# HiSoft Devpac

## The Assembler/Debugger

## for all ZX Spectrums

HiSoft

High Quality Software

# HiSoft Devpac

## Fast Interactive Z80 Development Kit

**System Requirements:**
ZX Spectrum (48K and 128K), ZX Spectrum Plus, ZX Spectrum Plus 2, Plus 3.
Additional versions available for Opus Discovery & Disciple Disc systems.



**Version 4.1 September 1987**

Set using an Apple Macintosh™ and Laserwriter.

# HiSoft Licence Statement

Thank you for buying software from HiSoft. We hope you enjoy the package and find it useful. Please read this note carefully as it contains important information concerning the efficient use and support of your HiSoft software.

## Copyright

Firstly, and most importantly, your HiSoft software is protected by English Copyright Law and International Treaties relating to intellectual property. This means that you must treat the software and its documentation just like you would treat a book. You know that if you make a copy of any book then you are breaking Copyright Law and that legal action can ensue; the same is true of your HiSoft software. You may move the package around from location to location and from one computer system to another as long as there is never any possibility of the product being used by two people at the same time or on two computer systems at the same time; this would be violationg HiSoft's copyright. The one, single exception to this is that HiSoft authorises you to make backup copies of the software for the sole purpose of protecting your investment from loss.

## Warranty

With respect to the physical computer cassette, computer disc and physical documentation enclosed, HiSoft warrants the same to be free from defects in materials and workmanship for a period of 90 days from the date of purchase. On notification of any such fault within the warranty period, HiSoft will replace the defective cassette, disc or documentation. The remedy for any breach of this warranty shall be limited to replacement and shall not encompass any other damages.

HiSoft specifically disclaims all other warranties, express or implied, including but not limited to implied warranties of merchantibility and fitness for purpose with respect to defects in the cassette, disc and documentation. In no event shall HiSoft be liable for any loss of profit or any commercial damage.

## Technical Support

We put a great deal of effort into all our software packages, tailoring them to work efficiently on each computer system; however, if you do have any problems then please feel free to contact us by post at the address below, giving as much detail as possible (listings are often helpful). We ask you to communicate by letter so that we can evaluate your problem carefully with the full facts in front of us; in general this speeds up the solution and enables you to get programming again as quickly as possible. Please note that, to obtain technical support, you must be registered with us; do this by filling out the enclosed Registration Card and mailing it to us. The Licence Agreement referred to on the Registration Card is this document.

If you are desperate and need to contact us by phone, we have a technical hour between 3 p.m. and 4 p.m. every weekday when our programmers are available to help you. Please try to keep your call as concise as possible and have all relevant information to hand. It often helps to be sitting in front of the computer when phoning, if this is possible.

Once again, thank you for investing in HiSoft software.

**HiSoft** **The Old School, Greenfield, Bedford MK45 5DE. (0525) 718181**

# HiSoft GENS4 ASSEMBLER/EDITOR CONTENTS

# SECTION 1 GETTING STARTED

## 1.1 Introduction and Loading Instructions

GENS4 is a powerful and easy-to-use Z80 assembler which is very close to the standard Zilog assembler in definition. Unlike many other assemblers available for the Spectrum, GENS4 is an extensive, professional piece of software and you are urged to study the following sections, together with the example in Appendix 3, very carefully before attempting to use the asssembler. If you are a complete novice, work through Appendix 3 first or consult one of the excellent books given in the Bibliography.

We have versions of Devpac 4, on disc, for the Disciple & Opus Discovery disc systems and the Spectrum Plus 3 computer. These versions work exactly as described here, simply replace the word *Microdrive* where it occurs with *Opus disc, Plus 3 disc* or *Disciple disc* as appropriate. There are some extra features for the Plus 3 version which are described in the additional leaflet.

GENS4 is roughly 10K bytes in length, once relocated, and uses its own internal stack so that it is a self-contained piece of software. It contains its own integral line editor which places the textfile immediately after the GENS4 code while the assembler's symbol table is created after the textfile. This when loading GENS4 you must allow enough room to include the assembler itself and the maximum symbol table and text size that you are likely you use in the current session. It will often be convenient, therefore, to load GENS4 into low memory.

There are two versions of the assembler on the cassette, both are on side 1. First comes the 51 characters-per-line version, followed by the regular 32 character-per-line version. Their names on the tape are `GENS4-51` and `GENS4` respectively and you should use whichever version suits you best, the 51-column code is some 400 bytes longer than the 32-column one.

To load GENS4, place the supplied tape in your cassette recorder and type:

```
LOAD "" CODE xxxxx [ENTER]           and press PLAY on the recorder or
LOAD "GENS4" CODE xxxxx [ENTER]      or
LOAD "GENS4-51" CODE xxxxx [ENTER]   if loading from disc
```

where `xxxxx` is the decimal address at which you want GENS4 to run.

Once you have loaded the GENS4 code into the computer you may enter the assembler by `RANDOMIZE USR xxxxx` where `xxxxx` is the address at which you loaded the assembler code. If at any subsequent time you wish to re-enter the assembler then you should simply execute address `xxxxx` which will preserve any previously-created textfile.

For example, say you want to load GENS4 so that it executes from address 26000 decimal - proceed as follows:

```
LOAD "" CODE 26000 [ENTER]
RANDOMIZE USR 26000 [ENTER]
```

To re-enter the assembler use `RANDOMIZE USR 26000` from within BASIC.

Once you have entered GEN, a help screen will appear and you will be prompted with a > sign, the editor's command prompt - consult **Section 3** for how to enter and edit text and **Section 2** for what to enter.

# 1.2 Making a Backup Copy

Once you have loaded GENS4 into your Spectrum's memory then you can make a backup copy of the assembler as follows:

```
SAVE "GENS4-51" CODE xxxxx,11392 [ENTER]      or
SAVE "GENS4" CODE xxxxx,11010 [ENTER]         to cassette

SAVE *"M";1;"GENS4-51" CODE xxxxx,11392 [ENTER]      or
SAVE *"M";1;"GENS4" CODE xxxxx,11010 [ENTER]         to Microdrive
```

where: `xxxxx` is the address at which you loaded GENS4. You should do this backup before entering GENS so as to preserve the relocation information within the program.

See the additional leaflet for back-up instructions for Devpac on the Plus 3.

Please note that we allow you to make a backup copy of GENS4 for your own use so that you can program with confidence. Please do not copy GENS4 to give (or worse, sell) to your friends, we supply very reasonably priced software and a full after-sales support service but if enough people copy our software we shall not be able to continue this; *please buy, don't steal*.

# SECTION 2 DETAILS OF GENS4

## 2.0 How GENS4 Works, Assembler Options, Listing Format

GENS4 is a fast, two-pass Z80 assembler which assembles all standard Z80 mnemonics and has added features which include macros, conditional assembly, many assembler commands and a binary-tree symbol table.

When you invoke an assembly, you use the `A` command like this:

```
A4,2000,1:TEST [ENTER]
```

The first number (4 above) after the `A` specifies the assembly options you want this time, these options are listed below. If you don't want any options then don't type a number, just a comma.

The second number (`2000` above) is the size of the assembler's symbol table, in decimal bytes. If you default this (by simply using a comma instead of a number) then GENS4 will choose a symbol table size that it thinks is suitable for the size of the text - normally this will be perfectly acceptable. However, when using the *Include* option, you may have to specify a larger than normal symbol table size; the assembler cannot predict the size of the file that will be included.

After the symbol table size you can type a microdrive filename (`1:TEST` above). If you do, then the resulting object code generated by the assembly will be saved to microdrive, automatically. It doesn't matter how much object code you generate, *all* will be saved. If you don't want this feature, then don't type a filename (don't type the comma either, otherwise GENS will think you have a blank filename). See the `A]` command in **Section 3** for more details of this feature and its effect on the use of `ORG`.

### Assembler Options

**Option 1** produce a symbol table listing at the end of the second pass of the assembly.

**Option 2** Do not generate any object code.

**Option 4** Produce an assembly listing, note this is the reverse of previous versions of the assembler!

**Option 8** Direct any assembly listing to the printer.

**Option 16** Simply place the object code, if generated, after the symbol table. The Location Counter is still updated by the `ORG` so that object code can be placed in one section of memory but designed to run elsewhere.

**Option 32** Turn off the check of where the object code is going - useful for speeding up assembly.

To combine options, simply add them together e.g. `A33` produces a fast assembly - no listing is generated, no checks are made to see where the object code is being placed and a symbol table listing is produced at the end.

Note that if you have used Option 16 then the `ENT` assembler directive will have no effect. You can work out where the object code has been placed if Option 16 has been specified by using the editor `Y` command to find out the end of the text (the second number displayed) and then adding to this the amount of symbol table allocated + 2.

Assembly takes place in two passes; during the first pass GENS4 searches for errors and compiles the symbol table, the second pass generates object code (if option 2 is not specified). During the first pass nothing is displayed on the screen or printer unless an error is detected, in which case the rogue line will be displayed with an error number below it (see **Appendix 1**). The assembly is paused - press `E` to return to the editor or any other key to continue the assembly from the next line.

At the end of the first pass the message:

```
Pass 1 errors: nn
```

will be displayed. If any errors have been detected the assembly will then halt and not proceed to the second pass. If any labels were referenced in the operand field but never declared in a label field then the message:

```
*WARNING* label absent
```

will be displayed for each missing label declaration.

It is during the second pass that object code is generated (unless generation has been turned off by Option 2 - see above). An assembler listing is not generated during this pass unless it has been switched on by Option 4 or the assembler command `*L+`.

In the 32-column version, the assembler listing is normally on two lines and is of the form:

```
C000 210100       25 label
                     LD   HL,1
1    6        15     21   26
```

whereas in the 51-column version the listing is continuous, in one line, wrapping round onto the next line if too long to fit on one.

The first entry in the line is the value of the Location Counter at the start of processing this line, unless the mnemonic in this line is one of the pseudo-mnemonics `ORG`, `EQU` or `ENT` (see **Section 2.7**) in which case the first entry will represent the value in the operand field of the instruction. This entry is normally displayed in hexadecimal but may be displayed in unsigned decimal through use of the assembler command `*D+` (see **Section 2.9**).

The next entry, from column 6, is up to 8 characters in length (representing up to 4 bytes) and is the object code produced by the current instruction - but see the `*C` assembler command below.

Then comes the line number - an integer in the range 1 to 32767 inclusive.

Columns 21-26 of the first line contain the first 6 characters of any label defined in this line.

After any label comes the mnemonic which is displayed from columns 21-24 (in the 32-column version, this will be on a new line unless `*C-` has been used).

Then comes the operand field from column 26 of this line and finally any comments that have been inserted at the end of the line with new lines being generated as necessary.

The `*C` assembler command may be used to produce a shorter assembly listing line - its effect is to omit the 9 characters representing the object code of the line thus enabling most assembler lines to fit on one 32-column screen line. See **Section 2.8** below.

# Modifying the Listing Format *32-column version only*

You can modify the form in which each line of the listing is split by POKEing 3 locations within the 32-column version of GENS4. Details of how to do this are given below. We distinguish between *assembly line* which is the current line of the assembly listing held in an internal buffer and *screen line* which is a line that actually appears on the screen. An assembly line will normally generate more than one screen line.

1. Location *Start of GENS4 + 51 (#33)* dictates at which column position - 5 the first screen line of the assembly line will be terminated. Change this byte to zero to cause the line to wrap round (useful if you have a full-width printer) or any other value (<256) to end the first screen line at a particular column.

2. Location *Start of GENS4 + 52 (#34)* gives the column position (from 1) at which each subsequent screen line of the assembly line is to start.

3. Location *Start of GENS4 + 53 (#35)* gives how many characters from the remainder of the assembly line are to be displayed on each screen line after the first screen line.

As an example, say you wanted the first screen line of each assembly line to contain 20 characters (i.e. not including the label field) and then each subsequent screen line to start at column 1 and fill the whole line. Also assume that you have loaded GENS4 at 26000 decimal. To effect these changes, execute the following POKE instructions from within BASIC:

```
POKE 2605120
POKE 26052,1       there must be at least one space at the
POKE 26053,31      start of each subsequent screen line.
```

The above modifications are only applicable if the *C command has not been used - use of the *C command causes lines to roll over where necessary.

The assembly listing may be paused at the end of a line by hitting [CAPS SHIFT] and [SPACE] together - subsequently hit E to return to the editor or any other key to continue the listing.

The only errors that can occur during the second pass are *ERROR* 10 (see **Appendix 1**) and Bad ORG! (which occurs when the object code will overwrite GENS4, the textfil or the symbol table - the detection of this can be turned off by Option 32). *ERROR* 10 is non-fatal and you may continue the assembly as for first pass errors whereas Bad ORG! is fatal and immediately returns control to the editor.

At the end of the second pass the message:

```
Pass 2 errors: nn
```

will be displayed followed by warnings of any absent labels - see above. The following message is now displayed:

```
Table used: xxxxx from yyyyy
```

This informs you of how much of the symbol table was used compared with how much was allocated.

At this point, if the assembler directive `ENT` has been used correctly, the message `Executes: nnnnn` is displayed. This shows the run address of the object code - you can execute the code by using the editor `R` command. Be careful using the `R` command unless you have just finished a successful assembly and seen the `Executes: nnnnn` message.

Finally, if option 1 has been specified, an alphabetic list of the labels used and their associated values will be produced. The number of entries displayed on one line may be changed by `POKE`ing *`Start of GENS4 + 50`* with the relevant value; the default is 2.

Control now returns to the editor.

# 2.1 Assembler Statement Format

Each line of text hat is to be processed by GENS4 should have the following format where certain fields are optional:

```
LABEL   MNEMONIC   OPERANDS     COMMENT
start   LD         HL,label     ;pick up 'label'
```

Spaces and tab characters (inserted by the editor) are generally ignored. The line is processed in the following way:

The first character of the line is checked and subsequent action depends on the nature of this character as indicated on the next page:

`;` the whole line is treated as a comment i.e. effectively ignored.

`*` expects the next character(s) to constitute an assembler command (see **Section 2.8**). Treats all characters after the command as a comment.

`<CR>` (end-of-line character) simply ignores the line.

_ (space or tab) if the first character is a space or a tab character then GENS4 expects the next non-space or non- tab character to be the start of a Z80 mnemonic.

If the first character of a line is any character other than those given above then the assembler expects a label to be present - see **Section 2.2**. After processing a valid label, or if the first character of the line is a space/tab, the assembler searches for the next non-space/tab character and expects this to be either an end-of-line character or the start of a Z80 mnemonic (see **Appendix 2**) of up to 4 characters in length and terminated by a space/tab or end-of-line character.

If the mnemonic is valid and requires one or more operands then spaces/tabs are skipped and the operand field is processed.

Labels may be present alone in an assembler statement; this is useful for increasing the readability of the listing.

Comments may occur anywhere after the operand field or, if a mnemonic takes no arguments, after the mnemonic field.

## 2.2 Labels

A label is a symbol which represents up to 16 bits of information. A label can be used to specify the address of a particular instruction or data area or it can be used as a constant via the `EQU` directive (see **Section 2.7**).

If a label is associated with a value greater than 8 bits and it is then used in a context where an 8 bit constant is applicable then the assembler will generate an error message e.g.

```
label   EQU     #1234
        LD      A,label
```

will cause `*ERROR*  10` to be generated when the second statement is processed during the second pass.

A label may contain any number of valid characters (see below) although only the first 6 are treated as significant; these first 6 characters must be unique since a label cannot be re-defined (`*ERROR* 4`). A label must not constitute a Reserved Word (see **Appendix 2** although a Reserved Word may be embedded as part of a label.

The characters which may be legally used within a label are `0-9`, `$` and `A-z`. Note that `A-z` includes all the upper and lower case alphabetics together with the characters `[`, `\`, `]`, `^`, `#` and `_`. A label must begin with an alphabetic character. Some examples of valid labels are:

```
LOOP
loop
a_long_label
L[1]
L[2]
a
LDIR      LDIR is not a Reserved Word.
two^5
```

# 2.3 Location Counter

The assembler maintains a Location Counter so that a symbol in the label field can be associated with an address and entered into the Symbol Table. This Location Counter may be set to any value via the `ORG` assembler directive (see **Section 2.7**).

The symbol `$` can be used to refer to the current value of the Location Counter e.g. `LD HL,$+5` would generate code that would load the register pair HL with a value 5 greater than the current Location Counter value.

# 2.4 Symbol Table

When a label is encoutered for the first time it is entered into a table along with two pointers which indicate, at a later time, how this label is related alphabetically to other labels within the table. If the first occurrence of the label is in the label field then its value (as given by the Location Counter or the value of the expression after an `EQU` assembler directive) is entered into the Symbol Table. Otherwise the value is entered whenever the symbol is subsequently found in the label field.

This type of symbol table is called a *Binary Tree Symbol Table*] and its structure enables symbols to be entered into and recovered from the table in a very short time - essential for large programs.

The size of an entry in the table varies from 8 bytes to 13 bytes depending on the length of the symbol.

If, during the first pass, a symbol is defined more than once then an error (`*ERROR* 4`) will be generated since the assembler does not know which value should be associated with the symbol.

If a symbol is never associated with a value then the message `*WARNING symbol absent` will be generated at the end of the assembly. The absence of a symbol definition does not prevent the assembly from continuing.

Note that only the first 6 characters of a symbol are entered into the Symbol Table in order to keep down the size of the table.

At the end of the assembly you will be given a message stating how much memory was used by the Symbol Table during this assembly - you may change how much memory is allocated to the Symbol Table when starting the assembly (see **Section 2.0**).

# 2.5 Expressions

An expression is an operand entry consisting of either a single **TERM** or a combination of terms each separated by an **OPERATOR**. The definitions of *term* and *operator* follow:

## TERM

decimal constant e.g. `1029`
hexadecimal constant e.g. `#405`
binary constant e.g. `%10000000101`
character constant e.g. `"a"`
label e.g. `L1029`

also `$` may be used to denote the current value of the Location Counter.

# OPERATOR

| | |
|---|---|
| + | addition |
| - | subtraction |
| & | logical AND |
| @ | logical OR |
| ! | logical XOR |
| * | integer multiplication |
| / | integer division |
| ? | MOD function ( `a ? b = a - (a/b)*b` ) |

Notes: # is used to denote the start of a hexadecimal number, `%` for a binary number and `"` for a character constant. When reading a number (decimal, hexadecimal or binary) GENS4 takes the least significant 16 bits of the number (i.e. MOD 65536) e.g. `70016` becomes `4480` and `#5A2C4` becomes `#A2C4`.

A wide variety of operators are provided but no operator precedence is observed; ***expressions are evaluated strictly from left to right.*** The operators `*`, `/` and `?` are provided merely for added convenience and not as part of a full expression handler which would increase the size of GENS4. If an expression is enclosed within parentheses then it is taken as representing a memory address as in the instruction `LD HL,(loc+5)` which would load the register pair HL with the 16 bit value contained at memory location `loc+5`.

Certain Z80 instructions (`JR` and `DJNZ`) expect operands which have an 8 bit value and not a 16 bit one - this is called relative addressing. When relative addresses are specified GENS4 automatically subtracts the value of the Location Counter at the next instruction from the value given in the operand field of the current instruction in order to obtain the relative address for the current instruction. The range of values allowed as a relative address are the Location Counter value of the next instruction -128 to +127.

If, instead, you wish to specify a relative offset from the Location Counter value of the current instruction then you should use the symbol `$` (a Reserved Word) followed by the required displacement. Since this is now relative to the current instruction's Location Counter value the displacement must be in the range -126 to +129 inclusive.

## Examples of valid expressions

```
#5000 - label
%1001101 ! %1011                    gives %1000110
#3456 ? #1000                       gives #456
4 + 5 * 3 - 8                       gives 19
$-label+8
2345 / 7 - 1                        gives 334]
"y"-";"+7
(5 * label - #1000 & %1111)
17 @ %1000                          gives 25
```

Note that spaces may be inserted between terms and operators and vice versa but not within terms.

If a multiplication operation would result in an absolute value greater than 32767 then `*ERROR* 15` is reported while if a division operation involves a division by zero then `*ERROR* 14` is given - otherwise overflow is ignored. All arithmetic uses the two's complement for where any numbers greater than 32767 are treated as negative e.g. 60000 = -5536 (60000-65536).

If a multiplication operation would result in an absolute value greater than 32767 then `*ERROR* 15` is reported while if a division operation involves a division by zero then `*ERROR* 14` is given - otherwise overflow is ignored. All arithmetic uses the two's complement for where any numbers greater than 32767 are treated as negative e.g. 60000 = -5536 (60000-65536).

# 2.6 Macros

Macros allow you to write shorter, more meaningful assembler programs but they must be used with care and must not be confused with subroutines. A macro definition consists of a series of assembler statements, together with the name of the macro; when this macro name is used subsequently in the mnemonic field then it will be replaced by all the assembler statements that made up the definition e.g. the macro `NSUB` may be defined thus:

```
NSUB    MAC
        OR      A
        SBC     HL,DE
        ADD     HL,DE
        ENDM
```

and then, whenever NSUB is used as a mnemonic, it will generate the three assembler statements OR A SBC HL,DE and ADD HL,DE. This saves you typing and makes your program easier to understand but you must remember that every occurrence of NSUB results in code being generated and it may be more efficient to use a CALL to a subroutine instead. Below, we give the format of macro definitions and invocations together with some more examples, please study these carefully.

A macro definition takes the following form:

```
Name    MAC
        .
        .
        macro definition
        .
        .
        ENDM
```

where Name is the macro name that will invoke the text of the macro whenever Name is used subsequently in the mnemonic field, MAC indicates the start of the macro definition and ENDM indicates the end of the definition.

Parameters of the macro may be referenced within the macro definition by the use of the equals sign followed by the parameter number (0-31 inclusive) e.g. the macro:

```
MOVE    MAC
        LD      HL,=0
        LD      DE,=0
        LD      BC,=0
        LDIR
        ENDM
```

takes 3 parameters, source address, destination address and length, loads the relevant values into HL, DE and BC and then performs the instruction LDIR. To invoke this macro at a later stage in your program, simply use the name of the macro in the mnemonic field followed by the values that you wish the 3 parameters to take e.g.

```
        MOVE 16384,16535,4096
```

We have used specific addresses in this example but we can, in fact, use any valid expression to specify the value of the macro parameter e.g.

```
        MOVE start,start+1,length
```

Think .... is the above a good use of a macro? Could it have been a subroutine?

Within the macro definition, the parameters may appear in any valid expression e.g.

```
HMS     MAC
        LD      HL,=0*3600
        LD      DE,=1*60
        ADD     HL,DE
        LD      DE,=2
        ADD     HL,DE
        ENDM
```

is a macro, taking 3 parameters - hours, minutes, seconds, that produces in register HL the total number of seconds specified by the parameters. You might use it like this:

```
Hours   EQU     2
Minutes EQU     30
Seconds EQU     12
Start   EQU     0

        HMS     Hours,Minutes,Seconds
        LD      DE,Start
        ADD     HL,DE           ;HL gives the finish time
```

Macros may not be nested so that you cannot define a macro within a macro definition nor can you invoke a macro within a macro definition.

At assembly time, whenever a macro name is encountered in the mnemonic field, the text of the macro is then assembled. Normally this text is not listed in the assembly listing - only the macro name is shown. However, you can force a listing of the expansion of the macro by using the assembler command `*M+` before you want macro expansions to be listed - use `*M-` to switch off this exansion.

If you run out of Macro Buffer space then a message will be displayed and the assembly aborted; use the editor's `C` command to allocate a larger Macro Buffer.

# 2.7 Assembler Directives

Certain *pseudo-mnemonics* are recognised by GENS4. These assembler directives, as they are called, have no effect on the Z80 processor at run-time i.e. they are not decoded into opcodes, they simply direct the assembler to take certain actions at assembly time. These actions have the effect of changing, in some way, the object code produced by GENS4.

Pseudo-mnemonics are assembled exactly like executable instructions; they may be preceded by a label (necessary for `EQU`) and followed by a comment. The directives available are:

## ORG expression

sets the Location Counter to the value of *expression*. If option 2 and option 16 are both not selected and an `ORG` would result in the overwriting of the GENS4 program, the textfile or the symbol table then the message `Bad ORG!` is displayed and the assembly is aborted. See **Section 2.0** for more details on how options 2 and 16 affect the use of `ORG`. See the `A` command in **Section 3** for some precautions on using `ORG` when automatically saving the object code.

## EQU expression

must be preceded by a label. Sets the value of the label to the value of `expression`. The expression cannot contain a symbol which has not yet been assigned a value (`*ERROR* 13`).

## DEFB expression,expression,....

each `expression` must evaluate to 8 bits; the byte at the address currently held by the Location Counter is set to the value of `expression` and the Location Counter advanced by 1. Repeats for each expression.

## DEFW expression,expression,....

sets the 'word' (two bytes) at the address currently held by the Location Counter to the value of `expression` and advances the Location Counter by 2. The lesser significant byte is placed first followed by the more significant byte. Repeats for each expression.

## DEFS expression

increases the Location Counter by the value of `expression` - equivalent to reserving a block of memory of size equal to the value of `expression`.

## DEFM "s"

defines the contents of n bytes of memory to be equal to the ASCII representation of the string s, where n is the length of the string and may be, in theory, in the range 1 to 255 inclusive although, in practice, the length of the string is limited by the length of the line you can enter from the editor. The first character in the operand field (`"` above) is taken as the string delimiter and the string `s` is defined as those characters between two delimiters; the end-of-line character also acts as a terminator of the string.

## ENT expression

this has no effect on the generated code, it is simply used to define an address to which the editor's `R` command will jump to. `ENT` expression sets this address to the value of expression - used in conjunction with the editor `R` command (see **Section 3**). There is no default for the execute address.

# 2.8 Conditional Pseudo-mnemonics

Conditional pseudo-mnemonics provide the programmer with the capability of including or not including certain sections of source text in the assembly process. This is made available through the use of IF, ELSE and END.

## IF expression

this evaluates `xpression`. If the result is zero then the assembly of subsequent lines is turned off until either an `ELSE` or an `END` pseudo-mnemonic is encountered. If the value of `expression` is non-zero then the assembly continues normally.

## ELSE

this pseudo-mnemonic simply flips the assembly on and off. If the assembly is on before the `ELSE` is encountered then it will subsequently be turned off and vice versa.

## END

END simply turns the assembly on.

Note: Conditional pseudo-mnemonics cannot be nested; no check is made for nested IF's so any attempt to nest these mnemonics will have unspecified results.

# 2.9 Assembler Commands

Assembler commands, like assembler directives, have no effect on the Z80 processor at runtime since they are not decoded into opcodes. However, unlike assembler directives, they also have no effect on the object code produced by the assembler - assembler commands simply modify the listing format. An assembler command is a line of the source text that begins with an asterisk *.

The letter after the asterisk determines the type of the command and must be in upper case. The remainder of the line may be any text except that the commands L and D expect a + or a - after the command.

The following commands are available:

## *E

(eject) causes three blank lines to be sent to the screen or printer - useful for separating modules.

## *Hs

causes string s to be taken as a heading which is printed after each eject (*E). *H automatically performs a *E.

## *S

causes the listing to be stopped at this line. The listing may be reactivated by pressing any key on the keyboard. Useful for reading addresses in the middle of the listing. Note: *S is still recognised after a *L-, *S does not halt printing.

## *L-

causes listing and printing to be turned off beginning with this line.

## *L+

causes listing and printing to be turned on starting with this line.

## *D+

causes the value of the Location Counter to be given in decimal at the beginning of each line instead of the normal hexadecimal. Unsigned decimal is used.

## *D-

reverts to using hexadecimal for the value of the Location Counter at the start of each line.

## *C-

Shorten the assembler listing starting from the next line. The listing is abbreviated by not including the display of the object code generated by the current line - this saves 9 characters and enables most assembler lines to fit within one 32-character screen line, thus improving readability.

## *C+

Revert to the full assembler listing as described in **Section 2.0**.

## *M+

Turn on the listing of macro expansions.

## *M-

Turn off the listing of macro expansions.

# *F filename

This is a very powerful command which allows you to assembler text from tape or microdrive - the textfile is read from the tape or microdrive into a buffer, a block at a time, and then assembled from the buffer; this allows you to create large amounts of object code since the text being assembled does not take up valuable memory space.

The filename (up to 10 characters) of the textfile you wish to 'include' at this point in the assembly may, optionally, be specified after the `F` and must be preceded with a space. If the file is on microdrive cartdridge then you indicate this by starting the filename with a drive number and a colon e.g

```
*F 2:TEST       to include from Microdrive Drive 2
*F TEST         to include from tape
```

If no filename is given then the first textfile found on the tape is included, this is not allowed for microdrive inclusion.

If you are including from microdrive then the text to be included should have been saved previously using the editor's `P`(ut) command in the normal way.

If including from tape then you must have saved the file previously to tape using the editor's `T` command and not the `P` command - this is necessary because a textfile to be included from take must be dumped out in blocks with sufficient length inter-block gaps to allow the assembly of the current block before the next block is loaded from the tape. The size of the block used by this command (and the editor's `T` command) is set using the editor's `C` command (see next section). The ability to select the size of this buffer enables you to optimise the size/speed ratio of any inclusion of text from tape; for example, if you are not intending to use the `F` command during an assembly then you may find it useful to specify a buffer size of 1 to minimise the space taken up by GENS4 and its workspace.

Whenever the assembler detects an `F` command it searches the tape or microdrive cartridge for the relevant file; this will happen in the first and second passes since the include text must be scanned in each pass. If including from tape, the the tape is then searched for an include file with the required filename, or for the first file. If an include file is found whose filename does not match that required then the message `Found filename` is displayed and searching continues, otherwise `Using filename` is displayed, the file loaded, block by block, and included.

See **Appendix 3** for an example of the use of this command.

Assembler commands, other than `*F`, are recognised only within the second pass.

If assembly has been tuned off by one the conditional pseudo-mnemonics then the effect of any assembler command is also turned off.

**Note:** The include facility is not available from tape on the Disciple, Opus and Plus 3 disc versions of Devpac 4. It is much faster and easier to include from disc instead. Side-effects of this are that, within these disc versions, the `C` editor command does not allow you to specify an include buffer size and the `T` command does not exist.

# SECTION 3
# THE INTEGRAL EDITOR

## 3.1 Introduction to the Editor

The editor supplied with all versions of GENS4 is a simple, line-based editor designed to work with all Z80 operating systems while maintaining ease of use and the ability to edit programs quickly and efficiently.

In order to reduce the size of the textfile, a certain amount of compression of spaces is performed by the editor. This takes place according to the following scheme: whenever a line is typed in from the keyboard it is entered, character by character into a buffer internal to the assembler; then, when the line is finished (i.e. you hit `[ENTER]`), it is transferred from the buffer into the textfile.

It is during this transfer that certain spaces are compressed: the line is scanned from its first character, if this is a space then a tab character is entered into the textfile and all subsequent spaces are skipped. If the first character is not a space then characters are transferred from the buffer to the textfile until a space is detected whereupon the action taken is the same as if the next character was the first character in the line. This is then repeated a further time with the result that tab characters are inserted at the front of the line or between the label and the mnemonic and between the mnemonic and the operands and between the operands and any comment. Of course, if any carriage return `[ENTER]` character is detected at any time then the transfer is finished and control returned to the editor.

This compression process is transparent and you may simply use cursor right (→) to produce a neatly tabulated textfile which, at the same time, is economic on storage.

Note that spaces are not compressed within comments and spaces should not be present within a label, mnemonic or operand field.

The editor is entered automatically when GENS4 is executed and displays a help screen, followed by the editor prompt >.

In response to the prompt you may enter a command line of the following format:

```
C N1, N2, S1, S2          followed by [ENTER]
```

`C` is the command to be executed (see **Section 3.2** below). `N1` is a number in the range 1 - 32767 inclusive. `N2` is a number in the range 1 - 32767 inclusive. `S1` is a string of characters with a maximum length of 20. `S2` is a string of characters with a maximum length of 20.

The comma is used to separate the various arguments (although this can be changed - see the `S` command) and spaces are ignored, except within the strings. None of the arguments are mandatory although some of the commands (e.g. the `D`elete command) will not proceed without `N1` and `N2` being specified.

The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line. The values of `N1` and `N2` are initially set to 10 and the strings are initially empty. If you enter an illegal command line such as `F- 1,100,HELLO` then the line will be ignored and the message `Pardon?` displayed - you should then retype the line correctly e.g. `F1,100,HELLO`. This error message will also be displayed if the length of `S2` exceeds 20; if the length of `S1` is greater than 20 then any excess characters are ignored.

Commands may be entered in upper or lower case.

While entering a command line certain key combinations may be used to edit the line viz. ← to delete to the beginning of the line, → to advance the cursor to the next tab position, `[CAPS SHIFT] 0` or `[DELETE]` to delete the previous character.

The following sub-section gives the various commands available within the editor - note that wherever an argument is enclosed by the symbols < > then that argument *must* be present for the command to proceed.

# 3.2 The Editor Commands

## 3.2.1 Text Insertion

Text may be inserted into the textfile either by typing a line number, a space and then the required text or by use of the `I` command. Note that if you type a line number followed by `[ENTER]` (i.e. without any text) then that line will be deleted from the text if it exists. Whenever text is being entered ← (delete to the beginning of the line), → (go to the next tab position) and `[EDIT]` (return to the command loop) may be employed.

The `[DELETE]` (`[CAPS SHIFT] 0`) key will produce a destructive backspace (but not beyond the beginning of the text line). Text is entered into an internal buffer within GENS4 and if this buffer should become full then you will be prevented from entering any more text - you must then use `[DELETE]` or ← to free space in the buffer. If, during text insertion, the editor detects that the end of text is nearing the top of RAM it displays the message `Bad Memory!`. This indicates that no more text can be inserted and that the current textfile, or at least part of it, should be saved to tape/microdrive for later retrieval.

## Command: I n,m

Use of this command gains entry to the automatic insert mode: you are prompted with line numbers starting at `n` and incrementing in steps of `m`. You enter the required text after the displayed line number, using the various control codes if desired and terminating the text line with `[ENTER]`. To exit from this mode use `[EDIT]`.

If you enter a line with a line number that already exists in the text then the existing line will be deleted and replaced with the new line, after you have pressed `[ENTER]`. If the automatic incrementing of the line number produces a line number greater than 32767 then the Insert mode will exit automatically.

If, when typing in text, you get to the end of a screen line without having entered 64 characters (the buffer size) then the screen will be scrolled up and you may continue typing on the next line - an automatic indentation will be given to the text so that the line numbers are effectively separated from the text.

## 3.2.2 Text Listing

Text may be inspected by use of the L command; the number of lines displayed at any one time during the execution of this command is fixed initially but may be changed through use of the K command.

## Command: L n,m

This lists the current text to the display device from line number n to line number m inclusive. The default value for n is always 1 and the default value for m is always 32767 i.e. default values are not taken from previously entered arguments.

To list the entire textfile simply use L without any arguments. Screen lines are formatted with a left hand margin so that the line number is clearly displayed. Tabulation of the line is automatic, resulting in a clear separation of the various fields with the line. The number of screen lines listed on the display device may be controlled through use of the K command - after listing a certain number of lines the list will pause (if not yet at line number m),hit [EDIT] to return to the main editor loop or any other key to continue the listing.

## Command: K n

K sets the number of screen lines to be listed to the display device before the display is paused as described in L above. The value (n MOD 256) is computed and stored. For example use K5 if you wish a subsequent List to produce five screen lines at a time.

## 3.2.3 Text Editing

Once some text has been created there will inevitably be a need to edit some lines. Various commands are provided to enable lines to be amended, deleted, moved and renumbered:

## Command: D <n,m>

All lines from n to m inclusive are deleted from the textfile. If n<m, or less than two arguments are specified, then no action will be taken; this is to help prevent careless mistakes. A single line may be deleted by making m=n; this can also be accomplished by simply typing the line number followed by [ENTER].

## Command: M n,m

This causes the text at line n to be entered at line m deleting any text that already exists there. Note that line n is left alone. So this command allows you to Move a line of text to another position within the textfile. If line number n does not exist then no action is taken.

## Command: N <n,m>

Use of the N command causes the textfile to be renumbered with a first line number of n and in line number steps of m. Both n and m must be present and if the renumbering would cause any line number to exceed 32767 then the original numbering is retained.

## Command: F n,m,f,s

The text existing within the line range n→m is searched for an occurrence of the string f - the 'find' string. If such an occurrence is found then the relevant text line is displayed and the Edit mode is entered - see below.

You may then use commands within the Edit mode to search for subsequent occurrences of the string f within the defined line range or to substitute the string s (the 'substitute' string) for the current occurrence of f and then search for the next occurrence of f; see below for more details. Note that the line range and the two strings may have been set up previously by any other command so that it may only be necessary to enter F to initiate the search - see the example in **Section 3.3** for clarification.

## Command: E n

Edit the line with line number n. If n does not exist then no action is taken; otherwise the line is copied into a buffer and displayed on the screen (with the line number), the line number is displayed again underneath the line and the Edit mode is entered. All subsequent editing takes place within the buffer and not in the text itself; thus the original line can be recovered at any time.

In this mode a pointer is imagined moving through the line (starting at the first character) and various sub-commands are supported which allow you to edit the line. The sub-commands are:

| | |
|---|---|
| (space) | increment the text pointer by one i.e. point to the next character in the line. You cannot step beyond the end of the line. |
| [DELETE] | decrement the text pointer by one to point at the previous character in the line. You cannot step backwards beyond the first character in the line. |
| → | step the text pointer forwards to the next tab position on each screen line. |
| [ENTER] | end the edit of this line keeping all the changes made. |
| Q | quit the edit of this line i.e. leave the edit ignoring all the changes made and leaving the line as it was before the edit was initiated. |
| R | reload the edit buffer from the text i.e. forget all changes made on this line and restore the line as it was originally. |
| L | list the rest of the line being edited i.e. the remainder of the line beyond the current pointer positions. You remain in the Edit mode with the pointer re-positioned at the start of the line. |
| K | kill (delete) the character at the current pointer position. |
| Z | delete all the characters from (and including) the current pointer position to the end of the line. |
| F | find the next occurrence of the 'find' string previously defined within a command line (see the F command above). This sub-command will automatically exit the edit on the current line (keeping the changes) if it does not find another occurrence of the 'find' string in the current line. If an occurrence of the 'find' string is detected in a subsequent line (within the previously specified line range) then the Edit mode will be entered for the line in which the string is found. Note that the text pointer is always positioned at the start of the found string. |
| S | substitute the previously defined 'substitute' string for the currently found occurrence of the 'find' string and then perform the sub-command F i.e. search for the next occurrence of the 'find' string. This, together with the above F sub-command, is used to step through the textfile optionally replacing occurrences of the 'find' string with the 'substitute' string - see **Section 3.3** for an example. |

| | |
|---|---|
| I | insert characters at the current pointer position. You will remain in this sub-mode until you press [ENTER] - this will return you to the main Edit mode with the pointer positioned after the last character inserted. Using [DELETE] within this sub-mode will cause the character to the left of the pointer to be deleted from the buffer while the use of → will advance the pointer to the next tab position, inserting spaces. |
| X | this advances the pointer to the end of the line and enters the insert sub-mode detailed above. |
| C | change sub-mode. This allows you to overwrite the character at the current pointer position and then advances the pointer by one. You remain in the change sub-mode until you press [ENTER] whence you are taken back to the Edit mode with the pointer positioned after the last character you changed. [DELETE] simply decrements the pointer by one i.e. moves it left while → has no effect. |

## 3.2.4 Tape/Microdrive Commands

Text may be saved to tape/microdrive or loaded from tape/microdrive using the commands P, G and T. Object code may be saved to tape/microdrive using the O command. Filenames may be up to 10 characters long.

## Command: P n,m,s

The line range n→m is saved to tape or microdrive under the filename specified by the string s. The text will be saved to microdrive if the filename begins with a drive number followed by a colon (:). Remember that these arguments may have been set by a previous command. Examples:

| | |
|---|---|
| P10,200,EXAMPLE | save lines 10-200 to tape as EXAMPLE |
| P500,900,1:TEXT | save lines 500-900 to microdrive 1 |

Before entering this command make sure that your tape recorder is switched on and in RECORD mode, if saving to tape. Do not use this command if you wish, at a later stage, to 'Include' the text from tape - use the T command instead. If you intend to 'include' from microdrive then you should use this P command.

When Putting to microdrive and the filename you have specified already exists on the cartridge, you will be asked:

```
File Exists Delete (Y/N)?
```

answer `Y` to delete the file and continue saving or any other key to return to the editor without saving the file.

## Command: G,,s

The tape or microdrive is searched for a file with a filename of `s`; when found, it is loaded at the end of the current text. If a null string is specified as the filename then the first textfile on the tape is loaded. For microdrive, you must specify a filename and it should begin with a drive number followed by a colon.

If using cassette, after you have netered the `G` command, the message `Start tape..` is displayed - you should now press PLAY on your recorder. A search is made for a textfile with the specified filename, or the first textfile if a null filename is given. If a match is made then the message `Using filename` is displayed, otherwise `Found filename` is shown and the search of the tape continues.

If using microdrive and the specified file cannot be found then the message `Absent` is displayed.

Note that if any textfile is already present in the memory then the textfile that is loaded from tape will be appended to the existing file and the whole file will be renumbered starting with line 1 in steps of 1.

## Command: Tn,m,s

Dump out a block of text, between the line numbers `n` and `m` inclusive, to tape in a format suitable for inclusion at a later stage via the assembler command `*F` - see Section 2.9. The file is dumped with the filename `s`. The dump takes place immediately you have pressed `[ENTER]` so you should ensure that your tape recorder is ready to record before entering this command line.

If you intend to include from microdrive then use the `P` command to save the text,as usual, and not this `T` command.

Note that this command is to be used only if you want to assemble the text from tape at a later stage. It is not available on disc-only versions of Devpac 4.

## Command: O,,s

Dump out your object code to cassette or microdrive. The filename `s` can be up to 8 characters in length and should begin with a drive number (1-8) and a colon if you wish to save the object code to microdrive.

Only the last 'block' of code produced by the assembler can be saved in this way i.e. if you have more than one `ORG` directive in your source program then only the code produced after the last `ORG` is saved.

Code must have been produced in memory before it can be saved using `O`.

It will often be more convenient to automatically dump the object code by using the `A,,filename` command rather than the `O` command. See below.

## 3.2.5 Assembling and Running from the Editor

## Command: Ao,s,f

This causes the text to be assembled from the first line in the textfile.

`o` allows you to specify the options to be used in this assembly, see **page 3** for details. Normally, you will be able to use the default options by typing a comma.

`s` gives you the facility to change the symbol table size for this assembly. Look at **page 3** and **Section 2.4** for discussions on the symbol table. Again, the default size will often be satisfactory, except when Including.

`f` should be a valid microdrive filename, starting with `d:` where `d` is the microdrive drive number of the file; the filename can be up to 10 characters long. The presence of a filename here causes the assembler to behave in a different way from normal.

Instead of simply assembling the object code into memory and stopping when the top of memory is reached, the assembler will now assemble into memory until it reaches the top of memory (you can set the top of memory using the `U` command) and then it will save the object code assembled so far to microdrive in the file you asked for.

The assembly will then continue from the bottom of memory again and this process will continue until all the program has been assembled and saved to microdrive.

There is thus no limit (except the available space on your microdrive cartridge) to the size of the program you can assemble.

There are a couple of important points regarding the use of the ORG directive when using this facility:

**1.** The ORG directive will cause object code to be placed at the ORG address initially *and* also after each time code has been saved to the object file unless option 16 is used to ensure that object code is placed directly after the symbol table.

Therefore, it will normally be sensible to use option 16 when assembling directly to microdrive since this gives the maximum size for your object code buffer, the *execution* addresses of your object code will not be affected.

**2.** You should avoid using more than one ORG in your program unless you pad out any intervening memory with zeros by using DEFS e.g.

```
    ORG 50000

;some code
    RET
    ORG 60000

;some more code
```

will not be saved to microdrive correctly because the second ORG effectively redefines the start of the object code buffer. However:

```
    ORG 50000

;some code
    RET
;pad out until 60000
    DEFS 60000-$

;some more code
```

will be saved correctly since the DEFS 60000-$ generates sufficient zeros to ensure that subsequent code starts at address 60000.

This is obviously inefficient in terms of the amount of code stored on microdrive but its simplicity keeps the assembler small and fast.

Examples of the A command:

```
A20,,1:TEST [ENTER]
```

assembles, with listing on, putting the object code immediately after the symbol table (thus maximising the object code buffer in memory), using a default symbol table size and saving the object code to microdrive 1 under the name TEST.

```
A,3000 [ENTER]
```

Assemble the program, using default options and with a symbol table size of 3000 bytes.

See **Section 2** for further details of what happens during an assemmbly.

## Command: R

If the source has been assembled without errors and an execute address has been specified by the use of the ENT assembler directive then the R command may be used to execute the object program. The object program can use a RET (#C9) instruction to return to the editor so long as the stack is in the same position at the end of the execution of the program as it was at the beginning.

Note that ENT will have no effect if **Option 16** has been specified for the assembly.

Before entering the code, interrupts are enabled and register IY is loaded with the value #5C3A, important for the Spectrum ROM interrupt routine.

## 3.2.6 Other Commands

## Command: B

This simply returns control to the operating system. To re-enter the assembler use RANDOMIZE USR xxxxx where xxxxx is the address at which you loaded GENS.

# Command: C

This allows you to configure the size of the Include (on tape-only versions of Devpac) and the Macro buffers.

The Include buffer is the buffer in which text is held when assembling directly from cassette or microdrive - the larger this buffer, the more text that will be read in from cassette or microdrive at one go and therefore the faster the assmbly will proceed. On the other hand, more memory is used. This there is a compromise to be made between speed of assembly and use of memory; the `C` command allows you to control this tradeoff by giving you the opportunity of setting the size of the Include buffer.

The Macro buffer is used to hold the text of any macro definitions that you may use.

The `C` command prompts you to enter the `Include buffer` size and then the `Macro buffer` size. In both cases simply enter the number of bytes (in decimal) that you wish to allocate, followed by `[ENTER]`. If you press `[ENTER]` by itself without entering a number then no action is taken. If you specify an Include buffer size then the size is forced to be a minimum of 256 bytes. You may abort the commands using `[CAPS SHIFT] 1`.

Note that, on disc-only versions of Devpac 4, you can only change the size of the Macro buffer.

`C` does not destroy your text; it is moved up and down in memory as the buffers change in size. It is best to allocate the buffers as large as you will need them at the start of a session.

# Command: S,,d

This command allows you to change the delimiter which is taken as separating the arguments in the command line. On entry to the editor the comma `,` is taken as the delimiter; this may be changed by the use of the `S` command to the first character of the specified string `d`.

Remember that once you have defined a new delimiter it must be used (even within the `S` command) until another one is specified. Please do not confuse this command with the Substitute sub-command within edit mode.

Note that the separator may not be a space.

## Command: Un

Allows you to set the top of memory to `n`. If `n` is not present (i.e. you just type `U [ENTER]`) then the current top of memory is displayed.

GENS4 will not allow your textfile or your object code to grow above top of momory and will report an error message if either gets close to it.

By default, the top of memory is taken initially as the top of the Spectrum's system stack.

## Command: V

The `V` command gives a display of useful information; the values of the default command parameters `N1` and `N2`, the default command delimiter, the start and end of text (in decimal) and the value of the first command string, `S1`.

## Command: W n,m

The `W` command causes the section of text between lines `n` and `m` inclusive to be output to the printer. If both `n` and `m` are defaulted then the whole textfile will be printed. The printing will pause after the number of lines set by the `K` command - press any key to continue printing.

## Command: X n

The `X` command gives a catalogue of microdrive `n`. In 51 column mode, the screen is cleared first. The catalogue is always displayed in 32 columns.

## Command: Z

The `Z` command effectively deletes all your text and therefore asks you whether you are sure that you want to proceed.; answer `Y` or `y` to delete the text. Apart from being a quick shortcut for `D1,32767`, the Zap command allows you to clean up your textfile if it has somehow become corrupted. For example, you might have loaded a code file in error.

The `Z` command obviates the need for a cold start entry point into GENS4.

## Command: H

Displays the help screen which is a list of the commands available in two columns with a capital letter indicating the command letter e.g. the command `V` displays the `current Values` of certain important parameters.

# 3.3 An Example of the use of the Editor

Let us assume that you have typed in the following program (using `I10,10`):

```
10 *h   16 BIT RANDOM NUMBERS
20
30 ;INPUT:  HL contains previous random number or seed.
40 ;OUTPUT: HL contains new randon number.
50
60 Random PUSH AF     ;save registers
70 PUSH BC
80          PUSH HL
90          ADD  HL,HL  ;*2
100        ADD  HL,HL  ;*4
110        ADD  HL,HL  ;*8
120        ADD  HL,HL  ;*16
130        ADD  HL,HL  ;*32
140        ADD  HL,HL  ;*64
150        PIP  BC     ;old random number
160        ADD  HL,DE
170        LD   DE,41
180        ADD  HL,DE
190        POP  BC     ;restore registers
200        POP  AF
210        REY
```

This program has a number of errors which are as follows:

Line 10: a lower case `h` has been used in the assembler command `*H`.
Line 40: `randon` instead of `random`.
Line 70: `PUSH BC` starts in the label field.
Line 150: `PIP` instead of `POP`.
Line 160: needs a comment (not an error - merely style).
Line 210: `REY` should be `RET`.

Also 2 extra lines of `ADD HL,HL` should be added between lines 140 and 150 and all references to the register pair `DE` in lines 160 and 180 should be to register pair `BC`.

To put all this right we can proceed as follows:

`E10 [ENTER]` then (1 space) `C` (enter change mode) `H[ENTER] [ENTER]`

`F40,40,randon,random[ENTER]`
then the `S` sub-command.

`E70 [ENTER]` then `I` (insert mode) (1 space)`[ENTER] [ENTER]`

```
I142,2 [ENTER]  141   ADD  HL,HL ;*128
                142   ADD  HL,HL ;*256
                [EDIT]
```

`F150,150,PIP,POP [ENTER]`
then the `S` sub-command.

`E160 [ENTER]` then `X` (2 spaces) `;*257 + 41 [ENTER][ENTER]`

`F160,180,DE,BC [ENTER]`
then repeated use of the sub-command `S`.

`E210 [ENTER]` → (2 spaces) `C` (change mode) `T [ENTER][ENTER]`

`N10,10 [ENTER]` to renumber the text.

You are strongly recommended to work through the above example actually using the editor.

# APPENDIX 1
# ERROR NUMBERS AND THEIR MEANINGS

| | |
|---|---|
| `*ERROR* 1` | Error in the context of this line. |
| `*ERROR* 2` | Mnemonic not recognised. |
| `*ERROR* 3` | Statement badly formed. |
| `*ERROR* 4` | Symbol defined more than once. |
| `*ERROR* 5` | Line contains an illegal character i.e. one which is not valid in a particular context. |
| `*ERROR* 6` | One of the operands in this line is illegal. |
| `*ERROR* 7` | A symbol in this line is a Reserved Word. |
| `*ERROR* 8` | Mismatch of registers. |
| `*ERROR* 9` | Too many registers in this line. |
| `*ERROR* 10` | An expression that should evaluate to 8 bits evaluates to more than 8 bits. |
| `*ERROR* 11` | The instructions `JP (IX+n)` and `JP (IY+n)` are illegal. |
| `*ERROR* 12` | Error in the formation of an assembler directive. |
| `*ERROR* 13` | Illegal forward reference i.e. an `EQU`ate has been made to a symbol which has not yet been defined. |
| `*ERROR* 14` | Division by zero. |
| `*ERROR* 15` | Overflow in a multiplication operation. |
| `*ERROR* 16` | Nested macro definition |
| `*ERROR* 17` | This identifier is not a macro. |
| `*ERROR* 18` | Nested macro call. |
| `*ERROR* 19` | Nested conditional statement. |

| | |
|---|---|
| `Bad ORG!` | An ORG has been made to an address that would corrupt GENS, its textfile or the Symbol Table. Control returns to the editor. |
| `Out of Table space!` | Occurs during the first pass if insufficient memory has been allowed for the Symbol Table. Control returns to the editor. |
| `Bad Memory!` | No room for any more text to be inserted i.e. the end of text is near the top of RAM. You should save the current textfile, or part of it. |

# APPENDIX 2
# RESERVED WORDS, MNEMONICS ETC.

The following is a list of the Reserved Words within GENS. These symbols may not be used as labels although they may form part of a label. All the Reserved Words are composed of capital letters.

```
A     B     C     D     E     H     L     I     R     $
AF          AF'         BC          DE          HL    IX
IY          SP          NC          Z           NZ    M
P           PE          PO
```

There now follows a list of the valid Z80 mnemonics, assembler directives and assembler commands. These also must be entered in capital letters.

```
ADC         ADD         AND         BIT         CALL
CCF         CP          CPD         CPDR        CPI
CPIR        CPL         DAA         DEC         DI
DJNZ        EI          EX          EXX         HALT
IM          IN          INC         IND         INDR
INI         INIR        JP          JR          LD
LDD         LDDR        LDI         LDIR        NEG
NOP         OR          OTDR        OTIR        OUT
OUTD        OUTI        POP         PUSH        RES
RET         RETI        RETN        RL          RLA
RLC         RLCA        RLD         RR          RRA
RRC         RRCA        RRD         RST         SBC
SCF         SET         SLA         SRA         SRL
SUB         XOR

DEFB        DEFM        DEFS        DEFW        ELSE
END         ENT         EQU         IF          ORG
MAC         ENDM

*D          *E          *H          *L          *S
*C          *F          *M
```

# APPENDIX 3
# A WORKED EXAMPLE

There follows an example of a typical session using GENS4 - if you are a newcomer to the world of assembler programs or if you are a little unsure how to use the editor/assembler then we urge you to work through this example carefully. Note that `[ENTER]` is used to indicate that you should press the `ENTER` key on the keyboard.

## Session objective:

To write and test a fast integer multiply routine, the text of which is to be saved to tape using the editor's `T` command so that it can easily be 'included' from tape in future programs.

## Session workplan:

1. Write the multiply routine as a subroutine and save it to tape using the editor's `P` command so that it can be easily retrieved and edited during this session, should bugs be present.

2. De-bug the multiply subroutine, editing as necessary.

3. Save the de-bugged routine to tape, using the editor's `T` command so that the routine may be 'included' from tape in other programs.

Before we start we must load GENS4 into the computer - do this by typing `LOAD "" CODE 26000 [ENTER]` to load the assembler at address 26000. Now type `RANDOMIZE USR 26000 [ENTER]`. You are now in editor mode ready to create assembly programs.

## Stage 1 - write the integer multiply routine

We use the editor's `I` command to insert the text using → (the tab character) to obtain a tabulated listing. We do not need to use →, a list of the text will always perform the tabulation for us. We have not indicated where tabs have been used below but you can assume that they are used before the mnemonic and between the mnemonic and the operand. Note that the addresses shown in the example assembler listings that follow may not correspond to those produced on your machine; they serve an illustrative purpose only.

```
I10,10 [ENTER]
10 ;A fast integer multiply [ENTER]
20 ;routine. Multiplies HL [ENTER]
30; by DE. Return the result [ENTER]
40 ;in HL. C flag set on an [ENTER]
50 ;overflow. [ENTER]
60 [ENTER]
70       ORG  #7F00 [ENTER]
80 [ENTER]
90 Mult  OR   A [ENTER]
100      SBC  HL,DE ;HL>DE? [ENTER]
110      ADD  HL,DE [ENTER]
120      JR   NC,Mul ;yes [ENTER]
130      EX   DE,HL [ENTER]
140 Mul  OR   D [ENTER]
150      SCF ;overflow if [ENTER]
160      RET  NZ ;DE>255 [ENTER]
170      OR   E ;times 0? [ENTER]
180      LD   E,D [ENTER]
190      JR   NZ,MU4; ;no [ENTER]
200      EX   DE,HL ;0 [ENTER]
210      RET [ENTER]
220 [ENTER]
230 ;Main routine. [ENTER]
240 [ENTER]
250 Mu2  EX   DE,HL [ENTER]
260      ADD  HL,DE [ENTER]
270      EX   DE,HL [ENTER]
280 Mu3  ADD  HL,HL [ENTER]
290      RET  C ;overflow [ENTER]
300 Mu4  RRA [ENTER]
310      JR   NC,Mu3 [ENTER]
320      OR   A [ENTER]
330      JR   NZ,Mu2 [ENTER]
340      ADD  HL,DE [ENTER]
350      RET [ENTER]
360 [EDIT]
```

The above will create the text of the routine, now save it to tape using:

```
>P10,350,Mult [ENTER]
```

Remember to have your tape recorder running and in RECORD mode before issuing the P command.

## Stage 2 - de-bug the routine

First, let's see the text assembles correctly. We will use option 2 so that no listing is produced and no object code generated.

```
>A2 [ENTER]

*HISOFT GENS4 ASSEMBLER*
Copyright HiSoft 1983,84,87
All Rights Reserved

Pass 1 errors: 00
Pass 2 errors: 00

*WARNING* MU4  absent Table used:    74  from   161
>
```

We see from this assembly that we have made a mistake in line 190 and entered MU4 instead of Mu4 which is the label we wish to branch to. So edit line 190:

```
>F190,190,MU4,Mu4 [ENTER]
  190          JR  NZ,  (now use the S sub-command)
```

Now assemble the text again and you should find that it assembles without errors. So now we must write some code to test the routine:

```
>N300,10 [ENTER]                (renumber so that we can write
                                 some more text)
I10,10 [ENTER]
10 ;Some code to test [ENTER]
20 ;the Mult routine. [ENTER]
30 [ENTER]
40      LD   HL,50 [ENTER]
50      LD   DE,20 [ENTER]
60      CALL Mult ;Multiply [ENTER]
70      LD   A,H [o/p result [ENTER]
80      CALL Aout [ENTER]
90      LD A,L [ENTER]
100     Call Aout [ENTER]
```

```
110     RET ;Return to editor [ENTER]
120 [ENTER]
130 ;Routine to o/p A in hex [ENTER]
140 [ENTER]
150Aout PUSH AF [ENTER]
160     RRCA [ENTER]
170     RRCA [ENTER]
180     RRCA [ENTER]
190     RECA [ENTER]
200     CALL Nibble [ENTER]
210     POP  AF [ENTER]
220 Nibble   AND  %1111 [ENTER]
230     ADD  A,#90 [ENTER]
240     DAA  [ENTER]
250     ADC  A,#40 [ENTER]
260     DAA  [ENTER]
270     LD   IY,#5C30 ;for ROM [ENTER]
280     RST  #10 ;ROM call [ENTER]
290     RET [ENTER]
300 [EDIT]
>
```

Now assemble the test routine and the `Mult` routine together.

```
>A2 [ENTER]

*HISOFT GENS4 ASSEMBLER*
Copyright HiSoft 1983,84,87
All Rights Reserved

7EAC  190        RECA
*ERROR* 02      (hit any key to continue)

Pass 1 errors: 01

Table used:    88  from   210
```

We have an error in our routine; `RECA` should be `RRCA` in line 190. So:

```
>E190
  190   RECA
  190   →   (1 space) C (enter change mode) R [ENTER] [ENTER]
>
```

Now assemble again, using the default options (just use `A [ENTER]`), and the text should assemble correctly Assuming it does, we are now in a position to test the working of our `Mult` routine so we need to tell the editor where it can execute the code from. We do this with the `ENT` directive

```
>35       ENT  $ [ENTER]
```

Now assemble the text again and the assembly should terminate correctly with the messages:

```
Table used:    88  from   210
Executes: 32416
>
```

or something similar. Now we can run our code using the editor's `R` command. We should expect it to multiply 50 by 20 producing `1000` which is `#3E8` in hexadecimal.

```
R [ENTER]
0032>
```

It doesn't work! Why not? List the lines 380 to 500 (`L380,500`). You will see that at line 430 the instruction is an `OR D` followed, effectively, by a `RET NZ`. What this is doing is a logical `OR` between the `D` register and the accumulator `A` and returning with an error flag set (the `C` flag) if the result is non-zero. The object of this is to ensure that `DE`<256 so that the multiplication does not overflow - it does this by checking that `D` is zero ... but the `OR` will only work correctly in this case if the accumulator `A` is zero to start with, and we have no guarantee that this is so.

We must ensure that `A` is zero before doing the `OR D`, otherwise we will get unpredictable overflow with the higher number returned as the result. From inspection of the code we see that the `OR A` at line 380 could be made into a `XOR A` thus setting the flags for the `SBC HL,DE` instruction and setting `A` to zero. So:

```
>E380 [ENTER]
380 Mult OR   A
380 → I (enter insert mode) X [ENTER] [ENTER]
>
```

Now assemble again and run the code, using `R`. The answer should now be correct - `#3E8`.

We can further check the routine by editing lines 40 and 50 to multiply different numbers, assembling and running; you should find that the routine now works.

Now we have perfected the routine we can save it to tape in 'Include' format:

```
>T300,999,Mult [ENTER]
```

Remember to start the recorder in RECORD mode before pressing `[ENTER]`.

If you want to include the program from microdrive rather than frop tape then you do not need to use the `T` command; programs saved with the normal `P` command can be included from microdrive.

Once the routine has been saved like this it may be included in a program as shown below:

```
      RET
510
520 ;Include the Mult routine here.
530
540 *F Mult
550
560 ;The next routine.
```

When the above text is assembled the assembler will ask you to `Start tape..` when it gets to line 540 on both the first and second pass. Therefore you should have the Mult dump cued up on the tape in both cases. This will normally mean rewinding the tape after the first pass. You could record two dumps of Mult on the tape, following each other, and use one for the first pass and the other for the second pass.

When including from microdrive, no messages are displayed, it all happens automatically.

Please study the above example carefully and try it out for yourself.

# HiSoft MONS Disassembler/Debugger CONTENTS

# SECTION 1
# GETTING STARTED

MONS4 is supplied in a relocatable form; you simply load it at the address that you wish it to execute from and then enter MONS4 via that address. If you wish to enter MONS4 again (having returned from MONS4 to BASIC) then you should execute the address at which you originally loaded the debugger.

Plus 3 owners should read the additional leaflet for details of an extra debugger which lives in the RAM disc and only needs 100 bytes in the normal 48K RAM.

Example:

Say you want to load MONS4 at address `#C000` (49152 decimal) - proceed as follows:

```
LOAD "" CODE 49152 [ENTER]
RANDOMIZE USR 49152 [ENTER]
```

To enter MONS4 again use:

```
RANDOMIZE USR 49152 [ENTER]
```

MONS4 is roughly 6K in length once it has been relocated but you should allow nearer 7K bytes on loading MONS4 owing to the table of relocation addresses after the main code. MONS4 contains its own internal stack so that it is a self-contained program.

MONS4 is loaded, by default, at 55000, although you may load it at any sensible address; it is usually convenient to load MONS into high memory.

## Making a Backup Copy

Once you have loaded MONS4 into your Spectrum's memory then you can make a backup copy of the package as follows:

```
SAVE "MONS4" CODE xxxxx,6656 [ENTER]          to cassette
SAVE *"M";1;"MONS4" CODE xxxxx,6656 [ENTER]   to Microdrive
SAVE *"M";1;"MONS4" CODE xxxxx,6780 [ENTER]   to Opus Disc
```

where: `xxxxx` is the address at which you loaded MONS4.

Please note that we allow you to make a backup copy of MONS4 for your own use so that you can program with confidence. Please do not copy MONS4 to give (or worse, sell) to your friends, we supply very reasonably-priced software and a full after-sales support service but if enough people copy our software we shall not be able to continue this; *please buy, don't steal*.

Having entered MONS4 you will be presented with the *front panel* display (see the **Appendix** for an example display). This consists of the Z80 registers and flags together with their contents plus a 24 byte sections of memory centred around the Memory Pointer, initially set to address 0. On the top line of the display is a disassembly of the instruction addressed by the Memory Pointer.

On entry to MONS4, all the addresses displayed within the Front Panel are given in hexadecimal format (i.e. to base 16); you can change this so that the addresses are shown in decimal by using the command `[SYMBOL SHIFT] 3` - see the next section. Note, however, that addresses must always be *entered* in hexadecimal. Commands are entered from the keyboard in response to the prompt > under the memory display and may be entered in upper or lower case.

Some commands, whose effect might be disastrous if used in error, require you to press `[SYMBOL SHIFT]` as well as the command letter. Throughout this manual the use of the `[SYMBOL SHIFT]` key may be represented by the symbol `^` e.g. `^Z` means hold the `[SYMBOL SHIFT]` and `Z` key down together.

Commands take effect immediately - there is no need to press `[ENTER]`. Invalid commands are simply ignored. The entire 'front panel' display is updated after each command is processed so that you can observe any results of the particular command.

Many commands require the input of a hexadecimal number - when entering a hexadecimal number you may enter as many hexadecimal digits (`0-9` and `A-F` or `a-f`) as you wish and terminate them with any non-hex digit. If the terminator is a valid command then the command is obeyed after any previous command has been processed. If the terminator is a minus sign '-' then the negative of the hexadecimal number entered is returned - in two's complement form e.g. `1800-` gives `E800`. If you enter more than 4 digits when typing a hexadecimal number then only the last 4 typed are retained and displayed on the screen.

To return to BASIC from MONS4 simply press `[STOP]`, i.e. `^A`.

***Spectrum Plus 3 owners should consult the additional instructions supplied with this manual before proceeding any further.***

# SECTION 2
# THE COMMANDS AVAILABLE

The following commands are available from within MONS4. In this section, whenever [ENTER] is used to terminate a hexadecimal number this in fact can be any non-hex character (see **Section 1**). Also _ is used to denote a space where applicable.

## [SYMBOL SHIFT] 3 or # — flip hex/dec

flip the number base in which addresses are displayed between base 16 (hexadecimal) and base 10 (decimal). On entry to MONS4, addresses are shown in hexadecimal, use ^3 to flip to a decimal display and ^3 again to revert to the hexadecimal format. This affects all addresses displayed by MONS4 including those generated by the dis-assembler but it does not change the display of memory contents - this is always given in hexadecimal.

## [SYMBOL SHIFT] 4 or $ — page disassembly

displays a page of dis-assembly starting from the address held in the Memory Pointer. Useful to look ahead of your current position to see what instructions are coming up. Hit ^4 again or [EDIT] to return to the Front Panel display or another key to get a further page of dis-assembly.

## [ENTER] — forward 1

increment the Memory Pointer by one so that the 24 byte memory display is now centred around an address one greater than it was previously.

## ↑ — back 1

decrement the Memory Pointer by one.

## ← — back 8

decrement the Memory Pointer by eight - used to step backwards quickly.

## → forward 8

increment the Memory Pointer by eight - used to step forwards quickly.

## , (comma) get stack

update the Memory Pointer so that it contains the address currently on the stack (indicated by SP). This is useful when you want to look around the return address of a called routine etc.

## G Get pattern

search memory for a specified string.

You are prompted with a : and you should then enter the first byte for which you want to search followed by [ENTER]. Now keep entering subsequent bytes (and [ENTER]) in response to the : until you have defined the whole string.

Then just press [ENTER] in response to the :;this will terminate the definition of the string and search memory, starting from the current Memory Pointer address, for the first occurrence of the specified string. When the string is found the front panel display will be updated so that the Memory Pointer is positioned at the first character of the string. Example:

Say that you wish to search memory, starting from #8000, for occurrences of the pattern #3E #FF (2 bytes) - proceed as follows:

| | |
|---|---|
| M:8000 [ENTER] | set the Memory Pointer to #8000. |
| G:3E [ENTER] | define the first byte of the string. |
| FF [ENTER] | define the second byte of the string. |
| [ENTER] | terminate the string. |

After the final [ENTER] (or any non-hex character) G proceeds to search memory from #8000 for the first occurrence of 3E #FF. When found the display is updated - to find subsequent occurrences of the string use the N command.

## H convert to Hex

You are prompted with : to enter a decimal number terminated by any non-digit (i.e. any character other than 0..9 inclusive). Once the number has been terminated, an = sign is displayed on the same line followed by the hexadecimal equivalent of the decimal number. Now hit any key to return to the command mode.

Example:

```
H:41472_=A2000
```
here a space was used as the terminator.

## I Intelligent copy

his is used to copy a block of memory from one location to another - it is intelligent in that the block of memory may be copied to locations where it would overlap its previous locations. I prompts for the inclusive start and end addresses of the block to be copied (First:,Last:) and then for the address to which the block is to be moved (To:); enter hexadecimal numbers in response to each of these prompts. If the start address is greater than the end address then the command is aborted - otherwise the block is moved as directed.

## J Jump to address

execute code from a specified address.

This command prompts, via :, for a hexadecimal number - once this is entered the internal stack is reset, the screen cleared and execution transferred to the specified address. If you wish to return to the front panel after executing code then set a breakpoint (see the W command) at the point where you wish to return to the display.

Example:

```
J:B000 [ENTER]
```
executes the code starting at #B000

You may abort this command before you terminate the address by using [EDIT]. Note that J corrupts the Z80 registers before executing the code; thus the machine code program should make no assumptions as to the values held in registers. If you wish to execute code with the registers set to particular values then you should use the [SYMBOL SHIFT] K command - see below.

continue execution from the address currently held in the Program Counter (PC).

This command will probably be used most frequently in conjunction with the W command - an example should help to clarify this usage:

say you are single-stepping (using ^Z) through the code given below and you have reached address #8920. you are now not interested in stepping through the subroutine at #9000 but wish to see how the flags are set up after the call to the subroutine at #8800.

```
891E 3EFF              LD   A,-1
8920 CD0090            CALL #9000
8923 2A0080            LD HL,(#8000)
8926 7E                LD A,(HL)
8927 111488            LD DE,#8814
892A CD0088            CALL #8800
892D 2003              JR NZ,labl
892F 320280            LD (#8002),A
8932 211488 labl       LD   HL,#8814
```

Proceed as follows: set a breakpoint, using W, at location #892D (remember to use M first to set the Memory Pointer) and then issue a ^K command. Execution continues from the address held in the PC which, in this case, is #8920. Execution will then continue until the address at which the breakpoint was set (#892D) at which point the display will be updated and you can inspect the state of the flags etc. after the call to the subroutine at #8800. Then you can resume single-stepping through the code.

So ^K is useful for executing code without first resetting the stack or corrupting the registers, as J does.

## L **List memory**

tabulate, or list, a block of memory starting from the address currently held in the Memory Pointer.

L clears the screen and displays the hexadecimal representation and ASCII equivalents of the 80 bytes of memory starting from the current value of the Memory Pointer. Addresses will be shown in either hexadecimal or decimal depending on the current state of the Front Panel (see ^3 above).

The display consists of 20 rows with 4 bytes per row, the ASCII being shown at the end of each row. For the purposes of the ASCII display and values above 127 are decremented by 128 and any values between 0 and 31 inclusive are shown as ..

At the end of a page of the list you have the option of returning to the main front panel display by pressing [EDIT] or continuing with the next page of 80 bytes by pressing any other key.

## M — set Memory address

set the Memory Pointer to a specified address.

You are prompted with : to enter a hexadecimal address (see **Section 1**). The Memory Pointer is then updated with the address entered and the memory display of the front panel changes accordingly.

M is useful as a prelude to entering code, tabulating memory etc.

## N — Next pattern

find the next occurrence of the hex string last specified by the G command.

G allows you to define a string and then searches for the first occurrence of it; if you want further occurrences of the string then use N. N begins searching from the Memory Pointer and updates the memory display when the next occurrence of the string is found.

## O — relative Offset

go to the destination of a relative displacement.

The command takes the byte currently addressed by the Memory Pointer, treats it as a relative displacement and updates the memory display accordingly.

Example:

say the Memory Pointer is set to #6800 and that the contents of locations #67FF and #6800 are #20 and #16 respectively - this could be intepreted as a JR NZ,$+24 instruction. To find out where this branch would go on a Non-Zero condition simply press O when the Memory Pointer is addressing the displacement byte #16. The display will then update to centre around #6817, the required destination of the branch.

Remember that relative displacements of greater then #7f (127) are treated as negative by the Z80 processor; O takes this into account.

See also the U command in connection with O.

## P — fill memory

fill memory between specified limits with a specified byte.

P prompts for First:, Last: and With:. Enter hexadecimal numbers in response to these prompts; respectively, the start and end addresses (inclusive) of the block that you wish to fill and the byte with which you want to fill the block of memory.

Example:

```
P
First:7000 [ENTER]
Last:77FF [ENTER]
With:55 [ENTER]
```

will fill locations #7000 to #77FF (inclusive) with the byte #55 (U).

If the start address is greater than the end address then P will be aborted.

## Q — flip register sets

On entry to the front panel display the set of registers displayed is the Standard register set (AF, HL, DE, BC). The use of Q will display the Alternate register set (AF', HL', DE', BC') which is distinguished from the Standard set by the single quote ' after the register name.

If Q is used when the Alternate register set is displayed then the Standard set will be shown.

set a breakpoint after the current instruction and continue execution.

Example:

```
9000 B1      OR   A
9001 C20098  CALL NZ,#9800
9004 010000  LD   BC,0
9800 21FFFF  LD   HL,-1
```

You are single-stepping the above code and have reached `#9001` with a non-zero value in register `A`, thus the Zero flag will be in a `NZ` state after the `OR A` instruction. If you now use `^Z` to continue single-stepping then execution will continue at address `#9800`, the address of the subroutine. If you do not wish to single-step through this routine then issue the `^T` command when at address `#9001` and the `CALL` will be obeyed automatically and execution stopped at address `#9004` for you to continue single-stepping.

Remember, `^T` sets a breakpoint after the current instruction and then issues a `^K` command.

See the `^Z` command for an extended example of single-stepping.

## T **disassemble**

dis-assemble a block of code, optionally to the printer and/or microdrive.

You are first prompted to enter the `First:` and `Last:` addresses of the code that you wish to dis-assemble; enter these in hexadecimal as detailed in **Section 1**. If the start address is greater than the end address then the command is aborted. After entering these addresses you will be prompted with `Printer?`; answer `Y` (capital `Y` only) to direct the dis-assembly to your Printer stream or any other value to send output to the screen.

Now you are prompted with `Text:`to enter, in hexadecimal, the start address of any textfile that you wish the dis-assembler to produce. If you do not want a textfile to be generated then simply press `[ENTER]` after this prompt. If you specify an address then a textfile of the dis-assembly will be produced, starting at that address, in a form suitable for use by GENS4. If you want to load a textfile with GENS4 then you should note down the start and end addresses of the text file, return to BASIC and save the text as a `CODE` file.

You will then be able to load it into the GENS4 editor directly using the G command.

Alternatively, instead of typing an address in response to Text:, you can enter a microdrive filename and the text will be saved directly to microdrive as it is disassembled. Example:

```
Test: 2:DCODE [ENTER]
```

will save the disassembly to microdrive 2 under the name DCODE. No listing will be produced on the screen if you are disassembling to microdrive. The file produced on the microdrive is directly-loadable using the assembler's G editor command.

If, at any stage when you are generating a textfile, the text would overwrite MONS4 then the dis-assembly is aborted - press any key to return to the Front Panel.

If you specified a textfile address you are now asked to specify a Workspace: address - this should be the start of a spare area of memory which is used as a primitive symbol table for any labels generated by the dis-assembler. The amount of memory needed is 2 bytes for each label generated. If you default by simply hitting [ENTER] then 4K of space below MONS4 is allocated.

After this, you are asked repeatedly for the First: and Last: (inclusive) addresses of any data areas that exist within the block that you wish to dis-assemble. Data areas are areas of, say, text that you do not wish to be interpreted as Z80 instructions - instead these data areas cause DEFB assembler directives to be generated by the dis-assembler.

If the value of the data byte is between 32 and 127 (#20 and #7F) inclusive then the ASCII interpretation of the byte is given e.g. #41 is changed to A] after a DEFB. When you have finished specifying data areas, or if you do not wish to specify any, simply type [ENTER] in response to both prompts. The T command uses an area at the end of MONS4 to store the data area addresses and so you may set as many data areas as there is memory available; each data area requires 4 bytes of storage. Note that using T destroys any breakpoints that were previously set - see the W command.

The byte after a RST 8 instruction is disassembled as DEFB nsince this byte is picked up by the Spectrum ROM and never executed.

The screen will now be cleared. If you asked for a textfile to be created then there will be a short delay (depending on how large a section of memory you wish to dis-assemble) while the symbol table is constructed during pass 1.

This having been done, the dis-assembly listing will appear on the screen or printer unless you are disassembling to microdrive when no listing is produced. You may pause the listing at the end of a line by hitting [ENTER] or [SPACE], subsequently hit [EDIT] to return to the front panel display or any other key to continue the dis-assembly.

If an invalid opcode is encountered then it is dis-assembled as NOP and flagged with an asterist * after the opcode in the listing. At the end of the dis-assembly the display will pause and, if you have asked for a textfile to be produced, the message End of text xxxxx will be displayed.

When the dis-assembly has finished, press any key to return to the front panel.

Labels are generated, where relevant (e.g. in C30078), in the form LXXXX where XXXX is the absolute hex address of the label, but only if the address concerned is within the limits of the dis-assembly. If the address lies outside this range then a label is not generated, simply the hexadecimal or decimal address is given.

For example, if we were dis-assembling between #7000 and #8000, then the instruction C30078 would be dis-assembled as JP L7800; on the other hand, if we were dis-assembling between #9000 and #9800 then the C30078 instruction would be dis-assembled as JP #7800 or JP 30720 if a decimal display is being used. If a particular address has been referenced in an instruction within the dis-assembly then its label will appear in the label field (before the mnemonic) of the dis-assembly of the instruction at that address but only if the listing is directed to a textfile.

Example:

```
T
First: 8B [ENTER]
Last:9E [ENTER]
Printer?Y
Text: [ENTER]
First:95 [ENTER]
Last:9E [ENTER]
First: [ENTER]
Last [ENTER]
```

would produce something like:

```
008B FE16          CP   #16
008D 3801          JR   C,L0090
008F 23            INC HL
0090 37            SCF
0091 225D5C        LD   (#5C5D),HL
0094 C9            RET
0095 BF524E        DEFB #BF,"R","N"
0098 C4494E        DEFB #C4,"I","N"
009B 4B4559        DEFB "K","E","Y"
009E A4            DEFB #A4
```

## U back to offset

used in conjunction with the O command.

Remember that O updates the memory display according to a relative displacement i.e. it shows the effect of a JR or DJNZ instruction. U is used to update the memory display back to where the last O was issued. Example:

```
 7200 47                71F3 77
 7201 20                71F4 C9
>7202 F2<              >71F5 F5<
 7203 06                71F6 C5
display 1              display 2
```

You are on display 1 and wish to know where the relative jump 20 F2 branches. So you press O and the memory display updates to display 2.

So you press O and the memory display updates to display 2. Now you investigate the code following #71F5 for a while and then wish to return to the code following the original relative jump in order to see what happens if the zero flag is set. So press U and the memory display will return to display 1. Note that you can only use U to return to the last occurrence of the O command, all previous uses of O are lost.

used in conjunction with the X command.

V is similar to the U command in effect except that it updates the memory display to where it was before the last X command was issued. Example:

```
 8702 AF            842D 18
 8703 CD            842E A2
>8704 2F<          >842F E5<
 8705 44            8430 21
display 1          display 2
```

You are on display 1 and wish to look at the subroutine at #842F. So you press X with the display centred as shown; the memory display updates to display 2. You look at this routine for a while and then wish to return to the code after the original call to the subroutine. So press V and display 1 will reappear. As with U you can use this command only to reach the address at which the last X command was issued, all previous addresses at which X was used are lost.

## W set a breakpoint

A *breakpoint*, as far as MONS4 is concerned, is simply a CALL instruction into a routine within MONS4 that displays the front panel thus enabling the programmer to halt the execution of a program and inspect the Z80 registers, flags and any relevant memory locations.

Thus, if you wish to halt the execution of a program at #9876, say, then use the M command to set the Memory Pointer to #9876 and then use W to set a breakpoint at that address. The 3 bytes of code that were originally at #9876 are saved and then replaced with a CALL instruction that halts the execution when obeyed. When this CALL instruction is reached it causes the original 3 bytes to be replaced at #9876 and the front panel to be displayed with all the registers and flags in the state they were just before the breakpoint was executed. You can now use any of the facilities of MONS4 in the usual way.

### Notes on using breakpoints:

When the breakpoint is met, MONS4 will emit a tone through the Spectrum's speaker and wait for you to hit a key before returning to the Front Panel.

MONS4 uses the area, at the end of itself, that originally contained the relocation addresses in order to store breakpoint information. This means that you may set as many breakpoints as there is memory available; each breakpoint requires 5 bytes of storage. When a breakpoint is executed MONS will automatically restore the memory contents that existed prior to the setting of that breakpoint.

Note that, since the T command also uses this area, all breakpoints are lost when the T command is used. Breakpoints can only be set in RAM. Since a breakpoint consists of a 3 byte CALL instruction a certain amount of care must be exercised in certain exceptional cases e.g. consider the code:

```
 8000 3E          8008 00
 8001 01          8009 00
 8002 18          800A 06
 8003 06          800B 02
>8004 AF<         800C 18
 8005 0E          800D F7
 8006 FF          800E 06
 8007 01          800F 44
```

Assuming the code on the previous page, if you set a breakpoint at #8004 and then begin execution of the code from location #8000 then register A will be loaded with the value 1, execution transferred to #800A, register B loaded with the value 2 and execution transferred to location #8005. But #8005 has been overwritten with the low byte of the breakpoint call and thus we now have corrupted code and unpredictable results will occur. This type of situation is rather unusual but you must attempt to guard against it - in this case single- stepping the code would provide the answer; see the ^Z command below for a detailed example of single-stepping.

## X — go to indirection

used to update the Memory Pointer with the destination of an absolute CALL or JP instruction.

X takes the 16 bit address specified by the byte at the Memory Pointer and the byte at the Memory Pointer+1 and then updates the memory display so that it is centred around that address. Remember that the low order half of the address is specified by the first byte and the high order half of the address is given by the second byte - Intel format. As an example, say you wish to look at the routine that the code CD0563 calls; set the Memory Pointer (using M) so that it addresses the 05 within the CALL instruction and then press X. The memory display will be updated so that it is centred around location #6305. See also the V command in connection with X.

Y gives you a new line on which you can enter ASCII characters directly from the keyboard. These characters are echoed and their hexadecimal equivalents are entered into memory starting from the current value of the Memory Pointer. The string of characters should be terminated by [EDIT] and DELETE ([CAPS SHIFT] 0) may be used to delete characters from the string.

When you have finished entering the ASCII characters (and typed [EDIT]) then the display is updated so that the Memory Pointer is positioned just after the end of the string as it was entered into memory.

## [SYMBOL SHIFT] Z single-step

Prior to the use of ^Z (or ^T) the Program Counter (PC) must be set to the address of the instruction that you wish to execute.

^Z simply executes the current instruction and then updates the front panel to reflect the changes caused by the executed instruction.

Note that you can single-step anywhere in the memory map (RAM or ROM) but that you cannot single-step the Interface 1 ROM.

There now follows an extended example which should clarify the use of many of the debugging commands available within MONS4 - you are urged to study it carefully and try it out for yourself.

# A Worked Example

Let us assume that we have the 3 sections of code shown below in the machine, the first section is the main program which loads HL and DE with numbers and then calls a routine to multiply them together (the second section) with the result in HL and finally calls a routine twice to output the result of the multiplication to the screen (third section). The code follows on the next page.

```
7080 2A0071          LD   HL,(#7200)   ;SECTION 1
7083 ED5B0272        LD   DE,(#7202)
7087 CD0071          CALL Mult
708A 7C              LD   A,H
708B CD1D71          CALL Aout
708E 7D              LD   A,L
708F CD1D71          CALL Aout
7092 210000          LD   HL,0

7100 AF       Mult   XOR  A            ;SECTION 2
7101 ED52            SBC  HL,DE
7103 19              ADD  HL,DE
7104 3001            JR   NC,Mul
7106 EB              EX   DE,HL
7107 B2       Mul    OR     D
7108 37              SCF
7109 C0              RET  NZ
710A B3              OR   E
710B 5A              LD   E,D
710C 2007            JR   NZ,Mu4
710E EB              EX   DE,HL
710F C9              RET
7110 EB       Mu2    EX   DE,HL
7111 19              ADD  HL,DE
7112 EB              EX   DE,HL
7113 29       Mu3    ADD  HL,HL
7114 D8              RET  C
7115 1F       Mu4    RRA
7116 30FB            JR   NC,Mu3
7118 B7              OR   A
7119 20F5            JR   NZ,Mu2
711B 19              ADD  HL,DE
711C C9              RET

711D F5       Aout   PUSH AF           ;SECTION 3
711E 0F              RRCA
711F 0F              RRCA
7120 0F              RRCA
7121 0F              RRCA
7122 CD2671          CALL Nibble
7125 F1              POP  AF
7126 E60F   Nibble   AND  %1111
7128 C690            ADD  A,#90
712A 27              DAA
712B CE40            ADC  A,#40
712D 27              DAA
712E FD213A5C        LD   IY,#5C3A
7132 D7              RST #10
7133 C9              RET

7200 1B2A            DEFW 10779
7202 0300            DEFW 3
```

Now we wish to investigate the above code either to see if it works or maybe how it works. We can do this with the following set of commands - it should be noted that this is merely one way of stepping through the code, it is not necessarily efficient but should serve to demonstrate single-stepping:

| | |
|---|---|
| `M:7080 [ENTER]` | set Memory Pointer to `#7080`. |
| `7080.` | set Program Counter to `#7080`. |
| `^Z` | single step. |
| `^Z` | single step. |
| `^Z` | follow the [f]CALL[/f]. |
| `M:7115 [ENTER]` | skip the pre-processing of the numbers. |
| `W` | set a breakpoint. |
| `^K` | continue execution from `#7100` up to breakpoint. |
| `^Z` | single step. |
| `^Z` | follow the relative jump. |
| `^Z` | single step. |
| `^Z` | ” |
| `^Z` | ” |
| `^Z` | ” |
| `^Z` | ” |
| `^Z` | ” |
| `^Z` | ” |
| `^Z` | return from the multiply routine. |
| `^Z` | single step. |
| `^Z` | follow the `CALL`. |
| `M:7128 [ENTER]` | set Memory Pointer to interesting bit. |
| `W` | set breakpoint. |
| `^K` | continue execution from `#711D` to breakpoint. |
| `^Z` | single step. |
| `^Z` | ” |
| `^Z` | ” |
| `^Z` | ” |
| `,` | have a look at the return address |
| `W` | set breakpoint here |
| `^K` | and continue. |
| `^Z` | single step. |
| `,` | return from `Aout` routine |
| `W` | |
| `^K` | |
| `^Z` | single step. |
| `^T` | obey the whole `CALL` to `Aout`. |

Please do work through the above example, first typing in the code of the routines (see **Modifying Memory** below), or using GENS4, and then obeying the commands detailed above. you will find the example invaluable as an aid to understanding how to trace a path through a program.

this command is exactly the same as the List command except that the output goes to the Printer stream instead of to the screen. Remember that, at the end of a page, you press [EDIT] to return to the front panel or any other key to get another page.

## Modifying Memory

The contents of the address given by the Memory Pointer may be modified by entering a hexadecimal number followed by a terminator (see **Section 1**). The two least significant hex digits (if only one digit is entered then it is padded to the left with a zero) are entered into the location currently addressed by the Memory Pointer and then the command (if any) specified by the terminator is obeyed. If the terminator is not a valid command then it is ignored. Examples:

| | |
|---|---|
| F2 [ENTER] | #F2 is entered & the Memory Pointer advanced by 1. |
| 123 [CAPS SHIFT] 8 | #23 is entered & the Memory Pointer advanced by 8. |
| EM:E00_ | #0E is entered at the current Memory Pointer and then the Memory Pointer is updated to #E00. Notice that a space (_) has been used to terminate the M command here. |
| 8CO | #8C is entered and then the Memory Pointer is updated (because of the use of the O command) to the destination of the relative offset #8C i.e. to its current value - 115. |
| 2A5D_ | #5D is entered and the Memory Pointer is not changed since the terminator is a space, not a command. |

# Modifying Registers

If a hexadecimal number is entered in response to the > prompt and is terminated by a period, `.`, then the number specified will be entered into the Z80 register currently addressed by the right arrow >.

On entry to MONS4 > points to the Program Counter (`PC`) and so using `.` as a terminator to a hex number initially will modify the Program Counter. Should you wish to modify any other register then use `.` by itself (not as a terminator) and the pointer > will cycle round the registers `PC` to `AF`. Note that it is not possible to address (and thus change) either the Stack Pointer (`SP`) or the `IR` registers.

Examples:

Assume that the register pointer > is initially addressing the PC.

| | |
|---|---|
| `.` | point to `IY`. |
| `.` | point to `IX`. |
| `0.` | set `IX` to zero. |
| `.` | point to `HL`. |
| `123.` | set `HL` to `#123`. |
| `.` | point to `DE`. |
| `.` | point to `BC`. |
| `E2A7.` | set `BC` to `#E2A7`. |
| `.` | point to `AF`. |
| `FF00.` | set `A` to `#FF` and reset all the flags. |
| `.` | point to the `PC`. |
| `8000.` | set the `PC` to `#8000`. |

Note that `.` can also be used to modify the Alternate register set if this is displayed. Use the `Q` command to flip the display of the register sets.

# APPENDIX
# AN EXAMPLE
# FRONT PANEL DISPLAY

```
  710C  2007      JR   NZ,#7115
>PC 710C   20 07 EB C9 EB 19 EB
 SP D0AF   8A 70 06 03 0A 03 0D
 IY CF6A   0D 11 0C 0F 09 18 18
 IX D09F   04 03 04 00 00 00 1B
 HL 2A1B   DF FE 29 28 02 CF 02
 DE 0000   F3 AF 11 FF FF C3 CB
 BC 0004   FF C3 CB 11 2A 5D 5C
 AF 0304       V
 IR 3F7C  ON

7100  AF   7108  37   7110  EB
7101  ED   7109  C0   7111  19
7102  52   710A  B3   7112  EB
7103  19   710B  5A   7113  29
7104  30  >710C  20<  7114  D8
7105  01   710D  07   7115  1F
7106  EB   710E  EB   7116  30
7107  B2   710F  C9   7117  FB
>
```

Shown above is a fairly typical *front panel* display - the display is one obtained while single-stepping the `Mult` routine given in the example of the `Z` command.

The first 9 lines of the display contain the Z80 registers; the name of the register first (`PC` to `IR`), then (for `PC` to `BC` the value presently held in the register and finally the contents of the 7 memory locations starting from the address held in the register. The Flag register is decoded to show the flags currently set in the bit order that they are used within the register - if the Flag register was set to `#FF` then the display following `AF` would look like `00FF  SZ H VNC` i.e. the sign, zero, half-carry, parity/overflow, add/subtract and carry flags are all set. To the right of the `IR` registers is the word `ON` or `OFF` that indicates the current state of the interrupts. A Register pointer > points to the register currently addressed; see **Section 2 - Modifying Registers**.

The 24 byte memory display below the register display is organised as the address (2 bytes, 4 characters) followed by the contents (1 byte, 2 characters) of the memory at that address. The display is centred around the current Memory Pointer value, indicated by > <.

Commands (see **Section 2**) are entered at the bottom of the screen in response to the prompt >. The display is updated after each command is processed.

# Bibliography

The following books are recommended when programming the Spectrum in assembler-language.


| | | |
|---|---|---|
| The Complete Spectrum ROM Disassembly | Dr. Ian Logan & Dr. F O'Hara | Melbourne House ISBN 0 86759 117 X |
| Programming the Z80 | Rodney Zaks | Sybex 1982 |
| Mastering Machine Code on the ZX Spectrum | Toni Baker | Interface |
| Z80 Assembly Language Programming Manual | Zilog | Zilog (UK) (0628) 39200 |
| Master your ZX Microdrive | Andrew Pennell | Sunshine ISBN 0 946407 19 X |

*Printed by Jiffy Print Limited, Luton, Beds.*