **`

An expression in the current sentence contains too great a depth of parenthesis.

`** file too long **`

The file you are trying to load is too big to fit into the currently allotted editor space. Use the SET SPACE immediate command to give yourself more room.

`** illegal file type **`

You have attempted to LOAD/IMPORT a file which was not produced by Genius’ SAVE/EXPORT command.

```
** illegal first operand **
** illegal operand **
** illegal second operand **
```

One of the operands given to an opcode mnemonic or assembler directive is not allowed in this position.

`** line range overlap **`

The line ranges given to the COPY or MOVE immediate commands overlap. This is not allowed.

`** line range too long **`

There is not enough space to COPY the line range specified to a new position. Use the SET SPACE command to give yourself more room.

`** line too long **`

The tokenised form of the whole line-numbered block you have just tried to enter is too big. You should split the block into smaller ones.

`** mismatched []’s **`

Parentheses in the expression are not paired correctly.

`** misplaced ( or ) **`

You have not correctly enclosed a Z80 operand in parentheses e.g. (HL), or there is a mismatch in parentheses enclosing a (hash extension) function’s argument list.

`** misplaced operand **`

You have used a reserved name in an expression e.g. one of the register names.

`** name too long **`

Names may be up to 240 characters long.

`** no buffer space **`

The LOAD ASCII immediate command uses a 2K input buffer in symbol table space. This message is issued if insufficient space is available. You could use SET SPACE to give the editor less room (increasing the symbol table space), or CLEAR the symbol table.

`** no line number **`

The syntax checker cannot find a line number for the statement in the current sentence to be entered under.

`** number expected **`

You have not given a (decimal) number as operand to an immediate command which requires one.

`** number to large **`

A number in the current sentence is too big. This can mean greater than 65534 for line numbers or 65535 for other numbers.

`** opcode expected **`

This error message will be issued whenever the syntax checker cannot find an opcode or an immediate command name to match the start of the input sentence.

`** operand expected **`

The expression parser expects an operand (number, symbol or parenthesised expression) after an operator. This error can be given if you have typed a symbol which cannot belong to an operand by mistake.

`** operand too many **`

You have given too many separate operands to an opcode or assembler directive.

`** operator expected **`

The expression parser expects an operator (see appendix 0) at the position indicated.

`** separator expected **`

The syntax checker expects a “,” or a “;” at the position indicated.

`** space expected **`

The syntax checker expects a space after an opcode mnemonic.

`** space too large **`

The amount of space requested in a SET SPACE command is too big (even if the symbol table was to be deleted).

`** space too low **`

You have requested less than 1024 bytes of space in a SET SPACE command. This is not allowed.

# Appendix C - Assembler Error Messages and Their Meanings

## *a) Warnings*

`** division by 0 **`

An expression in the statement contains an attempt to divide by zero.

`** ENDM with no MACRO **`

An ENDM pseudo-op has been found in the source without a preceding MACRO statement. It will be ignored.

`** label expected **`

The instruction in the statement expects a label to precede it. Such instructions are DL (or DEFL), EQU, CARGO. The instruction will be ignored; it has nothing to work on.

(MACRO also requires a label but omitting it generates an error.)

`** misplaced CARGO **`

A CARGO pseudo-op has been used while a local table is being referenced i.e. within a MACRO expansion or #FNC definition. Only global names may be declared as CARGO.

`** number out of range **`

The expression given as the operand of a RST instruction does not evaluate to one of 0, #8, #10, #18, #20, #28, #30 or #38.

`** number out of range 0, 2 **`

The expression given as operand to an IM instruction evaluates to something other than 0, 1 or 2.

`** number out of range 0,7 **`

The expression given as the bit number in a BIT, SET or RES instruction evaluates to something other than 0, 1, 2, 3, 4, 5, 6 or 7.

`** number out of range -128, 127 **`

The index expression in an instruction involving the IX or IY index registers is out of range. Or a relative jump has been specified to a location which is out of the range of the instruction. (JR, DJNZ or a conditional JR).

`** number out of range -128, 255 **`

An expression in the instruction which is required to be a byte-sized piece of immediate data has evaluated to a greater than byte-size value.

The range here is -128 to 255 to allow you to write either

```
LD A,255
```
or
```
LD A,-1
```

as you wish.

`** overflow **`

An overflow has occurred during evaluation of an expression in the statement.

`** PUT with tape/disc output **`

A PUT directive has been used while object code is being written to backing store. The PUT will be ignored.

This message will only be issued on pass 1.

`** undefined name **`

An expression in the statement contains a reference to an undefined name. For example writing:

```
name2: equ name1+#FE
```

Where “name1” is unknown will cause “name2” to be known but undefined. Thus a use of “name2” will give the “undefined name” error.

This warning will only be given on pass 2 in case of dependencies on as yet undefined names.

** unknown name **

An expression in the statement contains a reference to a name which you have not attempted to define. The name in question will be inserted in the symbol table and marked as “missing” so that you can use the immediate command “MISSING” to find all such names.

This warning will only be given on pass 2, since on pass 1 the assembler will not know if the name in question is to be defined further down the text.

### *b) Errors*

NOTE: The following four errors can also be given at the end of pass 1 when the assembler detects that, although the object code started off in a legal position it has since grown into protected memory. Assembly will be terminated. However since the object is not written into memory on pass 1 no harm will have been done.

** code in name table **

You have attempted to PUT the object code into the assembler’s symbol table.

** code in program **

You have attempted to PUT the object code into Genius.

** code too high **

You have attempted to PUT the object code above the assembler’s high limit i.e. into ROM variable space or the Amstrad jump block.

** code too low **

You have attempted to PUT the object code below the assembler’s low limit (default value is #40 on the AMSTRAD and RAMTOP on the Spectrum).

** label expected **

The pseudo-op MACRO requires a label which gives the macro’s name. This error will be given on pass 1.

** MACRO name expected **

You have attempted to use a non-existent macro (or have given the wrong name) in a “\macro name>” construction.

** misplaced MACRO name **

You have used a macro parameter construction (\<name>) outside a macro expansion.

** multiple name definition **

The name given in the label field of the statement has already been defined. (Note that local names, those within macros and functions, may be re-used in different local situations).

This error will be given on pass 1.

** undefined expression **

Some pseudo-ops and directives need their operand expression to be well defined even on pass 1. They are: ORG, COND, PUT, DS, *WHILE, *UNTIL, #DS. This error is fatal in all cases except DS and #DS.

This error will be given on pass 1.

### *c) Fatal Errors*

** bad nesting **

The assembler has detected a mismatch in one of the following pairs of pseudo-ops/directives.

```
COND-ENDC             COND- ELSE                ELSE-ENDC
                     MACRO- ENDM
                  *OPENOUT- *CLOSEOUT
                    *WHILE- *ENDW
                   *REPEAT- *UNTIL
                #FNC-#BEGIN #BEGIN-#END
```

This error can also occur at the end of pass 1 when the assembler finds that its stack is not empty as it should be.

** MACRO nesting **

You have attempted to define a macro while a local table is being referenced i.e. within a MACRO expansion or #FNC definition.

`** nesting too deep **`

There is no space left on the assembler's stack. This is used by the pseudo-op COND and the directives *OPENOUT, *WHILE, *RE PEAT, #FNC, #IF, #WHILE, #REPEAT.

`** no buffer space **`

There is no space left for the assembler to open an input/output buffer. Such buffers are required by *INCLUDE, *OPENOUT.

`** no table space **`

The assembler has run Out of room for the name table. This error can also be generated by uses of MACRO, *WHILE, *TITLE, #IF, #WHILE which use table space for storage.

`** tape/disc WHILE too long **`

A *WHILE loop being assembled from tape/disc (or microdrive) does not fit into one line-numbered line.

`** too few parameters **`

You have given a macro use construction (\ <macro name> ) too few parameters. (i.e. A parameter referenced during macro expansion has been found at a position in the macro's dummy parameter list which would mean its replacement is "off the end" of the actual parameter list.

`** unavailable command **`

You have attempted to assemble source containing # extension references without the hash command extensions loaded.

# Appendix D - Arithmetic Expressions and Operator Precedence

## *a) The Difference Between Signed and Unsigned Arithmetic*

All arithmetic operations are carried out on 16 bit integers. However, it is often possible to specify whether you want calculations to be done with signed or unsigned arithmetic e.g. “PRINT” uses signed, “UPRINT” uses unsigned. Signed arithmetic uses the twos complement method for sign representation. That is, if n is a positive number -n is represented internally by 65536-n. So -1, for instance, becomes 65535=#FFFF.

The difference between the two is really in the error checking performed during the operations. If we add two numbers together unsigned then overflow occurs when we go over 65535=#FFFF, on the other hand using signed arithmetic overflow occurs when we go over 32767=#7FFF, since we have then added two positive numbers and got a negative result!

## *b) Note on Expression Evaluation*

Genius’ expression evaluator detects uses of unknown or undefined names during the calculation of a result. Thus it is possible even on pass 1 to detect overflow and out-of-range errors correctly in some cases (those in which no names are used or where all names used are well defined). The assembler will report these errors as early as possible (pass 1) if it is sure that the expression already evaluates to its final result.

The evaluator cannot, however, tell the difference between large positive values and small negative ones. So the assembler will accept an instruction such as:

```
LD A,#FFFF
```

The actual code assembled will be (the sensible):

```
LD A,#FF
```

Writing

```
LD A,-1
```

is therefore acceptable.

## *c) Operator Precedence*

The following table gives the relative “binding power” of the arithmetic operators which may be used in Genius expressions (including those operators only available to compiler expressions).

The table is in order of decreasing precedence, all operators on the same line having equal precedence.

| | OPERATOR | EVALUATION |
|---|---|---|
| | [...] (...) | |
| (unaries) | & ++ _ * ! ^ - | right to left |
| (binaries) | * / % | left to right |
| | + - | left to right |
| | << >> @< @> | left to right |
| | > < >= <= | left to right |
| | ?= != | left to right |
| | & \| ^ | left to right |
| | && \|\| | left to right |
| | = | right to left |

### The Meaning Of Operator Precedence

When you work out the value of, for instance:

```
1+2*3
```

(and get the result 7) you have been applying the rules of operator precedence. The multiplication must be done before the addition because the “*” operator has higher precedence than the “+”. If you had intended to do the addition first you would have had to write:

```
        1+2*3
```

From the table above you can see that parentheses “[” and “]” have the highest binding power of all. So using them in this way causes the expression inside them to be evaluated before anything else.

This idea can be generalised to all the operators allowed in Genius’ expressions.

You will occasionally need to be careful in your use of some of these operators. Look at the following example from a hash extension expression.

```
#IF c=square(x)?=100
```

This would be evaluated as:

```
c=[square(x)?=100]
```

Which is really setting “c” to a TRUE or FALSE flag. What may have been intended is:

```
[c=square(x)] ?=100
```

Which first assigns “square(x)” to “c” and then tests to see if this was 100.

If in doubt use parenthesis “[” and “]”.

### *d) Operator Descriptions*

Unary operators:

| | |
|---|---|
| – | Unary minus or negation. e.g UPRINT -1 gives #FFFF, the twos complement representation of minus one, as result. |
| ! | Logical complement or NOT. “!” will turn a true flag into a false flag and vice versa. In general any non-zero value will be treated as “true” and only zero as “false”. |
| ^ | Bitwise (ones) complement. e.g. !%101010 gives %010101 as result. |
| * | This operator should be read as “contents of”. It has many uses in compiled expressions (see 5. “The Hash Extensions”) but in assembler expressions it acts as the “DEEK” function which may be found in many versions of BASIC. i.e. *<value> returns the value of the word in memory whose start address is <value>. |
| & | This operator should be read as “address of”. It is only available to compiled expressions and is used to find the address of a variables storage space. (See 5. “The Hash Extensions”). |
| ++ | This operator is only available to compiled expressions. It increments its operand and returns the new value of this operand i.e. the value after incrementation. (See 5. “The Hash Extensions”). |
| -- | This operator is only available to compiled expressions. It decrements its operand and returns the new value of this operand i.e. the value after decrementation. (See 5. “The Hash Extensions”). |

Binary Operators:

### Addition and Multiplication

| | |
|---|---|
| + | Addition |
| – | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modulus i.e. a%b returns the value of a modulo b. This will always be a positive integer, even when signed arithmetic is used, for example [-10] %3 will give 2 as result. |

## Shifts and Rotates

<< Shift left, i.e. x<<n shifts x left by n bits.

>> Shift right.

@< Rotate left. i.e. x@<n rotates x left by n bits.

@> Rotate right.

## Relational Operators

< Less than a<b returns a true flag (1) if a is less than b, and a false flag (0) otherwise.

> Greater than a>b returns a true flag (1) if a is greater than band a false flag (0) otherwise.

<= Less than or equal.

>= Greater than or equal.

## Logical Operators

Note that in general "true" or "true flag" may be taken to mean any non-zero value. While "false" or "false flag" means zero.

&& Logical AND. a&&b returns a value according to the truth table:

| a | b | a&&b |
|---|---|---|
| true | true | true |
| false | true | false |
| true | false | false |
| false | false | false |

|| Logical OR. a||b returns a value according to the following truth table:

| a | b | a\|\|b |
|---|---|---|
| true | true | true |
| false | true | true |
| true | false | true |
| false | false | false |

## Bitwise Logical Operators

& Bitwise AND. Operates like the logical AND (&&) on corresponding bits of its operands where a non-zero bit is taken to be "true" and a zero bit "false". Note that this is not the same as logical AND, for instance:

```
2 && 1 gives 1 (=true)
```

while

```
2 & 1 gives 0 (=false)
```

since in the second case the two non-zero bits are not in corresponding positions.

! Bitwise OR. Operates like the logical OR (||) on corresponding bits of its operands.

^ Bitwise Exclusive OR. i.e. This operator acts according to the following truth table on corresponding bits of its operands where a non-zero bit is taken to be "true", and a zero bit "false".

| a | b | a&&b |
|---|---|---|
| true | true | false |
| false | true | true |
| true | false | true |

| | | |
|---|---|---|
| false | false | false |

### Suffix Operators:

`++` This operator is available only to compiled expressions. It increments its operand but returns the original value of this operand i.e. the value before incrementation. (See 5. “The Hash Extensions”).

`--` This operator is available only to compiled expressions. It decrements its operand but returns the original value of this operand i.e. the value before decrementation. (See 5. “The Hash Extensions”).

### *e) Parenthesis, Arrays and Functions*

In both assembler and compiler (hash extension) expressions the square brackets are used to denote parenthesis. These are also used to introduce array indices in compiler expressions. Round brackets are used only for function references in compiler expressions.

A function reference in a compiler expression should be of the following form:

```
<function_name> ()
```

or

```
<function_name> (<parameter list>)
```

Where <parameter list> means a list of expressions separated by commas. (See ‘5. The Hash Extensions’).

## Appendix E - The Memory Map

The Genius assembler memory map is as shown below. The actual addresses involved will depend on which optional portions of the program you have loaded, and (on the AMSTRAD) what, if any, expansion ROMs are fitted.

```
High Memory
            Tool kit commands, if loaded
            Main Assembler Program
            Variable Storage Space
            Hash extensions, if loaded
            2K Tape/Disc Buffer
            Screen Buffer and Workspace
            Source Text File
            Symbol table growing downwards in memory
Low Memory
```

# Appendix F - Assembler Command Summary

| COMMAND | PARAMETER | ACTION |
|---|---|---|
| ASSEM | | Clear the assembler's symbol table and assemble the current source text. |
| ASSEMC | | Preserve the current symbol table and assemble the current source text. |
| ASSEML | | Selectively assemble a subroutine library. |
| BASE | <2, 8, 10 or 16> | Set the default base used by PRINT and UPRINT. |
| CAT | | Display tape/disc/microdrive directory. |
| CLEAR | | Clear the current symbol table. |
| CLS | | Clear the visible screen and the editor's copy in memory. |
| CODE | "<string>", <expression1>,<expression2> | Save the block of memory between expression1 and expression2 as a binary file named <string>. |
| COPY | <line range1>, <line range2> | Replace line range 2 with line range 1, retaining line range l. |
| DELETE | | Delete the whole source file. |
| DELETE | <line range> | Delete the specified line range from the source file. |
| DISC | | Direct the firmware to read from, and write to, disc (Amstrad disc only). |
| DISC.I N | | Direct the firmware to take input from disc (Amstrad disc only). |
| DISC.OUT | | Direct the firmware to write to disc (Amstrad disc only). |
| DRIVE A | | Set the current drive to "A" (Amstrad disc only). |
| DRIVE B | | Set the current drive to "B" (Amstrad disc only). |
| ERA | "<string>" | Erase file/files named <string> (Disc/Microdrive only). |
| EXECUTE | <expression> | Call the code at the address to which the expression evaluates. |
| EXPORT | "<string>" | Save the current symbol table to tape/disc/microdrive. |
| EXIT | | Return to whatever CALLed the editor/assembler. |
| FIND | "<string>", <Linerange> | String search. |
| FORM | | Issue a form feed to the printer. |
| IMPORT | "<string>" | Merge a previously EXPORTed table with the current table. |
| LENGTH | | Sets the length, in lines, of the printer page to 65536. |
| LENGTH | <decimal integer> | Set the length, in lines, of the printer page to the selected value. |
| LIST | | List the whole source file to the screen. |
| LIST | <line range> | List the specified line range of the source file to the screen. |
| LMISSING | | List to the printer, all those symbols which the program has referenced but not defined in the last piece of source code assembled, with a field width of 16. |
| LMISSING | <decimal integer> | List to the printer, all those symbols which the program has referenced but not defined in the last piece of source code assembled, with a field width given by the decimal integer. |
| LOAD | "<string>" | Load the file named <string>. |

| COMMAND | PARAMETER | ACTION |
|---|---|---|
| LOAD | "<string>", <line range> | Load the file named <string> and replace the text in line range with the contents of the file. |
| LOAD ASCII | "<string>" | Load a pure ASCII file named <string> and produce a tokenised file with 10 statements per line number. |
| LOAD ASCII | "<string>", <options> | Load a source file named <string> of the type indicated by <options> and produce a tokenised file, with 10 statements per line number. |
| LOAD ASCII | "<string>", <options> <decimal integer> | Load a source file named <string> of the type indicated by <options> and produce a tokenised file, with statements per line selected by the decimal integer. |
| LLI ST | | List the whole source file to the printer. |
| LLIST | <line range> | List the specified line range of the source file to the printer. |
| LTABLE | | List to the printer, the current symbol table, in the order of its ASCII sort with a field of width l6. |
| LTABLE | <decimal integer> | List to the printer, the current symbol table, in the order of its ASCII sort, with a field width given by the decimal integer. |
| LTABLEN | | List to the printer, the current symbol table, in numerical order, with a field width of l6. |
| LTABLEN | <decimal integer> | List to the printer, the current symbol table, in numerical order, with a field width given by the decimal integer. |
| LUNUSED | | List to the printer, all those symbols defined but not referenced, with a field width of 16. |
| LUNUSED | <decimal integer> | List to the printer, all those symbols defined but not referenced, with a field width given by the decimal integer. |
| MARGIN | | Set the left hand margin to zero. |
| MARGIN | <decimal integer> | Set the left hand margin to the selected value. |
| MDRV | | Direct the assembler to read from and write to, microdrive. Default drive is drive l (Spectrum only). |
| MDRV | <microdrive number> | Direct the assembler to read from and write to microdrive. Default drive is specified by <microdrive number> (Spectrum only). |
| MDRV.IN | | Direct the assembler to take input from microdrive (Spectrum only). |
| MDRV.OUT | | Direct the assembler to send output to microdrive (Spectrum only). |
| MISSING | | List to the screen, all those symbols, which the program has referenced but not defined in the last piece of source code assembled, with a field width of l6. |
| MISSING | <decimal integer> | List to the screen, all those symbols, which the program has referenced but not defined in the last piece of source code assembled, with a field width given by the decimal integer. |
| MODE | <0, l or 2> | Change screen mode (Amstrad only). |
| MOVE | <line range 1>, <line range 2> | Replace line range2with line range l, retaining line range l. |
| PRINT | <expression> | Print the signed value of the expression in the current base. |
| PRINT | <expression>, <base> | Print the signed value of the expression in the selected base. |

| COMMAND | PARAMETER | ACTION |
|---|---|---|
| REDUCE | | Remove all symbols which were not specified as CARGO during the last disassembly, from the symbol table. |
| REN | "<stringl>", "<string2>" | Rename a file named <string2>, <stringl> (Amstrad disc only). |
| RENUM | <new start>, <step>, <old start> | Renumber source text paragraphs. |
| REPLACE | "<string1>", "<stringl>" "<string2>" <line range> | Search and replace. |
| SAVE | "<string>" | Save the source text to a file named <string>. |
| SAVE | "<string>", <line range> | Save the given line range of source to a file named <string>. |
| SET SPACE | <decimal integer> | Adjust editing space. |
| STATS | | List memory map information. |
| TABLE | | List to the screen, the current symbol table, in the order of its ASCII sort, with a field width of l6. |
| TABLE | <decimal integer> | List to the screen, the current symbol table, in order of its ASCII sort, with a field width given by the decimal integer. |
| TABLEN | | List to the screen, the current symbol table, in numerical order, with a field width of l6. |
| TABLEN | <decimal integer> | List to the screen, the current symbol table, in numerical order, with a field width given by the decimal integer. |
| TAPE | | Direct the firmware to read from, and write to, tape. |
| TAPE.IN | | Direct the firmware to take input from tape. |
| TAPE.OUT | | Direct the firmware to write to tape. |
| UNUSED | | List to the screen, all those symbols defined but not referenced, with a field width of 16. |
| UNUSED | <decimal integer> | List to the screen, all those symbols defined but not referenced, with field width given by the decimal integer. |
| UPRINT | <expression> | Print the unsigned value of the expression in the current base. |
| UPRINT | <expression>, <base> | Print the unsigned value of the expression in the selected base. |
| VERIFY | "<string>" | Attempt to verify the source file named <string> against the source file in memory. |
| VERIFY | "<string>", <line range> | Attempt to verify the line range "<string>" of the source file named against the line range of the source file in memory. |
| WIDTH | | Set the width of the printer page to 65536. |
| WIDTH | <decimal integer> | Set the width of the printer to the selected value. |

# Appendix G - Spectrum 128K Assembler: Differences and Additions

### *Introduction*

The 128K version of the assembler has all the features of the standard version but because of the RAM paging facility, resides in fixed memory in RAM page 3. The extra memory is utilised in such a way that a number of files can be co-resident and large files can be assembled in RAM. This makes assembly very fast indeed and when, in addition, the monitor is co-resident, the program development cycle is greatly enhanced.

### *Tape Map*

On Tape I of the Genius package you will find the following files:

Side A:

| | |
|---|---|
| GEN128 | A BASIC loader. |
| ASM | Assembler object file. |

Side B:

| | |
|---|---|
| TRANS | Transfer loader (Spectrum only). |
| TRANSBIN | Transfer utility (Spectrum only). |
| SIEVE.ASM | Sieve of Eratosthenes example program in assembly language. |
| SIEVE.PHX | Sieve of Eratosthenes example program in PHOENIX. |
| ELLIPSE.ASM | Ellipse drawing program in assembly language. |
| ELLIPSE.PHX | Ellipse drawing program in PHOENIX. |
| MPAFNCS.PHX | Multiple precision arithmetic routines in PHOENIX. |

### *1. Operating Instructions*

Tape: To load the assembler from tape, use the up and down arrow keys to select the “TAPE LOADER” option and press ENTER. The assembler will load and auto-run.

Microdrive: If you have produced, and wish to load, a microdrive version of the assembler you should use the up and down arrow keys to select the “128K BASIC” option and press ENTER. Having entered BASIC the assembler can be loaded using:

```
LOAD *”m”;1;”GEN128”
```

**NOTE:** The Hash extensions and tools will always be loaded and initialised in the Spectrum 128k version.

### *2. Differences*

SET SPACE The SET SPACE command is no longer required. The screen buffer will be fixed and source will use pages 0, 1, 6, 7 (unless protected, see Section 3 below).

PUT <expression> [,<page>] This pseudo-op functions in the same way as that described in Section 4.3.5 of the main text, but has an additional optional parameter. If <page> is given, then the specified RAM page will be paged into #C000-#FFFF before any code is generated. If the <page> is not specified then a default value of 0 is used.

**NOTE:** Object code may only be written into protected RAM pages (or free memory below #C000). On entering the assembler, RAM page 0 is protected and can be used for the output of object code. If an attempt is made to PUT object code into memory containing the assembler or its workspace (source files etc.) then an error will be issued.

The lowest byte of memory that is free for object code storage is given by the low address displayed by the STATS command.

STATS

This immediate command is similar to that described in Section 2.9 of the main text, but has been extended to give further RAM paging information. The following will be displayed:

```
**low <addr>
<1st program name> <length>
<2nd program name> <length>
<Last program name> <length>
Page 0 <memory left> or '_' if protected
Page 1 <memory left> or '_' if protected
Page 6 <memory left> or '_' if protected
Page 7 <memory left> or '_' if protected
```

EXECUTE <expression> [,<page>]

This immediate command is similar to that described in Section 2.9 of the main text but an additional optional parameter is included. If <page> is given then the specified RAM page will be paged into #C000-#FFFF before the CALL is executed. If <page> is not specified then a default value of 0 is used.

CODE

This immediate command is similar to that described in Section 2.6 of the main text, but an additional optional parameter is included. If <page> is specified then the specified RAM page will be paged into #C000-#FFFF before any code is saved. If <page> is not specified then a default value of 0 is used.

### 3. Additional Commands

The assembler can keep more than one source file in memory at any one time. Each file (called a "program") will have a name associated with it. The editor will work on the current file, which on entering the assembler will be called "NONAME". In effect the extra memory is used as a RAM disc to speed up assembly and accommodate larger files. The following new commands are provided:

PROGRAM "<name>"

Make the program called <name> the current file (all edits will now be carried out on this file). If the assembler cannot find this program in memory, a new empty file is created. This new file is then the current file.

**NOTE:** All editor commands including "DELETE", "COPY" and "FIND" will only apply to the current file.

WIPE "<name>"

Deletes the specified program from memory. If an attempt is made to WIPE the current program (the one being edited) then a "can't wipe" error message will be displayed.

PROTECT <page>

This will prevent the assembler from using the specified RAM page for its source files. <Page> must be one of the integers 0, 1, 6 or 7. On entering the assembler page 0 will be protected, i.e. the assembler will use pages 1, 6 and 7 for its source code.

USE <page>

Frees a previously protected RAM page for use by assembler source files. <Page> must be one of the integers 0, 1, 6 or 7.

### 4. Assembly

The only change to the assembler is in the use of the *include directive. The syntax for using the directive is the same (see 4.2.3 of the main manual) but its effect may be different.

When the assembler finds a *include "<filename>" on pass lit first searches for a program (source file in RAM) called <filename>. If it finds a program with this name then it includes this file in the object code. If no program has the name specified then the assembler will search through the files in the current input device (tape or microdrive, see "2.6 Tape, Disc and Microdrive Commands"). If a file with the correct name is found the assembler will load the file into RAM if enough space is available and include the file from RAM on passes l and 2. If there is insufficient room the file will be included from tape or microdrive on both passes in the usual way.

**NOTE:** Files which were loaded into RAM from a *include directive will remain in RAM after assembly has finished, as programs with the same name as the file loaded from tape or microdrive. Use the STATS immediate command to see which files have been loaded and the WIPE command to remove any you do not want.

# Z80 Monitor And Analyser

by Andrew Foord, Kevin Kambleton and Chris Smith

## *Introduction*

The Monitor/Analyser is an essential tool in the debugging of machine code programs. It has all the facilities of a normal monitor plus many new features. The standard commands allow memory to be examined, searched, edited, moved or filled; programs can be run at normal speed; breakpoints set and machine code disassembled to the screen or printer. Additional features include slow running of programs, single stepping, disassembly to a file (so that it can be loaded into the Assembler) and the Analyser. The Analyser allows ten highly selective “stop conditions” to be defined. While the users machine code program is running under the control of the monitor, the Analyser will test each “stop condition”. The monitor will stop executing the users program if any of the “stop conditions” is fulfilled. You may, for example, want to know when your program writes to a certain area of memory or when a register takes on a particular value. The occurrence of conditions such as these can be detected by defining the appropriate “stop conditions”. The user is at liberty to define the ten stop conditions as he wishes and the analyser provides a variety of functions which may be invoked in the definitions.

This manual has been written primarily for the 48k Spectrum and 64kAmstrad variants but most of the manual is also applicable to the Spectrum 128k. However, the differences in command syntax and functions that exist between the 48k and 128k Spectrums versions have been condensed into Appendix C. Consequently, the 128k Spectrum user is advised to consult Appendix C at all times when using this manual.

At first glance it may appear that the monitor was designed to cater for a minority of highly advanced users. In fact all of the features are aimed at simplifying and speeding up the task of program development. We believe that it’s “friendliness” makes this monitor particularly suitable for newcomers to machine code programming.

## *Tape Map*

On Tape 2 you will find the following files:

AMSTRAD:

| | | |
|---|---|---|
| Side A | MON | BASIC - loader and relocator for low version |
| | LOWVER | CODE - low version of monitor |
| | RELOCOBJ | CODE - relocator code and table for low version |
| Side B | MON | BASIC - loader and relocator for high version |
| | HIGHVER | CODE - high version of monitor |
| | RELOCOBJ | CODE - relocation code and table for high version |

SPECTRUM:

| | | |
|---|---|---|
| Side A | MON | BASIC - loader |
| | LOWVER | CODE - low version of monitor |
| | RELOCOBJ | CODE - relocation code and table for low version |
| | RELOCATE | BASIC - relocator |
| Side B | MON | BASIC - loader |
| | HIGHVER | CODE - high version of monitor |
| | RELOCOBJ | CODE - relocation code and table for high version |
| | RELOCATE | BASIC - relocator |

# 1. Operating Instructions

Debugging a machine code program requires that the monitor/analyser program must reside in memory with the program being debugged in such a way that they do not overlap or interfere with each other. In order that the user may locate his program anywhere in memory there must be flexibility in the monitor/analyser execution address. High and low versions of the monitor/analyser have been provided but if neither of these is appropriate

to your needs you may wish to create a suitable version using the relocator program provided. The analyser functions are an optional extra to those provided in the monitor so the user may decide to maximise the available program space by deciding to do without the analyser section of the program. The low version of the monitor/analyser is organised with the analyser code on the high end whereas the high version has the analyser section on the low end. The user can therefore choose a monitor only version of the program which fits into either the lowest or highest part of memory.

| | |
|---|---|
| SPECTRUM | the low version occupies memory from 25000 to 40589 inclusive<br>the high version occupies memory from 49946 to 65535 inclusive |
| AMSTRAD | the low version occupies memory from 5924 to 19947 inclusive<br>the high version occupies memory from 28595 to 42618 inclusive |

If you wish to create a relocated version you may use either side A or side B depending upon which configuration is required.

If you wish to use the low version of the monitor/analyser insert and rewind side A of the monitor tape (tape 2).

If you wish to use the high version of the monitor/analyser insert and rewind side B of tape 2.

| | |
|---|---|
| SPECTRUM | type LOAD "" followed by ENTER |
| AMSTRAD | type RUN "MON" followed by ENTER |

The prompt will appear "`Relocate? (Y/N)`"

Type "Y" or "y" followed by ENTER if you wish to create a relocated version, type "N" or "n" followed by ENTER to load the ordinary version (first time users should type N). When the ordinary version has loaded another prompt will appear "Using analyser (Y/N) ?". Type "Y" or "y" followed by ENTER if you wish to use the monitor and analyser. Type "N" or "n" followed by ENTER and the monitor alone will execute. The monitor screen will appear.

**NOTE:** Spectrum users will need to respond to the prompt "Relocate ? (Y/N)" fairly quickly or stop and start the tape manually, otherwise the tape may run too far and prevent loading.

## *1.1 Relocating*

We suggest first time users skip this section and move onto "2. Screen Layout".

If you are creating a relocated version the ordinary version of the monitor/analyser will be loaded followed by the data required to perform the relocation. A prompt will appear "Relocate at?". You must now enter the address at which you wish to create the relocated version of the monitor/analyser. The new version will be created. For example if you wish to create a version of the monitor/analyser occupying memory from 30000 simply type 30000 followed by ENTER, after the prompt.

**NOTE:** The monitor will be physically moved in memory and so any other programs currently in memory may be corrupted. You will then be given the option to include the analyser before entering the monitor itself.

**NOTE:** Spectrum If and when the Monitor is exited and control is returned to BASIC, then BASIC will use memory from the CLEAR address downwards for its machine stack. Therefore BASIC could overwrite part of the user program and if this is to be avoided a further clear (to an address below the user program) may be necessary.

**NOTE:** Amstrad If and when the monitor is exited and control returns to BASIC, then BASIC will use memory from the MEMORY address downwards. Therefore BASIC could overwrite part of the user program and if this is to be avoided a further MEMORY (to an address below the user program) may be necessary.

## 1.2 Low and High Monitor/Analysers

Although relocated versions of the monitor/analyser run automatically, if you need to exit and re-enter you may calculate the execution addresses from the following information. Amstrad users may also use the command MON to enter the monitor alone and AMON to enter the monitor/analyser.

As previously mentioned the monitor/analyser has been provided in two distinct forms:

### 1.2.1 The 'Low' Version

This has been assembled so that the monitor section of the program is lower in memory than the optional analyser section consequently the low version is suited to use at the low end of memory because it allows the user the largest possible contiguous memory area above the monitor/analyser program. If the analyser is not used the area available to the user is increased by approximately 2k (Amstrad users should study the section "Tape and Disc Buffer").

| LowVersion | Starts at | Monitor ends at | Analyser ends at | Enter Monitor only | Enter Monitor / Analyser |
|---|---|---|---|---|---|
| SPECTRUM | 25000 | 37924 | 40589 | 25000 | 25003 |
| AMSTRAD | 5924 | 17142 | 19947 | 5924 | 5927 |

When relocating the low version to a new start address X, the entry points become X for monitor only and X+3 for monitor and analyser.

### 1.2.2 The 'High' Version

In contrast to the low version the high version has been assembled so that the optional analyser section resides lower in memory than the monitor section so this version is suited to use at the high end of memory because it allows the user the largest possible contiguous memory area below the monitor/analyser program. Once again not using the analyser will yield nearly 3k of memory for the user (Amstrad users should study the section "Tape and Disc Buffer").

| High Version | Starts at | Monitor ends at | Analyser ends at | Enter Monitor only | Enter Monitor / Analyser |
|---|---|---|---|---|---|
| SPECTRUM | 49946 | 52610 | 65535 | 52611 | 52614 |
| AMSTRAD | 28595 | 31399 | 42618 | 31400 | 31403 |

**NOTE:** Spectrum — When relocating the high version at address X the entry points become (X+2665) for the monitor only and (X+2668) for the monitor and analyser.

**NOTE:** Amstrad: — When relocating the high version at address X the entry points become (X+2805) for the monitor only and (X+2808) for the monitor and analyser.

## 1.3 Extension ROMs (Amstrad Only)

When extension ROMs, such as the DDI disc interface or the RS232 serial interface, are fitted to an AMSTRAD the upper limit of available memory is reduced because the ROMs reserve memory for their own use.

For example, printing HIMEM will yield different results for different configurations of machine:

| | HIMEM |
|---|---|
| STANDARD 464 | 43903 |
| STANDARD 464 with DD1 | 42619 |
| STANDARD 464 with RS232 | 41539 |
| STANDARD 464 with DD1 and with R5232 | 40255 |

The high version of the monitor/analyser has been assembled to occupy the top of memory for a standard 464 with DD1, i.e. it fills memory to 42618.

It follows that if you have extension ROMs fitted which bring HIMEM below 42619 the high version of the monitor/analyser provided will not be compatible with your system. Therefore, you will have to relocate below your HIMEM. Alternatively, if you have no extension ROMs fitted you may wish to relocate to fill memory up to 43903, the value of HIMEM on a standard 464 machine. You may find out the value of HIMEM by typing ?HIMEM followed by ENTER.

To relocate relative to HIMEM simply subtract the overall length of the monitor/analyser from your value of HIMEM. The overall length of the monitor/analyser is 14024 bytes. So, for example, to relocate the monitor / analyser to memory at 43903 use the address 29879 when relocating because 43903 - 14024 = 29879. For further information see the section on relocating the monitor/analyser.

## *1.4 Tape and Disc Buffer (Amstrad only)*

When the FLIST and LOAD (for ASCII files) commands are used the monitor/analyser, in accordance with the Amstrad firmware routines, makes use of a 2048 byte buffer for interfacing to tape/disc. Consequently, if the user is prepared to forego the use of the FLIST command he may use the buffer area for his own purposes. In the versions provided the buffer area has been located on the end of the code:

| The “low” version | The “high” version |
|---|---|
| 5924 MONITOR | 26547 TAPE BUFFER (OPTIONAL)<br>2048 bytes used to FLIST |
| 17143 ANALYSER (OPTIONAL) | 28595 ANALYSER (OPTIONAL) |
| 19948 TAPE BUFFER (OPTIONAL)<br>2048 bytes | 31400 MONITOR |
| 21995 END | 42618 END |
| 21996 | 42619 |

From the table above we can see that the user of the low version may use memory from 19948 for his own purposes if he/she is prepared to forego FLIST and LOAD commands. Similarly the user of the high version may use memory up to and including 28594.

You may wish to use the monitor alone i.e. without the analyser but with the TAPE buffer moved to the end of the monitor code. To do this enter the monitor at the “monitor only” entry point i.e. for the low version type:

```
CALL 5924 followed by ENTER
```

for the high version type:

```
CALL 31400 followed by ENTER
```

Now move the buffer using the “BUFFER = “ command i.e. for the low version type:

```
BUFFER=17143
```

for the high version type:

```
BUFFER=29352 (=31400-2048)
```

If you are using a relocated version of the monitor/analyser then it is possible to achieve the same result i.e. dispense with the analyser and redefine the buffer.

e.g. For a relocated LOW version at address X type CALL X followed by ENTER then type BUFFER= X+11219 followed by ENTER. So if X = 9000 type CALL 9000 followed by ENTER then type BUFFER=20219 followed by ENTER.

e.g. For a relocated HIGH version at address Y type CALL Y+2805 followed by ENTER then type BUFFER=Y+757 followed by ENTER. So if Y=20000 type CALL 22805 followed by ENTER then type BUFFER=20757 followed by ENTER

**NOTE:**

i) The FLIST and LOAD commands will still operate at any time but they will use the current buffer area.

ii) The BUFFER= command may be used to set up different buffer areas to those in the table above which are simply the default values included at assembly time.

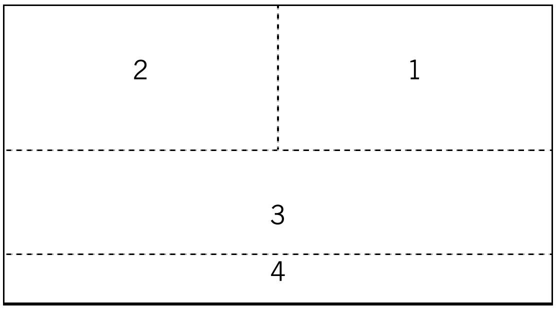
iii) The LOADing of binary, as opposed to ASCII, files does not use the buffer.

**SPECTRUM users**: The Spectrum version of the monitor/analyser uses the lower third of screen memory during FLIST so no buffer space has been allocated during assembly The FLIST buffer is fixed so the "BUFFER =" command is not available on the Spectrum version.

# 2. Screen Layout

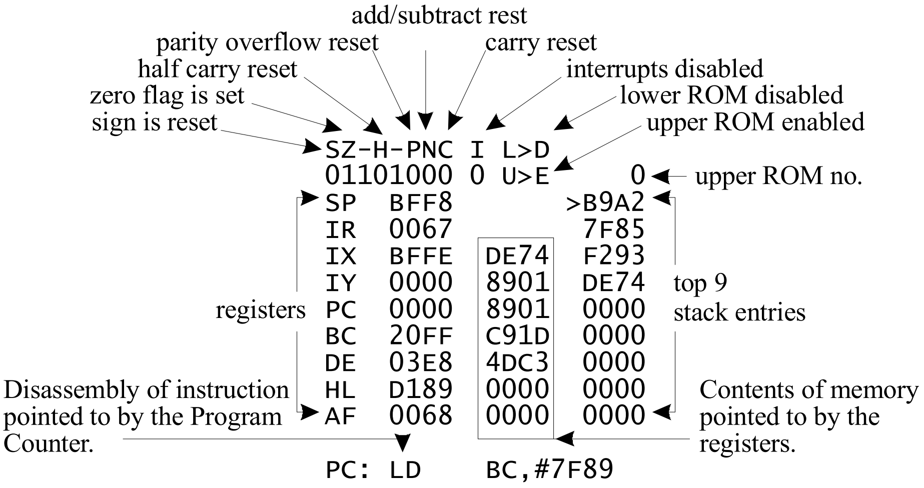
## 2.1 Spectrum and Amstrad 40 Column Mode

The screen is split into four windows:



**Window Number 1:** This window displays the current state of the Z80 registers, an example is given below. The top two lines give the state of the flags (S-sign, Z-zero, H-half carry, P-parity/overflow, N-add/subtract and C-carry, blanks indicate the unused flags), the state of ROM switching and the state of the interrupt flip-flop. In the case shown, the zero flag is set and all the others are clear and interrupts are disabled. Down the right hand side of the window the top 9 entries on the stack are displayed (as pointed to by SP). The left hand side displays the register values and the central column shows the two bytes pointed to by that register pair. All numbers are printed in hexadecimal. The disassembly of the instruction pointed to by the PC is printed at the bottom of the window.

### 2.1.1 Amstrad



```
SZ-H-PNC I L>D
01101000 0 U>E    0
SP  BFF8         >B9A2
IR  0067          7F85
IX  BFFE  DE74    F293
IY  0000  8901    DE74
PC  0000  8901    0000
BC  20FF  C91D    0000
DE  03E8  4DC3    0000
HL  D189  0000    0000
AF  0068  0000    0000
PC: LD    BC,#7F89
```

*Top right hand corner - Amstrad*

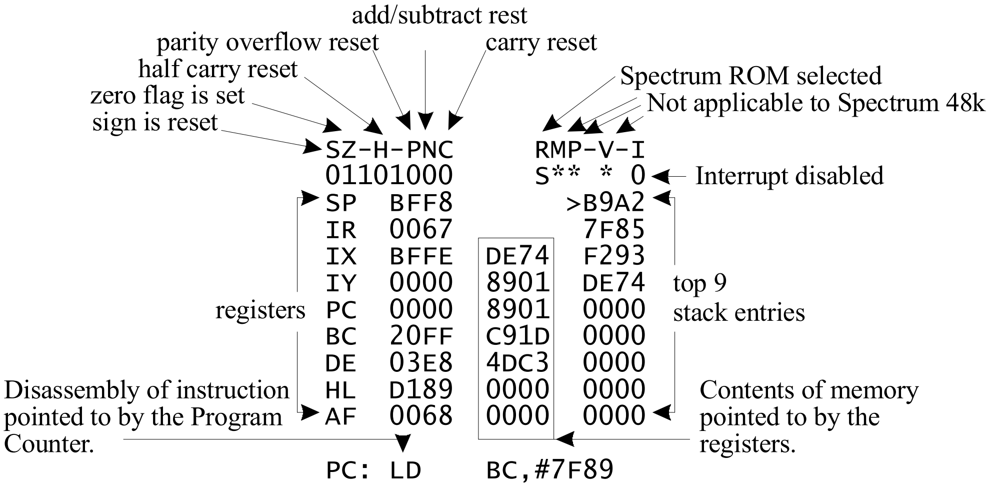
The states of the ROMs are shown in the top right hand corner of the screen, ‘L’ stands for lower ROM, ‘U’ for upper. A ‘D’ indicates that a ROM is disabled and an ‘E’ means it is enabled. The 0 next to the U>E in the example indicates that upper ROM number 0 is currently selected.

I　　This indicates interrupt status.

1 = enabled
0 = disabled

## 2.1.2 Spectrum

The top right hand corner of window 1 on the Spectrum differs from the same area on the Amstrad screen. Note Spectrum 128k users should see Appendix C.



```
SZ-H-PNC          RMP-V-I
01101000          S** * 0
SP   BFF8           >B9A2
IR   0067            7F85
IX   BFFE   DE74     F293
IY   0000   8901     DE74
PC   0000   8901     0000
BC   20FF   C91D     0000
DE   03E8   4DC3     0000
HL   D189   0000     0000
AF   0068   0000     0000
PC: LD      BC,#7F89
```

*Top right hand corner - Spectrum*

This is used to display the memory configuration and interrupt status. There are five indicators which are displayed as RMP-V-I. Each indicator has the following relevance:

R This indicates which ROM is currently paged in. There are three possibilities.

S = Standard Spectrum ROM
I = Interface 1 (Shadow) ROM
D = Spectrum 128k (Derby) ROM

**NOTE:** The standard 48k Spectrum obviously does not support the third option!

M This indicates the memory page currently selected for the memory pointer for Spectrum 128k users. 48k users will find an asterisk displayed.

P This indicates the memory page currently selected for the program counter for Spectrum 128k users. 48k users will find an asterisk displayed.

V This indicates which of the possible screen bases has been selected for 128k users. 48k users will find an asterisk displayed.

I This indicates interrupt status.

1 = enabled
0 = disabled

**WINDOW NUMBER 2:** The window in the top left hand corner is used for disassembly and the printing of results by some of the Monitor's commands.

**WINDOW NUMBER 3:** Below the top two windows there is a hex and ASCII display of a section of memory, the area displayed is indicated by the value of the Memory Pointer, whose exact location is shown by a cursor on the middle line. The example shown here was taken from the Spectrum. Amstrad users should not be alarmed by differences in the character set.

```
MEM: DI                 PC: DI

  FFE8 00 00 00 00 00 00 00 00  ........
  FFF0 00 00 00 00 00 00 00 00  ........
  FFF8 00 00 00 00 00 00 00 00  ........
  0000>F3<AF 11 FF FF C3 CB 11  S/.©©CK.
  0008 2A 5D 5C 22 5F 5C 18 43  *]\"_\.C
  0010 C3 F2 15 FF FF FF FF FF  Cr.©©©©©
  0018 2A 5D 5C 7E CD 7D 00 00  *]\~M}.P
  0020 CD 74 00 18 F7 FF FF FF  Mt..w©©©
```

**WINDOW NUMBER 4:** The bottom two lines of the display are used for printing error messages and the inputting of commands.

Commands such as LIST use the whole screen to print disassembly, when this happens the screen is cleared before anything is printed on the screen.

### 2.1.3. 80 Column Mode - Amstrad only

The 80 column mode splits the screen into two halves, the left hand side uses the same format as the 40 column mode, the right hand side is for commands which would use the whole screen in the 40 column mode, this allows the register display and memory display to stay on the screen while disassembling, displaying memory or listing breakpoints.

## 3. The Editor

The Monitor's Editor allows the input of up to 39 characters at a time on the bottom line of the display. The following keys can be used to edit the line:

| Amstrad | Spectrum | Operation |
|---|---|---|
| left and right cursor keys | left and right cursor keys | Move cursor along input line. |
| DEL | caps shift 0 | Delete previous character |
| CLR | symbol shift 0 | Delete current character. |
| COPY | caps shift 1 | Insert a space. |
| CTRL L | symbol shift A | Clear the editing area. |
| ENTER | ENTER | Execute the command in the editing area. If there is an error in the command, control is passed to the editor and the line can be corrected. Pressing ENTER with nothing in the editing area causes the screen to be cleared and the register and memory display to be updated. |

Often it is required to breakout of commands, on the Amstrad the ESCAPE key can be used to leave commands, the escape key on the Spectrum is symbol-shift & A.

In the lower half of the screen there is the memory display. On the middle line of the dump a cursor is printed which shows the current position of the memory pointer. The memory pointer can be moved around memory using the following keys:

| Amstrad | Spectrum | Operation |
|---|---|---|
| SHIFT left cursor | symbol-shift 0 | Decrement the memory pointer. |
| SHIFT right cursor | symbol-shift E | Increment the memory pointer. |
| SHIFT up cursor | symbol-shift W | Subtract 8 from the memory pointer. |
| SHIFT down | symbol-shift S | Add 8 to the memory pointer. |
| SHIFT - COPY | caps-shift 9 | Advance the memory pointer to the next machine code instruction, NOT the next byte. |

The memory pointer can be set to a particular value using the MEM = command.

# 4. Entering Commands

Command names can be entered in upper or lower case, if the command name is followed by an equals sign (e.g. “BUFFER=”) then there should be no spaces between the name and the equals sign. Note that there is a single space between “EX” and “AF” in the “EX AF” command.

Numbers can be entered in one of four bases as shown below:

| | | | |
|---|---|---|---|
| Decimal | e.g. | 4785 | |
| Hexadecimal | | #12B1 | (preceded by a ‘#’) |
| Octal | | @11261 | (preceded by a ‘@’) |
| Binary | | %1001010110001 | (preceded by a ‘%’) |

A single ASCII character can be used to represent a number by enclosing it in quotes, e.g. “A” is equivalent to the number 65.

“ON” and “OFF” have the values 1 and 0 respectively and can be used instead of these numbers. The word “MEM”, as a parameter, returns the current value of the memory pointer and “AF”, “BC”, “’DE”, “HL”, “IX”, “IY”, “SP” and “PC” can also be used as parameters and will return the value of the corresponding register pair. This makes the following commands possible:

| | |
|---|---|
| PC= MEM | (gives PC the value of the memory pointer) |
| LIST HL | (disassembles from the address held in the register pair HL) |

## 4.1 Command Syntax

The following notation is used for command syntax: parameters are enclosed in ‘<’and’>’ type brackets and optional parameters are enclosed in square brackets.

For example, FRED <x> [,<y>] would mean that the command whose name is FRED can be followed by one or two parameters, one called <x> and the other, an optional parameter called <y>. The meaning of the parameter names used are given below:

| | |
|---|---|
| <byte> | A number in the range 0 to 255(#FF) |
| <word> | A number in the range 0 to 65535 (#FFFF) |
| <count> | As <word> |
| <addr> | An address in the range 0 to 65535 (#FFFF) |
| <start> | As <addr> |
| <finish> | As <addr> |
| <flag> | Ether 0 or 1 (or ON or OFF) |
| <db number> | A DB block number in the range I to 8 |
| <brk number> | A breakpoint number in the range I to 8 |
| <filename> | A string of characters enclosed in quotes, e.g. “fred”. On the Amstrad this can be up to 16 characters long and on the Spectrum this can be 10 characters.<br><option number> An option in the range I to 8 |

**Amstrad only:**

| | |
|---|---|
| <ROM number> | An upper ROM number in the range 0 to 251 |

**Spectrum only:**

| | |
|---|---|
| <drive> | A microdrive number in the range I to 8 |

## 5. Monitor Commands

NOTE: The use of square brackets indicates optional parameters. Spectrum users should also note that the ESCAPE key should be read as symbol shift A.

```
<reg name>= <byte>                e.g.    C= %10110
```

Assigns a value to a single register, where <reg name> is one of A, B, C, D, E, H, L or F (the flag register).

```
<reg pair>= <word>                e.g.    BC= 30000
```

Gives a value to a register pair. <reg pair> must be one of AF, BC, DE, HL, IX, IY, SP or PC.

```
(<reg pair>)= <byte>              e.g.    (HL)= #6D
```

Places the specified byte in memory at the location pointed to by the given register pair. The register pair can be one of BC, DE, HL, IX, IY or PC.

```
EXX
```

Performs the equivalent of the Z80 “EXX” instruction on the Monitor’s copy of the registers. The register display now shows an apostrophe after the register names BC, DE and HL to indicate that the alternate register set is now the main set. Typing EXX again returns to the normal set. See also the EX AF command below.

```
EX AF
```

Performs the equivalent of the Z80 “EX AF,AF’” instruction on the monitor’s copy of the register pair AF. The register display will now show the alternate value of AF and an apostrophe printed after the register name indicates this. Typing EX AF again toggles the display back to the normal value of AF. See also the EXX command.

```
MEM= <addr>                       e.g.    MEM= #4000
```

Sets the memory pointer to a particular value. The memory dump display is updated by this command.

```
DATA <byte list>                  e.g.    DATA “hello”,13
```

Places the list of bytes at the memory location pointed to by the memory pointer. The memory pointer is incremented after each byte is placed in memory. The memory dump display is updated by this command. Strings of more than one character can be used in this command and each character in the string is considered as a different byte value.

```
.<byte list>                      e.g.    .#3E,“*”,#CD,#5A,#BB
```

The “.” command is just a shorthand version of the DATA command and is otherwise identical.

```
EXIT
```

Returns control to whatever called the Monitor. Note that as long as the Monitor’s workspace is not over-written, all breakpoint definitions, data areas, analyser programs and other user defined options will be unchanged on re-entry to the Monitor. Note also that breakpoints are recognised if the code they are in is executed from outside the Monitor (see the section on breakpoints).

```
FILL <start>,<finish>,<byte>      e.g.    FILL #4000,#5FFF,0
```

The block of memory from <start> to <finish> inclusive is filled with the value of <byte>.

```
DUMP [<start>[,<finish>]]         e.g.    DUMP
                                  or      DUMP #1000
                                  or      DUMP #1000,#10FF
```

Gives a hex and ASCII dump of memory onto the screen. With no parameters the dump starts from the current value of the memory pointer and continues until the ESCAPE key (symbol-shift A on the Spectrum) is pressed. With one parameter the dump starts from the specified address and continues until the ESCAPE key is pressed. With two parameters the dump starts from the first address and continues until either the ESCAPE key is pressed or the second address is reached. Pressing a key during the listing will pause printing until another key is pressed (other than the ESCAPE key).

```
LDUMP [<start> [,<finish>]]       e.g.    LDUMP
                                  or      LDUMP 30000
                                  or      LDUMP 30000,31000
```

This command is the same as the DUMP command except that the output is also sent to the printer. Option number 2 selects whether or not the printer output is also sent to the screen (see the OPTION command).

```
MOVE <start1>,<finish1>,<start2>  e.g.    MOVE #1000,#13FF,#A000
```

The block of memory from <start1> to <finish1> inclusive is copied into the area staring at <start2>.

Note that copying is ‘intelligent’ in that the two blocks can overlap.

`CHECK <start1>,<finsh1>,<start2>` e.g. `CHECK 1000,1010,#A0F0`

This command verifies that two blocks of memory are identical. The area from <start1> to <finish1> is compared with the area starting at <start2>. If the two blocks are the same then a “Blocks Identical” message is given, but if they are different a “Failed at <addr>” message is printed, where <addr> is the address of the first mismatch.

`SEARCH <start>,<finish>,<bytelist>` e.g. `SEARCH #8000,#A7FF,#5A,#BB`

The block of memory defined by <start> and <finish> inclusive is searched for the first occurrence of the specified string of bytes. If no match is found then “String Not Found” is printed otherwise “Found at <addr>” is printed (where <addr> is the address of the first match). If a match is found then the memory pointer is set to the address of this match and the memory dump display is updated. To find any further occurrences of the string the NEXT command should be used (see below).

`NEXT`

The search specified by the most recent SEARCH command is continued so that the next occurrence of the string can be found. If no search command had been used before, or the last search had failed to find any more matches, a “No Search String” error is given. On the Amstrad CTRL and N will achieve the same effect as typing NEXT, on the Spectrum pressing symbol-shift Y will perform this operation.

`JUMP [<addr>]` e.g. `JUMP`
or `JUMP #1000`

The machine code at the specified address (or at the value of PC if no address is supplied) is executed at normal Z80 speed. On entry to the code the actual Z80 registers are set to the values of the Monitor’s copy of the registers. If one or more breakpoints have been defined then these breakpoints are loaded into the RAM. The only way to return to the monitor from the JUMP command is via a breakpoint (see the section on breakpoints).

**NOTE:** Executing a JUMP or CALL will clear the slow run workspace. This means that the table of previous addresses used by the TRACE and LTRACE commands will be lost.

`CALL [<addr>]` e.g. `CALL`
or `CALL#1000`

This command is the same as the JUMP command except a return address to the monitor is placed on the stack before the machine code is executed. On returning from the machine code the Z80 registers are stored in the Monitor’s copy of the registers (including the alternate set) and the current ROM states are also stored away. Breakpoints will be recognised during a CALL command.

**NOTE:** Executing a JUMP or CALL will clear the slow run workspace. This means that the table of previous addresses used by the TRACE and LTRACE commands will be lost.

`? <word>` e.g. `? @7466`

The ? command can be used for converting between number bases. The value of <word> is printed in hex, decimal, octal and binary in the top left hand window.

`PUSH <word>` e.g. `PUSH #ABCD`

The value of <word> is placed on the stack and the stack pointer decremented by two. The register display is updated by this command.

`POP`

The value of SP incremented by two, this has the effect of ‘popping’ a value off of the stack (although the value ‘popped’ off is not remembered).

`OPTION <option number>,<flag>` e.g. `OPTION 3,ON`

The OPTION command allows various options used within the Monitor to be selected, the option number parameter is between l and 8 representing the eight different options which are listed below. Initially all options are off.

**Option** **Effect**

1. If this option is off, then all disassembly will be. in hexadecimal, otherwise, if on, it will be in decimal. This option also affects some other number printing: numbers in error messages, addresses in the listing of breakpoints or data areas and the value of the stack after an EVAL command.

2. If off, then printer output is also sent to the screen. If on, printer output is only sent to the printer. This option affects the LDUMP, LLIST and PDEF commands.

3. If on, then on the third pass of an FLIST command, the disassembly that is being sent to the tape/disc/microdrive file will also be sent to the screen. If this option is off only the current address is printed during the third pass. See the FLIST command.

4. If off, then during a slow run or single stepping, if a call or jump to a ROM routine is encountered it will be immediately executed and not single stepped. On the Amstrad these calls are those to the firmware block from #B900 to #BF00, on the Spectrum this is a call below #4000 (16384). If this option is turned on then ROM routines can be slow run normally. See the section on the debugger for more details.

5. If off then the file created by the FLIST command will contain labels to allow the source code produced to be re-assembled at a different address. Turning on the option turns off the labels so that the file contains just absolute addresses. A source file produced with no labels will be shorter in length than one with labels but in most cases it is advisable to use labels.

6. If off, then the screen is cleared at the start of a JUMP, CALL or SLOW command but if on the routine to be executed is entered with the screen still displaying the Monitor's display.

7. This option is for the Spectrum only, if off then printer output is sent to the ZX printer (if one is connected) and if on it is sent to the Kempston Centronics Interface (if connected).

8. If off, then a line feed is sent to the printer with every carriage return, if on, then only the carriage return is sent.

`CAT`
List the tape/disc/microdrive directory. This command does not work with Spectrum tape.

`ERA "<string>"` e.g. `ERA "DEMO`
e.g. `ERA"4: FRED" (Spectrum only)`

Erase the specified file from disc/microdrive. Wildcards can be used with Amstrad disc (see Amstrad disc manual). Spectrum users may erase a file on any microdrive by including "n:" in the filename string where n is the microdrive number.

`DI`
Disable Interrupts. This may be necessary before single stepping or slow running certain programs.

`EI`
Enable Interrupts.

When any user code is executed during a slow run, single step or after a JUMP or CALL, interrupts will be either enabled or disabled according to the state of a flag which is maintained by the monitor/analyser. The flag may be set/reset by using the DI and EI commands described above. The flag is also set/reset when the user program returns control to the. monitor either after a single step or after a slow run is halted or when a breakpoint or return from CALL occurs. The monitor determines whether interrupts areenabled by executing two successive LD A,R instructions and testing the P/V flags.

An analyser word "|" has been provided to allow the state of interrupts to be evaluated from within the analyser.

Amstrad users should remember that if interrupts are enabled the alternate register set must be valid because. ROM and MODE information is kept in BC'. If interrupts are disabled the alternate register set may be used. The monitor/analyser, when selecting user ROM states prior to executing user code, uses BC' (or BC if a single. EXX has been executed) to determine which ROMs should be enabled.

## *5.1 Amstrad Specific Commands*

`ROM <flag1>,<flag2> [,<rom number>]`
This command enables or disables the ROMs on the Amstrad. <Flag1> selects the state of the lower ROM, if "0" then it is disabled, if "1" it is enabled. <Flag2> sets the state of the currently selected upper ROM. If <rom number> is given then a particular upper ROM can be selected. The ROM command affects all of the Monitor's commands that read from memory, these include.: DUMP, LIST, MOVE, SEARCH and CHECK. When machine code is executed or single stepped, the selected ROM states are set on entry and the Monitor's copy of the ROM states updated on exit from the code. Here are some examples of the ROM command :

| | |
|---|---|
| `ROM 0,1` | lower disabled, upper enabled, |
| `ROM ON,OFF` | lower enabled, upper disabled, |
| `ROM ON,ON,13` | lower enabled, upper ROM number 13 selected and enabled. |

`BUFFER= <address>` e.g. `BUFFER=#A000`
Sets where in memory the tape or disc buffer will lie., this buffer is 2k long and is used during the FLIST command and during the LOAD command if an ASCII file is loaded.

`MODE <mode number>`
Selects either 40 or 80 column screen mode:

`MODE` 1 gives 40 column
`MODE` 2 gives 80 column

`MAP`
The current memory usage is listed in the disassembly window. The complete list will appear as follows:

| | | START | END | |
|---|---|---|---|---|
| Line 1 | GMON | #1724 | #4DEB | Monitor/analyser code |
| Line 2 | WORK | #0000 | #0000 | Workspace |
| Line 3 | PROG | #9000 | #9FFF | Analyser program space |
| Line 4 | VSCR | #8000 | #84FF | Virtual screen storage. |
| Line 5 | BUFF | #4DEC | #55EB | Buffer |
| Line 6 | OPTION | 00010010 | | |
| Line 7 | VS | OFF | | Virtual screen on/off |
| Line 8 | ANAL | ON | | Analyser on/off |

If the virtual screen memory has not been defined then lines 4 and 7 will be omitted.

If the analyser code is not included or the program space is undefined then line 3 will be omitted.

The numbers declared in the END column of the list are inclusive, in other words the monitor occupies memory from #1724 up to and including #4DEB.

`DISC`
Direct the firmware to make disc the current input/output device.

`TAPE`
Direct the firmware to make tape the current input/output device.

`LOAD <filename>,<addr>` e.g. `LOAD “fred.wow”,#8000`
The LOAD command tries to load the specified file in from tape or disc and place it in memory at the given address. The monitor’s PC register is made equal to the execution address of the file (if it has one.).

`SAVE <filename>,<start>,` e.g. `SAVE “fred.yuk”,#8000,#8FFF`
`<finish>[,<exec>]` or `SAVE “fred.yuk”,#8000,#8FFF,#809D`
The block of memory from <start> to <finish> is saved to tape or disc as a binary file. An execution address may by specified but if not the execution address is made equal to the start address. Once saved the file can be loaded into memory either straight from BASIC or by using the LOAD command from the. Monitor.

`DRIVE <letter>` e.g. `DRIVE A`
Direct the firmware to make the disc drive <letter> the current input/output drive.

`|<command>`
This command gives access to any RSX’s which are currently logged on including disc and tape commands. See your Amstrad manuals for syntax.

464 users should notice that string parameters are passed directly so the construction @a$ is neither necessary nor valid e.g. to erase. a file called FRED type:

```
|ERA “FRED” or ERA,“FRED”
```

Spaces and commas are treated as separators.

## *5.2 Spectrum Specific Commands*

`ROM <flag>` e.g. `ROM ON`
Pages in or out the shadow ROM of the interface I (if fitted). When the shadow ROM is on, listing and dumping of memory will read the shadow ROM and not the main ROM, with the shadow ROM enabled routines in the shadow ROM can be single stepped, slow run and fast executed. Executing the instructions at #8 and #1708 will page in the shadow ROM and executing the instruction at #700 will cause the shadow ROM to be paged out.

MAP
The current memory usage is listed in the disassembly window. The complete list will appear as follows:

| | | START | END | |
|---|---|---|---|---|
| Line 1 | GMON | #61A8 | #9E8D | Monitor/analyser code |
| Line 2 | WORK | #A000 | #A2FF | Workspace |
| Line 3 | PROG | #A300 | #A900 | Analyser program space |
| Line 4 | VSCR | #B000 | #CAFF | Virtual screen storage. |
| Line 5 | OPTION | 00010010 | | |
| Line 6 | VS | OFF | | Virtual screen on/off |
| Line 7 | ANAL | ON | | Analyser on/off |

If the virtual screen memory has not been defined then lines 4 and 7 will be omitted.

If the analyser code is not included or the program space is undefined then line 3 will be omitted.

The number declared in the END column of the list is inclusive, in other words the work space occupies the memory from #A000 up to and including #A2FF.

MDRV [<drive>]
Direct the firmware to make the specified microdrive the current input/output device. If <drive> is not specified input/output defaults to drive 1.

TAPE
Direct the. firmware to make. tape the current input/output device.

LOAD <filename>[,<addr>] e.g. LOAD “fred”
LOAD “fred”3,30000

The LOAD command tries to load the specified file in from tape/microdrive and place it in memory starting at the address <addr>. If <addr> is not supplied then a code file will be loaded in at the place in memory it was saved from, but other file types will be loaded to the address pointed to by MEM.

SAVE <filename>,<start>,<finish> e.g. SAVE “notagain”,30000,30199

This command saves to tape/microdrive the block of memory defined by <start> and <finish>. A “start>finish” error will be produced if <start> address is greater than the <finish> address.

Note 1: All Spectrum tape/microdrive commands will be. halted by pressing the SPACE key.

Note 2: If microdrive is selected input/output may be directed to any of the 8 possible microdrives irrespective of which one is presently selected. This is achieved by preceding the filename with n: where n is the number of the drive which is required. In other words when drive l is selected, for example, a file FRED may be loaded from drive 4 by typing:

```
LOAD “4:FRED”,#8000
```

and similarly a file may be saved on drive. 6 by typing:

```
SAVE “6:TOM”,#9000,#A000
```

Note 3: The monitor uses ROM routines for SAVEing so the contents of the shadow ROM cannot be SAVEd directly. If you wish to save the. contents of the shadow ROM move the area required to free RAM then save it from there. Although the contents of the shadow ROM cannot be saved directly they can be disassembled to tape or microdrive using the FLIST command.

## *5.3 Virtual Screens*

One of the most useful additions offered by the monitor is the virtual screen facility. It is provided with graphics applications in mind but will be found to have uses in almost every sphere of program development. The commands for controlling the virtual screen are described here and their use is demonstrated in example 5 (see section 10).

When the user’s program is RUN using the SLOW, CALL or JUMP command, the user’s screen is restored before execution. When the monitor is re-entered via a breakpoint, STOP or RETurn (from a CALL) the user’s screen is stored in memory before the monitor’s front panel is displayed. When this facility is enabled, single stepping will cause the user’s screen to be temporarily displayed so that it can be updated by the program.

**NOTE:** Under normal circumstances this facility will only be used with CALLs, JUMPs and SLOW 0. SLOW 1, 2 and 3 cause the screen to update after each instruction and this may interfere with the user screen. Bear in mind that screen updates do not affect the top left hand corner of the screen (the area used for disassembly) and so it is possible to combine front panel displays with graphical operations. Occasionally this mode of use will be invaluable.

`SCRN`
Display the virtual screen until a key is pressed.

`SCRN <flag>` e.g. `SCRN 1`
or `SCRN ON`
Enable or disable the virtual screen facility. If <flag> = l the virtual screen is enabled, if <flag> = 0 it is disabled. If the virtual screen is off then slow running will be faster.

`SCRN= <addr>` e.g. `SCRN=#D000`
Set the base address for the virtual screen storage.

SPECTRUM users:

Note 1: If this command is to be executed the user must find 6912 free bytes at which to store the virtual screen whilst the monitor front panel is being displayed. The SCRN= command is used to indicate where to store the data.

Note 2: This command produces a user screen in memory which will be cleared with all attributes set to the current value in the system variable ATTR-T.

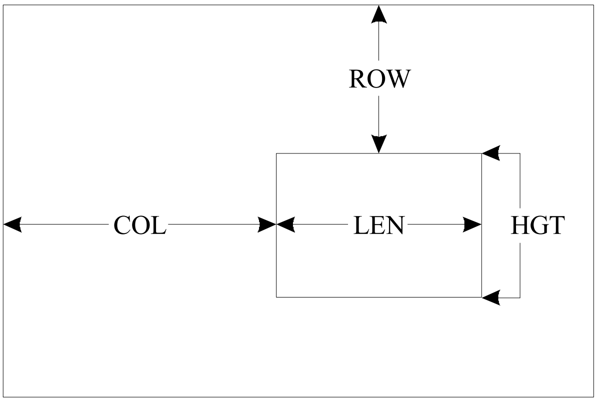
Note 3: Use this facility with great care!

AMSTRAD users:

On your machine the effect of this command is simply to define the start address of screen storage.

## 5.3.1 AMSTRAD only

Although the AMSTRAD screen occupies memory from #C000 to #FFFF (16k) only 16000 bytes are needed to duplicate the screen data. However, 16000 bytes is a lot so, to prevent memory problems, the user may define a window in the screen which will then become the subject of all virtual screen operations. Therefore the amount of memory required for using the virtual screen facility will depend on the size of the window.



The window has four parameters all of which are measured in AMSTRAD characters i.e. they are MODE dependent. The four parameters are COL, ROW, LEN and HGT and they may be assigned values either by the commands detailed below or from within the analyser (see later).

The amount of memory required is a function of LEN and HGT i.e. the area of the window.

The values of COL and ROW do not effect the amount of memory required.

`COL= <number of characters>` e.g. `COL=20`
Define the left hand edge of the virtual screen window. The initial value is 15.

`LEN= <number of characters>` e.g. `LEN=8`
Define the width of the virtual screen window. The initial value is 10.

`ROW= <number of character lines>` e.g. `ROW=2`
Define the top edge of the virtual screen window. The initial value is 8.

`HGT= <number of character lines>` e.g. `HGT= 16`

Define the height of the virtual screen window. The initial value is 8.

The validity of the COL, ROW, LEN and HGT parameters is related to the current MODE of the AMSTRAD because they are measured in characters rather than bytes. In MODE 0 for example the maximum number of characters is 20 so COL + ROW must be less than 21.

Whenever the screen is about to be moved, i.e. when the screen is on and a SCRN, SLOW, JUMP, CALL or single step is executed, the parameters defining the window are checked. If the parameters are invalid, an error message will be displayed. If parameters are to be valid then they must obey the following relationships:

```
COL + LEN <=     20 in MODE 0
            40 in MODE l
            80 in MODE 2
```

ROW + HGT <= 25 (the maximum number of rows is always 25 whichever MODE)

`SCRCLR`

This command has the effect of clearing the virtual screen memory. The amount of memory cleared will depend upon the current value of HGT and LEN. If HGT or LEN are changed after performing an SCRCLR then a further SCRCLR may be necessary.

`CTRL & V`

The location of the current window will be highlighted until any other key is pressed.

# 6. Disassembler Commands

```
DISS [<start>[,<finish>]]            e.g.   DISS
                                     or     DISS #A000
                                     or     DISS #A000,#A0FF
```

Disassembles to the window in the top left hand corner of the screen. With no parameters, the disassembly starts at the current value of the memory pointer (MEM) and continues until the ESCAPE key is pressed. With one parameter, disassembly starts at the specified address and continues until the ESCAPE key is pressed. With two parameters, disassembly starts at the first address and continues until either the second address is reached or the ESCAPE key is pressed. Pressing any key (including the ESCAPE key) pauses the printing until another key is pressed (other than the ESCAPE key, which will return control back to the editor). The ESCAPE key on the Spectrum is symbol-shift and A pressed together.

```
LIST [<start>[,<finish>]]            e.g.   LIST
                                     or     LIST 1000
                                     or     LIST 1000,2000
```

This command is the same as the DISS command except the disassembly is printed using the whole screen (in 40 column mode) or to the right hand half of the screen (in 80 column mode).

```
LLIST[<start>[,<finish>]]            e.g.   LLIST
                                     or     LLIST #8000
                                     or     LLIST #8000,#80A7
```

This command is the same as the LIST command except that the disassembly is sent to the printer in addition to the screen. If option bit 1 is set, the output is only sent to the printer (see the OPTION command).

```
DB [<db number>[,<start>,<finish>]]  e.g.   DB
                                     or     DB 3
                                     or     DB 7,#8000,#83FF
```

The DB command allows sections of memory to be defined as areas of data instead of machine code. During disassembly any data areas are listed using defined bytes (DBs). Up to 8 areas of memory can be defined as data areas and are numbered from 1 to 8. This command affects the LIST, LLIST and FLIST commands.

With no parameters the DB command lists the current selection of data areas already defined by the user. With one parameter, a number from 1 to 8, the specified data area is removed from the list of data areas. With three parameters, a number from 1 to 8, a start address and finish address, a data area can be defined.

```
FLIST <filename>,<start>,<finish>    e.g.   FLIST “fred”,#100,#1FF
```

The FLIST command sends disassembly to either an Amstrad tape or disc file, Spectrum tape or Spectrum microdrive. The file produced can then be loaded into the Assembler, altered and re-assembled. The file produced will contain labels instead of addresses so that the code can be re-assembled at a different address. Addresses within the range of the code disassembled will be converted to labels and those out of this range will be kept as absolute addresses. For example, suppose the following piece of code were placed at address #7000:

```
#7000 LD    B,#10
#7002 PUSH  BC
#7003 CALL  #ABCD
#7006 POP   BC
#7007 DJNZ  #7002
#7009 CALL  #7040
#700C JP    Z,#7000
#700F RET
```

If it were then FLISTed the resulting listing would appear if the file was then loaded into the Assembler:

```
L7000 : LD   B,#10
L7002 : PUSH BC
        CALL #ABCD
        POP  BC
        DJNZ L7002
        CALL #7040
        JP   Z,L7000
        RET
```

The insertion of labels is optional; setting option 5 ON, will turn off the labels and absolute addresses will always be inserted. If the labels are used then workspace must be set aside so that the addresses of labels can be stored during the operation of the command. This is done by using the WORK= command and is explained later. If no workspace has been set aside then a “Workspace Undefined” error will be given and if there are more labels than can be fitted in the given amount of workspace an “Insufficient Workspace” error will be given.

To FLIST an area of memory the Monitor will disassemble it three times: the first pass sets up the labels, the second pass calculates the length of the file including the labels and the third pass saves the file. If option number 3 is on then during the third pass, the file being saved is also printed on the screen. During all passes the address of the current instruction is printed on the screen so you have an idea of how far the command has reached in producing the file. After the second pass has been completed the length of the source code that is about to be produced is printed. The ESCAPE key will abandon the command at any time (but if you are in pass three only part of the file will have been saved).

There are three points to note when using this command:

Note 1. Not only are CALL and JUMP operands given labels but so are instructions such as LD BC,#1234 (if the value of the operand were to fall in the range of the code being disassembled). This will only be a problem if the code is re-assembled at a different address because the value of the label L1234 will change and if the number #1234 was intended and not the address then the register pair will now be loaded with the wrong value. To solve this problem change any value like this to absolute addresses before reassembling.

Note 2. If a label address falls in the middle of an instruction it will not be defined. The example below demonstrates this. On the left is the code disassembled and on the right the source code produced:

```
#8000 LD    HL,#8007    L8000: LD   HL,L8007
#8003 LD    (HL),E             LD   (HL),E
#8004 INC   HL                 INC  HL
#8005 LD    (HL),D             LD   (HL),D
#8006 CALL  #4321              CALL #4321
#8009 JR    #8000              JR   L8000
```

The label L8007 has not been defined since it falls in the middle of the CALL #4321 instruction. To solve this either define the label elsewhere or remove the label and put in an absolute address, the solution will depend on whether #8007 was intended as a number or an address (an address in this case).

Note 3. If a program contains a table of addresses and you attempt to relocate the program by using FLIST and then re-assemble, the addresses in the table will, of course, not be relocated (assuming the table had been defined as an area of data). Tables of addresses such as this will have to be altered by hand from the assembler by replacing the DBs with DWs and inserting either labels or absolute addresses. For example, the following program looks up an address in a table and jumps to it (the source code produced is shown on the right).

The C register contains 0,1 or 2 on entry and the corresponding routine at #8020,#8030 or #8040 is jumped to, the area from #800C to #8011 has been defined as a data area:

```
#8000 LD    B,0                LD   B,#0
#8002 LD    HL,#800C           LD   HL,L800C
#8005 ADD   HL,BC              ADD  HL,BC
#8006 ADD   HL,BC              ADD  HL,BC
#8007 LD    A,(HL)             LD   A,(HL)
#8008 INC   HL                 INC  HL
#8009 LD    H,(HL)             LD   H,(HL)
#800A LD    L,A                LD   L,A
#800B JP    (HL)               JP   (HL)
#800C DW    #8020       L8OOC: DB #20,#80,#30,#80
#800E DW    #8030              DB #40,#80
#8010 DW    #8040
```

The source code would then have to be altered if the code was to be re-assembled at a different address, the two lines of DBs could be changed to:

```
L8OOC: DW L8020,L8030,L8040
```

(Assuming L8020, L8030 and L8040 were defined elsewhere).

SPECTRUM only: The flexibility allowed for saving to any of the microdrives by typing “n:filename” previously described in the section on LOAD and SAVE also applies when FLISTing.

`WORK= <start>,<finish>` e.g. `WORK=40000,40100`

The WORK= command sets aside an area of memory as workspace for use by the FLIST command and the SLOW/TRACE commands (see debugger section).

`WIDTH= <word>` e.g. `WIDTH=50`

Set the width of the printer page in characters. On entry to the monitor this has a default value of 65535.

`LENGTH= <word>` e.g. `LENGTH=80`

Sets the length of the print page in characters. On entry to the monitor this has a default value of 65535.

# 7. Debug Commands

## *7.1 Single Stepping*

From the Monitor's editor the following keys can be used to perform single stepping operations:

| KEY | Operation performed |
|---|---|
| Amstrad/Spectrum | |
| CTRL I/caps & down arrow | Increment the value of PC |
| CTRL D/caps & up arrow | Decrement the value of PC |
| CTRL K/symbol-shift G | Skip the next instruction, make PC point to the address of the next instruction. |
| CTRL S/symbol-shift D | Single step the instruction pointed to by PC and make PC point to the next instruction. |
| CTRL E/symbol-shift F | Single step the instruction pointed to by PC unless the instruction is a CALL or RST, in which case the subroutine pointed to by the instruction will be executed at normal Z80 speed. Control will return back to the Monitor on exit from the subroutine and PC made to point to the instruction after the CALL or RST. |

### 7.1.1 Single Stepping and in line parameters

Before a single step is performed the instruction to be executed is copied into a special location in the monitor code. The fast execution of CALLs and RSTs when single stepping is not suited to the passing of in line parameters because in line parameters are not copied into the special locations mentioned previously e.g.

```
CALL  OUT-PARS
DB    4,8,12,16,20
```

If this code is executed using the CTRL E/SS&F type of single step the code “CALL OUT-PARS” will be copied into the monitor area for single stepping but the DB data will not. OUT-PARS is expecting to find operands at the return address but they will be missing and monitor code will be used as in line data instead. This means that routines which expect in line data (e.g. Spectrum RST 8) cannot be fast executed using CTRL E/SS&F and consequently slow modes 4 to 7 and 12 to 15 which perform fast execution of CALLs and RSTs cannot be used for code containing CALLs or RST followed by in line data.

If the user is single stepping or slow running code which makes CALLs or RSTs into the ROM areas of memory then Option Bit 4 should be switched ON to avoid the automatic fast execution of ROM routines which might expect in line parameters. A case in point is the RST 8 error routine in the Spectrum. As the RST 8 is in the ROM, unless option 4 is switched ON, the monitor will endeavour to fast execute it. However, the RST 8 is always accompanied by an in line parameter so a crash will occur but if option 4 is switched ON RST 8's can be single stepped.

Each time an instruction is executed the register values are loaded into the actual Z80 register values, the instruction executed and then the new register values saved away.

### 7.1.2 Using Single Stepping with the Amstrad

If while single stepping the instruction to be stepped is a CALL or JUMP to a firmware routine then the routine will be executed at normal speed and not single stepped. This is like using CTRL E, but the difference is that JUMPs are also executed (PC is set to the address on top of the stack and the stack pointer is incremented by 2 after the routine has finished). The execution of firmware calls in this way is an option and is set by option number 4, if clear, they are executed and if set, no special action is taken (see the OPTION command).

### 7.1.3 Using Single Stepping with the Spectrum

If a routine which calls or jumps to the ROM is single stepped, the call or jump will be executed at normal speed and not single stepped. This is like using the symbol-shift F type of stepping but the difference is that jumps are also executed (PC is set to the address on top of the stack and the stack pointer is incremented by 2 after the routine has finished). This execution of ROM routines is an option and can be turned off by turning on option number 4. If option number 4 is on then ROM routines are single stepped.

### 7.1.4 The Alternate Registers and Single Stepping

If an EXX or EX AF,AF' instruction is single stepped, the Monitor commands EXX and EX AF are used instead of the Z80 instruction so that the register display indicates that the alternate registers are now the main set (by printing an apostrophe). The monitor can only keep track of the register sets during single stepping and slow running. If a single (or odd number of) EXX (or EX AF,AF') is executed (by a JUMP or CALL) at normal Z80 speed then the monitor will be ignorant of it. This is particularly relevant to AMSTRAD users because BC' normally holds the ROM state.

## *7.2 Slow Running*

A slow run is a continuous single stepping. There are four different modes of screen update when slow running:

0. No screen update after each instruction is executed
1. Only the memory dump display is updated after each instruction
2. Only the register display is updated after each instruction
3. Both register display and memory dump are updated after each instruction

Mode 0 may seem pointless since you could use the Monitor's JUMP or CALL commands and execute it at normal speed but slow running has several advantages:

1. The ESCAPE key can be pressed at any time and control will return back to the monitor (the ESCAPE key on the Spectrum is symbol shift A).
2. When debugging programs it is useful to see what is happening in 'slow motion', especially graphic routines.
3. Special breakpoints can be used while slow running which do things other than just halting the program (see the section on breakpoints).
4. When slow running, the Monitor can be made to remember the addresses of every instruction it executed in the order that they were executed, this is via the TRACE command and is useful for back tracking from a breakpoint.

There are eight modes of slow running, numbered 0 to 7. Modes 0 to 3 use the CTRL S/ symbol shift D type of single stepping and modes 4 to 7 use the CTRL E/symbol shift F type. The execution of ROM routines applies exactly the same as it does for single stepping, remember the. limitations regarding in line parameters. The table below shows what each mode does:

| **Mode** | **Screen Update After Each Instruction** | **Stepping Type** |
|---|---|---|
| | | Amstrad/Spectrum |
| 0 | none | CTRL S/symbol-shift D |
| 1 | just memory display | CTRL S/symbol-shift D |
| 2 | just register display | CTRL S/symbol-shift D |
| 3 | both memory display and register display | CTRL S/symbol-shift D |
| 4 | none | CTRL E/symbol-shift F |
| 5 | just memory display | CTRL E/symbol-shift F |
| 6 | just register display | CTRL E/symbol-shift F |
| 7 | both memory display and register display | CTRL E/symbol-shift F |

## *7.3 Slow Running Commands*

`SLOW <slow mode number>` e.g. `SLOW 0`

Starts a program slow running from the address held in PC. Execution stops when either the ESCAPE key is pressed or a breakpoint is met (not all breakpoints cause programs to halt, seethe section on breakpoints). The value of <slow mode number> must be from 0 to 7. If a breakpoint has been set at the current PC value then any attempt to, slow run will stop instantly without executing any instructions.

Therefore if you have used fast execution to reach the current PC value and want to continue in a slow mode you must first single step over the breakpoint or switch the breakpoint off before continuing.

**NOTE:** Memory screen flags are reset at the start of a slow run (see Analyser – section 9.3).

## *7.4 Trace*

`TRACE [<number of instructions>]` e.g. `TRACE 1000`

During a slow run the value of PC for each instruction is stored in memory, the TRACE command prints out these addresses and the instructions stored at each address. The number of PC addresses remembered is dependent on how much workspace has been defined (using the WORK= command), two bytes of workspace are required for each address. If no workspace has been defined then during a slow run no addresses will be saved away and the TRACE command cannot be used (a "Workspace Undefined" error will be given). Each time that a SLOW command is executed the workspace is cleared so only one slow run can be remembered at a time.

The TRACE command lists from the oldest PC address it can remember and works forward to the point at which it halted. If no parameter is given then all the addresses stored are used, otherwise the specified number of most recent addresses is used. The listing can be paused by pressing a key and restarted again by pressing another (other than the ESCAPE key, which will return control back to the editor). If you have defined a large amount of workspace and the last slow run went on for a long time, the TRACE command might take a while to list all the PC addresses and get to the point at which the slow run had halted.

If you have used an FLIST command (which also uses the workspace), since the last slow run, the workspace will be unprintable and a "Workspace Undefined" error will be given.

`LTRACE <number of instructions>` e.g. `LTRACE 50`

LTRACE works in exactly the same way as TRACE but output is to the printer.

## *7.5 Breakpoints*

Breakpoints are special controls inserted into a program to make it change its mode of execution, it could cause the program to stop and return to the monitor (a normal breakpoint) or it could change the mode of execution to one of the slow modes (a special breakpoint). Up to eight breakpoints can be defined at any one time.

Breakpoints are implemented differently depending on what mode of execution you are using at the time, there are two modes: slow running and fast execution (JUMP or CALL commands).

### 7.5.1 Fast Execution

When testing machine code at normal Z80 speed using either the CALL or JUMP commands, breakpoints are implemented by placing a Z80 CALL instruction at the address where the breakpoint is required, this means that these breakpoints will only work in RAM. The address of the CALL instruction points to the Monitor's breakpoint handling routine. The CALL instructions are only placed in RAM at the start of a CALL or JUMP command and are removed on exit, this makes the CALLs 'invisible' to disassembly and memory dumps. Breakpoints are also loaded into RAM when the Monitor is exited so that programs can be executed from outside the Monitor but breakpoints still recognised.

Since a Z80 CALL instruction takes up three bytes of memory a problem may occur. Consider the section of program below:

```
#8000 CP     175
#8002 JR     NC,#8006
#8004 XOR    A
#8005 RET
#8006 LD     A,1
#8008 RET
```

This will set A=0 if A was initially less than 175, otherwise it sets A=1. Suppose you wanted to put a breakpoint at #8005 to halt when A was less than 175. You might run your program using the CALL command. If so a Z80 CALL instruction is put in your routine at #8005, but a CALL is 3 bytes long, so the routine at #8006 becomes corrupted and will no longer work correctly. To solve this problem either use slow running (breakpoints while slow running do not corrupt RAM) or just be extra careful in the placing of breakpoints when fast executing.

### 7.5.2 Slow Running

Breakpoints can be in ROM or RAM when slow running since nothing is placed in the code being run. Each time an instruction is single stepped the value of PC is compared with all the breakpoint addresses, a match indicates that one has been encountered.

### 7.5.3 Breakpoint Types

There are 18 different types of breakpoint; ALL TYPES ARE RECOGNISED IN BOTH SLOW AND FAST MODES. One type is the normal breakpoint which stops execution when it is encountered. The other 17 have special functions and are numbered 0 to 16. Types 0 to 7, when encountered, change the mode of execution to the slow mode corresponding to the breakpoint number. Types 8 to 15 are breakpoints with counters, each time the breakpoint is encountered the count is decremented, when the count reaches zero execution is halted. Type 16, the slow-fast breakpoint allows the user to switch the mode of operation of the monitor from slow to normal Z80 speed. Note that although 18 different types of breakpoint can be defined the total number of breakpoints permitted is 8. The table below shows the differences between the 17 special breakpoints:

| Type | Effect |
|---|---|
| 0 | Continue in slow mode 0 |
| 1 | Continue in slow mode l |
| 2 | Continue in slow mode 2 |
| 3 | Continue in slow mode 3 |
| 4 | Continue in slow mode 4 |
| 5 | Continue in slow mode 5 |
| 6 | Continue in slow mode 6 |
| 7 | Continue in slow mode 7 |
| 8 | Decrement count, halt if zero else slow mode 0 |
| 9 | Decrement count, halt if zero else slow mode 1 |
| 10 | Decrement count, halt if zero else slow mode 2 |
| 11 | Decrement count, halt if zero else slow mode 3 |
| 12 | Decrement count, halt if zero else slow mode 4 |
| 13 | Decrement count, halt if zero else slow mode S |
| 14 | Decrement count, halt if zero else slow mode 6 |
| 15 | Decrement count, halt if zero else slow mode 7 |
| 16 | Continue in fast execution mode. |

Breakpoints, once defined, can be turned off and on; a breakpoint in the off state is ignored if encountered during the execution of a program.

**NOTE:** When a slow to fast breakpoint (type 16) is encountered the slow run workspace is NOT cleared. Executing CALL or JUMP does however clear the slow run workspace.

### 7.5.4 Encountering Breakpoints

When a breakpoint is encountered the program is stopped and the message "Press a key" is displayed at the foot of the screen. If the user is slow to respond the monitor will 'BEEP'. So if you are debugging sound programs on the Amstrad you must respond before the first 'BEEP', because CHR$ (7) which causes the 'BEEP' also flushes the sound queue. After a key is pressed, the message "Breakpoint <number>" is printed (where <number> is the number of the breakpoint that caused execution to halt). PC is made equal to the address at which the breakpoint occurred. To continue from a breakpoint either turn the breakpoint off and continue with JUMP, CALL or SLOW, or if you require the breakpoint to stay active then single step past the breakpoint and then continue execution (remember if you are fast executing, a breakpoint is three bytes long and so at least the next three bytes will need to be single stepped before execution can continue). When using the type 16 breakpoint remember that once it has been encountered control will only return to the monitor if another breakpoint is encountered. Type 16 breakpoints have no effect while fast running.

Note that if any key (or the Amstrad joystick) is being pressed continuously when a breakpoint occurs the message "Release key/joystick" will be displayed. This is to prevent the monitor from responding to any backlog of user key presses.

## *7.6 Breakpoint Commands*

`BREAK <brknumber>,<flag>,<addr>` e.g. `BREAK2,ON,#1005`

Defines a breakpoint, with the given number, as a normal breakpoint with the specified address. The breakpoint number must be in the range 1 to 8 and if it was previously defined the old definition will be lost. The flag indicates the state the breakpoint will be left in after it has been executed. If <flag> = 0 then the breakpoint will be turned off after it has been encountered (and must be turned back on by the user using the BRK command). If <flag> = 1 then the breakpoint stays active after it has been executed. Whatever the value of <flag>, initially, the breakpoint will be on.

```
DEFBRK <brk number>,<type>,          e.g.   DEFBRK 4,0,#A000
            <addr> [,<count>]        or     DEFBRK 4,8,#A000,10
```

Defines a special function breakpoint, <brk number> is a breakpoint number from 1 to 8 and <type> is the breakpoint type from 0 to 16. If the given breakpoint is already defined the old definition is lost. <Addr> is the address of the breakpoint. A <count> parameter is required if the breakpoint is of type 8 to 15, this is the initial value of the breakpoint's counter. The breakpoint will be initially 'on' but can be turned off using the BRK command.

Breakpoint types 0 to 7 always stay on after they have been executed, so do types 8 to 15 as long as the count has not reached zero but, when it does the breakpoint will be turned off.

`BRK <brknumber>,<flag>` e.g. `BRK 1 ,OFF`

Sets the state of a given breakpoint. If <flag> = 0 then the breakpoint is turned off (but its definition is still remembered), if <flag>=1 then the breakpoint is turned on. If the breakpoint has not yet been defined then "Breakpoint Undefined" is printed. Turning on a counting type breakpoint will reset its count to the starting value it had when it was first defined.

`DELETE <brknumber>` e.g. `DELETE 8`

Removes a breakpoint definition from the list of the breakpoints. The breakpoint is then considered undefined.

`LBRK`

Lists the eight breakpoint definitions giving their type (either BRK for normal breakpoints or the. type. number for special breakpoints), the address, current state and, if the breakpoint is a special function type 8 to 15, its initial and current count values are also printed.

## 8. Monitor Error Messages

`ENTER COMMAND`

The last command was completed successfully and the monitor is waiting for a new command to be entered.

`COMMAND NOT KNOWN`

The Monitor cannot find the command name you have entered in its list of commands (you may be trying to use an Analyser command without the Analyser being present).

`NUMBER TOO BIG`

A number entered is out of the range 0 to 65535 (0 to #FFFF).

`OUTOF RANGE`

A number entered is out of range for a particular parameter, for example when a <flag> parameter is required a number either 0 or 1 (or ON or OFF) is required, entering any other value will produce this error.

`BAD NUMBER`

The Monitor is trying to read a parameter in as a number but cannot understand what has been entered.

`BAD STRING`

This error occurs when either the Monitor is expecting a string but a number has been entered instead, or if a string has no closing quote, or if a string of more than one character has been entered (unless in the DATA or SEARCH commands which allow strings to be more than one character in length).

`STRING TOO LONG`

Filenames cannot be longer than 16 characters on the Amstrad and 10 characters on the Spectrum. Byte lists for the SEARCH command cannot be longer than 20 bytes long.

`START > FINISH`

When <start> and <finish> parameters are required to define a block of memory the <finish> parameter must always be greater than or equal to the <start> parameter.

`TOO MANY OPERANDS`

The command entered expects fewer operands than have been supplied.

`TOO FEW OPERANDS`

The command entered expects more operands than have been supplied.

`FAILED AT <ADDR>`

This error is given by the CHECK command and gives the address of the first difference in the two blocks.

`FOUND AT <ADDR>`

Given by the SEARCH and NEXT commands, this message gives the address of the match.

`STRING NOT FOUND`

The SEARCH or NEXT command could not find any more matches of the search string.

`NO SEARCH STRING`

A NEXT command has been used either without a SEARCH command being used beforehand or after a search which had failed to find any more matches.

`COMMAND ABANDONED`

**Amstrad** - The LOAD, SAVE and FLIST commands produce this error if a tape or disc error occurs, the actual reason for the error will be printed by the firmware. Some other commands produce this error to indicate the ESCAPE key was used to terminate the command.

**Spectrum** - During a LOAD, SAVE or FLIST command this error will occur if break is pressed (shift & space), it is also used by other commands to indicate that ESCAPE (symbol-shift A) has been pressed and the command has been abandoned.

`BAD FILE`

Spectrum only, this indicates that the file being loaded from tape is corrupted in some way, equivalent to Basic's tape loading error.

`FILE ALREADY OPEN`

Amstrad only - If a file is left open and a LOAD, SAVE or FLIST command is used, this error is produced, the open file will be closed and so next time the LOAD, SAVE or FLIST command is entered the error should not be produced.

`WORKSPACE UNDEFINED`

No workspace has been defined before using an FLIST or TRACE command or the workspace is unprintable for the TRACE command.

`INSUFFICIENT WORKSPACE`

If during an FLI ST command there becomes too many labels to fit in the current allocation of workspace, this error will be produced.
This error message is also produced if an attempt to use the virtual screen is made before the memory has been defined.

`FILE NOT FOUND`

Spectrum only, this error occurs only during LOAD if the named file cannot be found on the selected drive.

`BAD OPCODE`

You have tried to single step or slow run an instruction that the Monitor could not disassemble, i.e. an instruction that is printed as `DB <n>; BAD`, where <n> is the byte it could not disassemble.

`BREAKPOINT UNDEFINED`

Using a BRK or DELETE command with a breakpoint that has not yet been defined will produce this error.

`BREAKPOINT <n>`

Breakpoint number <n> has been encountered and has caused execution to stop.

# 9. The Analyser

The Analyser allows the definition of intelligent breakpoints which cause a program that is slow running to stop when a particular condition occurs. The conditions that can be monitored are: the state of the flags, values of registers, the contents of memory and if memory has been written to or read from.

Conditions are set up using a subset of the language called FORTH (for those who already know FORTH, the words already defined for the Analyser Forth are listed at the end of this section).

The following discussion on the monitor/analyser includes examples with references to absolute numbers which are only suitable for the low version. Any version, high, low or relocated, of the. monitor/analyser may be used in the same fashion if the numbers are altered accordingly.

Before anything can be done with the Analyser some space for user defined definitions must be reserved, this is done using the PROG=<start>,<finish> command (where <start> and <finish> define a block of memory to be used as program space). To start with about 250 bytes of program space will be sufficient. As the low version of the. monitor/ analyser is being used define the required memory area as follows:

```
SPECTRUM:   PROG=42000,42249
AMSTRAD:    PROG=25000,25249
```

## 9.0.1 Introducing Analyser Forth

The analyser uses a dialect of Forth as the breakpoint controlling language for three reasons. Firstly, a Forth compiler is easy to implement and does not require a lot of memory to run in. Secondly, the code generated is very compact. Thirdly, and most importantly, Forth programs execute very rapidly. This latter quality is essential if the analyser is to be of any practical use.

Those users who are familiar with Forth should have no problems with this dialect but should still read this section carefully. We do not recommend newcomers to the language purchasing a text on Forth because the analyser uses only a very small subset of the words and the examples in this section are aimed to provide sufficient tutorial.

## 9.0.2 Using Analyser Forth

When processing a forth definition the analyser works along the definition from left to right dealing with each number or operator in turn. Numbers are simply placed on the "stack". Operators are interpreted and the required function performed. "Stack" in the context of this discussion is the analyser stack, not the Z80 stack. How would the analyser evaluate the following string?

```
6 2 3 4 + * +
```

As each term is encountered, working from left to right it is processed.

| | **Stack** |
|---|---|
| 6 is a number so it is placed on the stack | 6 |
| 2 is a number so it is placed on the stack | 6 2 |
| 3 is a number so it is placed on the stack | 6 2 3 |
| 4 is a number so it is placed on the stack | 6 2 3 4 |

+ is an operator so it is interpreted. The. effect of the + operator is to take the last two stack entries, add them and put the result on the stack:

| | | |
|---|---|---|
| | 4 is removed from stack | 6 2 3 |
| | 3 is removed from stack | 6 2 |
| the sum | 7 is placed on the stack | 6 2 7 |

* is an operator so it is interpreted. The effect of the * operator is to take the last two stack entries, multiply them and put the product on the stack:

| | | |
|---|---|---|
| | 7 is removed from stack | 6 2 |
| | 2 is removed from stack | 6 |
| the product | l4 is placed on the stack | 6 14 |

+ is an operator so it is interpreted. The last two stack entries are removed, added and the result is placed on the stack:

|  |  |  |
|---|---|---|
|  | 14 is removed from stack | 6 |
|  | 6 is removed from stack |  |
| the sum | 20 is placed on the stack | 20 |

The EVAL command allows a line of Analyser Forth to be executed immediately and shall be used to introduce the language to you. To verify that the analysis above is correct type.:

```
EVAL 6 2 3 4 + * + followed by ENTER
```

The following will appear (#14 = 20):

```
STATE OF STACK:
#14
ENTER COMMAND
```

Forth uses a stack to store all its data. The EVAL command will clear the stack before it evaluates any string. To get Forth to do anything for you, you must put data on the stack first, for example you might want to add the numbers 2 and 3 together using the EVAL command. First you must place the two numbers on the stack, to do this type:

```
EVAL 2 3 (and press ENTER)
```

The following will appear (as long as program space has been defined):

```
STATE OF STACK:
#3 #2
ENTER COMMAND
```

This shows you that the Analyser can place numbers on the stack. The stack is known as a first-in-last-out structure because you can keep adding items but if you want to get at something that was placed on the stack earlier on you must remove everything above it first. For example, say you place the numbers 1, 2 and 3 on the stack in that order. 1 is on the bottom of the stack and 3 is on the top; to get at 1 again you must first remove the 3 and then the 2. So numbers placed on the stack are read off in the opposite order to which they went on.

Now to add the two numbers together, Forth has a word called '+' which it understands to mean addition. It takes the top two stack entries and adds them together, placing the result on the top of the stack, so now type:

```
EVAL 2 3 +
```

and, hey presto, you get the answer »5 (note the Analyser is printing the stack in hexadecimal - type OPTION 1,ON to select the decimal printing mode).

To demonstrate how the stack works, try:

```
EVAL 1 2 3 +
```

This leaves 5 and 1 on the stack, showing that the two most recent entries are added together. Now try:

```
EVAL 3 2 1 - +
```

The answer should be 4, to see why, look at the diagram below (the 'state of stack' diagram shows the bottom of the stack to the left):

| **Word** | **State of Stack** | **Explanation** |
|---|---|---|
| EVAL | empty | clears the stack |
| 3 | 3 | puts 3 on the top of the stack |
| 2 | 32 | adds 2 to the top of the stack |
| 1 | 32 1 | adds 1 to the top of the stack |
| - | 31 | 2 minus l equals l |
| + | 4 | 3 plus l equals4 |

Note that '-' is another Forth word, this time meaning subtraction (you can find all the defined words listed at the end of this section).

Instead of putting numbers on the stack you could use the values of the Z80 registers. For example, EVAL HL BC + will place the values of HL and BC on the stack, add them together and put the answer back on the stack. Try the following:

```
HL=100
BC=200
EVAL HL BC +
```

The answer should be 300 (decimal). Executing a word that requires items on the stack when none are there will cause a “Stack Empty” error to be printed. For example, EVAL + will give such an error.

While numbers are on the stack we can move them around. For example SWAP swaps the top two stack items over, so the following:

```
EVAL 1 2 3 SWAP
```

will leave the numbers on the stack in the order 1 3 2 (working from the bottom of the stack going up). ROT rotates the top three items so

```
EVAL 1 2 3 4 ROT
```

produces 1 3 4 2 (working from the bottom up again), this has taken the third item to the top of the stack.

### *9.0.3 Defining Words*

So far we have been using words such as +, -, HL and SWAP. These are all predefined words but you can define your own using the WORD command, for example:

```
WORD FRED: DUP *
```

This defines a word called FRED which gives the square of the number held on top of the stack. DUP is a defined word which makes a copy of the top stack item onto the stack and ‘*’ is multiply. Note that when you enter this line nothing is executed, the definition is just stored away in the program space. To list any definitions you have made and to see how much space you have left use the LDEF (list definitions) command. Now try:

```
EVAL 4 FRED 5 FRED
```

You should get the answers 16 and 25 (since 4 squared is 16 and 5 squared is 25), the diagram below shows what happens:

| **Word** | **State of Stack** | **Explanation** |
|---|---|---|
| EVAL | empty | stack cleared |
| 4 | 4 | 4 placed on stack |
| FRED » DUP | 4 4 | another copy placed on stack |
| * | 16 | multiply top two items |
| 5 « | 16 5 | 5 placed on stack |
| FRED » DUP | 16 5 5 | another copy placed on stack |
| * | 16 25 | multiply top two entries |
| <end> « | | |

The definition FRED can only be used after it has been defined, otherwise. a “Word Not Known” error is given. A word may be redefined by using the WORD command but there are things to note.

1. If an error is made during the redefinition of a word the old definition will be retained.
2. If a correct redefinition is made the warning “Word Redefined” will appear after the redefinition has been made. The old definition will have been lost by the time the warning appears.
3. During the redefinition of a word, other definitions which had not been made at the time of the original definition, may be incorporated into the new definition.

User word names must be made up of alpha-numeric characters only. Since word names could consist of numeric characters only it is possible (but not advised) to redefine numbers, for example try this:

```
WORD 10: 11
EVAL 10 10 +
```

The answer printed will be 22 since you have redefined the number l0 to give the value 11.

We still have FRED defined, we can now use it in a new definition, say SUMS, which will square the top two stack items and add them together:

```
WORD SUMS: FRED SWAP FRED +
EVAL 4 5 SUMS
```

and will give the answer 41 (since 4*4+5*5=41). We can use FRED again to square the result of SUMS to give us a new word which we can call SQUARES:

```
WORD SQUARES: SUMS FRED
EVAL 2 3 SQUARES
```

giving the answer 169 (2*2+3*3=13 and 13*13=169).

How the result 169 has been calculated may be clarified. The string typed in originally is:

```
EVAL 2 3 SQUARES
```

Inside the analyser SQUARES is replaced by its definition. In effect the string becomes:

```
EVAL 2 3 SUMS FRED
```

Now the word SUMS can be replaced by its definition. The string becomes:

```
EVAL 2 3 FRED SWAP FRED + FRED
```

Finally the word FRED may be replaced so the string becomes:

```
EVAL 2 3 DUP * SWAP DUP * + DUP *
```

Working from left to right through this string:

EVAL is a command to clear the stack and evaluate the subsequent string.

| **EVAL** | | **Stack** |
|---|---|---|
| 2 | a number so it is placed on stack | 2 |
| 3 | a number so it is placed on stack | 2 3 |
| DUP | an operator. Duplicate the last stack entry | 2 3 3 |
| * | an operator. Remove last two entries and multiply them, put result on stack | 2 9 |
| SWAP | an operator. Remove last two entries then put them on stack in reverse order | 9 2 |
| DUP | an operator. Duplicate the last stack entry | 9 2 2 |
| * | an operator. Remove last two entries, multiply them, put result on stack | 9 4 |
| + | an operator. Add last two entries, result on stack | 13 |
| DUP | an operator. Duplicate the last entry | 13 13 |
| * | an operator. Multiply | 169 |

Our collection of definitions now looks like this:

```
FRED:     DUP*
10:       11
SUMS:     FRED SWAP FRED +
SQUARES:  SUMS FRED
```

We can see that words can be defined in terms of both predefined words and other, user defined, words and each definition can be used more than once.

To remove any old definitions from the program space use the CLEAR command, this removes all user definitions, so use it with care.

To get the Analyser to test for conditions while a program is running, a STOP must be defined. This is a word with the name STOP followed by a single digit from 0 to 9, e.g. STOP 1 or STOP 7. STOPs can be used like any other word definitions but during SLOW running the STOPs are evaluated after every instruction has been executed. AFTER EVALUATING A STOP DEFINITION THERE SHOULD BE ONE NUMBER LEFT ON THE STACK, if not an error will occur. If this value is non-zero the program will stop and the number of the STOP (0 to 9) in which the halt was caused will be printed.

**NOTE:** When a STOP definition is actually processed the PC is pointing at the next instruction to be executed but the. flags and registers are all set to the values that they held at the. end of the last instructions executed. This is clarified by example 4-‘STACK CHECKER’.

To explain the use of STOPs we require a demonstration program. Type in the following list of commands

```
Amstrad                           Spectrum

MEM= #6978                        MEM=#ABE0
DATA #21,0,#C0,#75,#23,#7C        DATA #21,0,#40,#75,#23,#7C
DATA #FE,0,#20,#F9,#C9            DATA #FE,#58,#20,#F9,#C9
```

This will load a short machine code program into the. memory at #6978/ABE0 (at #6978 for the Amstrad, at #ABE0 for Spectrum). The disassembly of the code is given below:

```
#6978 LD HL,#C000          #ABE0 LD HL,#4000
#697B LD (HL),L            #ABE3 LD (HL),L
#697C INC HL               #ABE4 INC HL
#697D LD A,H               #ABE5 LD A,H
#697E CP #0                #ABE6 CP#58
#6980 JR NZ,#697B          #ABE8 JR NZ,#ABE3
#6982 RET                  #ABEA RET
```

The program fills the screen with a pattern. HL is used as a pointer into the screen memory. On the Amstrad when H=0 address #0000 has been reached and so the routine finishes. On the Spectrum the end of the screen memory is at #5800 and so H is compared with #58 to see if the screen end has been reached. To test the routine use the command:

```
Spectrum:    CALL #ABE0
Amstrad:     CALL #6978
```

Say we wanted the routine to stop when HL was equal to #C010 (#4010 on the Spectrum). To do this we need to define a STOP using the Analyser. First remove any old definitions that are no longer needed:

```
Amstrad                     Spectrum

CLEAR                       CLEAR
WORD STOP 0: HL #C010=      WORD STOP 0 : HL #4010=
```

STOP0 has now been set up to detect when HL=#C010 (Amstrad) or #4010 (Spectrum).

The Forth word '=' compares the top two stack items, if they are equal 1 is put on the stack otherwise a 0 is placed on the stack.

The Analyser can only be used while slow running not fast executing, so to run the program type:

```
Spectrum:    PC = #ABE0
Amstrad:     PC = #6978

and then:    SLOW 0
```

Part of the screen will be filled, then the Monitor will print "Press any key", pressing a key updates the display and at the bottom of the screen the message "Stop number 0" will be displayed, the current instruction will be an INC HL instruction. This is the point where HL was made equal to the required address. Note that PC still points to the instruction that was executed, to continue, use the CTRL K (symbol-shift G) operation to skip past this instruction before using SLOW again. In this case trying to continue will cause the Analyser to stop the program again since the condition set by STOP 0 is still true. To continueeither CLEAR thedefinitions or type ANALYSER OFF which will stop the Analyser checking conditions but will keep any definitions defined. Typing ANALYSER ON or CLEAR switches the Analyser back on.

ADDR, RD, WR and ACF are Analyser words that allow the detection of reading or writing of memory (see the list of definitions for an explanation). Using the same program as above type in the following definition:

```
Amstrad                     Spectrum

CLEAR                       CLEAR
WORD SCREEN: ADDR #C010=    WORD SCREEN: ADDR #4010=
WORD STOP 0: SCREEN WR&     WORD STOP 0: SCREEN WR&
```

The definition SCREEN returns 1 if the last instruction accessed the required memory location otherwise it returns 0. The word & is used to ensure that the STOP occurs if SCREEN is true and the memory was written to. Run the program now using:

```
Spectrum:    PC = #ABE0
Amstrad:     PC = #6978

and then:    SLOW 0
```

The Analyser will now stop at the LD (HL),L instruction when HL=#C010 (or HL=&4010). The Analyser can be used to check that a particular value is written to memory:

```
CLEAR
WORD VALUE: ADDR C@175=
    WORD STOP 0: VALUE WR&
```

The Analyser will check if the value written to memory is 175 and if so it will halt. There are several things to note about the Analyser:

1. The logical operator '&' and the bitwise operator OR can be used to chain conditions together, for example "condition1 AND (condition2 OR condition3)" would be converted to Forth as:

   ```
   condition2 condition3 OR condition1 &
   ```

2. Words in FORTH definitions need to be separated, in the examples above a spacehas been used but commas and colons could be used instead. The WORD command can have any of the three separators after the word being defined, but in the examples above a colon has been used to make the definitions clearer.

3. The. WORD command can be. abbreviated to $<word name> <definition>, for example:

   ```
   $FRED 1 1 +
   ```

4. More than one STOP can be defined at a time, since STOP 0 to STOP 9 are valid STOPs, then up to 10 STOPs can be defined at any one time. But remember the more STOPs there are the slower SLOW running will be!

## *9.1 Analyser Commands*

`PROG= <start>,<finish>` e.g. `PROG=#0000,#1000`

Defines the position in memory of the Analyser's PROGram space. The command will erase all previous definitions and enable the Analyser (see the ANALYSER command). Take care to ensure that PROG space is not in user code or monitor code.

`CLEAR`

Resets the program space and erases all previous commands. A "Workspace Undefined" error will be given if a PROG= has not been used before hand. The Analyser will be enabled by the use of this command (see the ANALYSER command).

`ANALYSER <flag>` e.g. `ANALYSEROFF`

Enables or disables the Analyser from working during a slow run. If <flag> = 0 then no checking occurs during a slow run.

`WORD <wordname> <definition>` e.g. `WORD FRED HL C@10=`
or `$ <wordname> <definition>` or `$FRED HL C@10=`

Defines an Analyser word, <word name> must be made of alpha-numeric characters only and should not have been defined before. <Definition> is a list of at least one number or Analyser Forth word.

`EVAL <definition>` e.g. `EVAL FRED`

Evaluates the given definition and then prints the state of the stack.

`LDEF`

Lists the current user defined definitions and prints how much free memory is left in the program space. Note that the space left in the program space is used for the stack during execution time. If there is insufficient space left an "out of Stack Space" error will be given.

`PDEF`

As LDEF but output is to the printer.

`DEFSAVE "<string>"`

Save the current analyser definitions to a tape, disc or microdrive file named <string>. Amstrad users will require a 2k buffer and this will require the use of the BUFFER= command. It is often a good idea to set BUFFER to the second half of PROG SPACE (see FLIST).

`DEFLOAD "<string>"`

Load a previously saved file of analyser definitions named <string> from tape, disc or microdrive. Note that Amstrad users require a 2k buffer (see DEFSAVE).

**NOTE: Amstrad**

When the definitions are saved, they are first disassembled back to their ASCII form (hence the 2k buffer). When they are re-loaded, they are re-compiled back into PROG SPACE. This means that a PROG= will need to be executed before a DEFLOAD or a PROG SPACE UNDEFINED error will occur. It is also possible that a DEFLOAD may be terminated with an INSUFFICIENT SPACE error if the current PROG SPACE is not large enough to accommodate the previously saved definitions. If these errors occur reset PROG SPACE and try again.

**NOTE: Spectrum**

The Spectrum version does not DEFSAVE ASCII files in the same way as the Amstrad and so the DEFSAVE, DEFLOAD sequence has to be treated in a different way. The Monitor/Analyser must be relocated to the same address when DEFLOADing a particular file, as it was when DEFSAVing the particular file. DEFLOAD also resets PROG SPACE automatically to the size and position that it had at the time of the DEFSAVing. This means that the Spectrum user must be extra cautious when setting PROG SPACE prior to typing definitions because it is not possible to increase PROG SPACE without losing the definitions. As mentioned previously in SAVE and LOAD, a microdrive number may be included in the filename string to direct the monitor to access that drive rather than the current microdrive..

`EDIT <wordname>`
Edit a previously defined analyser/stop definition.

The $ is automatically inserted at the start of the word definition which is displayed on the edit line with the cursor pointing to the first character of the word definition. The line is then edited in the normal way. Pressing ENTER will cause the new definition to be processed. There are several points to note:

i) If the word name itself is not edited then the previous definition is replaced by the new definition. The 'Word Redefined' message is issued to indicate a successful redefinition.

ii) If the word name itself is edited then the new word is created and the previous definition is preserved.

iii) Spaces will automatically be inserted between each word in a definition when it is displayed to the editor line and a colon is automatically inserted at the end of the word name. This means that a word that was originally entered as

```
$FRED #1 #2 #3
```

would be displayed on the edit line as

```
$FRED: #1 #2 #3
```

Occasionally this will cause a definition to exceed the 40 characters allowed. If this is the case, then the definition will be automatically truncated at 40 characters (including carriage returns). For this reason, care should be exercised when compacting definitions.

iv) It is also possible, although unlikely, that a 'No space' error will be given when the edited line is ENTERed. If this does occur then Amstrad users should save the definitions, increase PROG SPACE, reload the definitions and try again. Spectrum users cannot surmount this problem and it is therefore advisable to ensure ample PROG SPACE before beginning a session. As an indication, the STACK CHECKER example used in this manual would require about 500 bytes of program space to ensure sufficient editing space. 1k of PROG SPACE should cover almost any eventuality.

## *9.2 Analyser Errors*

`PROGRAM SPACE UNDEFINED`
An analyser command has been used without setting up program space with the PROG= command.

`OUT OF PROGRAM SPACE`
This error can occur in a WORD command because insufficient program space has been defined and the current definition will not fit in the space that is left. This error can also occur with the EVAL command since the definition is compiled into the program space before it is executed.

`WORD NOT KNOWN`
You are trying to use a word in a definition which has not yet been defined.

`NO DEFINITION`
You are trying to define a word without any definition.

`STACK EMPTY`
The current word requires at least one operand on the stack but there are insufficient left on the stack for it to use.

`STACK NOT EMPTY`

This error can occur when a STOP has been executed and more than one number is left on the stack at the end of the definition.

`DIVISION BY ZERO`

The word '/' has been used with a divisor of 0.

`INSUFFICIENT STACK SPACE`

The stack is too big to fit in the spare space left in the program space.

`NO SPACE`

An attempt has been made to use the EDIT command with insufficient PROC SPACE defined - see EDIT.

`WORD REDEFINED`

A pre-defined analyser definition has been successfully re-defined - see EDIT.

## *9.3 Analyser Forth Reserved Words*

In the following list of definitions a stack diagram is given for each word, this represents how the word effects the stack. In these diagrams the '>' sign splits the 'before' and 'after' parts. Numbers on the stack are represented by nl, n2 and n3, these are 16 bit integers and so have the range 0 to 65535. Sometimes words use flags and fl, f2 and f3 represent these, a flag uses zero to represent false and a non-zero value (usually 1) to represent true. Here is an example:

```
n1 n2 > f1
```

This would mean that the definition would expect at least two numbers on the stack, where n2 is the top stack item and n1 is the next stack item. After the word had been executed n1 and n2 would have been removed leaving a flag (which has the value 0 or 1) on the top of the stack. A "Stack Empty" error is given if there are insufficient items on the stack when the command is executed.

### 9.3.1 Defined Words

### Register Values

`AF, BC, DE, HL, IX, IY, SP, PC` > nl

These eight words place the current value of the corresponding register pair on the stack.

`A, B, C, D, E, H, L, F` > nl

Place the value of the indicated register on the stack.

`CF, PVF, NF, ZF, SF, HF` > fl

Place a flag on the stack which reflects the value of one of the Z80 flags as shown below:

| | |
|---|---|
| CF | Carry flag |
| PVF | Parity/Overflow flag |
| NF | Add/Subtract flag |
| HF | Half carry flag |
| ZF | Zero flag |
| SF | Sign flag |

The flag value placed on the stack will be 0 if the flag was clear or 1 if it was set.

`I` > nl

Interrupts enabled/disabled. The flag value placed on the stack will be 0 if interrupts are disabled but 1 if they are enabled.

### Memory Addressing

ADDR > nl

ADDR places the address of the last memory location accessed on the stack. If the most recent instruction did not write to or read from memory then ADDR will return the value 0. Not all memory accessed by an instruction is recorded, the following do not effect ADDR:

| | |
|---|---|
| Block moves | (LDI, LDD, LDIR and LDDR) |
| Block compares | (CPI, CPD, CPIR and CPDR) |
| Block inputs and outputs | (INI, IND, INIR, OUTI, OUTD, OTIR and OTDR) |

| | |
|---|---|
| Stack operations | (PUSH, POP, CALL, RST and RET) |

The following are recognised and effect ADDR:

| | | |
|---|---|---|
| Absolute addressing | e.g. | LD (#4321),HL<br>LD SP, (#ABCD) |
| Register indirect addressing | e.g. | LD (BC),A<br>ADD A, (HL)<br>BIT 4, (IX-3)<br>RLD and RRD |

ADDR can be used from the EVAL command and it will reflect the actions of the instruction PC is pointing to at that moment.

`RD, WR, ACF` > f1

All three instructions leave a flag on the stack. RD leaves a true flag (1) if the last instruction read from memory otherwise it leaves false (0). WR leaves a true flag if the last instruction wrote to memory. ACF leaves a true flag if the last instruction accessed memory (i.e. either read or write).

Some instructions read and write to the same location, they are listed below:

| | |
|---|---|
| Increments or decrements of a memory location | e.g. INC (IX+ 12) |
| Rotations of memory locations | e.g. SLA (HL) |
| Bit set/resets of memory locations | e.g. SET 4, (IY-1) |

**NOTE:** Memory access flags RD,WR and ACF are reset at the start of a slow run.

## Arithmetic Operations

| | | | |
|---|---|---|---|
| + | addition | n1 n2>n3 | (n3=n1+n2) |
| - | subtraction | n1 n2>n3 | (n3=n1-n2) |
| * | multiplication | n1 n2>n3 | (n3=n1*n2) |
| / | division | n1 n2>n3 | (n3=n1/n2) |

These four mathematical operators use the top two stack items as their operands, if there are less than two numbers on the stack at the beginning of the word a “Stack Empty” error is given. The operands are removed before the answer is placed on the stack.

Note that the Analyser stores its numbers as 16 bit integers, i.e. a number from 0 to 65535 and so negative numbers, decimals and numbers greater than 65535 cannot be entered. The Analyser has no overflow error and so only the least significant 16 bits of any answer is ever remembered, the following examples demonstrate this:

| | |
|---|---|
| EVAL 65535 1 + | gives the answer 0 |
| EVAL 0 1 - | gives the answer 65535 |
| EVAL 500 500 * | gives the answer 53392 (i.e. the 16 LSB’s of 250000) |

Since numbers are stored as integers, the result of division will be rounded down to a whole number, e.g. 10 4 / will give the answer 2 instead of 2.5.

## Logic Operators

| | | |
|---|---|---|
| AND | bitwise AND | n1 n2 > n3 |
| OR | bitwise OR | n1 n2 > n3 |
| XOR | bitwise exclusive-OR | n1 n2 > n3 |
| NOT | bitwise complement | n1 > n2 |

These four operators perform the given bitwise logic operations on the top two stack items (only one for NOT) and after removing the operands the answer is placed back on the stack. The logic operations are performed on the full 16 bits of a number and the tables below show the effect on the individual bits for the four operations:

```
0 AND 0 = 0   0 OR 0 = 0   0 XOR 0 = 0   NOT 0 = 1
0 AND 1 = 0   0 OR 1 = 1   0 XOR 1 = 1   NOT 1 = 0
1 AND 0 = 0   1 OR 0 = 1   1 XOR 0 = 1
1 AND 1 = 1   1 OR 1 = 1   1 XOR 1 = 0
```

## Relational Operators

| | | |
|---|---|---|
| “ | n1 equal to n2? | n1 n2 > f1 |
| > | n1 greater than n2? | n1 n2 > f1 |
| < | n1 less than n2? | n1 n2 > f1 |
| >= | n1 greater than or equal to n2? | n1 n2 > f1 |
| <= | n1 less than or equal to n2? | n1 n2 > f1 |
| <> | n1 not equal to n2? | n1 n2 > f1 |
| 0= | n1 equal to 0? | n1 > f1 |
| 0> | n1 not equal to 0? | n1 > f1 |

These operators compare either the top two numbers on the stack, or the top stack item with 0. Fl is either 1 or 0 to represent true or false. A “Stack Empty” error is given if there are insufficient items on the stack. As with the other operators the operands are removed from the stack before the answer is placed on the stack.

& Logical AND f1 f2 > f3

If the top two stack systems are both non-zero, a true flag (1) is put in their place, otherwise they are replaced by a false flag (0). The & operator can be used to chain conditions together, where any non-zero value on the stack represents a true condition and a zero value represents false. This word acts differently to the word AND since 1 2 AND would produce the answer 0, even though 1 and 2 both represent true conditions. The bitwise OR word can be used for chaining conditions together and 0 = can be used to perform the logical NOT. For example you may want to convert the following to Analyser Forth:

> “Stop if condition 1 is false and either condition 2 or condition 3 are true”

this is translated to:

> (NOT condition 1) AND (condition 2 OR condition 3) and can be written in Analyser Forth as:
>
> STOP1: condition 1 0 = condition2 condition3 OR &

## Other Operators

?SCR1 f1 >

If the flag on the top of the stack is true then the memory dump display will be updated immediately, if the flag is false no action is taken.

e.g. to update the stack when HL = #0100 we would use the following structure. Note, the zero on the end is to prevent a “stop0” occuring:

```
$STOP0: HL#100=?SCR10
```

?SCR2 f1 >

This word may be used in the same fashion as ?SCR1 but the register display will be updated rather than the memory dump.

! store word n1 n2 >

This command can be used to deposit a value in memory. The word value n1 is stored at the address n2. Both numbers are dropped from the stack.

e.g. to store the value #1234 at the address #6000 we would use the following structure:

```
#1234 #6000 !
```

The effect would be that:

| address | data |
|---|---|
| #6000 | #34 |
| #6001 | #12 |

C! storebyte n1 n2 >

This command is similar to ! but only stores one byte. The least significant byte of n1 is stored at the address n2. Both numbers are dropped from the stack. Use capital C only.

MEM > n1

The address held in the monitor system variable MEM is put onto the stack.

This command allows the memory pointer to be incremented and decremented whilst slow-running.

MEM! n1 >

This will set the memory pointer to the last value on the stack.

?PAUSE f1 >

If the flag on the stack is true the analyser stops and waits for a key-press. Pressing ESC/SS&A will cause the slow run to be aborted. Pressing any other key will allow the slow run to continue. The flag is removed from the stack.

e.g. to pause the analyser if the IX register becomes equal the the PC value we would use the following definition:

```
#STOP0: PC IX =?PAUSE
```

Note, however, that the evaluation of STOP words must leave a flag on the stack. If there is no flag a “stack empty” error will occur. The above definition would leave the stack empty; so to prevent the error occuring we must alter it to leave a flag for the stop condition.

To stop the analyser immediately after key-press, whether ESC/SS&A or not, use the definition:

```
$STOP0: PC IX =?PAUSE 1
```

but to continue after the key-press, unless ESC/SS&A is pressed, use the definition:

```
$STOP0: PC IX “ ?PAUSE 0
```

By adding a 1 after the ?PAUSE the whole point of the pause is negated, the analyser stops irrespective of which key is pressed and irrespective of whether a pause occurs or not, i.e. only one instruction will be executed. However, by adding 0 after the pause the required pause function is obtained, only when ESC/SS&A is pressed in response to the pause will the analyser stop the slow run.

`IF` f1 >

If the flag on the stack is false the word currently being evaluated will be prematurely terminated. This permits the definition of words which will be executed conditionally, see the later examples. The IF command removes one flag from the stack.

e.g. `$STOP3: HL #4000 = IF HL MEM!`

When evaluating this definition IF HL is not equal to #4000, the functions after the IF will not be considered but IF HL= #4000 the value of HL will be assigned to the MEMory pointer.

Again, remember that a flag must be left on the stack by the evaluation of the STOP, so it is necessary to insert another function. As the IF may cause a premature termination of the evaluation of STOP3 we must insert the function before the IF so the definition becomes:

```
$STOP3: 0 HL#4000 = IF HL MEM!
```

By inserting 0 the evaluation of STOP3 will yield a result 0. Therefore, STOP3: will never cause the analyser to stop but when HL = #4000 the MEMory dump pointer will be set to #4000.

`C@` byte fetch n1 > n2

The address n1 is replaced on the stack by the byte stored at n1. e.g. to find what HL is pointing to use HL C@ or to find what (IX+10) is pointing to use IX 10 + C@.

`@` word fetch n1 > n2

This word is like the C@ word except that a 16-bit word is fetched instead of an 8-bit byte. The word is formed from the two consecutive locations pointed to by the address held on the stack. For example, to fetch the address held at locations 23100 (low byte) and 23101 (high byte) use: 23100 @. Note that a 16-bit word is placed on the stack and not two separate bytes.

`BIT` bit test n1 n2 > f1

This word expects two numbers on the stack, n1 and n2. The result, f1, is a copy of the n2th bit of n1. n2 should be in the range 0 to 15 (since numbers are stored in 16 bits) and is taken modulus 16 if greater than 15 (i.e. 31 represents bit 15 etc.).

## Stack Operations

`DUP` n1 > n1 n1

Duplicates the top item on the stack.

`SWAP` n1 n2 > n2 n1

Swaps the top two stack items over.

`OVER` n1 n2 > n1 n2 n1

The second from top stack item is copied onto the top of the stack without removing it from its original position.

ROT n1 n2 n3 > n2 n3 n1

Rotates the top three stack items around as shown in the diagram.

## Other Words

ON has the value1 >1

Places 1 on the stack.

OFF has the value 0 >0

Places 0 on the stack.

DROP n1 >

Remove the top number from stack.

NOP >

This is a dummy command that does nothing. It is equivalent to a Z80 NOP.

KEY > n1

Stack the value of the last key pressed. This facility is included to allow the analyser to be externally controlled. This is particularly useful if you are using a complex definition that slows program execution. This facility can be used to switch the definition on and off as required. If no key is being pressed the value ‘0’ will be stacked.

For example, suppose we wished the analyser to execute a word called CHECK1 before the “A” key were pressed, and then to execute a word called CHECK2 after the “A” key were pressed. We could use the following definition:

```
TESTA KEY DUP “A” = SWAP “a” = OR DUP
```

This would leave 2 true flags on the stack if the “A” key were pressed or 2 false flags otherwise.

```
TEST1 TESTA IF CHECK2
```

This will use the first flag which, if true, (“A” has been pressed), would cause CHECK2 to execute.

```
CHECK TEST1 NOT IF CHECK1
```

This is the complete definition which will execute CHECK1 if “A” has not been pressed and CHECK2 if “A” has been pressed.

CALL

The word CALL allows extra functions to be added to the Analyser which cannot be made up from other, already defined words. CALL calls the machine code routine whose address is held on top of the stack. On entry to the routine the following registers are set up:

HL: Address of a routine that will pop the top item off the analyser’s stack and place it in the register pair BC, this routine is called DROPBC.

DE: Address of a routine that places the value of BC onto the Analyser’s stack, this routine is called STKBC.

IX: Address of a table containing the values of the registers in the program currently being slow run, the table is set out as follows:

| | |
|---|---|
| IX+0,1 | SP |
| IX+2,3 | IX |
| IX+4,5 | IY |
| IX+6,7 | PC |
| IX+8,9 | BC |
| IX+10,11 | DE |
| IX+12,13 | HL |
| IX+14 | flag byte |
| IX+15 | A |
| IX+16,17 | BC’ |
| IX+18,19 | DE’ |
| IX+20,21 | HL’ |
| IX+22 | F’ |
| IX+23 | A’ |

The routines STKBC and DROPBC can be used to pass parameters into and out of the machine code using the Analyser’s stack, the layout of any code used within the CALL word should be as follows:

1. Read required parameters off the stack using DROPBC.

2. Perform required operation.
3. Put any return parameters back on the stack using STKBC.

For example, to write a routine which will ‘beep’ if the top stack item is a true flag (i.e. nonzero) the following will be required:

**Amstrad:**

```
BEEP:   CALL DROPBC ; Fetch parameter from stack
        LD   A,C    ;
        OR   B      ; If BC = 0 then just return
        RET  Z      ;
        LD   A,7    ; Otherwise use firmware to print a chr$(7)
        JP   #BB5A  ;
DROPBC: JP   (HL)   ; This routine jumps to the routine whose
                    ; address is held in HL (i.e. DROPBC)
```

**Spectrum:**

```
BEEP:   CALL DROPBC ; Fetch parameter from stack
        LD   A,C    ;
        OR   B      ; If BC = 0 then just return
        RET  Z      ;
        LD   HL,#100; Otherwise use ROM routine to make a beep
        LD   DE#80  ;
        JP   #3B5   ;
DROPBC: JP   (HL)   ; This routine jumps to the routine whose
                    ; address is held in HL (i.e. DROPBC)
```

If the appropriate routine from above was assembled in a free area of memory e.g. #ABE0 on the Spectrum or #6978 on the Amstrad an Analyser word BEEP could be defined which would perform ‘conditional beeps’:

```
Spectrum:   $BEEP #ABE0 CALL
Amstrad:    $BEEP #6978 CALL
```

and could be used thus:

```
$STOP1 PC #A000 = BEEP 0
```

This would cause a beep sound to be made each time the location at #A000 was slow run. Note that the zero on the end is required because the STOP word expects a value on the stack at the end of the definition and 0 tells the Analyser not to stop.

**AMSTRAD only**

The virtual screen window may be defined from within the analyser using the following set of commands. Each of these commands removes one entry from the stack.

```
COL!  n1 >
ROW!  n1 >
LEN!  n1 >
HGT!  n1 >
```

The appropriate window parameter is set to the value on the stack e.g.

EVAL 1 COL! will set the window column to 1.

EVALl 2 3 4 COL! ROW! HGT! LEN!

will set the window column to 4
will set the window row to 3
will set the window height to 2
will set the window length to 1

## 9.4 Useful Definitions

Given below are a list of useful words that can be defined and then used to make up more powerful definitions (none of the definitions already exist in the Analyser and so must be defined by you when you require them):

RANGE range checking n1 n2 n3 > f1

```
$RANGE: ROT SWAP OVER >= ROT ROT <= &
```

Checks if the value n1 is in the range of n2 to n3 inclusive, for example HL #8000 #80FF. RANGE checks if HL lies in the range #8000 to #8OFF inclusive.

| Word | State of Stack |
|---|---|
| RANGE | n1 n2 n3 |
| ROT | n2 n3 n1 |
| SWAP | n2 n1 n3 |
| OVER | n2 n1 n3 n1 |
| >= | n2 n1 f1 |
| ROT | n1 f1 n2 |
| ROT | f1 n2 n1 |
| <= | f1 f2 |
| & | f3 |

MEMWR Memory Protection n1 n2 > f1

```
=$MEMWR: ADDR >= SWAP ADDR <= & WR &
```

This word can be used to protect areas of memory from being written to by the machine code being slow run. n1 and n2 represent the start and finish addresses of an area of memory that should not be written to. For example:

```
$STOP9: #8000 #8FFF MEM WR #0 #1FF MEM WR OR
```

This stop definition would cause the program to stop if it starts to write to the area from #8000 to #8FFF inclusive and #0 to #1FF inclusive.

LXOR Logical exclusive-or f1 f2 > f3

```
$LXOR: 0 > SWAP 0 > XOR
```

The word LXOR is different to XOR since it treats the two numbers on top of the stack as flags (0=false, 0<>TRUE). The logical XOR can be used to chain conditions, for example you might want a program to stop if either condition1 or condition2 were true but not if both are true, this would be written:

$STOP0: condition1 condition2 LXOR

PRINT prints the value on top ofihe stack n1 >

The following definition, with the associated piece of machine code, will print, in hexadecimal, the value on top of the stack in the top left hand corner of the screen. Assume the code is at either #ABE0 for Spectrum or #6978 for the Amstrad (it could be any piece of free RAM). The new definition will be called PRINT.

The machine code part:

**Amstrad:**

```
#6978 #CD #99 #69 CALL #6999 ; Call the DROPBC routine
#697B #3E #1E     LD   A,30
#697D #CD #5A #BB CALL #BB5A ; send a character 30 to the VDU
#6980 #78         LD   A,B
#6981 #85 #85 #69 CALL #6985 ; print most significant byte
#6984 #79         LD   A,C   ; print least significant byte
#6985 #F5         PUSH AF    ; Routine which prints the
#6986 #07         RLCA       ; contents of A in hexadecimal
#6987 #07         RLCA
#6988 #07         RLCA
#6989 #07         RLCA
#698A #CD #8E #69 CALL #698E
#698D #F1         POP  AF
#698E #E6 #0F     AND  #0F
#6990 #C6 #90     ADD  A,#90
#6992 #27         DAA
#6993 #CE #40     ADC  A,#40
#6995 #27         DAA
#6996 #C3 #5A #BB JP   #BB5A ; firmware’s print routine
#6999 #E9         JP   (HL)  ; HL holds the address of DROPBC
```

**Spectrum:**

```
#ABE0 #CD #01 #AC CALL #AC01 ; Call the DROPBC routine
#ABE3 #3E #16     LD   A,22
#ABE5 #D7         RST  #10   ; print an AT control character
#ABE6 #AF         XOR  A
```

```
#ABE7 #D7         RST  #10   ; followed by two 0's, puts print
#ABE8 #D7         RST  #10   ; position at 0,0
#ABE9 #78         LD   A,B
#ABEA #CD #EE #AB CALL #ABEE ; print most significant byte
#ABED #79         LD   A,C   ; print least significant byte
#ABEE #F5         PUSH AF    ; print the value of A
#ABEF #07         RLCA       ; in hexadecimal
#ABF0 #07         RLCA
#ABF1 #07         RLCA
#ABF2 #07         RLCA
#ABF3 #CD #F7 #AB CALL #ABF7
#ABF6 #F1         POP  AF
#ABF7 #E6 #0F     AND  #F
#ABF9 #C6 #90     ADD  A,#90
#ABFB #27         DAA
#ABFC #CE #40     ADC  A,#40
#ABFE #27         DAA
#ABFF #D7         RST  #10   ; print character using ROM routine
#AC00 #C9         RET
#AC01 #E9         JP   (HL)  ; HL contains address of DROPBC
```

(Spectrum - ensure before running the above program that IY has the value #5C3A or it will not work!)

The Analyser definition for the new word is:

Spectrum: `$PRINT #ABE0 CALL`
Amstrad: `$PRINT #6978 CALL`

The word PRINT will now print the number on top of the stack in hexadecimal in the top left hand corner of the screen. For example to monitor the contents of a particular memory location while slow running the following definition could be used:

`$STOP1 address @ PRINT 0`

Where 'address' is the address to be monitored and the 0 on the end ensures that the stop condition is never met and so the Analyser does not halt.

ALTBC get the value of BC' > n1
There are no words in the Analyser to read the values of any of the alternate registers, CALL can be used to get around this. ALTBC fetches the value of BC'. CALL can be used since on entry to the CALL's machine code, IX points to a table of register values which includes the alternate registers (see definition of the CALL word for a layout of this table).

The code to perform the ALTBC is given below:

**Amstrad:**

```
#6978 #DD #4E #10 LD   C,(IX+16) ; fetch the value of BC' from the table
#697B #DD #46 #11 LD   B,(IX+17)
#697E #D5         PUSH DE        ; DE holds the address of STKBC
#697F #C9         RET            ; exit via STKBC
```

**Spectrum:**

```
#ABEO #DD #4E #10 LD   C,(IX+16)
#ABE3 #DD #46 #11 LD   B,(IX+17)
#ABE6 #D5         PUSH DE
#ABE7 #C9         RET
```

The Analyser definition for the above will be:

Spectrum: `$ALTBC #ABE0 CALL`
Amstrad: `$ALTBC #6978 CALL`

The ALTBC word will now place the value of BC' on the stack and can be used just as though it were one of the other register words. To define words that get the value of other registers just read different parts of the register table.

# 10. Examples

For most real time applications it is not practical to slow run the entire program so a breakpoint structure has been provided that allows discrete sections of the code to be analysed while the remainder runs at full Z80 speed.

By no means do the examples here exhaust all of the facilities contained within the monitor/analyser but they do exhibit the workings of the system to some degree. The programs used on the following examples are simple because the examples are included to demonstrate the monitor/analyser in action as clearly as possible.

### *Example 1 - demonstrates SIOW and fast execution*

Load the low version of the monitor/analyser provided and enter it by typing:

```
Amstrad:     CALL 5927
Spectrum:    RANDOMIZE USR 25003
```

Now set the MEMory pointer so that the example program can be entered using the DATA command

```
Spectrum:    MEM=#ABE0
Amstrad:     MEM=#6978
```

Now enter the example program using the DATA command.

```
Source                          Amstrad               Spectrum

LOOP      LD B,5                #06,#05               #06,#05
          LD IX,WORD1           #DD,#21,#93,#69       #DD,#21,#FB,#AB
          CALL SEARCH           #CD,#83,#69           #CD,#EB,#AB
          JR LOOP               #18,#F5               #18,#F5
SEARCH    PUSH IX               #DD,#E5               #DD,#E5
          POP HL                #E1                   #E1
LOOP1     BIT 7,(HL)            #CB,#7E               #CB,#7E
          INC HL                #23                   #23
          JR Z,LOOP1            #28,#FB               #28,#FB
LOOP2     PUSH HL               #E5                   #E5
          POP IX                #DD,#E1               #DD,#E1
          DJNZ LOOP1            #10,#F6               #10,#F6
          RET                   #C9                   #C9
          DEFS 2                #00,#00               #00,#00
WORDI     DEFB “ONE”,#80        “ONE”,#80             “ONE”,#80
          DEFB “TWO”,#80        “TWO”,#80             “TWO”,#80
          DEFB “THREE”,#80      “THREE”,#80           “THREE”,#80
          DEFB “FOUR”,#80       “FOUR”,#80            “FOUR”,#80
          DEFB “FIVE”,#80       “FIVE”,#80            “FIVE” ,#80
          DEFB “SIX”,#80        “SIX”,#80             “SIX”,#80
```

Review the program using the LIST or DISS command to check that it is correct. As this program will be used again you may wish to save this code in case you have any mishaps, if so type:

```
Amstrad:     SAVE “EXAMP”,#6978,#69B8
Spectrum:    SAVE “EXAMP”,#ABE0,#AC20
```

Now set the program counter:

```
Spectrum:    PC=#ABE0
Amstrad:     PC=#6978
```

Now define two breakpoints:

Spectrum: `DEFBRK 1,16,#ABEE` – slow to fast type
`DEFBRK 2,3,#ABF3` – slow mode 3
Amstrad: `DEFBRK 1,16,#6986` – slow to fast type
`DEFBRK 2,3,#698B` – slow mode 3

Now type SLOW3 or JUMP.

The program will run under control of the monitor. The program contains a continuous loop. The program will run in slow mode 3 from LOOP 2 to LOOP 1, and at full speed from LOOP 1 to LOOP 2. Therefore, the display will only exhibit register values etc., for the section of code after LOOP 2 and before LOOP 1.

If left, this program would continue to run forever. Therefore, we must intervene. While the monitor is running in any slow mode it may be halted by typing:

```
Amstrad:     ESC
Spectrum:    SS & A
```

### *Example 2 - Using the Analyser to update the screen*

It is unusual to require the front panel to be updated on every instruction. The program typed in for example 1 will now be used to demonstrate how MEMory display may be updated less often under the control of the analyser. The analyser will be programmed so that the MEMory display will be updated only when the IX register changes value and the MEMory pointer will follow the value in IX whenever an update occurs.

To use the analyser a program space must be defined.

```
Spectrum:    PROG= 42000,42499
Amstrad:     PROG= 25000,25499
```

To use the TRACE facility we must also define a workspace for the trace.

```
Spectrum:    WORK= 42500,43000
Amstrad:     WORK= 25500,26000
```

Now reset the program counter to the start of the example program byt yping:

```
Spectrum:    PC=#ABE0
Amstrad:     PC=#6978
```

Bearing in mind that the faster execution is obtained in slow mode 0 define two breakpoints as follows:

```
Spectrum:    DEFBRK 1,16,#ABEE - slow to fast
             DEFBRK 2,0,#ABF3 - continue in slow mode 0
Amstrad:     DEFBRK 1,16,#6986 - slow to fast
             DEFBRK 2,0,#698B - continue in slow mode 0
```

Type ANALYSER ON.

Define two analyser STOP words as follows:

```
Spectrum:    $STOP0: IX MEM! 0
             $STOP1: #ABF9 @ IX <> ? SCR1 IX #ABF9 ! 0
Amstrad:     $STOP0: IX MEM! 0
             $STOP1: #6991 @ IX <> ? SCR1 IX #6991 ! 0
```

Both the STOP definitions above are terminated with a zero for the same reason. The analyser expects to find a single flag on the stack when a definition has been evaluated. If the zero's are omitted from the above definitions they will leave the stack empty. The value zero has been chosen so that the value left on the stack, i.e. zero is false, therefore, the analyser will not stop the program.

Now set the monitor into action by typing either SLOW 0 or JUMP. Observe the following effects:

a) The program runs fast (full Z80 speed) from LOOP1 to LOOP2.

b) The program runs under the control of the monitor in slow mode 0 from LOOP2 to LOOP1

c) Whenever IX is assigned a new value the MEMory display is updated.

d) Despite the definition of two analyser STOP words, the analyser does not stop because both STOP words evaluate to '0' (see above).

e) Each time STOP0 is evaluated (after every instruction during the SLOW 0 phase) the value of IX is copied into the MEMory pointer.

f) Each time STOP1 is evaluated the old value of IX, stored at either #AB99 on the SPECTRUM or at #6991 on the Amstrad, is compared with the program value and if they differ the memory display is updated then the program value of IX is stored so it becomes the old value of IX for the next time that STOP1 is evaluated.

### *Example 3 - Debugging*

The following section shows how the Analyser can be used to track down bugs, it assumes that the area of memory around

```
   #ABE0 on the Spectrum
or #6978 on the Amstrad
```

is free. To demonstrate the Analyser we need a piece of machine code with an error in it.

Specification: The program given below SHOULD add up the ten numbers held in the bytes from #ABE0 to #ABE9 on the Spectrum (#6978 to #6981 on the Amstrad) and place the total in the last element of the table:

**Spectrum**

```
#ABEA LD   B,10      ; Length of table is 10 bytes
#ABEC LD   HL,#ABE0  ; HL points to the start of the table
#ABEF XOR  A         ; Keep the total in A
#ABF0 ADD  A, (HL)   ; Start of the loop which adds up the ten values
#ABF1 INC  HL        ; Go on to next element in the table
#ABF2 DJNZ #ABF0     ; Repeat loop ten times
#ABF4 LD   (HL),A    ; Store result in last element
#ABF5 RET
```

**Amstrad**

```
#6982 LD   B,10      ; Length of table is 10 bytes
#6984 LD   HL,#6978  ; HL points to the start of the table
#6987 XOR  A         ; Keep the total in A
#6988 ADD  A, (HL)   ; Start of the loop which adds up the ten values
#6989 INC  HL        ; Go on to next element in the table
#698A DJNZ #6988     ; Repeat loop ten times
#698C LD   (HL),A    ; Store result in last element
#698D RET
```

To enter this program in memory type the following:

**Spectrum** **Amstrad**

```
MEM=#ABEA                      MEM=#6982
DATA 6,#A,#21,#E0,#AB,#AF      DATA 6,#A,#21,#78,#69,#AF
DATA #86,#23,#10,#FC,#77,#C9   DATA #86,#23,#10,#FC,#77,#C9
```

The program requires ten numbers in the ten locations from #ABE0 on the Spectrum (from #6978 on the Amstrad). Place some data there by typing:

**Spectrum** **Amstrad**

```
MEM=#ABE0                      MEM=#6978
DATA 1,2,3,4,5                 DATA 1,2,3,4,5
DATA 6,7,8,9,0                 DATA 6,7,8,9,0
```

To make sure that we have the program and data in the memory we can check it by listing it, but first define the memory used as a data area using the command:

**Spectrum** **Amstrad**

```
DB 1 #ABE0 #ABE9               DB 1 #6978 #6981
```

and use the following to list it:

**Spectrum** **Amstrad**

```
LIST #ABE0,#ABF5               LIST #6978,#698D
```

The program can now be run using:

**Spectrum:** `CALL #ABEA`
**Amstrad:** `CALL #6982`

The answer 45 (#2D) should have been placed in the last location of the table, either #ABE9 or #6981 (the location originally containing 0). Examining the location shows that it still holds the value 0, a bug! If we list the program using LIST we also notice that the program has become corrupted, the first byte has changed from #6 to #2D. Assuming we don't know immediately why the program isn't working we can use the Analyser to find out why the total is not being placed in the correct place in the data area and why the first byte of the program is being corrupted.

First define some program space:

```
    Spectrum                    Amstrad
    PROG=42000,42200            PROG=25000,25200
```

We could start out by trying to find out why the byte at #ABE9/#6981 (Spectrum/Amstrad) was corrupted.

In this case there is only one instruction which writes to memory, the instruction at #ABF4/#698C. Using the Analyser here illustrates how a similar bug in a much longer program could be found, in a longer program memory would be written to in many places and only one would be causing the bug.

The following definition will set up the Analyser to detect when this byte is being written to:

```
    Spectrum                    Amstrad
    $STOP1 ADDR #ABEA = WR &    $STOP1 ADDR #6982 = WR &
```

Before we can run the program again the first byte of the program should be returned to its original value and the PC should be reset.

```
    Spectrum                    Amstrad
    MEM=#ABEA                   MEM=#6982
    DATA 6                      DATA 6
    PC=#ABEA                    PC=#6982
```

Now run the program again,

```
    SLOW 0 (remember that the Analyser can only be used when slow running)
```

The program will stop with “Press any key” as usual but in addition gives the error message “Stop number #1” indicating that STOP 1 has been fulfilled. The instruction executed will be the LD (HL),A instruction located at the end of the addition program. This will tell us that the answer is being placed in the wrong place, i.e. #ABEA instead of #ABE9 on the Spectrum, #6982 instead of #6981 on the Amstrad

To see exactly what is happening we can single step the program but since the program loops around several times this will waste time, especially since we can assume that the addition part of the program is working. Suppose we want to single step from when the last byte of the table is read, up to there we can slow run the program.

The following definition sets up STOP1 to detect when the last byte in the table is read (#ABE9/#6981 )

```
    Spectrum                    Amstrad
    CLEAR                       CLEAR
    $STOP1 ADDR #ABE9 = RD &    $STOP1 ADDR #6981 = RD &
```

Now run the program again (remember to correct the program first):

```
    Spectrum                    Amstrad
    MEM=#ABEA                   MEM=#6982
    DATA6                       DATA6
    PC=#ABEA                    PC=#6982
    SLOW 0                      SLOW 0
```

The program will now stop at the ADD A,(HL) instruction at #ABF0/#6988 with HL=#ABE9/ #6981. We can now use single stepping to investigate what happens from here, what should happen is that the total (held in A) should be loaded into #ABE9/#6981 (the current value of HL). The next instruction is an INC HL, single step this (use CTRL S or symbol shift D). Single step the DJNZ instruction and you will reach the LD (HL),A. Note that HL now has the value #ABEA/#6982 and not #ABE9/#6981, this is because of the INC HL just executed, the error has been found! We cannot remove the INC HL because it is needed in the loop, instead a DEC HL is needed between the DJNZ instruction and the LD (HL),A. To insert this extra instruction use:

```
    Spectrum                    Amstrad
    MEM=#ABF4                   MEM=#698C
    DATA #2B,#77,#C9            DATA #2B,#77,#C9
```

Now run the program:

| Spectrum | Amstrad |
|---|---|
| `CALL #ABEA` | `CALL #6982` |

afterwards, examine the byte at #ABE9/#6981, it should now contain the value 45 (#2D), the sum of 1, 2, 3, 4, 5, 6, 7, 8, 9 and 0. Type:

| Spectrum | Amstrad |
|---|---|
| `LIST #ABE0,#ABF6` | `LIST#6978,#698E` |

and you will see that the program has not been corrupted. Running the program again will give the result 90 (#5A) since the sum is now that of 1, 2, 3, 4, 5, 6, 7, 8, 9, 45.

### *Example 4 - Stack Checker*

The following example is included firstly because it makes fairly comprehensive use of the analyser's predefined words, and secondly because it performs a test that some programmers may find useful at some stage.

Under normal circumstances a subroutine is exited with the stack in the same state as it was when the routine was entered. A common source of error (and one which is often difficult to trace) occurs when routines are CALLed recursively with conditional RETurns. The definition in this example will cause a screen update and pause, if a RETurn (conditional or otherwise) is encountered, with the stack in a different state to that which was produced by the most recent successful CALL. The definition accommodates nested CALLs and checks to see whether a conditional CALL was actually executed. All this amounts to is a definition to check that a subroutine contains matching sets of PUSHes and POPs.

This example assumes that the low version of the monitor is in use and consists of the following word definitions.

| **Definition** | **Description** |
|---|---|
| `$ILAST #AC40` (Spectrum)<br>`$ILAST #6A00` (Amstrad) | ILAST is a constant which evaluates to the address of the temporary storage of the last opcode executed. ILAST is therefore treated as a variable of sorts. |
| `$SP1 #AC41` (Spectrum)<br>`$SP1 #6A01` (Amstrad) | SP1 is a constant which evaluates to the address of the temporary storage of the stack pointer when the last opcode was executed. SP1 can also be thought of as a variable. |
| `$SPOINTER #AC43` (Spectrum)<br>`$SPOINTER #6A03` (Amstrad) | SPOINTER is a constant which evaluates to the address of the pointer into the table of stack pointer values stored. Each time a CALL is successfully executed the SP (after the call) is added to the table and the value in SPOINTER is incremented by 2. SPOINTER can also be thought of as a variable. Each time a RETurn is encountered the value in SPOINTER is decremented by 2 to point to the previous SP value and thus accommodates nesting. |
| `$UPDATE IF 1 ? SCR1 1 ? SCR2 1 ?PAUSE` | UPDATE will test the flag on the top of the stack and if a true flag is found, will update the register display and memory display, then wail to a key before continuing. |
| `$QC ILAST C@ # CD =` | Leave a true flag if the last instruction was a CALL, or a false flag if otherwise. |
| `$QCC ILAST C@ #C7 AND #C4 =` | Leave a true flag is the last instruction was a conditional CALL, or a false flag if otherwise. The opcode is bitwise ANDed with #C7 (%11000111) and if the result is #C4 (%11000100) then it was one of the 8 conditional CALLs. |
| `$QRST ILAST C@ #C7 AND #C7 =` | Leave a true flag if the last instruction was a restart (RST), or false flag if otherwise. The |

opcode is bitwise ANDed with #C7 (%11000111) and if the result is #C7 (%11000111) then it was one of the B restarts.

```
$QCALL QC QCC QRST OR OR SP1 @ SP <>
&
```

Test to see whether: a CALL, OR a conditional CALL, OR a restart, has been executed, AND the stack has been changed (i.e. a conditional CALL was successfully executed). This will therefore leave a true flag if the last instruction was a successful CALL or a false flag if it was not.

```
$QRET PC C@ #C9 =
```

Leave a true flag if the current instruction (about to be executed) is a RETurn, or a false flag if otherwise.

```
$QRETC PC C@ #C7 AND #C0 =
```

Leave a true flag if the current instruction (about to be executed) is a conditional RETurn or a false flag if otherwise. The opcode is bitwise ANDed with #C7 (%11000111) and if the result is #C0 then it is one of the 8 conditional RETurns.

```
$QRETURN QRET QRETC OR
```

Test to see whether the current instruction is a RETurn or a conditional RETurn and leave a true flag if it is either or a false flag if it is neither.

```
$SPS1 SPOINTER @DUP SP SWAP !
```

The first half of the definition for SPS. The contents of SPOINTER are placed on the stack (pointer into the stack pointer table), this value is copied onto the top of the stack and SP is placed in the table at the specified position. The other copy of the contents of SPOINTER is left on the stack for use in the second half of SPS.

```
$SPS SPS1 2 + SPOINTER !
```

The value left on the stack by SPS1 (pointer into the table) is incremented by two (to point to the next vacant space in the table) and the new pointer is put back into SPOINTER.

```
$SPF SPOINTER @ 2 - DUP SPOINTER ! @
```

Fetch the pointer into the table on to the stack, decrement it by two (points to the previous value in the table), copy it onto the top of the stack, store one of them back into SPOINTER and fetch the SP value from the address pointed to by the other.

```
$ISP PC C@ ILAST C ! SP SP1 !
```

Put the current opcode into the last instruction location and the current SP into the last SP location. Before executing the next instruction, these will be used as the ‘last instruction’ and ‘last stack pointer’ values.

```
$CHECK QCALL ISP IF SPS
```

Tests to see if a successful CALL has been made and stacks a flag, updates ILAST and SP1 and then, if a successful CALL was made, update the table.

```
$RUN CHECK QRETURN IF SPF SP <>
UPDATE
```

This is a full definition which checks to see if a CALL/RETURN sequence is interrupted by a net stack change.

```
$STOP0 RUN 0
```

Defines the STOP condition. The 0 after the RUN ensures that the STOP condition is never actually met.

## Using the Example Definition

To begin with, a sample program is needed. The following program has no practical significance but illustrates the use of the stack checker quite clearly. Since we are using the low version of the monitor we will use memory from #ABE0 on the Spectrum or from #6978 on the Amstrad.

**Spectrum:**

```
#ABE0 #3E #20                LD   A,#20
#ABE2 #CD #E7 #AB  LINE 2:   CALL #ABE7
#ABE5 #18 #08                JR   #ABEF
#ABE7 #E5          LINE 4:   PUSH HL
#ABE8 #3D                    DEC  A
#ABE9 #C4 #E2 #AB  LINE 6:   CALL NZ,#ABE2
#ABEC #C8          LINE 7:   RET  Z
#ABED #E1                    POP  HL
#ABEE #C9                    RET
#ABEF #0                     NOP
```

**Amstrad:**

```
#6978 #3E #20                LD   A,#20
#697A #CD #7F #69  LINE 2:   CALL #697F
#697D #18 #08                JR   #6987
#697F #E5          LINE 4:   PUSH HL
#6980 #3D                    DEC  A
#6981 #C4#7A#69    LINE 6:   CALL NZ,#697A
#6984 #C8          LINE 7:   RET  Z
#6985 #E1                    POP  HL
#6986 #C9                    RET
#6987 #0                     NOP
```

The program begins by setting A to hold a counter with a value of 32. The CALL at LINE 2, CALLs LINE 4 which stacks HL, decrements the counter, and if non-zero, CALLs LINE 2 to loop round again. This cycle continues until A holds zero. At this point the CALL at LINE 6 is not executed and control drops to the RET Z at LINE 7. At this point the analyser checks the stack to see what position it was in when the last CALL (from LINE 2) was executed. In this example a PUSH HL has occurred and so the stack has been changed and the analyser updates the screen and waits for a key. Note that even if the zero flag were not set at the RET Z instruction, the analyser will STOP because it knows that if the zero flag had been set an error was likely and this may therefore be a potential error which needs to be drawn to the user's attention. The STOP condition could be redefined as an exercise, so that a STOP only occurs if the RET is successfully executed.

To run the example itself:

(i) Spectrum: `PROG=42000,42200`
Amstrad: `PROG=25000,25200`

to reserve program space for the definitions.

(ii) Type in each of the definitions previously listed.

(iii) Use the "." command to enter the sample program in the same manner as the previous examples.

(iv) Type: `EVAL 0 ILAST C! <ENTER>`

to initialise ILAST.

(v) Type: `EVAL SP SP1 ! <ENTER>`

to initialise SP1.

(vi) Spectrum: `EVAL #AC45 SPOINTER ! <ENTER>`
Amstrad: `EVAL #6A05 SPOINTER ! <ENTER>`

to initialise SPOINTER.

(vii) Spectrum: `PC=#ABE0 <ENTER>`
Amstrad: `PC=#6978 <ENTER>`

to set the PC to the start of the sample program.

(viii) Type: `ANALYSER ON <ENTER>`

to ensure the analyser is activated.

(ix) Type: `SLOW 0 <ENTER>`

to set the program running under the scrutiny of the analyser.

## *Example 5 - Virtual Screen Usage*

**Spectrum:**

Type MEM=#ABE0 and enter the following program data:

```
DATA #21,0,#40              LD HL,#4000
DATA #36,#66          LOOP: LD (HL),#66
DATA #23                    INC HL
DATA #7C                    LD A,H
DATA #FE,#58                CP #58
DATA #20,#F8                JR NZ,LOOP
DATA #C9                    RET
```

We will now define the memory area for the virtual screen by typing:

`SCRN= #E000 <ENTER>` (There must be 6912 free bytes from #E000 onwards)

and then type

```
SCRN ON <ENTER>
```

To verify that a virtual screen has been set up type:

```
SCRN <ENTER>
```

The whole screen should go blank; press any key to continue. Now type:

```
PC=#ABE0 <ENTER>
CALL <ENTER>
```

The screen should go blank, then fill up with a candy stripe pattern. Finally the message "press a key" will appear at the foot of the screen. Press any key and the monitor screen will be recreated. Now type

```
SCRN <ENTER>
```

The screen will now display the contents of the virtual screen memory. Press any key to continue.

**Amstrad:**

Type MEM=#6978 and enter the following program data:

```
DATA #21,0,#C0              LD HL,#C000
DATA #36,#55          LOOP: LD(HL),#55
DATA #23                    INC HL
DATA #AF                    XOR A
DATA #B4                    OR H
DATA #20,#F9                JR NZ,LOOP
DATA #C9                    RET
```

We will now define the memory area for the virtual screen by typing:

```
SCRN=#7000 <ENTER>
```

and then:

```
SCRN ON <ENTER>
```

We will define a screen window 6 characters wide and 8 characters high, in the top left corner of the screen by typing:

```
COL=0 <ENTER>
ROW=0 <ENTER>
HGT=8 <ENTER>
LEN=6 <ENTER>
```

Press CTRL and V together to invert the window to verify that it is correct. Now type:

```
SCRCLR <ENTER>
```

This command clears the memory from #7000 to #72FF, the amount required for saving the current window. Type

```
DISS #6978 <ENTER>
```

to check the program. Now type

```
SCRN <ENTER>
```

The window will go blank. Press any key and the window will be restored. Now type

```
PC=#6978 <ENTER>
CALL <ENTER>
```

The screen should go blank then fill up with a candy stripe pattern. The message “press a key” will appear at the bottom of the screen. Press any key and the monitor screen will be recreated. Now type

```
SCRN <ENTER>
```

The contents of the virtual screen memory will be displayed in the window until any other key is pressed then the monitor screen will be replaced.

# Appendix A – Key Summary

**Abbreviations:**

S = Shift
CS = Caps shift
SS = Symbol shift

| **Amstrad** | **Spectrum** | |
|---|---|---|
| ← | CS & 5 | Command line cursor left. |
| → | CS & 8 | Command line cursor right. |
| DEL | CS & 0 | Delete character left of cursor. |
| CLR | SS & 0 | Delete character at cursor. |
| COPY | CS & 1 | Insert a space, move characters from cursor to the right. |
| CTRL&L | SS & A | Clear the command input line. |
| ESC | SS & A (STOP) | Used to exit from SLOW running, LISTIng, DISSassembling and DUMPing. |
| ENTER | ENTER | Execute the command in the command input line, irrespective of cursor position. |
| S & ← | SS & Q | Decrement the MEMory pointer. |
| S & → | SS & E | Increment the MEMory pointer. |
| S & ↑ | SS & W | Subtract 8 from MEMory pointer (up 1 line of MEMory dump). |
| S & ↓ | SS & S | Add 8 to MEMory pointer(down 1 line of MEMory dump). |
| S & COPY | CS & 9 | Advance MEMory pointer to next instruction. |
| CTRL & N | SS & Y | Equivalent to “NEXT” when searching. |
| CTRL & I | CS & 6 | Increment the PC register. |
| CTRL & D | CS & 7 | Decrement the PC register. |
| CTRL & K | SS & G | Advance PC to the next instruction. |
| CTRL & S | SS & D | Single step, i.e. execute the instruction pointed to by PC, advance PC to the next instruction. |
| CTRL & E | SS & F | Single step the instruction pointed to by PC except CALL’s or RST’s which are executed at normal speed, advance PC to the next instruction. |
| CTRL & V | - | Highlight the virtual screen window (AMSTRAD only) |

## Appendix B – Monitor Command Summary

| Command | Parameter | Action |
|---|---|---|
| A= | <byte> | Assigns value in <byte> to the A register. |
| AF= | <word> | Assigns value in <word> to the AF register pair. |
| ANALYSER | <flag> | Switch the analyser on or off according to <flag>. <flag> = 0 turns analyser off. |
| B= | <byte> | Assigns value in <byte> to the B register. |
| BC= | <word> | Assigns value in <word> to the BC register pair. |
| (BC) | <byte> | Places the value <byte> in memory at the address in BC. |
| BREAK | <brk number>, <flag>, <addr> | Define breakpoint number <brk number>. If the address <addr> is met, the monitor will resume control. The breakpoint may be on or off, set by <flag>. |
| BRK | <brk number>, <flag> | The breakpoint <brk number> is switched on or off according to the value of <flag>. |
| BUFFER= | <address> | The 2048 bytes of memory starting at <address> will be used as a buffer during FLIST or during the load of ASCII files (Amstrad only). |
| C= | <byte> | Assigns value in <byte> to the C register. |
| CALL | | A call to the address in PC is made. Code executes at normal Z80 speed but a return address, to the monitor, is placed on the stack. |
| CALL | <addr> | As CALL but control is passed to the machine code at <addr> rather than PC. |
| CAT | | List the tape/disc/microdrive directory to the screen. |
| CHECK | <start1>, <finish1 >, <start2> | Compares byte for byte, the contents of memory, lying between <start1> and <finish1> inclusive, with the area starting at address <start2>. |
| CLEAR | | Resets program space and switches analyser on. |
| CTRL & N<br>SS &Y | | Shorthand for NEXT. |
| D= | <byte> | Assigns value in <byte> to the D register. |
| DATA | <byte list> | The byte list is placed into memory. |
| DB | | Lists the current selection of data areas. |
| DB | <db number> | Deletes the specified data area <db number>. |
| DB | <db number>, <start>, <finish> | Defines a data area. |
| DE= | <word> | Assigns value in <word> to the DE register pair. |
| (DE)= | <byte> | Places the value <byte> in memory at the |

| Command | Parameter | Action |
|---|---|---|
| | | address in DE. |
| DEFBRK | <brk number>, <type>, <addr> | Define SPECIAL breakpoint number <brk number>. If the address <addr> is met, the monitor will act according to the value of <type>. |
| DEFBRK | <brk number>, <type>, <addr>, <count> | Define SPECIAL breakpoint number <brk number>. If the address <addr> is encountered then <count> is decremented. When <count> reaches zero the monitor will act according to the value of <type>. |
| DEFLOAD | "<string>" | Load the analyser definitions from the file named "<string>". |
| DEFSAVE | "<string>" | Save the current analyser definitions to a file named "<string>". |
| DELETE | <brk number> | Removes the breakpoint <brk number> definition. |
| DI | | Disable interrupts. |
| DISC | | Select disc as the current input/output device. |
| DISS | | Disassemble the contents of memory, starting at address MEM, to the top left hand window of the screen until ESC/SS & A is pressed. |
| DISS | <start> | As "DISS" but starting at the address <start> rather than MEM until ESC/SS &A is pressed. |
| DISS | <start>, <finish> | As "DISS <start>" until either the address <finish> is reached or until ESC/SS&A is pressed. |
| DRIVE | <letter> | Direct firmware to required drive (Amstrad only). |
| DUMP | | Output contents of memory, starting at address MEM, to the SCREEN, until ESC/SS &A is pressed. |
| DUMP | <start> | Output contents of memory, starting at address <start>, to the SCREEN, until ESC/SS & A is pressed. |
| DUMP | <start>, <finish> | Output contents of memory, starting at address <start>, to the SCREEN, until either address <finish> is reached, or until ESC/SS & A is pressed. |
| E= | <byte> | Assigns value in <byte> to the E register. |
| EDIT | <word name> | Edit the specified analyser definition. |
| EI | | Enable interrupts. |
| ERA | "<string>" | Delete from disc/microdrive, the file named "<string>" (not tape versions). |
| EVAL | <definition> | Evaluate the <definition> and print state of analyser stack. |
| EX AF | | The alternative "AF" register is displayed. Swap AF for AF' or vice versa. |
| EXIT | | Return from the monitor to the calling routine. |

| Command | Parameter | Action |
|---|---|---|
| EXX | | The alternative register set is displayed. Swap BC, DE & HL for BC', DE' and HL' or vice versa. |
| F= | <byte> | Assigns value in <byte> to the F register. |
| FILL | <start>, <finish>, <byte> | The value <byte> is placed in the memory area ranging from <start> to <finish> inclusive. |
| FLIST | <filename>, <start>, <finish> | Disassembles the contents of memory lying between <start> and <finish> inclusive to tape, disc or microdrive. |
| H= | <byte> | Assigns value in <byte> to the H register. |
| HL= | <word> | Assigns value in <word> to the HL register pair. |
| (HL)= | <byte> | Places the value <byte> in memory at the address in HL. |
| IX= | <word> | Assigns value in <word> to the IX register pair. |
| (IX)= | <byte> | Places the value <byte> in memory at the address in IX. |
| IY= | <word> | Assigns value in <word> to the IY register pair. |
| (IY)= | <byte> | Places the value <byte> in memory at the address in IY. |
| JUMP | | Monitor passes control to the machine code starting at the address in PC. Code will execute at normal Z80 speed. Unless "breakpoints" are encountered the monitor will not be re-entered. |
| JUMP | <addr> | As JUMP, but control is passed to the machine code at <addr> rather than PC. |
| L= | <byte> | Assigns value in <byte> to the L register. |
| LBRK | | Lists the 8 breakpoint definitions. |
| LDEF | | List the user's analyser definitions. |
| LDUMP | | As "DUMP" but output to printer (duplicate to screen is controlled by OPTION 2). |
| LDUMP | <start> | As "DUMP <start>" but output to printer. (Duplicate to screen is controlled by OPTION 2). |
| LDUMP | <start>, <finish> | As "DUMP <start>,<finish>" but output to printer. (Duplicate to screen is control led by OPTION 2). |
| LIST | | As "DISS" but uses the whole screen. |
| LIST | <start> | As "DISS <start>" but uses the whole screen. |
| LIST | <start>, <finish> | As "DISS <start>, <finish>" but uses the whole screen. |
| LLIST | | As "LIST" but output to the printer. |
| LLIST | <start> | As "LIST <start> " but outputto the printer. |

| Command | Parameter | Action |
|---|---|---|
| LLIST | <start>,<finish> | As "LIST <start>, <finish>" but output to the printer. |
| LOAD | <filename> | Load the specified file from tape or microdrive to the address found in the header block (Spectrum only). |
| LOAD | <filename>, <addr> | Load the specified file from tape or microdrive to the address <addr> (Spectrum only). |
| LOAD | <filename>, <addr> | The specified file is loaded into memory starting at address <addr>. PC may be set to the execution address (Amstrad only). |
| LTRACE | | The contents of the Trace memory will be listed to the printer. |
| LTRACE | <number> | The specified number of instructions are listed to the printer. The most recent values are used. |
| MAP | | Display the current memory usage. |
| MDRV | | Select microdrive number 1 as the current input/output device. |
| MDRV | <drive> | Select the indicated microdrive as the current input/output device. |
| MEM= | <addr> | The memory pointer is set to the value <addr> and updates MEMory dump accordingly. |
| MODE | <mode number>, | Selects screen mode according to <mode number>, 1=40 column, 2=80 column (Amstrad only). |
| MOVE | <start1>, <finish1>, <start2> | Copy the contents of memory area lying between <start1> and <finish1> inclusive, to the area starting at address <start2>. |
| NEXT | | After a match is found the SEARCH is continued. |
| OPTION | <option number>, <flag> | The various options <option number> may be switched on or off, depending on the value in <flag>. |
| | 1, <flag> | ON = decimal<br>OFF = hexadecimal |
| | 2, <flag> | ON = printer only<br>OFF = printer and screen |
| | 3, <flag> | ON = FLIST to file and screen<br>OFF = FLIST to file only |
| | 4, <flag> | ON = slow run ROM calls and jumps<br>OFF = fast execute ROM calls and jumps |
| | 5, <flag> | ON = FLIST without labels<br>OFF = FLIST with labels |
| | 6, <flag> | ON = maintain screen during JUMP, CALL or SLOW<br>OFF = clear screen before JUMP, CALL or SLOW |
| | 7, <flag> | ON = printer output to KEMPSTON Centronics Interface (Spectrum only) |

| Command | Parameter | Action |
|---|---|---|
| | | OFF = printer output to ZX printer |
| | 8, <flag> | ON = carriage returns only<br>OFF = carriage returns are followed by line feeds |
| (PC)= | <byte> | Place the value <byte> in memory at the address in PC. |
| PDEF | | List the user's analyser definitions to the printer. |
| POP | | The SP register (stack pointer) is incremented twice. |
| PROG= | <addrl >, <addr2> | Define the memory between <addr1> and <addr2> as the analyser program space. |
| PUSH | <word> | The value <word> is placed on the stack and the stack pointer is decremented twice. |
| ROM | <flag> | Switches either the main ROM or the shadow ROM on depending on the state of <flag> (Spectrum only). |
| ROM | <flag1>, <flag2> | Switch the lower ROM either on or off, according to the state of <flag 1>. Switch the currently selected upper ROM either on or off, according to the state of <flag2> (Amstrad only). |
| ROM | <flag1>, <flag2>, <rom number> | As "ROM<flagl>,<flag2>" but alter the currently selected upper ROM to <rom number> (Amstrad only). |
| SAVE | <filename>, <start>, <finish> | The contents of memory lying between <start> and <finish> inclusive is saved, to tape, disc or microdrive with the specified <filename>. |
| SAVE | <filename>, <start>, <finish>, <exec> | As "SAVE <filename>, <start>, <finish>" but an execution address <exec> is saved also, see LOAD above (Amstrad only). |
| SCRCLR | | Clear the virtual screen memory (Amstrad only). |
| SCRN | | View the contents of the virtual screen. |
| SCRN | <flag> | Switch the virtual screen on or off according to <flag>. |
| SCRN= | <addr> | define the memory starting at <addr> to be virtual screen. |
| SEARCH | <start>, <finish>, <byte list> | The memory area defined by the address <start> and <finish> inclusive, is searched for the <byte list>. |
| SLOW | <slow mode number> | Start a program slow running with update of monitor display according to <slow mode number> until ESC/SS & A is pressed or a breakpoint is reached. |
| SP= | <word> | Assigns value in <word> to the SP register pair. |
| TAPE | | Select tape as the current input/output device. |

| Command | Parameter | Action |
|---|---|---|
| TRACE | | The contents of the TRACE memory will be displayed using the whole screen. |
| TRACE | <number> | The specified number of instructions are listed. The most recent values are used. |
| WORD | <word name>, <definition> | Define an analyser word. |
| WORK= | <start>, <finish> | Sets a side a workspace for FLIST. |
| . | <byte list> | A shorthand version of DATA. |
| ? | <word> | Convert and display the value <word> in the 4 bases, binary, octal, decimal and hexadecimal. |
| S | <word name>, <definition> | Define an analyser word. |

# Appendix C – Spectrum 128K Extensions

## *Introduction*

The 128k version of the monitor maintains all the features of the standard Spectrum monitor, although the syntax of some of the commands is extended to accommodate the RAM paging facilities.

The extended monitor may reside in any fixed RAM page and this means that to all intents and purposes, once loaded, it can be treated as a ROM, i.e. the full Z80 address space can be utilised. The assembler also sits in a fixed RAM page and so it is possible to have the assembler and monitor co-resident for quick and easy program development.

## *Tape Map*

On Tape 2 you will Genius package you will find the following files:

AMSTRAD:

| | | |
|---|---|---|
| Side A | MON | A BASIC loader. |
| | MON OBJ | Monitor object file. |
| Side B | BACKUP of Side A. | |

## *1. Opertating Instructions*

**Tape:** To load the monitor from tape, use the up and down arrow keys to select the 'TAPE LOADER' option, and press ENTER. The monitor will load, then the question "Which Page?" will appear. Enter a page number 0-7. The monitor will execute.

**Microdrive:** If you have produced, and wish to load, a microdrive version of the monitor you should use the up and down arrow keys to select the '128k BASIC' option and press ENTER. Having entered BASIC the monitor can be loaded using:

```
LOAD *"m";1;"MON128"
```

**NOTE:** Although the monitor can run in any RAM page, some pages have specific uses e.g. 5 and 7 which are screen contended. Realistically the users choice is restricted to 0, 1, 2, 3, 4, 6 and possibly 7 if one screen is sufficient.

## *2. Screen Layout*

The screen layout is basically the same as that employed by the standard Spectrum version but the second set of system flags, displayed to the right of the Z80 flags, are all used. There are five flags displayed, each with its status displayed directly beneath it. The flags have the following options and significance.

`R:` This is the ROM status and indicates which of the 3 ROMs is currently paged in. The ROMs are indicated by D (128k ROM), S (Spectrum ROM) and I (interface 1 ROM).

`M:` This indicates the RAM page that the MEM pointer is assuming to be paged into the Z80's address space at C000 hex. Values are in the range 0 to 7.

`P:` This indicates the RAM page that the PC is assuming to be paged into the Z80's address space at C000 hex. Values are in the range 0 to 7. **NOTE:** This means that MEM and PC may point to the same physical address, but display different instructions.

`V:` This indicates which screen the Spectrum is using for its video memory. Screen 0 or 1 is indicated by a '0' or '1' respectively.

`I:` This indicates the interrupt status of the user's program. '0' indicates that the user's program has interrupts disabled and a '1' indicates interrupts enabled.

## *3. 128K Monitor Command Syntax*

The following commands require revised syntax:

`ROM <letter>` e.g. `ROM D`

Page in the selected ROM `D` = 128k ROM

S = Spectrum ROM
I = Interface 1 ROM (shadow ROM)

```
RAM <number>                          e.g.   RAM 7
```

Page in the selected RAM page. There are 8 RAM pages in physical memory and each of these can be paged into the Z80 address space of address C000-FFFF hex. In fact RAM pages 2 and 5 always occupy Z80 address space anyway and so there is little point in executing RAM 2 or RAM 5. The following is a summary:

| **Physical Memory** | | **Z80 Memory** | |
|---|---|---|---|
| Page 7 | (screen 1) | Page 0-7 | C000-FFFF |
| Page 6 | - | Page 2 | 8000-BFFF |
| Page 5 | (screen 0) | Page 5 | 4000-7FFF |
| Page 4 | - | | |
| Page 3 | - | | |
| Page 2 | - | | |
| Page 1 | - | | |
| Page 0 | - | | |

```
BREAK <brknumber>,<flag>,<addr>,<page>     e.g.   BREAK,2,ON,#C070,7
```

Carries out the same function as the BREAK command described in Section 7.6 of the main text but sets the break point in the appropriate RAM page. Note that <page> will be irrelevant if <addr> is not in the range C000-FFFF hex, but should still be specified.

```
DEFBRK <brk number>,<flag>,<addr>,    e.g.   DEFBRK 4,0,#D000,6
                   <page>[,<coumt>]   or     DEFBRK 4,0,#D000,6,10
```

Carries out the same function as the DEFBRK command described in Section 7.6 of the main text but sets the break point in the appropriate RAM page. Note that <page> will be irrelevant if <addr> is not in the range C000-FFFF hex, but should still be specified.

```
CALL <addr>[,<page>]                  e.g.   CALL #D000 or CALL #D000,7
```

Carries out the same functions as the CALL command described in Section 5 of the main text. The second parameter is optional and, if provided, will select the indicated RAM page before executing the CALL.

```
CHECK <start1>,<finish1>,<start2>     e.g.   CHECK #B000,#B7FF,#9000
                          [,<page>]   or     CHECK #C000,#D000,#C000,6
```

This command verifies that two blocks of memory are identical. One of the two blocks is defined by <start1>, <finish1> and will lie in the MEMory page. The second block will be defined by <start2>. If the optional page parameter is entered the second block will lie in that <page>, otherwise the second block will also lie in the MEMory page.

```
JUMP <addr>[,<page>]                  e.g.   JUMP #D000
                                      or     JUMP #D000,7
```

Carries out the same function as the JUMP command described in Section 5 of the main text. The second parameter is optional and, if provided, will select the indicated RAM page before executing the JUMP.

```
MEM=<addr>[,<page>]                   e.g.   MEM=#D000
                                      or     MEM=#D000,7
```

Carries out the same function as the MEM= command described in Section 5 of the main text. The second parameter is optional and, if provided, will select the indicated RAM page before updating the display.

```
MEMP=<page>                           e.g.   MEMP=7
```

Select the indicated RAM page without changing the memory pointer address.

```
SCRN=<addr>[,<page>]                  e.g.   SCRN=#D000,0
```

Carries out the same function as the SCRN= command described in Section 5 of the main text. The second parameter is optional and, if provided, will tell the system to select the indicated RAM page during virtual screen operations.

```
DB <db number>,<start>,<finish>,<page>
```

A data area may be defined in any page.

```
MAP
```

This command, like the standard Spectrum, allows the. user to view the current state of memory usage within the monitor. The data is displayed in the top left area of the screen and has the following appearance:

|  |  | START | END | P |  |
|---|---|---|---|---|---|
| Line 1 | GMON | #61A8 | #6728 | 1 | Monitor/analyser code |
| Line 2 |  | & | #C000 |  | #FC36   1 |
| Line 3 | WORK | #FE1C | #FFFF | 1 | Workspace |
| Line 4 | PROG | #FC4D | #FE1B | 1 | Analyser program space |
| Line 5 | VSCR | #C000 | #DAFF | 6 | Virtual screen storage. |
| Line 6 | OPTION | 00000000 |  |  |  |
| Line 7 | VS | OFF |  |  | Virtual screen on/off |
| Line 8 | ANAL | OFF |  |  | Analyser on/off |

These values are typical but they may vary from version to version.

TRACE
Like the standard Spectrum but this trace also shows the program page from which each instruction was executed.

WORK=<start>,<fiNish>,<page>
At start up WORK will lie in the same page as the monitor. If this workspace is insufficient it may be moved to another page in which case the memory available for PROG= will be larger accordingly.

**User Notes**

(i) 'Prog' space must always be. in the same page as the monitor.

(ii) The monitor uses the 128k system variable at #5B5C to keep track of page switching operations. The user is warned therefore that any page switching must be duplicated in this variable otherwise the monitor will not keep track. The 128K itself expects to find the current page data in this system variable so any failure to conform may result in a bug.

## *4. 128K Analyser – Additional Words*

MPAGE! n1 >
Sets the memory page used by the memory pointer to the last value on the stack. n1 should be in the range 0 to 7. If n1 is greater than 7 then ni mode 8 is used.

MPAGE > n1
Puts the currently selected memory page onto the stack.

PPAGE > n1
Puts a value on the stack which gives the currently selected program page (used by user programs), the screen select, the ROM select and the Spectrum 48k/Spectrum 128k lock switch. These are contained in the following bits.

**Bits**

|  |  |  |  |
|---|---|---|---|
| 0, 1, 2 | Use RAM Page | 0, 1, 2, 3, 4, 5, 6 or 7 |  |
| 3 | Screen Select | 0 = page 5, | 1 = page 7 |
| 4 | ROM Select | 0 = 128K ROM, | 1 = Spectrum ROM |
| 5 | Spectrum Lock | 0 = I28K machine, | 1 = 48K Spectrum |