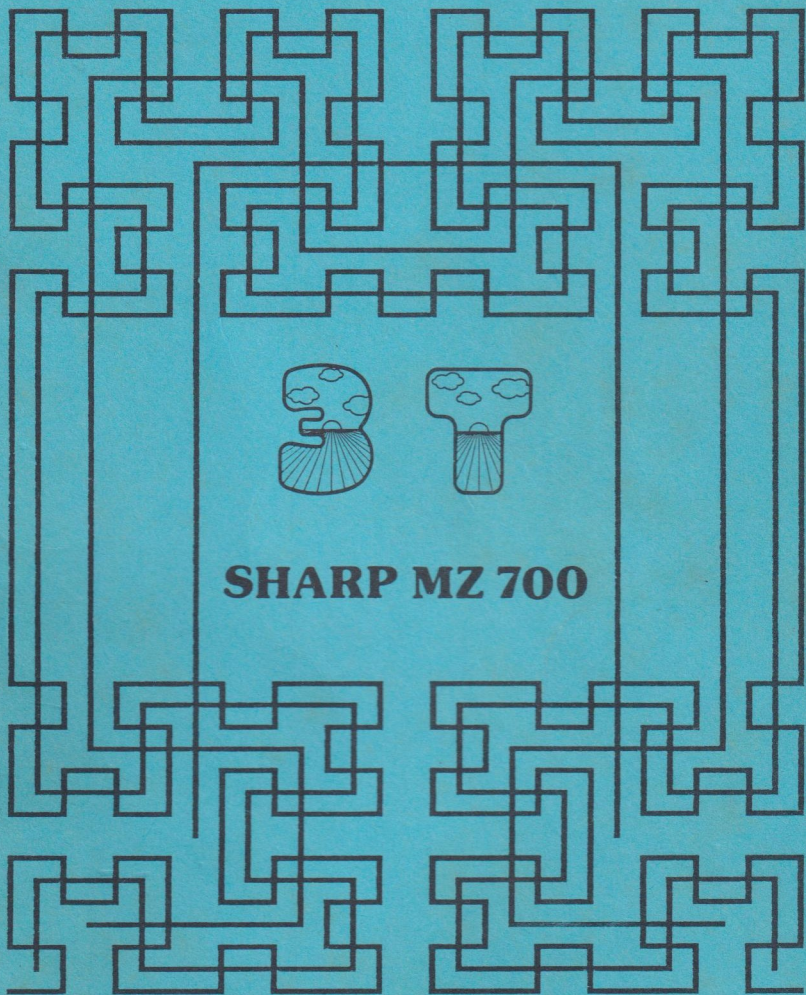


WISOT



SHARP MZ 700

DEVAPAC

CONTENTS

PAGE

PART 1

GENZ3

<u>SECTION 1</u>	<u>GETTING STARTED</u>	1
<u>SECTION 2</u>	<u>DETAILS OF GENZ3</u>	3
2.0	How GENZ3 Works	3
2.1	Assembler statement Format	5
2.2	Labels	6
2.3	Location counter	6
2.4	Symbol Table	7
2.5	Expressions	7
2.6	Assembler Directives	9
2.7	Conditional Pseudo-mnemonic	10
2.8	Assembler Commands	11
<u>SECTION 3</u>	<u>THE INTEGRAL EDITOR</u>	13
3.1	Introduction to the Editor	13
3.2	The Editor Commands	14
3.2.1	Text Insertion	14
3.2.2	Text Listing	14
3.2.3	Text Editing	15
3.2.4	Tape Commands	17
3.2.5	Assembling and Running	18
3.2.6	Other Commands	18
3.3	An example of the use of the Editor	19
<u>APPENDIX 1</u>	<u>ERROR NUMBERS AND THEIR MEANINGS</u>	21
<u>APPENDIX 2</u>	<u>RESERVED WORDS, MNEMONICS</u>	22
<u>APPENDIX 3</u>	<u>A WORKED EXAMPLE</u>	23

PART 2

MONZ3

PAGE

<u>SECTION 1</u>	<u>GETTING STARTED</u>	31
<u>SECTION 2</u>	<u>THE COMMANDS AVAILABLE</u>	33
	HASH or SHIFT/3	33
	'\$' or SHIFT/4	33
	CURSOR RIGHT or <cr>	33
	CURSOR LEFT	33
	CURSOR UP	33
	CURSOR DOWN	33
	':'(colon)	33
	'G'	34
	'H'	34
	'I'	34
	'J'	35
	'K'	35
	'L'	35
	'M'	36
	'N'	36
	'O'	36
	'P'	36
	'Q'	37
	'R'	37
	'> or SHIFT/.'	37
	'T'	38
	'U'	39
	'V'	40
	'!'-exclamation mark	41
	'W'	41
	'X'	41
	'Y'	42
	'Z'	42
	Example of single-stepping	43
	'u' or SHIFT/0	43
	Modifying Memory	45
	Modifying Registers	45
<u>APPENDIX</u>	<u>AN EXAMPLE FRONT PANEL</u> <u>DISPLAY</u>	46



© Copyright David Link, 1984

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, including photocopying and recording, without the written permission of the copyright holder. Such written permission must also be obtained before any part of this publication is stored in a retrieval system of any nature.

The information contained in this document is to be used only for modifying the reader's personal copy of DEVPAC 3.

It is an infringement of the copyright pertaining to DEVPAC 3 and associated documentation to copy, by any means whatsoever, any part of DEVPAC 3 for any reason other than for the purposes of making a security back-up copy of the object code.

SECTION 1 GETTING STARTED.

GENZ3 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, GENZ3 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.

GENZ3 is less than 8K bytes in length 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 GENZ3 code while the assembler's symbol table is created after the textfile.

To load GENZ3 into the computer simply load the GENZ3 tape into your tape recorder, label uppermost, rewind the tape to the start and type L <CR> from within the monitor and press PLAY on your tape recorder.

Once you have loaded the GENZ3 code into the computer it will auto-execute (see below); if at any subsequent time you wish to re-enter the assembler then you should execute address 1200 (use J1200 <CR> from within the monitor) for a warm start (preserving any previously created textfile).

Once you have entered GENZ3 for the first time you will first be prompted with the message 'Buffer size?': you should enter a number between 0 and 9, inclusive, followed by ENTER, or default by simply hitting ENTER alone, in which case the number 4 will be assumed. The number you enter, or default, is taken as the size of the 'Include' buffer in 256 byte units (see Section 2.8 for more details on the 'Include' option). Thus, if you want to minimise the space occupied by GENZ3 and its workspace (and don't care too much about using the 'Include' option efficiently) then you should reply to the prompt with '0' and then ENTER - this will assign the smallest possible 'Include' buffer, which is 64 bytes.

After you have replied to the 'Buffer size?' question 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.

SECTION 2 DETAILS OF GENZ3.

2.0 How GENZ3 Works.

GENZ3 is a fast, two-pass Z80 assembler which is easily adaptable to run on most Z80 systems. The assembler assembles all standard Z80 mnemonics and has added features which include 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 GENZ3 will choose a symbol table size that it thinks is suitable for the size of the text - normally this will be perfectly acceptable. Note that 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:

- | | |
|-----------|--|
| 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 GENZ3 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	7	16	22	29	34

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 2.6) 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 2.8).

The next entry, from column 7, is up to 8 characters in length (representing up to 4 bytes) and is the object code produced by the current instruction.

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

Columns 22-27 of the first line contain the first 6 characters of any label defined in this line.

The mnemonic of this line is displayed in columns 29-32 while any operands and subsequent comments are shown starting at column 34.

The assembly listing may be paused at the end of a line by hitting SHIFT/BREAK together; subsequently hit 'E' to return to the editor or any other key to continue the listing.

The only errors that can occur during the second pass are *ERROR* 10 (see Appendix 1) and 'Bad ORG!' (which occurs when the object code will overwrite GENZ3, 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 yyyy

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.

Control now returns to the editor.

2.1 Assembler Statement Format.

Each line of text that is to be processed by GENZ3 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, except for *H and *T commands.
- '' (end-of-line character) simply ignores the line.
- '' (space or tab) if the first character is a space or a tab character then GENZ3 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; 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 2.6).

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

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

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

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

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

Some examples of valid labels are:

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

2.3 Location Counter.

The assembler maintains a Location Counter so that a symbol in the label field can be associated with an address and entered into the Symbol Table. This Location Counter may be set to any value via the ORG assembler directive (see Section 2.6).

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 or the value of the expression after an EQU assembler directive) is entered into the Symbol Table. Otherwise the value is entered whenever the symbol is subsequently found in the label field.

This type of symbol table is called a Binary Tree Symbol Table and its structure enables symbols to be entered into and recovered from the table in a very short time - essential for large programs. The size of an entry in the table varies from 8 bytes to 13 bytes depending on the length of the symbol.

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

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

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

At the end of the assembly you will be given a message stating how much memory was used by the Symbol Table during this assembly - you may change how much memory is allocated to the Symbol Table 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) GENZ3 takes the least significant 16 bits of the number (i.e. MOD 65536) e.g. 70016 becomes 4480 and #5A2C4 becomes #A2C4.

A wide variety of operators are provided but no operator precedence is observed - expressions are evaluated strictly from left to right. The operators '*', '/' and '?' are provided merely for added convenience and not as part of a full expression handler which would increase the size of GENZ3.

If an expression is enclosed within parentheses then it is taken as representing a memory address as in the instruction LD HL,(loc+5) which would load the register pair HL with the 16 bit value contained at memory location 'loc+5'.

Certain Z80 instructions (JR and DJNZ) expect operands which have an 8 bit value and not a 16 bit one - this is called relative addressing. When relative addresses are specified GENZ3 automatically subtracts the value of the Location Counter at the next instruction from the value given in the operand field of the current instruction in order to obtain the relative address for the current instruction. The range of values allowed as a relative address are the Location Counter value of the next instruction -128 to +127.

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

Examples of valid expressions

```
#5000 - label
%1001101 ! %1011      gives %1000110
#3456 ? #1000          gives #456
4 + 5 * 3 - 8          gives 19
$-label+8
2345 / 7 - 1           gives 334
"A"+128
"y"-" "; "+7
(5 * label - #1000 & %1111)
17 @ %1000             gives 25
```

Note that spaces may be inserted between TERMS and OPERATORS and vice versa but not within TERMS.

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

2.6 Assembler Directives.

Certain 'pseudo-mnemonics' are recognised by GENZ3. 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 GENZ3.

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

ORG expression

sets the Location Counter to the value of 'expression'. If option 2 and option 16 are both not selected and an ORG would result in the overwriting of the GENZ3 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,expression,....

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

DEFS expression

increases the Location Counter by the value of 'expression' - equivalent to reserving a block of memory of size equal to the value of 'expression'. The block of memory that is 'skipped over' is filled with zeroes.

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

sets the execute address of the assembled object code to the value of expression - used in conjunction with the editor 'R' command (see Section 3). There is no default for the execute address.

2.7 Conditional Pseudo-mnemonic.

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.

2.8 Assembler Commands.

Assembler commands, like assembler directives, have no effect on the Z80 processor at runtime since they are not decoded into opcodes. However, unlike assembler directives, they also have no effect on the object code produced by the assembler - assembler commands simply modify the listing format.

An assembler command is a line of the source text that begins with an asterisk '*'. The letter after the asterisk determines the type of the command and must be in upper case. The remainder of the line may be any text except that the commands 'L' and 'D' expect a '+' or a '-' after the command.

The following commands are available:

*E

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

*Hs

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

*S

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

*L-

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

*L+

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

*D+

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

*D-

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

*F filename

This is a very powerful command which allows you to assemble text from tape - the textfile is read from the tape 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 8 characters) of the textfile you wish to 'include' at this point in the assembly must be specified after the 'F' and must be preceded with a space.

Any textfile that you wish to include via this option must have been previously dumped to tape using the editor's 'T' command and not the 'P' command - this is necessary because a textfile to be included 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 determined on the initial entry to GENZ3 from your response to the 'Buffer size?' prompt, see Section 1. The number you enter (from 0 to 9) following this question is taken as the size of the include buffer in 256 byte units with a default of 4*256 bytes. The ability to select the size of this buffer enables you to optimise the size/speed ratio of any inclusion of text from tape; for example, if you are not intending to use the 'F' command during an assembly then you may find it useful to specify a buffer size of 0 to minimise the space taken up by GENZ3 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.

When you use the include option the file will be read twice; once in the first and once in the second pass since the include text must be scanned in each pass. The tape is searched for a file with the correct filename and which is in 'Include' format - if a file is found which is not of the correct type or whose filename does not match that required then the name of the file will be displayed and searching continues, otherwise the name will be displayed and the file will be included, block by block.

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 GENZ3 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 <CR>), it is transferred from the buffer into the textfile. It is during this transfer that certain spaces are compressed: the line is scanned from its first character, if this is a space then a tab character is entered into the textfile and all subsequent spaces are skipped. If the first character is not a space then characters are transferred from the buffer to the textfile until a space is detected whereupon the action taken is the same as if the next character was the first character in the line. This is then repeated a further time with the result that tab characters are inserted at the front of the line or between the label and the mnemonic and between the mnemonic and the operands and between the operands and any comment. Of course, if any carriage return <CR> 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.

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 GENZ3 is executed and displays the message:

Copyright Hisoft 1983,4
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 followed by <CR>

- 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 'Delete' command) will not proceed without N1 and N2 being specified. The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line. The values of N1 and N2

are initially set to 10 and the strings are initially empty. If you enter an illegal command line such as F-1,100,HELLO then the line will be ignored and the message 'Pardon?' displayed - you should then retype the line correctly e.g. F1,100,HELLO. This error message will also be displayed if the length of S2 exceeds 20; if the length of S1 is greater than 20 then any excess characters are ignored.

Commands may be entered in upper or lower case.

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 <CR> (i.e. without any text) then that line will be deleted from the text if it exists.

The key will produce a destructive backspace (but not beyond the beginning of the text line). Text is entered into an internal buffer within GENZ3 and if this buffer should become full then you will be prevented from entering any more text - you must then use 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 <CR>. To exit from this mode use <SHIFT/BREAK>.

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 <CR>. 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 80 characters (the buffer size) then the screen will be scrolled up and you may continue typing on the next line - an automatic indentation will be given to the text so that the line numbers are effectively separated from the text.

3.2.2 Text Listing.

Text may be inspected by use of the 'L' command; the number of lines displayed at any one time during the execution of this command is fixed, at 12, initially but may be changed

through use of the 'K' command.

Command: L n,m

This lists the current text to the display device from line number *n* to line number *m* inclusive. The default value for *n* is always 1 and the default value for *m* is always 32767 i.e. default values are not taken from previously entered arguments. To list the entire textfile simply use 'L' without any arguments. Screen lines are formatted with a left hand margin so that the line number is clearly displayed. Tabulation of the line is automatic, resulting in a clear separation of the various fields with the line. The number of screen lines listed on the display device may be controlled through use of the 'K' command - after listing a certain number of lines the list will pause (if not yet at line number *m*), hit <SHIFT/BREAK> 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 128) is computed and stored. For example use K5 if you wish a subsequent 'L'ist to produce five screen lines at a time.

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.

3.2.3 Text Editing.

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

Command: D <n,m>

All lines from *n* to *m* inclusive are deleted from the textfile. If *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 <CR>.

Command: M n,m

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

Command: N <n,m>

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

Command: F n₁m₁f,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 'E' below.

Note that the line range and the two strings may have been set up previously by any other command so that it may only be necessary to enter 'F' to initiate the search - see the example in Section 3.3 for clarification.

Command: E n

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

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

' ' (space) - increment the text pointer by one i.e. point to the next character in the line. You cannot step beyond the end of the line.

 - 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.

<CR> - 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 <CR> - this will return you to the main Edit mode with the pointer positioned after the last character that you inserted. Using within this sub-mode will cause the character to the left of the pointer to be deleted from the buffer. In this mode the cursor changes to a '*'.

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 <CR> whence you are taken back to the Edit mode with the pointer positioned after the last character you changed. within this sub-mode simply decrements the pointer by one i.e. moves it left. While in Change mode the cursor changes to a '+'.
Command: S

3.2.4 Tape Commands.

Text may be saved to tape or loaded from tape using the commands 'P', 'G' and 'T'.

Command: P n,m,s

The line range defined by $n < x < m$ is saved to tape under the filename specified by the string s. Remember that these arguments may have been set by a previous command. Before entering this command make sure that your tape recorder is switched on and in RECORD mode. Do not use this command if you wish, at a later stage, to 'Include' the text - use the 'T' command instead.

Command: G,s

The tape is searched for a file with a filename of s; when found, it is loaded at the end of the current text.

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

Command: Tn,m,s

Dump out a block of text, between the line numbers n and m inclusive, to tape in a format suitable for inclusion at a later stage via the assembler command *F - see Section 2.8. The file is dumped with the filename s.

Note that you should not use this command if you simply wish to append the text later, neither should you use the 'P' command if you wish to 'include' the text.

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 be used to execute the object program. The object program can use a RET (#C9) instruction to return to the editor so long as the stack is in the same position at the end of the execution of the program as it was at the beginning. Note that ENT will have no effect if Option 16 has been specified for the assembly.

3.2.6 Other Commands.

Command: B

This simply returns control to the operating system. To re-enter the assembler use J1200 <CR> from within the monitor.

Command: S,,d

This command allows you to change the delimiter which is taken as separating the arguments in the command line. On entry to the editor the comma ',' is taken as the delimiter; this may be changed by the use of the 'S' command to the first character of the specified string d. Remember that once you have defined a new delimiter it must be used (even within the 'S' command) until another one is specified. Use 'V' to discover the current delimiter.

Note that the separator may not be a space.

Command: V

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

Command: X

'X' simply causes the start and end address of the textfile to be displayed in decimal. This is useful if you want to see how much memory you have left after the textfile.

GENZ3 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 #1209. Thus, if you wish to 'patch in' a textfile (perhaps produced by MONZ3) 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 GENZ3 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 #6A02. Then you should, from within the monitor, do the following:

```
M1209 <CR>
44 02 <CR>
36 6A <CR>
00 <SHIFT/BREAK>
J1200 <CR>
```

You will now be able to work with the textfile normally from within the 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 random 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 <CR>	then _ (space) C(enter change mode) H <CR> <CR>
F40,40,randon,randon <CR>	then the 'S' sub-command.
E70 <CR>	then I(insert mode) _ (space) <CR> <CR>
	142 _ADD HL,HL ;*128 <CR>
	144 _ADD HL,HL ;*256 <CR>
	<SHIFT/BREAK (to exit.)
F150,150,PIP,POP <CR>	then the 'S' sub-command.
E160 <CR>	then X _;*257 + 41 <CR> <CR>
F160,180,DE,BC <CR>	then repeated use of the sub-command 'S'.
E210 <CR>	then X T <CR> <CR>
N10,10 <CR>	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	This line contains an illegal character i.e. a character 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.
Bad ORG!	An ORG has been made to an address that would corrupt GENZ, 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!	This is displayed if there is 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, to tape.

APPENDIX 2 RESERVED WORDS, MNEMONICS ETC.

The following is a list of the Reserved Words within GENZ. 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								
*D	*E	*H	*L	*S	*C	*F		

APPENDIX 3 A WORKED EXAMPLE.

There follows an example of a typical session using GENZ3 - 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 <CR> is used to indicate that you should press the <CR> key on the keyboard.

Session objectives:

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' in future programs.

Session workplan:

1. Write the multiply routine as a subroutine and save it to tape using the editor's 'P' command so that it can be easily retrieved and edited during this session, should bugs be present.
2. De-bug the multiply subroutine, editing as necessary.
3. Save the de-bugged routine to tape, using the editor's 'T' command so that the routine may be 'included' in other programs.

Before we start we must load GENZ3 into the computer - do this by typing L <CR> from within the monitor and start the tape. In answer to the prompt 'Buffer size?' type 1 followed by <CR> to create an 'include' buffer of size 1*256 bytes. 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. 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 <CR>
 10 ;A fast integer multiply <CR>
 20 ;routine. Multiplies HL <CR>
 30 ;by DE. Return the result <CR>
 40 ;in HL. C flag set on an <CR>
 50 ;overflow. <CR>
 60 <CR>
 70      ORG    #7F00 <CR>
 80 <CR>
 90 Mult  OR     A <CR>
100      SBC    HL,DE ;HL>DE?<CR>
110      ADD    HL,DE <CR>
120      JR     NC,Mul1 ;yes <CR>
130      EX     DE,HL <CR>
140 Mul1  OR     D <CR>
150      SCF    ;overflow if<CR>
160      RET    NZ ;DE>255 <CR>
170      OR     E ;times 0?<CR>
180      LD     E,D <CR>
```

```

190      JR    NZ,MU4 ;no <CR>
200      EX    DE,HL ;0 <CR>
210      RET <CR>
220 <CR>
230 ;Main routine. <CR>
240 <CR>
250 Mu2.   EX    DE,HL <CR>
260      ADD   HL,DE <CR>
270      EX    DE,HL <CR>
280 Mu3    ADD   HL,HL <CR>
290      RET   C ;overflow <CR>
300 Mu4    RRA <CR>
310      JR    NC,Mu3 <CR>
320      OR    A <CR>
330      JR    NZ,Mu2 <CR>
340      ADD   HL,DE <CR>
350      RET <CR>
360 <SHIFT/BREAK>
>P10,350,MULT <CR>
>

```

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

Stage 2 - de-bug 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 <CR>
Table size: <CR>          {default the symbol table size}
Options: 6 <CR>

```

```

*HISOFT GENZ3 ASSEMBLER - SHARP MZ700*
Copyright Hisoft 1983,4
All Rights Reserved

```

Pass 1 errors: 00

Pass 2 errors: 00

```

*WARNING* MU4   absent
Table used:    74  from    162

```

>

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 <CR>
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:

```

>N310,10 <CR>          {renumber so that we can write some more text}

```



```

>I10,10 <CR>
10 ;Some code to test <CR>
20 ;the Mult routine. <CR>
30 <CR>
40     LD    HL,50 <CR>
50     LD    DE,20 <CR>
60     CALL  Mult ;Multiply <CR>
70     LD    A,H ;o/p result <CR>
80     CALL  Aout <CR>
90     LD    A,L <CR>
100    CALL  Aout <CR>
110    RET ;Return to editor <CR>
120 <CR>
130 ;Routine to o/p A in hex <CR>
140 <CR>
150 Aout    PUSH AF <CR>
160         RRCA <CR>
170         RRCA <CR>
180         RRCA <CR>
190         RECA <CR>
200         CALL Nibble <CR>
210         POP AF <CR>
220 Nibble  AND  %1111 <CR>
230         ADD  A,#90 <CR>
240         DAA  <CR>
250         ADC  A,#40 <CR>
260         DAA  <CR>
270         OUT  (#E4) <CR>
280         CALL #12 ;Monitor o/p <CR>
290         OUT  (#E0),A <CR>
300         RET <CR>
310 <SHIFT/BREAK>

```

Now assemble the test routine and the Mult routine together.

```

>A
Table size: <CR>
Options: 6 <CR>

```

```

*HISOFT GENZ3 ASSEMBLER - SHARP MZ700*
Copyright HISOFT 1983,4
All rights reserved

```

```

2853 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 <CR> <CR>

```

>

Now assemble again, using simply option 4 (no list), and the text should assemble correctly. Assuming it does, we are now in a position to test the working of our Mult routine so we need to tell the editor where it can execute the code from. We do this with the ENT directive:

```
>35      ENT  $ <CR>
```

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

```
Table used:      88  from    211
Executes: 10311
```

>

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 <CR>
0032>
```

It doesn't work! Why not? List the lines 390 to 510 (L390,510). You will see that at line 440 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 390 could be made into a XOR A thus setting the flags for the SBC HL,DE instruction and setting A to zero. So:

```
>E390 <CR>
      390 Mult    OR  A
      390 _____I{enter insert mode}X <CR> <CR>
>
```

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:

```
>T310,999,MULT <CR>
```

Remember to start the recorder in RECORD mode before pressing <CR>. 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 search the tape when it gets to line 540 on both the first and second pass. Therefore you should have the Mult dump cued up on the tape in both cases. This will normally mean rewinding the tape after the first pass. You could record two dumps of Mult on the tape, following each other, and use one for the first pass and the other for the second pass.

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

MON23

SECTION 1: INTRODUCTION

One of the main purposes of this document is to provide a clear and concise overview of the project's goals and objectives. It is intended to serve as a reference for all team members and to ensure that everyone is on the same page from the start.

The project is a collaborative effort involving several key stakeholders, including the project manager, team members, and external partners. The primary goal is to develop a comprehensive plan that addresses all aspects of the project, from initial planning to final execution.

With a well-defined plan, the project team can effectively manage resources, track progress, and address any challenges that arise. This document will serve as the foundation for all project-related decisions and actions.

The project is organized into several phases, each with its own set of tasks and deliverables. The phases are designed to be sequential, allowing the team to build upon previous work and ensure a smooth transition between stages.

Each phase is supported by a detailed schedule and resource allocation plan. This ensures that the team has a clear understanding of what needs to be done, when it needs to be done, and who is responsible for it.

The project team is committed to maintaining open communication and providing regular updates to all stakeholders. This will help to keep everyone informed and engaged throughout the project's duration.

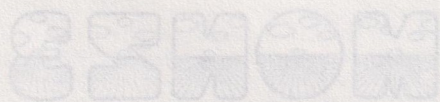
The project is a complex undertaking, but with the right plan and team, it is achievable. This document provides the roadmap for success, ensuring that all team members are aligned and working towards the same goals.

The project team is committed to delivering high-quality results on time and within budget. This document serves as a guide to ensure that all project activities are carried out in a professional and efficient manner.

The project is a collaborative effort, and the success of the project depends on the contributions of all team members. This document provides a framework for collaboration, ensuring that everyone's input is valued and that the project progresses smoothly.

The information contained in this document is for internal use only and should be handled with care. It is not to be distributed outside the project team without proper authorization.

It is an infringement of the copyright pertaining to this document to copy or reproduce any part of the document for any purpose other than the purposes of internal use. The project team reserves the right to take legal action against anyone who violates this policy.



© Copyright David Link, 1984

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, including photocopying and recording, without the written permission of the copyright holder. Such written permission must also be obtained before any part of this publication is stored in a retrieval system of any nature.

The information contained in this document is to be used only for modifying the reader's personal copy of DEVFAC 3.

It is an infringement of the copyright pertaining to DEVFAC 3 and associated documentation to copy, by any means whatsoever, any part of DEVFAC 3 for any reason other than for the purposes of making a security back-up copy of the object code.

SECTION 1 GETTING STARTED.

MONZ3 is supplied in a relocatable form; you simply load it from tape, execute it and specify the address that you wish it to execute from and it will relocate itself to that address. If you wish to enter MONZ3 again (having returned from MONZ3 to the monitor) then you should execute the specified execute address.

To load MONZ3 into the computer simply load the MONZ3 tape into your tape recorder, label side uppermost, rewind the tape and type L <CR> from within the monitor followed by PLAY on your tape recorder. Once the package has loaded (at #1200) it will execute itself and prompt you to enter the address at which you want to run MONZ3. You should enter a decimal number between 4608 (#1200) and 47000. MONZ3 will then relocate itself to the specified address and auto-execute.

MONZ3 is roughly 5K in length once it has been relocated but you should allow a little more than 6K bytes on loading MONZ3 owing to the table of relocation addresses which comes after the main code. MONZ3 contains its own internal stack so that it is a self-contained program.

Once you have entered MONZ3 the message '*MONZ3 c Copyright Hisoft 1984*' 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 32 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 dis-assembly of the instruction addressed by the Memory Pointer.

On entry to MONZ3, 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 HASH (<SHIFT/3>) - see the next section. Note, however, that addresses must always be entered in hexadecimal.

Commands are entered from the keyboard in response to the prompt '^' under the memory display and may be entered in upper or lower case.

Commands take effect immediately - there is no need to terminate them with <CR>. 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 monitor from MONZ3 then simply press <SHIFT/BREAK>.

SECTION 2 THE COMMANDS AVAILABLE.

The following commands are available from within MONZ3. In this section, whenever <CR> 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.

HASH or SHIFT/3

flip the number base in which addresses are displayed between base 16 (hexadecimal) and base 10 (denary). On entry to MONZ3, addresses are shown in hexadecimal, use SHIFT/3 to flip to a decimal display and SHIFT/3 again to revert to the hexadecimal format. This affects all addresses displayed by MONZ3 including those generated by the dis-assembler but it does not change the display of memory contents - this is always given in hexadecimal.

'\$' or SHIFT/4

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

CURSOR RIGHT or <CR>

increment the Memory Pointer by one so that the 32 byte memory display is now centred around an address one greater than it was previously.

CURSOR LEFT

decrement the Memory Pointer by one.

CURSOR UP

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

CURSOR DOWN

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

';' (colon)

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 <CR> - now keep entering subsequent bytes (and <CR>) in response to the ':' until you have defined the whole string. Then just press <CR> 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 #2000, for occurrences of the pattern #3E #FF (2 bytes) - proceed as follows;

M:2000 <CR>	set the Memory Pointer to #2000.
G:3E <CR>	define the first byte of the string.
FF <CR>	define the second byte of the string.
<CR>	terminate the string.

After the final <CR> (or any non-hex character) 'G' proceeds to search memory from #2000 for the first occurrence of #3E #FF. When found the display is updated - to find subsequent occurrences of the string use the 'N' command.

'H'

convert 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 'J', for a hexadecimal number - once this is entered 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 '?' command) at the point where you wish to return to the display.

Example:

J:B000 <CR> executes the code starting at #B000.

You may abort this command before you terminate the address by using <INST>.

'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 '?' 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 #4920. You are now not interested in stepping through the subroutine at #5000 but wish to see how the flags are set up after the call to the subroutine at #4800.

491E 3EFF	LD A,-1
4920 CD0050	CALL #5000
4923 2A0040	LD HL, (#4000)
4926 7E	LD A,(HL)
4927 111448	LD DE,#4814
492A CD0048	CALL #4800
492D 2003	JR NZ,lab1
492F 320240	LD (#4002),A
4932 211448 lab1	LD HL,#4814

Proceed as follows: set a breakpoint, using '?', at location #492D (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 #4920. Execution will then continue until the address at which the breakpoint was set (#492D) at which point the display will be updated and you can inspect the state of the flags etc. after the call to the subroutine at #4800. Then you can resume single-stepping through the code.

'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 equivalents of the 192 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 SHIFT/3 above). The display consists of 24 rows with 8 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 <INST> or continuing with the next page of 192 bytes by pressing any other key (other than SHIFT/BREAK).

'M'

set the Memory Pointer to a specified address.

You are prompted with 'i' 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 essential 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 than #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 <CR>
Last:77FF <CR>
With:55 <CR>

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.

'R'

read an object code file from tape.

You are first prompted to enter the filename of the file that you wish to load from tape and the 'Start:' address at which you want the file to load. If you wish the machine code to load at the address at which it was dumped out (from MONZ or GENZ) then simply hit <CR> in response to the 'Start:' question.

Once you have answered these two questions the tape will be searched for a machine code file with the correct filename; when found the file is loaded to the specified address.

If you wish to abort the command then simply press <INST> in response to either of the prompts.

> or SHIFT/.

set a breakpoint after the current instruction and continue execution.

Example:

5000 B7	OR A
5001 C20058	CALL NZ, #5800
5004 010000	LD BC, 0
5800 21FFFF	LD HL, -1

You are single-stepping the above code and have reached #5001 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 #5800,

the address of the subroutine. If you do not wish to single-step through this routine then issue the > command when at address #5001 and the CALL will be obeyed automatically and execution stopped at address #5004 for you to continue single-stepping.

Remember, 'X' 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'

dis-assemble 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 dis-assemble - enter these in hexadecimal as detailed in Section 1. If the start address is greater than the end address then the command is aborted. After entering these addresses you will be prompted with 'Printer?'; answer 'Y' (capital 'Y' only) to direct the dis-assembly to your Printer stream or any other value to send output to the screen.

Now you are prompted with 'Text' to enter, in hexadecimal, the start address of any textfile that you wish the dis-assembler to produce. If you do not want a textfile to be generated then simply press <CR> after this prompt. If you specify an address then a textfile of the dis-assembly will be produced, starting at that address, in a form suitable for use by GENZ3. If you want to use a textfile with GENZ3 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 GENZ3. You must also tell GENZ3 where the end of the textfile is; do this by taking the 'End of text' address given by the dis-assembler (see below) and patching it into the TEXTEND location of GENZ3 - see the GENZ3 manual, Section 3.2. Then you must enter GENZ3 by the warm start entry point, to preserve the text.

If, at any stage when you are generating a textfile, the text would overwrite MONZ3 then the dis-assembly is aborted - press any key to return to the Front Panel.

If you specified a textfile address you are now asked to specify a 'Workspace' address - this should be the start of a spare area of memory which is used as a primitive symbol table for any labels generated by the dis-assembler. The amount of memory needed is 2 bytes for each label generated. You must enter an address here and cannot default a value.

After this, you are asked repeatedly for the 'First' and 'Last' (inclusive) addresses of any data areas that exist within the block that you wish to dis-assemble. Data areas are areas of, say, text that you do not wish to be interpreted as Z80 instructions - instead these data areas cause DEFB assembler directives to be generated by the dis-assembler. If the value of the data byte is between 32 and 127 (#20 and #7F) inclusive then the ASCII interpretation of the byte is given e.g. #41 is changed to "A" after a DEFB. When you have finished specifying data areas, or if you do not wish to specify any, simply type <CR> in response to both prompts. The 'T' command uses an area at the end of MONZ3 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 '?' 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 dis-assemble)

while the symbol table is constructed. This having been done, the dis-assembly listing will appear on the screen or printer - you may pause the listing at the end of a line by hitting any key, subsequently hit <INST> to return to the 'front panel' display or any other key (except SHIFT/BREAK) to continue the dis-assembly. If an invalid opcode is encountered then it is dis-assembled as NOP and flagged with an asterisk '*' after the opcode in the listing.

At the end of the dis-assembly the display will pause and, if you have asked for a textfile to be produced, the message 'End of text xxxxx' will be displayed; xxxxx is the address (in hexadecimal or decimal) that should be POKEd (low order byte first) into the GENZ3 location TEXTEND in order that the assembler can pick up this dis-assembled textfile on a warm start. When the dis-assembly has finished, press any key to return to the 'front panel' display, apart from SHIFT/BREAK which will return you to the monitor.

Labels are generated, where relevant (e.g. in C30078), in the form LXXXX where 'XXXX' is the absolute hex address of the label, but only if the address concerned is within the limits of the dis-assembly. If the address lies outside this range then a label is not generated, simply the hexadecimal or decimal address is given. For example, if we were dis-assembling between #7000 and #8000, then the instruction C30078 would be dis-assembled as JP L7800; on the other hand, if we were dis-assembling between #9000 and #9800 then the C30078 instruction would be dis-assembled as JP #7800 or JP 30720 if a decimal display is being used. If a particular address has been referenced in an instruction within the dis-assembly then its label will appear in the label field (before the mnemonic) of the dis-assembly of the instruction at that address but only if the listing is directed to a textfile. Example:

```
T
First:8B <CR>
Last:9E <CR>
Printer?Y
Text: <CR>
First:95 <CR>
Last:9E <CR>
First: <CR>
Last: <CR>
```

```
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 44	8430 21
display 1	display 2

You are on display 1 and wish to look at the subroutine at #842F. So you press 'X' with the display centred as shown; the memory display 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.

':' - exclamation mark, SHIFT/1

sets a breakpoint at the Memory Pointer.

A 'breakpoint', as far as MONZ3 is concerned, is simply a CALL instruction into a routine within MONZ3 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 #6876, say, then use the 'M' command to set the Memory Pointer to #6876 and then use ':' to set a breakpoint at that address. The 3 bytes of code that were originally at #6876 are saved and then replaced with a CALL instruction that halts the execution when obeyed. When this CALL instruction is reached it causes the original 3 bytes to be replaced at #6876 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 MONZ3 in the usual way.

Notes:

MONZ3 uses the area, at the end of itself, that originally contained the relocation addresses in order to store breakpoint information. This means that you may set as many breakpoints as there is memory available; each breakpoint requires 5 bytes of storage. When a breakpoint is executed MONZ3 will automatically restore the memory

contents that existed prior to the setting of that breakpoint. Note that, since the 'T' command also uses this area, all breakpoints are lost when the 'T' command is used. Breakpoints can only be set in RAM. Since a breakpoint consists of a 3 byte CALL instruction a certain amount of care must be exercised in certain exceptional cases e.g. consider the code:

6000 3E	6008 00
6001 01	6009 00
6002 18	600A 06
6003 06	600B 02
>6004 AF<	600C 18
6005 0E	600D F7
6006 FF	600E 06
6007 01	600F 44

If you set a breakpoint at #6004 and then begin execution of the code from location #6000 then register A will be loaded with the value 1, execution transferred to #600A, register B loaded with the value 2 and execution transferred to location #6005. But #6005 has been overwritten with the low byte of the breakpoint call and thus we now have corrupted code and unpredictable results will occur. This type of situation is rather unusual but you must attempt to guard against it - in this case single-stepping the code would provide the answer; see the 'Z' command below for a detailed example of single-stepping.

'W'

write a block of memory to tape.

You are first prompted for the 'Name:' that you wish the file to have and then the 'First:' and 'Last:' addresses (inclusive) of the block that you wish to save to tape. When you have entered these, the machine code within the specified block of memory is saved to tape under the specified filename; remember to have your tape recorder set up in RECORD mode. The block of code may be re-loaded subsequently (at a different address if required) by using the 'R' command.

This command may be aborted at any time by pressing <INST>.

'X'

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

'X' takes the the 16 bit address specified by the byte at the Memory Pointer and the byte at the Memory Pointer +1 and then updates the memory display so that it is centred around that address. Remember that the low order half of the address is specified by the first byte and the high order half of the address is given by the second byte - Intel format. 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 <INST> and may be used to delete characters from the string. When you have finished entering the ASCII characters (and typed <INST>) 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.

'Z'

single-step.

Prior to the use of 'Z' (or 'S') 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).

There now follows an extended example which should clarify the use of many of the debugging commands available within MONZ3 - you are urged to study it carefully and try it out for yourself.

Let us assume that we have the 3 sections of code shown on the next page 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   XOR    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              ;SECTION 3
711E 0F          RRCA
711F 0F          RRCA
7120 0F          RRCA
7121 0F          RRCA
7122 CD2671      CALL    Nibble
7125 F1          POP     AF
7126 E60F        Nibble AND     %1111
7128 C690        ADD     A,#90
712A 27          DAA
712B CE40        ADC     A,#40
712D 27          DAA
712E CD1200      CALL    #12      ;monitor call
7131 C9          RET
.
.
.
7200 1B2A        DEFW    10779
7202 0200        DEFW     2

```

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 <CR>          set Memory Pointer to #7080.
7080.                set Program Counter to #7080.
Z                    single step.
Z                    single step.
Z                    follow the CALL.
M:7115 <CR>          skip the pre-processing of the numbers.
!                    set a breakpoint.
K                    continue execution from #7100 up to breakpoint.
Z                    single step.
Z                    follow the relative jump.
Z                    single step.
Z                    "
Z                    "
Z                    "
Z                    "
Z                    "
Z                    return from multiply routine.
Z                    single step.
Z                    follow the CALL.
M:7128 <CR>          set Memory Pointer to interesting bit.
!                    set breakpoint.
K                    continue execution from #711D to breakpoint.
Z                    single step.
Z                    "
Z                    "
Z                    "
:                    have a look at the return address
!                    set breakpoint there
K                    and continue.
Z                    single step.
:                    return from Aout routine
!                    by setting breakpoint
K                    and continuing.
Z                    single step.
>                    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 GENZ3, 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.

π' or SHIFT/O

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 <INS> to return to the 'front panel' or any other key (except SHIFT/BREAK) 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 <CR>	#F2 is entered and the Memory Pointer advanced by 1.
123	#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.
8C0	#8C is entered and then the Memory Pointer is updated (because of the use of the 'O' command) to the destination of the relative offset #8C i.e. to its current value - 115.
2A5D_	#5D is entered and the Memory Pointer is not changed since the terminator is a space, not a command.

Modifying Registers.

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

On entry to MONZ3 '>' points to the Program Counter (PC) and so using '.' as a terminator to a hex number initially will modify the Program Counter. Should you wish to modify any other register then use '.' by itself (not as a terminator) and the pointer '>' will cycle round the registers PC to AF. Note that it is not possible to address (and thus change) either the Stack Pointer (SP) or the IR registers.

Examples:

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

.	point to IY.
.	point to IX.
0.	set IX to zero.
.	point to HL.
123.	set HL to #123.
.	point to DE.
.	point to BC.
E2A7.	set BC to #E2A7.
.	point to AF.
FF00.	set A to #FF and reset all the flags.
.	point to the PC.
6000.	set the PC to #6000.

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

APPENDIX AN EXAMPLE FRONT PANEL DISPLAY.

```

710C 2007      JR    NZ,#7115
>PC 710C      20 07 EB C9 EB 19 EB 29 D8
SP 10EE      8A 70 B9 C1 45 C1 36 C1 15
IY 0000      C3 4A 00 C3 E6 07 C3 0E 09
IX 0000      C3 4A 00 C3 E6 07 C3 0E 09
HL 2A1B      CB 6E E1 2B 0A E5 23 CD B0
DE 0000      C3 4A 00 C3 E6 07 C3 0E 09
BC 0004      E6 07 C3 0E 09 C3 18 09 C3
AF 0304      V
IR 003E

7100 AF 710B 37 7110 EB 711B B7
7101 ED 7109 C0 7111 19 7119 20
7102 52 710A B3 7112 EB 711A F5
7103 19 710B 5A 7113 29 711B 19
7104 30 >710C 20< 7114 D8 711C C9
7105 01 710D 07 7115 1F 711D F5
7106 EB 710E EB 7116 30 711E 0F
7107 B2 710F C9 7117 FB 711F 0F
>_

```

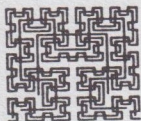
Shown above is a fairly typical 'front panel' display - the display is one actually obtained while single-stepping the Mult routine given in the example of the 'Z' command.

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 10 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.

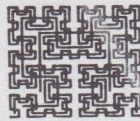
The 32 byte memory display below the register display is organised as the address (2 bytes, 4 characters) followed by the contents (1 byte, 2 characters) of the memory at that address. The display is centred around the current Memory Pointer value, indicated by '>' <.

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.



HISOFT

180 High Street North
Dunstable, Beds. LU6 1AT
Tel: (0582) 696421



NOTES

