HISOFT

ZX Spectrum Programmer's Manual

D E V P A C

(Typed in 1998 by Nele Abels-Ludwig. Help to keep Spectrum software usable. Transcribe technical manuals!)

(Transcribed to Rich Text Format in 1999 by Robert J. Baker using Microsoft Word 97. My thanks to Nele Abels-Ludwig — see above — without whom a difficult task would have been impossible!)

Contents **Page**

PART 2MONS3

GENS
SECTION 1 GETTING STARTED

Introduction and Loading Instructions

GENS3 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 microcomputers, GENS3 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 assembler. If you are a complete novice, work through Appendix 3 first.

GENS3 is roughly 9K 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 GENS3 code while the assembler's symbol table is created after the textfile. Thus when loading GENS3 you must allow enough room to include the assembler itself and the maximum symbol table and text size that you are likely to use in the current session. It will often be convenient, therefore, to load GENS3 into low memory.

To load GENS3 proceed as follows:

Place the supplied tape in your cassette recorder, type:

```
LOAD "" CODE xxxxx↵
```

and press PLAY on the recorder — where `xxxxx` is the decimal address at which you want GENS3 to run.

Once you have loaded the GENS3 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 execute the address `xxxxx + 56` for a cold start (destroying any text) or address `xxxxx` for a warm start (preserving any previously created textfile).

For example, say you want to load GENS3 so that it executes from address 26000 decimal — proceed as follows:

```
LOAD "" CODE 26000↵
RANDOMIZE USR 26000↵
```

To re-enter the assembler use `RANDOMIZE USR 26056` for a cold start and `RANDOMIZE USR 26000` for a warm start.

Once you have entered GENS3, 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.

Making a Backup Copy

Once you have loaded GENS3 into your Spectrum's memory then you can make a backup copy of the assembler as follows:

```
SAVE "GENS3" CODE xxxxx,10034↵      to cassette
SAVE*"M";1;"GENS3" CODE xxxxx,10034↵..... to Microdrive
```

where: `xxxxx` is the address at which you loaded GENS3.

Please note that we allow you to make a backup copy of GENS3 for your own use so that you can program with confidence. Please do not copy GENS3 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 GENS3

2.0 How GENS3 Works, Assembler Options, Listing Format etc.

GENS3 is a fast, two-pass Z80 assembler, which assembles all standard Z80 mnemonics and has added features that include macros, conditional assembly, many assembler commands and a binary-tree symbol table.

When you invoke an assembly (using the editor 'A' command — see Section 3) you will first be asked to specify 'Table size: ' in decimal. This is the amount of space that will be allocated to the symbol table during the assembly. If you default (by simply hitting ENTER) then GENS3 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 'Table size: ' you will be asked for any 'Options: ' that you require. Enter these in decimal adding the option numbers together if you want more than one option. The options available are:

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	Do not produce an assembly listing.
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.

Example: Option 36 produces a fast assembly — no listing is generated and no checks are made to see where the object code is being placed.

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 'X' 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 GENS3 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 generated during this pass unless it has been switched off by Option 4 or the assembler command *L-. The assembler listing is normally of the form:

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

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.

The following two paragraphs apply only to systems with narrow screen widths — on systems with a screen width greater than 40 each assembler listing line is contained wholly on one screen line.

After any label comes a new line — on this line the mnemonic is displayed from columns 21-24. 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 above format aids readability of the assembler listing on such a narrow screen width as that of the SPECTRUM. GENS3 does not redefine the screen width of the SPECTRUM because this would increase the space occupied by GENS3 and would be restrictive in that the standard output routines of the SPECTRUM ROM could not be used.

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 screen line. See Section 2.8 below.

Modifying the Listing Format

It is possible to modify the form in which each line of the listing is split by POKEing 3 locations within GENS3. 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 GENS3 + 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 GENS3 + 52 (#34)' gives the column position (starting at 1) at which each subsequent screen line of the assembly line is to start.
3. Location 'Start of GENS3 + 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 GENS3 at 26000 decimal. To effect these changes, execute the following POKE instructions from within BASIC:

```
POKE 26051,20↵
POKE 26052,1↵
POKE 26053,31↵
```

there must be at least one space at the start of the 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 would overwrite GENS3, the textfile 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 POKEing 'Start of GENS3 + 50' with the relevant value; the default is 2.

Control now returns to the editor.

2.1 Assembler Statement Format.

Each line of text that is to be processed by GENS3 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 below:

';'	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 GENS3 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' included all the upper and lower case alphabetics together with the characters [, \,], ^, £ and _. A label must begin with an alphabetic character. [£ is CHR$ 96 on the Speccy — RJB]

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 encountered 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 of 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 by responding to the 'Table: ' prompt 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) GENS3 takes the least significant 16 bits of the number (i.e. MOD 65535) 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 GENS3.

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 that have an 8-bit value and not a 16 bit one — this is called relative addressing. When relative addresses are specified GENS3 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 is 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
"A"+128
"y"-";"+7
(5 * label - #1000 & %1111)
17 @ %1000 .....gives 25
```

Note that the 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 = -55336 (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 invocation 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). For example, the macro:

```
MOVE    MAC
        LD      HL,=0
        LD      DE,=1
        LD      BC,=2
        LDIR
        ENDM
```

takes three parameters, source address, destination address and length, loads the relevant values in 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 three parameters to take e.g.

```
MOVE 16384,16385,4069
```

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 three 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 — subsequently use *M- to switch off this expansion.

If you run out of Macro Buffer space then a message will be displayed and the assembly aborted; use the editor's C command after saving your text, to allocate a larger macro buffer.

2.7 Assembler Directives

Certain 'pseudo-mnemonics' are recognised by GENS3. 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 GENS3.

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 'expression'. I f option 2 and option 16 are both not selected and an ORG would result in the overwriting of the GENS3 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.

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,expresion,...
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 object code, it is simply used to define an address to which the editor's R command will jump. 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 'expression'. 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 IFs 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 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 the 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 assemble text from tape to 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 cartridge 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 tape 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 inclu-

sion 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 0 to minimise the space taken up by GENS3 and its workspace.

Note that the buffer size specified in the session in which you dumped out a file to be included must be the same as the buffer size given in the session in which you are actually including the text.

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 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 the 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 turned off by one of the conditional pseudo-mnemonics then the effect of any assembler command is also turned off.

SECTION 3 THE INTEGRAL EDITOR

3.1 Introduction to the Editor.

The editor supplied with all versions of GENS3 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 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 to the user who may simply use tab control characters (CAPS SHIFT 8) 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 GENS3 is executed and displays the message:

```
Copyright Hisoft 1983,84
All rights reserved
```

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↵
```

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 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. CAPS SHIFT 5 to delete to the beginning of the line, CAPS SHIFT 8 to advance the cursor to the next tab position, CAPS SHIFT 0 to delete the previous character.

The following sub-section details 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 CAPS SHIFT 5 (delete to the beginning of the line), CAPS SHIFT 8 (go to the next tab position) and CAPS SHIFT 1 (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 GENS3 and if this buffer should become full then you will be prevented from entering any more text — you must then use CAPS SHIFT 0 or CAPS SHIFT 5 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 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 CAPS SHIFT 1.

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 '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 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 CAPS SHIFT 1 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 'L'ist 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 line to be amended, deleted, moved and renumbered:

Command: D <n,m>
All lines from n to m inclusive are deleted from the textfile. If m<n, 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 'M'ove a line of text to another position within the textfile. [It's actually a copy rather than a move as such — RJB] 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 <= x <= 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 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.

CAPS SHIFT 0 — 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.

CAPS SHIFT 5 — step the text pointer backwards to the previous tab position on each screen line.

CAPS SHIFT 8 — 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 position. 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 CAPS SHIFT 0 (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 CAPS SHIFT 8 will advance the pointer to the next tab position, inserting spaces.

X — this advances the pointer to the end of the line and automatically 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. CAPS SHIFT 0 within this sub-mode simply decrements the pointer by one i.e. moves it left while CAPS SHIFT 8 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.

Command: P n,m,s
The line range defined by n <= x <= 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     saves lines 10-200 to tape as EXAMPLE
P500,900,1:TEST     saves 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 'P'utting 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 entered the 'G' command, the message 'Start tape...' is displayed — you should now press PLAY on your recorder. A search is now 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: T n,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.

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'.

3.2.5 Assembling and Running from the Editor.

Command: A
This causes the text to be assembled from the first line in the textfile. See Section 2 for further details.

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 b 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.

3.2.6 Other Commands.

Command: B
This simply returns control to the operating system. To re-enter the assembler use either a cold or a warm start — see Section 1.

Command: C
This allows you to configure the size of the Include and 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 assembly will proceed. On the other hand, more memory is used. Thus, there is a compromise to be made between speed of assembly and use of memory; the 'C' command allows you to control this trade-off 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 command using CAPS SHIFT 1.

Please note that 'C' effectively performs a cold start on your text, deleting it. Thus, remember to save any existing text before using the 'C' command. It is best to allocate the buffers as large as you will need them at the start of a session.

Command: Q
This command allows you to convert textfiles produced by GENS1 to the compressed text format of GENS3. Simply load up the GENS1 textfile, using the 'G' command, use the 'Q' command to convert the file and then dump the

compressed file out using the 'P' command. 'Q' takes no arguments and may take a substantial time to complete the conversion of the file.

Command: S,,d
This command allows you to change the delimiter that 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 'S'ubstitute sub-command within edit mode.

Note that the separator may not be a space.

Command: V
The 'V' command displays the current values of N1, N2, S1 and S2 plus the current command delimiter i.e. the two default line numbers, the default strings and the delimiter. This is useful before entering any command in which you are going to use default values, to check that these values are correct.

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
'X' simply causes the start and end address of the textfile to be displayed in decimal. This is useful if you wish to save the text from within BASIC, or if you want to see how much memory you have left after the textfile. GENS3 always expects the text to start at the first address given by the 'X' command and holds the end address of the text in location TEXTEND which is at 'Start of GENS3 + 54'. Thus, if you wish to 'patch in' a textfile (perhaps produced by MONS3) you must move the textfile to the address specified by the first address displayed by the 'X' command, modify TEXTEND to contain the end address of the file and finally enter GENS3 via a warm start. For example, say you have generated a textfile in the correct place and that it ends (the address after the final end-of-line marker) at #9A02. Then, assuming that you have loaded GENS3 at 24064, you should, from BASIC, POKE 24064+54,2 (#02) and POKE 24064+55,154 (#9A) and then enter GENS3 by RANDOMIZE USR 24125. You will now be able to work with the textfile normally from within the editor.

Command: H,,s
Allows you to verify a textfile that has been saved to microdrive. Simply use H,,n:filename where n is the drive number and filename is the name you gave to the textfile on that drive. The file will be opened and each record in turn inspected to make sure that it can be read from the drive; record numbers are displayed as they are checked. If a particular record number fails the check then 'File not found' will be displayed and control returned to BASIC. If verification is successful then control will be returned to the assembler's editor.

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 to 180 should be to register pair BC.

To put all this right we can proceed as follows:

```
E10↵            then ▲(space) C(enter change mode)
                H↵↵
F40,40,randon,random↵          then the 'S' sub-command.
E70↵            then I(insert mode) ▲▲▲▲▲▲▲ (7 spaces)↵↵
I142,2↵         142▲▲▲▲▲▲▲ADD▲▲HL,HL▲▲▲▲▲;*128↵
                144▲▲▲▲▲▲▲▲ADD▲▲HL,HL▲▲▲▲▲;*256
CAPS SHIFT 1
F150,150,PIP,POP↵  then the 'S' sub-command.
E160↵           then X▲▲;*257 + 41↵↵
F160,180,DE,BC↵    then repeated use of the sub-command 'S'.
E210↵           C (change mode) T↵↵
N10,10↵         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 EQUate 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 immediately 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. Note that 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. Note that 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 GENS3 — if you are a newcomer to the world of assembler programs or if you are simply a little unsure how to use the editor/assembler then we urge you to work through this example carefully. Note that ↵ is used to indicate that you should press ENTER 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 sessions, should bugs be present.
2. Debug 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 GENS3 into the computer — do this by typing LOAD"" CODE 26000↵ to load the assembler at address 26000. Now type RANDOMIZE USR 26000↵. You will now be prompted with a '>' sign — you are 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 CAPS SHIFT 8 (the tab character) to obtain a tabulated listing. We do not need to use CAPS SHIFT 8, 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↵
 10 ; A fast integer multiply↵
 20 ; routine. Multiplies HL↵
 30 ; by DE. Return the result↵
 40 ; in HL. C flag set on an↵
 50 ; overflow.↵
 60↵
 70          ORG   #7F00↵
 80↵
 90 Mult     OR    A↵
100          SBC   HL,DE          ;HL>DE?↵
110          ADD   HL,DE↵
120          JR    NC,Mu1         ;yes↵
130          EX    DE,HL↵
140 Mu1      OR    D↵
150          SCF                  ;overflow if↵
160          RET   NZ             ;DE>255↵
170          OR    E              ;times 0?↵
180          LD    E,D↵
190          JR    NZ,MU4         ;no↵
200          EX    DE,HL          ;0↵
210          RET↵
220↵
230 ;Main routine↵
240↵
250 Mu2      EX    DE,HL↵
260          ADD   HL,DE↵
270          EX    DE,HL↵
280 Mu3      ADD   HL,HL↵
290          RET   C              ;overflow↵
300 Mu4      RRA↵
310          JR    NC,Mu3↵
320          OR    A↵
330          JR    NZ,Mu2↵
340          ADD   HL,DE↵
```

```
350           RET↵
360 CAPS SHIFT 1
>P10,350,Mult↵
>
```

The above will create the text of the routine and save it to tape. Remember to have your tape recorder running and in RECORD mode before issuing the 'P' command.

Stage 2 — debug the routine.

First, let's see if the text assembles correctly. We will use option 6 so that no listing is produced and no object code generated.

```
>A↵
Table size:↵ {default the symbol table size}
Options: 6↵
*HISOFT GENS3 ASSEMBLER*
Copyright Hisoft 1983,84
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↵
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↵ {renumber so that we can write some more text}
>I10,10↵
 10 ;Some code to test↵
 20 ;the Mult routine.↵
 30↵
 40           LD     HL,50↵
 50           LD     DE,20↵
 60           CALL   Mult           ;Multiply↵
 70           LD     A,H            ;o/p result↵
 80           CALL   Aout↵
 90           LD     A,L↵
100           CALL   Aout↵
110           RET                   ;Return to editor↵
120↵
130 ;Routine to o/p A in hex↵
140↵
150 Aout      PUSH   AF↵
160           RRCA↵
170           RRCA↵
180           RRCA↵
190           RECA↵
200           CALL   Nibble↵
210           POP    AF↵
220 Nibble    AND    %1111↵
230           ADD    A,#90↵
240           DAA↵
250           ADC    A,#40↵
260           DAA↵
270           LD     IY,#5C3A       ; for ROM↵
```

```
280             RST   #10             ; ROM call↵
290             RET↵
300 CAPS SHIFT 1
>
```

Now assemble the test routine and the Mult routine together.

```
>A↵
Table size:↵
Options: 6↵

*HISOFT GENS3 ASSEMBLER*
Copyright HISOFT 1983,84
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 ▲▲▲▲▲▲▲▲▲C {enter change mode} R↵↵
```

Now assemble again, using simply option 4 (no list), and then 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   $↵
```

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

```
Table used:       88 from   211
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↵
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 an XOR A thus setting the flags for the SBC HL,DE instruction *and* setting A to zero. So:

```
>E380↵
380 Mult      OR    A
380 ▲▲▲▲▲▲▲I {enter insert mode} X↵↵
```

Now assemble again (option 4) 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 and then assembling and running — you should find that the routine works perfectly.

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

```
>T300,999,Mult↵
```

Remember to start the recorder in RECORD mode before pressing ENTER. Once the routine has been saved like this it may be included in a program as shown below:

```
500             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 the second passes. 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, one following the other and use one for the first pass and the other for the second pass.

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

APPENDIX 4 AN EXTENDED CAT PROGRAM

We present below an assembly listing of a program that produces a CATalogue of a Microdrive cartridge similar to that produced by ZX BASIC'S CAT command but with extra information viz.

Type of file:

D — a data or print-type file
P — a BASIC program file
B — a CODE file
S — a string array
N — a numeric array

Also, for a CODE file, the length and start addresses are displayed in decimal while, for a program file, the length and autoexecution address are displayed in decimal.

To use the program either type in the hex codes directly (using MONS3 or POKE from BASIC) or use GENS3 to assemble the source of the program. Note that the program is positioned at 60000 — you may of course change this and re-assemble.

Once the code is in the Spectrum's memory you may execute it from within BASIC or patch it into the GENS3 editor.

To use the program from within BASIC, first POKE the Microdrive number that you wish to catalogue into location 60345 (assuming that the code starts at 60000) and then RANDOMIZE USR 60000.

To use the catalogue program from within GENS3 you must patch the dummy 'Z' command to jump to the CATalogue code. To do this patch, load up GENS3 normally and then POKE the address of the extended CATalogue routine + 2 into locations 'Start of GENS3 + 7790' and 'Start of GENS3 + 7791' (low order byte first). For example, say you have loaded GENS3 at 26000 and the extended CATalogue routine is at 60000. Then to effect the patch simply POKE 33790,98 and POKE 33791,234 and then enter GENS3 in the normal way. You can then use Zn↵ from within the editor to CATalogue drive number n.

Below is the listing of the extended CATalogue program:

```
Microdrive Extended CATalogue   28 February 1984
                    1 *H Microdrive Extended CATalogue   28 February 1984
                    2 *C+
                    3 ;Produced with Hisoft's GEN assembler
                    4
                    5 ;Copyright Hisoft 1984
                    6 ;With many thanks to Andrew Pennell for his
                    7 ;Microdrive 'bible' and the Stream 14 routine
                    8
                    9 ;Equates
                   10
000D                11 CR       EQU   13
                   12
0002                13 PRINT    EQU   2
0043                14 RECFLG   EQU   67
                   15
15D4                16 WAIT_K   EQU   #15D4
1655                17 MAKE_S   EQU   #1655
                   18
5C16                19 STRMS    EQU   #5C16
5C3A                20 ERR_NR   EQU   #5C3A
5C4F                21 CHANS    EQU   #5C4F
5C53                22 PROG     EQU   #5C53
                   23
5CD6                24 D_STR1   EQU   #5CD6
5CD8                25 S_STR1   EQU   #5CD8
5CDA                26 N_STR1   EQU   #5CDA
5CDC                27 T_STR1   EQU   #5CDC
5CE6                28 HD_00    EQU   #5CE6
5CE7                29 HD_0B    EQU   #5CE7
5CE9                30 HD_0D    EQU   #5CE9
5CEB                31 HD_0F    EQU   #5CEB
5CED                32 HD_11    EQU   #5CED
```

```
                    33
0022                34 OPEN_M   EQU   #22
0023                35 CLOSE_   EQU   #23
0031                36 NEWVAR   EQU   #31
0032                37 SHADOW   EQU   #32
                    38
                    39 ;For GENS3M2 only.
                    40
1C81                41 NUM1of   EQU   7297
                    42
                    43
EA60                44          ORG   60000
                    45
                    46 ;BASIC entry point
                    47
EA60 1817           48          JR    BasEnt
                    49
                    50 ;GENS3M2 entry point
                    51
EA62 E1             52          POP   HL
EA63 54             53          LD    D,H
EA64 5D             54          LD    E,L
EA65 E5             55          PUSH  HL
EA66 21811C         56          LD    HL,NUM1off
EA69 19             57          ADD   HL,DE
EA6A 7E             58          LD    A,(HL)
EA6B 23             59          INC   HL
EA6C B6             60          OR    (HL)
EA6D 2002           61          JR    NZ,NumSet
EA6F 3E01           62          LD    A,1
EA71 E60F           63 NumSet   AND   %00001111
EA73 2600           64          LD    H,0
EA75 6F             65          LD    L,A
EA76 22B9EB         66          LD    (DRIVE),HL
                    67
                    68 ;Initialise
                    69
EA79 210AEC         70 BasEnt   LD    HL,SPACE
EA7C 2208EC         71          LD    (POINTER),HL
EA7F FD213A5C       72          LD    IY,#5C3A
                    73
EA83 21C5EB         74          LD    HL,SIGNON
EA86 0634           75          LD    B,SIGNEND-SIGNON
EA88 CD9AEB         76          CALL  WRstring
                    77
                    78 ;Set up new channel and attach to Stream 14
                    79
EA8B 2A535C         80          LD    HL,(PROG)
EA8E 2B             81          DEC   HL
EA8F E5             82          PUSH  HL
EA90 010B00         83          LD    BC,11
EA93 CD5516         84          CALL  MAKE_SPACE
EA96 2191EB         85          LD    HL,CH14out
EA99 D1             86          POP   DE
EA9A D5             87          PUSH  DE
EA9B EB             88          EX    DE,HL
EA9C 73             89          LD    (HL),E
EA9D 23             90          INC   HL
EA9E 72             91          LD    (HL),D
EA9F 23             92          INC   HL
EAA0 EB             93          EX    DE,HL
EAA1 21FDEB         94          LD    HL,C14INFO
EAA4 010900         95          LD    BC,11-2
```

```
EAA7 EDB0        96          LDIR
EAA9 E1          97          POP   HL
EAAA 23          98          INC   HL
EAAB ED4B4F5C    99          LD    BC,(CHANS)
EAAF B7         100          OR    A
EAB0 ED42       101          SBC   HL,BC
EAB2 22325C     102          LD    (STRMS+28),HL
                103
                104 ;Now read in bare catalogue
                105
EAB5 D9         106          EXX
EAB6 E5         107          PUSH  HL
EAB7 D9         108          EXX
                109
EAB8 CF         110          RST   8
EAB9 31         111          DEFB  NEWVARS
                112
EABA 3E0E       113          LD    A,14
EABC 32D85C     114          LD    (S_STR1),A
EABF 2AB9EB     115          LD    HL,(DRIVE)
EAC2 22D65C     116          LD    (D_STR1),HL
EAC5 2A06EC     117          LD    HL,(CAT)
EAC8 22ED5C     118          LD    (HD_11),HL
EACB FB         119          EI
EACC CF         120          RST   8
EACD 32         121          DEFB  SHADOW
                122
                123 ;Now process bare catalogue
                124
EACE 210AEC     125          LD    HL,SPACE
EAD1 060B       126          LD    B,11
EAD3 CD9AEB     127          CALL  WRstring
EAD6 060F       128 CatL1    LD    B,15
EAD8 C5         129 CatLoo   PUSH  BC
EAD9 23         130          INC   HL
EADA 7E         131          LD    A,(HL)
EADB FE0D       132          CP    CR              ;finished?
EADD 2879       133          JR    Z,CatEnd
EADF 2B         134          DEC   HL
EAE0 060B       135          LD    B,11
EAE2 E5         136          PUSH  HL
EAE3 CD9AEB     137          CALL  WRstring
EAE6 E3         138          EX    (SP),HL
EAE7 23         139          INC   HL
EAE8 EB         140          EX    DE,HL
EAE9 21DC5C     141          LD    HL,T_STR1
EAEC 73         142          LD    (HL),E
EAED 23         143          INC   HL
EAEE 72         144          LD    (HL),D
EAEF 2AB9EB     145          LD    HL,(DRIVE)
EAF2 22D65C     146          LD    (D_STR1),HL
EAF5 210A00     147          LD    HL,10
EAF8 22DA5C     148          LD    (N_STR1),HL
EAFB FB         149          EI
EAFC CF         150          RST   8
EAFD 22         151          DEFB  OPEN_M
EAFE CDB5EB     152          CALL  Space
EB01 DDCB4356   153          BIT   PRINT,(IX+RECFLG)
EB05 2007       154          JR    NZ,NotPrint
EB07 3E44       155          LD    A,"D"
EB09 CDACEB     156          CALL  CONOUT
EB0C 1841       157          JR    CatBack
EB0E DDE5       158 NotPri   PUSH  IX
```

```
EB10 D1          159           POP   DE
EB11 215200      160           LD    HL,82
EB14 19          161           ADD   HL,DE
EB15 EB          162           EX    DE,HL
EB16 1A          163           LD    A,(DE)
EB17 13          164           INC   DE
EB18 21F9EB      165           LD    HL,TYPETAB
EB1B 4F          166           LD    C,A
EB1C 0600        167           LD    B,0
EB1E 09          168           ADD   HL,BC
EB1F 7E          169           LD    A,(HL)
EB20 CDACEB      170           CALL  CONOUT
EB23 CDB5EB      171           CALL  Space
EB26 EB          172           EX    DE,HL
EB27 79          173           LD    A,C
EB28 B7          174           OR    A
EB29 2013        175           JR    NZ,NotProg
EB2B 23          176           INC   HL
EB2C 23          177           INC   HL
EB2D 23          178           INC   HL
EB2E 23          179           INC   HL
EB2F 5E          180           LD    E,(HL)
EB30 23          181           INC   HL
EB31 56          182           LD    D,(HL)
EB32 23          183           INC   HL
EB33 CD66EB      184           CALL  DEOUTS
EB36 5E          185           LD    E,(HL)
EB37 23          186           INC   HL
EB38 56          187           LD    D,(HL)
EB39 CD66EB      188           CALL  DEOUTS
EB3C 1811        189           JR    CatBack
EB3E FE03        190 NotPro    CP    3
EB40 200D        191           JR    NZ,CatBack
EB42 5E          192           LD    E,(HL)
EB43 23          193           INC   HL
EB44 56          194           LD    D,(HL)
EB45 23          195           INC   HL
EB46 CD66EB      196           CALL  DEOUTS
EB49 5E          197           LD    E,(HL)
EB4A 23          198           INC   HL
EB4B 56          199           LD    D,(HL)
EB4C CD66EB      200           CALL  DEOUTS
EB4F CF          201 CatBac    RST   8
EB50 23          202           DEFB  CLOSE_M
EB51 E1          203           POP   HL
EB52 C1          204           POP   BC
EB53 1083        205           DJNZ  CatLoop
EB55 C3D6EA      206           JP    CatL1
                 207
EB58 C1          208 CatEnd    POP   BC
EB59 2B          209           DEC   HL
EB5A 0605        210           LD    B,5
EB5C CD9AEB      211           CALL  WRstring
EB5F D9          212           EXX
EB60 E1          213           POP   HL
EB61 D9          214           EXX
EB62 010000      215           LD    BC,0
EB65 C9          216           RET
                 217
                 218 ;Output DE in decimal
                 219
EB66 E5          220 DEOUTS    PUSH  HL
EB67 DDE5        221           PUSH  IX
```

```
EB69 EB          222           EX    DE,HL
EB6A 0605        223           LD    B,5
EB6C DD21BBEB    224           LD    IX,TENTAB
EB70 DD5E00      225 Deloop    LD    E,(IX)
EB73 DD5601      226           LD    D,(IX+1)
EB76 3EFF        227           LD    A,-1
EB78 3C          228 TenLoo    INC   A
EB79 B7          229           OR    A
EB7A ED52        230           SBC   HL,DE
EB7C 30FA        231           JR    NC,TenLoop
EB7E 19          232           ADD   HL,DE
EB7F F630        233           OR    #30
EB81 CDACEB      234           CALL  CONOUT
EB84 DD23        235           INC   IX
EB86 DD23        236           INC   IX
EB88 10E6        237           DJNZ  DEloop
EB8A CDB5EB      238           CALL  Space
EB8D DDE1        239           POP   IX
EB8F E1          240           POP   HL
EB90 C9          241           RET
                 242
                 243 ;Output for Stream 14
                 244
EB91 2A08EC      245 CH14ou    LD    HL,(POINTER)
EB94 77          246           LD    (HL),A
EB95 23          247           INC   HL
EB96 2208EC      248           LD    (POINTER),HL
EB99 C9          249           RET
                 250
                 251 ;Write a string of length B from (HL)
                 252
EB9A 7E          253 Wrstri    LD    A,(HL)
EB9B CDACEB      254           CALL  CONOUT
EB9E 23          255           INC   HL
EB9F 10F9        256           DJNZ  WRstring
EBA1 C9          257           RET
                 258
                 259 ;Open a stream
                 260
EBA2 E5          261 ChOpen    PUSH  HL
EBA3 D5          262           PUSH DE
EBA4 C5          263           PUSH BC
EBA5 CD0116      264           CALL #1601
EBA8 C1          265           POP   BC
EBA9 D1          266           POP   DE
EBAA E1          267           POP   HL
EBAB C9          268           RET
                 269
                 270 ;Output to stream 2
                 271
EBAC F5          272 CONOUT    PUSH  AF
EBAD 3E02        273           LD    A,2
EBAF CDA2EB      274           CALL  ChOpen
EBB2 F1          275           POP   AF
EBB3 D7          276           RST   #10
EBB4 C9          277           RET
                 278
                 279 ;Output a space to stream 2
                 280
EBB5 3E20        281 Space     LD    A,"▲"
EBB7 18F3        282           JR    CONOUT
                 283
                 284
```

```
                    285 ;Storage
                    286
EBB9 0100           287 DRIVE    DEFW  1
                    288
EBBB 1027E803       289 TENTAB   DEFW  10000,1000,100,10,1
EBBF 64000A00
EBC3 0100
                    290
EBC5 0D             291 SIGNON   DEFB  CR
EBC6 4869736F       292          DEFM  "Hisoft Extended CAT Listing"
EBCA 66742045
EBCE 7874656E
EBD2 64656420
EBD6 43415420
EBDA 4C697374
EBDE 696E67
EBE1 0D             293          DEFB  CR
EBE2 436F7079       294          DEFM  "Copyright Hisoft 1984"
EBE6 72696768
EBEA 74204869
EBEE 736F6674
EBF2 20313938
EBF6 34
EBF7 0D0D           295          DEFB  CR,CR
EBF9                296 SIGNEN   EQU   $
                    297
EBF9 504E5342       298 TYPETA DEFM "PNSB"
                    299
EBFD C415           300 C14INF   DEFW  #15C4
EBFF 5A             301          DEFB  "Z"
EC00 28002800       302          DEFW  #28,#28,11
                    303
                    304 ;*** this value may change with new Interface 1 ROM***
                    305
EC06 581C           306 CAT      DEFW  #1C58    *)
                    307
                    308 ;*************************************************
                    309
EC08                310 POINTE   DEFS  2
EC0A                311 SPACE    DEFS  512
                    312
```

***) 2nd issue #1C52**

***) 3rd issue #1C54**

MONS
SECTION 1 GETTING STARTED

MONS3 is supplied in a relocatable form; you simply load it at the address that you wish it to execute from and then enter MONS3 via that address. If you wish to enter MONS3 again (having returned from MONS3 to BASIC) then you should execute the address at which you originally loaded the debugger.

Example:

Say you want to load MONS3 at address #C000 (49152 decimal) — proceed as follows:

```
LOAD "MONS3" CODE 49152↵
RANDOMIZE USR 49152↵
```

To enter MONS3 again use:

```
RANDOMIZE USR 49152↵
```

MONS3 is roughly 5K in length once it has been relocated but you should allow nearer 6K bytes on loading MONS3 owing to the table of relocation addresses which comes after the main code. MONS3 contains its own internal stack so that it is a self-contained program.

Making a Backup Copy.

Once you have loaded MONS3 into your Spectrum's memory then you can make a backup copy of the package as follows:

```
SAVE "MONS3" CODE xxxxx,6068↵                    to cassette
SAVE *"M";1;"MONS3" CODE xxxxx,6068↵             to Microdrive
```

where: xxxxx is the address at which you loaded MONS3.

Please note that we allow you to make a backup copy of MONS3 for your own use so that you can program with confidence. Please do not copy MONS3 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.

Once you have entered MONS3 the message '*MONS3 © Copyright 1983*' will appear for a few seconds to be replaced by a '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 section of memory centred (using '>' and '<') around the current value of the Memory Pointer which is 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 MONS3, 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 then 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 terminate them with ↵. 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.

If, at any stage, you wish to return to the BASIC interpreter from MONS then simply press CAPS SHIFT 1

IMPORTANT NOTE: MONS3 disables interrupts in order to ensure correct functioning — the user should ensure that interrupts are not enabled during a session with MONS3.

SECTION 2 THE COMMANDS AVAILABLE.

The following commands are available from within MONS3. 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
flip the number base in which addresses are displayed between base 16 (hexadecimal) and base 10 (decimal). On entry to MONS3, 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 MONS3 including those generated by the disassembler but it does not change the display of memory contents — this is always given in hexadecimal.

SYMBOL SHIFT 4 or '$'
display a page of disassembly starting from the address held in the Memory Pointer. Useful to look ahead on your current position to see what instructions are coming up. Hit ^4 again to return to the 'Front Panel' display or another key to get a further page of disassembly.

ENTER
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.

CAPS SHIFT 7
decrement the Memory Pointer by one.

CAPS SHIFT 5
decrement the Memory Pointer by eight — used to step backwards quickly.

CAPS SHIFT 8
increment the Memory Pointer by eight — used to step forwards quickly.

',' (comma)
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'
search memory for a specified string ('G'et a 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↵	set the Memory Pointer to #8000
G:3E↵	define the first byte of the string.
FF↵	define the second byte of the string.
↵	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 'N' command.

'H'
convert a decimal number to its hexadecimal equivalent.

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▲=A200 here a space was used as the terminator.

'I'
intelligent copy.

This 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'
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↵        executes the code starting at #B000
```

You may abort this command before you terminate the address by using CAPS SHIFT 5. Note that 'J' corrupts the Z80 registers before executing the code; thus the machine program should make no assumptions as to the values held in the 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.

'SYMBOL SHIFT K'
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 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'
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 equivalence 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 any 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 CAPS SHIFT 5 or continuing with the next page of 80 bytes by pressing any other key (other than CAPS SHIFT 1).

'M'
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'
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'
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 interpreted 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 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↵
Last: 77FF↵
With: 55↵
```

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.

'SYMBOL SHIFT T'
set a breakpoint after the current instruction and continue execution.

Example:

```
9000 B7          OR      A
9001 C20098      CALL    NZ,#9800
9004 010000      LD      BC,0
9008 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. [This is what in D86 is called the "ProcStep" command. — RJB]

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 a block of code, optionally to the printer.

You are first prompted to enter the 'First:' and 'Last:' addresses of the code that you wish to disassemble — 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 disassembly 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 disassembler 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 disassembly will be produced, starting at that address, in a form suitable for use by GENS3. If you want to use a textfile with GENS3 then you must either generate it at, or move it to, the first address given by the assembler editor's 'X' command because this is the address of the start of the text expected by GENS3. You must also tell GENS3 where the end of the textfile is; do this by taking the 'End of text' address given by the disassembler (see below) and patching it into the TEXTEND location of GENS3 — see the GENS3 manual, Section 3.2. Then you must enter GENS3 by the warm start entry point, to preserve the text.

If, at any stage when you are generating a textfile, the text would overwrite MONS3 then the disassembly 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 disassembler. The amount of memory needed is 2 bytes for each label generated. If you default by simply hitting ENTER then an address of #6000 (hex) is assumed.

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 disassemble. 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 disassembler. If the value of the byte is between #20 and #7E (inclusive), 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 MONS3 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 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 be disassembled) while the symbol table is constructed. This having been done, the disassembly listing will appear on the screen or printer — you may pause the listing at the end of a line by hitting ENTER or SPACE, subsequently hit CAPS SHIFT 5 to return to the 'front panel' display or any other key (except CAPS SHIFT 1) to continue the disassembly. If an invalid [or unofficial — RJB] opcode is encountered then it is disassembled as NOP and flagged with an asterisk '*' after the opcode in the listing.

At the end of the disassembly the display will pause and, if you have asked for a textfile to be produced, the message 'End of text xxxxx' will be displayed; xxxxx is the address (in hexadecimal or decimal) that should be POKEd (low order byte first) into the GENS3 location TEXTEND in order that the assembler can pick up this disassembled textfile on a warm start. When the disassembly has finished, press any key to return to the 'front panel' display, apart from CAPS SHIFT 1, which will return you to BASIC.

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 disassembly. 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 disassembling between #7000 and #8000, then the instruction C30078 would be disassembled as JP L7800; on the other hand, if we were disassembling between #9000 and #9800 then the C30078 instruction would be disassembled 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 disassembly then its label will appear in the label field (before the mnemonic) of the disassembly of the instruction at that address — but only if the listing is directed to a textfile. Example:

```
T
First:8B↵
Last:9E↵
Printer?Y
Text:↵
First:95↵
Last:9E↵
First:↵
Last:↵

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'
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. 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.

'V'
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 84          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 then 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'
sets a breakpoint at the Memory Pointer.

A 'breakpoint', as far as MONS3 is concerned, is simply a CALL instruction into a routine within MONS3 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 three 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 three 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 MONS3 in the usual way.

Notes: When the breakpoint is encountered, MONS3 will emit a short tone through the Spectrum's speaker and wait for you to hit a key before returning to the Front Panel.

MONS3 uses the area, at the end of itself, which 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 five bytes of storage. When a breakpoint is executed MONS3 will automatically restore that 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 three-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
```

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 cor-

rupted 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'
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 ["little-endian" — RJB] format. 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'
enter ASCII from the Memory Pointer.

'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 CAPS SHIFT 5 and DELETE (CAPS SHIFT 0) may be used to delete characters from the string. When you have finished entering the ASCII characters (and typed CAPS SHIFT 5) 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') both the Program Counter (PC) and the Memory Pointer 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 should ensure that interrupts are not enabled at any time. You cannot single-step the Interface 1 ROM. [MONS will crash if you attempt to single-step an unofficial instruction — RJB]

Example Session

There now follows an extended example, which should clarify the use of many of the debugging, commands available within MONS3 — you are urged to study it carefully and try it out for yourself.

Let us assume that we have the three 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).

```
7080 2A0072             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    OR    A               ; SECTION 2
7101 ED52               SBC   HL,DE
7103 19                 ADD   HL,DE
7104 3001               JR    NC,Mu1
7106 EB                 EX    DE,HL
7107 B2         Mu1     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
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↵	set Memory Pointer to #7080
7080.	set Program Counter to #7080
^Z	single step.
^Z	single step.
^Z	follow the CALL.
M:7115↵	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	ditto.
^Z	ditto.
^Z	ditto.
^Z	ditto.
^Z	ditto.
^Z	ditto.
^Z	return from multiply routine.
^Z	single step.
^Z	follow the CALL.
M:7128↵	set Memory Pointer to interesting bit.
^K	continue execution from #711D to breakpoint.
^Z	single step.
^Z	ditto.
^Z	ditto.
^Z	ditto.
,	have a look at the return address.
W	set breakpoint there

^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 GENS3, 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.

'"' SYMBOL SHIFT P
this command is exactly the same as the 'L'ist 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 CAPS SHIFT 5 to return to the 'front panel' or any other key (except CAPS SHIFT 1) 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↵	#F2 is entered and the Memory Pointer advanced by 1.
123 CAP SHIFT 8	#23 is entered and 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 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 MONS3 '>' 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 (nor thus change) 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 PC.
8000.	set the PC to #8000

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

```
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

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 actually 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.

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 on the bottom line of the screen in response to the prompt '>'. The display is updated after each command is processed.