HiSoft Devpac80

Fast Interactive CP/M Development Kit

System Requirements:
Z80 disc system running CP/M 2 or CP/M 3 with at least 36K TPA.

Copyright © HiSoft 1987
Version 2 May 1987

First printing May 1987
Second printing October 1987

Set using an Apple Macintosh™ and Laserwriter™ with Aldus Pagemaker™.

HiSoft GEN80

Fast Interactive CP/M Assembler

System Requirements:
Z80 disc system running CP/M 2 or CP/M 3 with at least 36K TPA.

Version 2 May 1987

First printing May 1987
Second printing October 1987

Set using an Apple Macintosh™ and Laserwriter™ with Aldus Pagemaker™.

Contents

SECTION 1
Introduction to GEN80

GEN80 is a fast, full-feature, macro assembler for CP/M systems. It conforms very closely to both the Microsoft M80™ and Zilog™ assembler syntaxes allowing a wide range of assembler directives and commands and producing either directly-executable `.COM` files or linkable `.REL` files. Full expression handling is included together with conditional assembly, source include and extensive error reporting.

The various sections of this manual are now described to allow you to make efficient use of them.

Section 2 of this manual is a comprehensive guide to **GEN80** giving information on how to use and get the most out of every feature. Everybody except the most experienced assembler programmer should read this section as it contains many valuable examples of the use of **GEN80**.

Section 3 is concerned with using the installation program for **GEN80**; you do not need to read this section unless you wish to change the default top-of-file options used by **GEN80**.

Section 4 is a quick reference guide to **GEN80** for use after you have familiarised yourself with the assembler.

If after reading **Section 2** you are still unsure how to use the assembler or you are unfamiliar with Z80 programming then you may find it useful to work through the **Devpac80** tutorial and/or consult one of the books given in the Bibliography.

If you are an experienced programmer then you may find that **Section 1.1** covers all the details you need to use **GEN80** easily and efficiently.

1.1For Experienced Programmers

This section is included near the front of the manual to introduce the experienced assembly language programmer to the bare essentials for assembling a file. The details that follow should enable such a programmer to get to grips with **GEN80** immediately. Naturally, the requisite section of the manual should be consulted in case of problems.

1) The normal and default filetype for **GEN80** files is `.GEN`

2) The various fields in the source file (label, mnemonic, operand) should be separated by white space. White space is defined as any number of tab or space characters.

3) Labels may be of any length and may optionally be terminated with a colon which will be stripped before entry into the symbol table.

4) Mnemonics should start in column 2 or after (thus a space or tab in column 1 is sufficient in the source file).

5) A command line of `GEN80 <filename>` will normally suffice for assembly. Alternatively you can run the assembler from the menu system provided by **HDE**, top-of-file options may be included in the first line of your program so that the only reason for using **GEN80** from outside **HDE** is to assign different drives to your source and object files. If the source file is of type `.GEN` then the type may be omitted. By default an executable (or `.COM`) file is produced which will, in this case, be on the same disc and have the same name as the source file.

6) You can obey Microsoft M80™ or Zilog™ assembler syntax with few problems; consult the **Quick Reference Guide** in case of difficulty.

SECTION 2 GEN80 Reference

2.1 Getting Started

There are two ways of invoking **GEN80** from within your CP/M system; firstly you can run the interactive editor by typing:

```
HDE TEST [RETURN]
```

where `TEST.GEN` is the file you wish to assemble. A menu will appear and you press `A` to assemble the program. **GEN80** assembles the `Main file` which can be seen on the menu. It produces, by default, an object code file on the same drive as the source file and with an extension of `.COM`, ready to run. **GEN80** then returns to the menu after asking you to hit any key. Alternatively, you can run **GEN80** straight from CP/M by typing:

```
GEN80 {object file=} source file {;options} [RETURN]
```

{ } means optional.

This allows you to specify the object file on a different disc drive from the source file (or with a different name) and lets you enter options at assembly time rather than having the options built-in to the source file.

We have included a small example file called `TEST.GEN` on your disc which you should have copied to your work disc so, using your working disc in drive A, try to two methods now. Type:

```
HDE TEST [RETURN]
A
```
any key
```
Q
```
and then type:

```
GEN80 object=test;l+ [RETURN]
```

The first method produced a file called TEST.COM (you can run it from the menu using R or from CP/M by typing TEST [RETURN]). The second method made a file called OBJECT.COM and turned the list on.

Having seen how to get **GEN80** assembling there follows a slightly technical discussion of how it works.

2.2 How GEN80 Works

GEN80 divides your available memory into three areas, one area for source text, another area for resulting object code and the third area for the Symbol Table, in that order. The size of these areas is normally fixed by the assembler in a sensible ratio although you may change the size of the Symbol Table buffer (using option *B) on any run of **GEN80.** If object code generation is inhibited then the source text is given all the available memory not allocated to the Symbol Table.

GEN80 is a two pass assembler; it begins by reading as much of the source text as will fit into the relevant memory area. This may be all of the source. The assembler then enters its first pass in which it searches for errors within the text and compiles its symbol table in memory. When the last line of text from memory has been processed **GEN80** checks to see if all the text has been read from the disc - if not, the source text area in memory is filled with fresh text from the disc and the first pass continues. This path may be altered through use of the *Include* assembler command (*I) which allows source from a specified disc file to be assembled; when the source from this new file is exhausted then assembly will continue from after the include line in the original file. This is all handled automatically and is transparent to the user.

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 message and the filename of the file in which the error was detected. The assembly then continues, displaying error messages as appropriate. It will often be useful to direct these messages to a disc file for later inspection as well as the screen (see **Section 2.3**).

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, unless you have specified that the 2nd pass be forced using the option `*F`.

If any labels were referenced in an operand field but never declared in the label field then the message

```
*WARNING* label absent
```

will now be displayed (with `label` being the name of the undeclared label).

If errors or warnings occurred and you ran **GEN80** from within the menu system then the message:

```
        Error(s) found, hit a key for the editor:
```

will appear. Hit a key and the editor will appear with your source file. You can now use the command `Goto Next Error` to skip to the rogue line, correct it, exit the editor and re-assemble from the menu. More details of this are given in the Editor and Tutorial sections.

If you ran **GEN80** from CP/M then you will be returned to CP/M.

If no warnings or errors were detected (or the 2nd pass was forced), then the assembly now proceeds to the second pass.

It is during the second pass that object code is generated, if required. Code is fed to the object code buffer in memory and if this area becomes full then it is emptied to the specified object code file on disc and re-initialised. An assembler listing, see **Section 2.13**, is generated during the second pass unless this has been switched off. The only syntax error that can occur during the second pass is the

```
Out of range
```

error and the action taken following this is the same as given above for first pass errors.

The assembler listing may be paused at any time by using [CTRL]-S and restarted by using any key except [CTRL]-C which will abort the listing.

At the end of the second pass the message:

```
Pass 2 errors: nn
```

will be displayed followed by a repeat of any warnings for any absent labels detected during the first pass. You will now be informed of how many ORG assembler directives were issued; this is done since COM files must be continuous and the use of more than one ORG implies a discontinuity of object code. The form of the message is:

```
*WARNING* ORGs used:  nn
```

Finally the assembly will terminate with the message:

```
Symbol table used:xK out of yK.
```

where x is the number of kilobytes used by the Symbol Table and y is the number of kilobytes that was allocated to the table.

Note: If at any time during the first pass the Symbol Table becomes full then it will not overflow to disc. Instead the message

```
Used all xK bytes of Symbol Table!
```

will be reported and the assembly aborted.

If errors were detected on the second pass then the action taken is the same as that at the end of the first pass i.e. you are either returned to the editor of CP/M depending on how you invoked **GEN80**.

2.3 Top-of-File Options

There are a large number of options available within **GEN80** for controlling the assembly process. They may be broadly divided into those that must appear at the top of the source file and those that may appear anywhere in the source file.

The top-of-file options are described here and the others, which are called assembler commands, appear later. There are three of the top-of-file options that belong to both groups and these are accordingly described in both sections.

There are two places that are considered the top of the file:

1) On the command line when using **GEN80** directly from CP/M. When the options appear here, they must be preceded by a ; and separated from each other by tabs, spaces or commas e.g.

   ```
   GEN80 test;N, R+, K  [RETURN]
   ```

2) On the very first line of the source file. When the options appear here, they *must* be preceded by a * and separated from each other by tabs, spaces or commas e.g.

   ```
   *List+, Maclist On Print +
   ```

   When developing interactively with **HDE**, this is the only way you can specify the top-of-file options.

If there are no options required then it is wise to leave a blank line at the front of the file. The assembly options are divided into two groups:- Switches and Global Options (which *must* appear at the top of file).

SWITCHES consist of a letter indicating the command (optionally followed by the rest of a word) followed by white space (space or tab characters) and then one of ON, OFF, + or -. ON and OFF may be entered in lower case if you wish. This format allows the flexiblity to be either terse or to make things clear to the inexperienced user. For example:

```
GEN80 TEST;Listing off
GEN80 TEST;L -             (Note that if + or - is used)
GEN80 TEST;List-           (white space need not be present)
```

will all have the same effect (of switching off the listing).

The six switches are:-

List

This specifies whether assembly listing is generated. A counter is maintained during the second pass of the assembly, the state of which dictates whether listing is on or off. A `List ON` command adds 1 to the counter and a `List OFF` command subtracts 1. If the counter is zero or positive then listing is on, and if it is negative then listing is off. The default starting value for the counter is -1 (i.e. listing off). This system allows a considerable degree of control over listing permitting, as an example, the overriding of the `List OFF` which normally appears at the head of a library file by a preceding `List ON` in the main file. If the user does not require such control, then alternating `ON` and `OFF` commands will, of course, work as expected. The **List** switch is also an assembly command (i.e. it may appear anywhere in the file) and is also mentioned in **Section 2.11**.

Maclist

This specifies whether the lines generated by the expansion of macro calls are listed or not. The default is `Maclist OFF`. The **Maclist** switch is also an assembly command (i.e. it may appear anywhere in the file) and is also mentioned in **Section 2.11**.

Printer

This specifies whether the assembler listing (if listing is being generated) is output to the printer (via the CP/M logical device `LST:`). The default is `Printer OFF`. The Printer switch is also an assembly command (i.e. it may appear anywhere in the file) and is also mentioned in **Section 2.11**.

Relocate

This allows to to choose to generate either `.COM` or `.REL` object code files. `.COM` files are files that can be executed directly from CP/M once they are produced, they should start at address `#100` (decimal 256). `.REL` object files cannot be executed directly, they consist of a stream of bits and not sensible Z80 opcodes.

The purpose of .REL files is to allow linking of files together, two or more .REL files may be joined together using a standard linker (e.g. LINK.COM supplied with Amstrad CP/M Plus) or Microsoft's L80™.

The default setting of this switch is R- i.e. so relocatable output is off and a .COM file is produced. If R+ is used then other assembler directives are allowed viz. ASEG, CSEG, DSEG, PUBLIC, EXTERNAL, .PHASE, and .DEPHASE. These are specific to relocatable code output and are explained in more detail later; if you attempt to use these directives when R+ has *not* been used then you will get an error. You can also use .REL files to make Resident System Extensions under CP/M Plus.

Quick

This somewhat arbitrarily-named option specifies whether or not a .ERR file is generated by the assembler when errors are detected.

If you have used Q+ then all error information will be dumped to a .ERR file whose filename is the same as the source filename; this error information is then used by the interactive editor (**HDE**) to show you the errors in your source file when you use the Goto Next Error command. If you use Q- then no .ERR file is produced.

Upper case

U+ switches case-sensitivity off so that the assembler upper-cases all characters in labels; U- turns case-sensitivity on. The default is U-.

The following **GLOBAL OPTIONS** may only appear at the top of the file.

DirectInput

This extremely powerful option may only appear on the command line and not in the file and therefore cannot be used interactively. It allows you to enter text from the keyboard just as though it was in a file. If this option has been specified, **GEN80** will print the prompt

```
Direct mode:
At Front (Y/N)?
```

and on receipt of an answer (Yes or No) will then accept input from the keyboard prior to considering the first line of the main file.

You should type instructions just as though to a file and all the normal CP/M line-editing functions are available. On receipt of a blank line (i.e. just [RETURN] alone), **GEN80** will make a temporary file on the logged-in disc whose identifier is GENTEMP.$$$ and whose contents are whatever has been typed at the keyboard. **GEN80**'s activities now depend on the answer to the original prompt.

If you replied Y then **GEN80** will act as though the first line of the file was

```
*I GENTEMP.$$$
```

i.e. the text input from the keyboard will be assembled at the front of the file. **GEN80** will then continue to assemble the main file as normal. If you replied N then **GEN80** will ignore the temporary file and continue to assemble the main file as normal. In both cases (but more obviously the second case) you are free to include the line

```
*I GENTEMP.$$$
```

explicitly in the main (or any other) file (see Assembler commands below for the meaning of *I). The temporary file will be deleted after assembly is completed. The default setting is that DirectInput is not accepted.

The DirectInput option can be used in many ways:- a label controlling conditional assembly may be specified without altering the main file, registers may be set up specifically for testing of program modules, an ORG statement may be typed to test or verify the position-independence of code etc.

ForceSecond

If this option is specified then the Second Pass of the assembly process is forced even if there are errors or warnings in the first pass. This option will generally be used if a print-file is being made (see below) so all errors can be inspected and corrected at on go. An additional use of the option is to find any

```
Out of range
```

errors (e.g. a relative jump that is out of range).

This type of error will only occur on the second pass and will thus be missed if the assembly is aborted after the first pass due to other errors in the file. The default setting is that the second pass is not forced.

KillObject

If this option is specified then if the object file already exists on the disc, it will be deleted without asking the user. If the option is not specified then the user will be prompted whether to delete the previous object file or not. The default setting is that previous object files are deleted automatically.

NoObject

This specifies whether an object file (.COM or .REL) is generated or not. Using this option to inhibit generation of object code may be used for a fast test assembly to check that there are no syntax errors in the file. The default is that object code is generated.

TablePrint

If this option is specified then a Symbol Table, showing all labels in alphabetical order (with their values) is output after the end of the second pass. If this option is selected with listing off and a print-file is made (see below) then a disc file will be produced consisting only of the source file labels. This can be extremely convenient and useful for debugging or reference purposes. The default setting is that no Symbol Table list is produced.

Generate SYM file

The G option dictates whether or not a .SYM file is created and what length of symbols go into it. The reason for wanting a .SYM file is that the debugger,**ProMON**, will use any .SYM file corresponding to an object program that is being debugged to extract the symbols of the program so that you can see your program labels while debugging.

Two types of `.SYM` file may be produced; one containing up to 6 character labels, upper-cased (for compatibility with the `.SYM` files created by the linkers **LINK** and **L80**) or one with up to 10 character labels, upper- and lower-cased for maximum readability whilst debugging. You get the first by using `G 6` and the second by `G 10` which is the default. Alternatively, you can generate no `.SYM` file by using `G 0` to turn off the symbol dump.

Virtual Disking

This option allows the user who has only one disc drive in his system to retain full control over which discs are used for various files. If this option is specified then the letter that is normally used in CP/M to denote a drive is now used to denote a disc, and the logged-in drive is used throughout. Thus the source file might read:-

```
*Virtualdisking, WritePRNfile B:file

*Include C:module1
*Include C:module2
*Include C:module3
```

and the command line might be:-

```
A>GEN80 D:file=file [RETURN]
```

This rather complex example simply means:-

a) Drive A: is used throughout
b) The main source file is taken from the same disc as **GEN80**. This is because no discname is specified for it on the command line.
c) A print-file is produced on another disc (B:).
d) The includes are taken from another disc (C:).
e) The object file is written to a fourth disc (D:).

Whenever a disk-change is required (which would be rather often in the above slightly far-fetched example) **GEN80** halts and prompts you to insert a disc. This should, of course, be put into the logged-in drive and then a key is pressed to restart **GEN80**.

Virtual disking allows the owner of a one-drive system both to assemble large files and to keep different types of files on different discs (e.g. a certain disc is used for `.PRN` files alone).

WritePRNfile

If this option is specified then a file with filetype `.PRN` will be created containing whatever assembly listing and error messages that would otherwise have gone to the screen. You may specify a filename after **WritePRNfile** (separated by a tab, space or comma).

If no file is specified then the drive and filename are the same as for the source file, otherwise if no drive is specified then the currently logged-in drive is used. If errors result from the assembly then the messages are sent both to the screen and the file. The default setting is that no print-file is created. No other assembler option may follow the **WritePRNfile** option.

SizeOfLabels

This specifies the number of characters in labels that are treated as significant and requires a numeric parameter. This should be a decimal number separated by a space, tab, or comma from the option itself. The value given is the number of characters that will be entered into the Symbol Table and thus space considerations form the upper limit for label length e.g.

`Size 6`	for a small symbol table.
`Size 12`	for a very readable listing but large symbol table.

Whatever value `n` is given, the first `n` characters of all labels must be unique or a

```
Re-defined symbol
```

error will occur. The default value is 10, which should be sufficient for most purposes.

BufferSymbols

This is used to specify the amount of memory used by the Symbol Table and requires a numeric parameter. This should be a decimal number separated by a space, tab, or comma from the option itself. The value given is the amount of memory in kilobytes that the Symbol Table may occupy. The default is 38% of the available RAM. The amount of space used and allocated is displayed at the end of assembly. For fast assembly it is best to specify a table-size just larger than that required.

As an example, if a test assembly reports that the Symbol Table size used was `7K` then you should subsequently specify a size of `7` for most efficient and speedy assembly e.g. use

`B 7`   as the option.

Remember that when you use options in your source file you must start the line on which you use the options with an asterisk (`*`) e.g.

```
*List ON, R+, S 12
```

2.4 Assembler Statement Format

Each statement that is to be processed by **GEN80** should have the following format:

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

Excess spaces and tab characters (over and above the ones used to separate the various fields) are ignored.

GEN80 processes a source line in the following way:

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

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

* expects the next character to be the first letter of an assembler command - see **Section 2.11**. There may be more than one command on a line and the commands should then be separated by tab, space or comma characters.

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

(space)
if the first character is a space or a tab then **GEN80** expects the next non-space/tab character to be the start of a mnemonic, macro or comment.

If the first character is any other than those given above then the assembler expects a label to be present. For the format of a label see **Section 2.5** below.

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.

When found it either expects the character to be an end-of-line character (in which case processing of the line ends) or the following 1-plus characters to be a mnemonic or macro terminated by white space; for a list of mnemonics see **Section 4.3**. If the mnemonic or macro is valid and requires one or more operands then spaces/tabs are skipped and the operand field processed. Each mnemonic has a definite number of operands associated with it.

Comments may occur anywhere after the operand field or (if a mnemonic takes no arguments) after the mnemonic field and may be, theoretically, of any length.

2.5 Labels

A label is a symbol which represents up to 16 bits of information. It can be used to specify either the address of a data area or particular instruction or it can be used simply to specify data. If a label has been associated with a value greater than 8 bits and it is then used where an 8 bit constant is applicable then the assembler will generate an error message e.g. the text:

```
Label    EQU  #1234
         LD   A,Label
```

will generate the error

```
Out of range in <source filename>
```

when processing the second statement on the second pass.

A label can contain any number of valid characters. It is, however open to the user to specify how many of the characters are significant. As an example, assume you specify (by use of the `S` assembly command) that labels should be six characters in length. Now, although labels may be any length in the actual source file, only the first six are entered into the symbol table and thus two labels whose first six characters are the same (even though subsequent characters may differ) will be seen by **GEN80** as identical.

Thus if the label length is `S` (default value 10) the first `S` characters of all labels must be unique since a label may not be re-defined (unless the `DEFL` pseudo-operand is used. See **Section 2.10**).

A label must not constitute a Reserved Word, **see Section 4.2**, although a Reserved Word may be embedded as a part of a label.

The characters which are legal in a label are: `0-9` `$` and `A-z` although a label may not start with a decimal digit. A label may also start with a period (`.`) for compatibility. Note that `A-z` includes all the upper and lower case alphabetics and the characters `[ \ ] ^ ' _`. A label may optionally be terminated with a colon, which will be stripped from the label. This feature is included for compatibility with source files from other assemblers.

Some examples of valid labels:

`LOOP`	(These 2 labels are distinct)
`loop`	(as case is distinguished)
`a_long_label`	
`A_Label_no1`	(not distinct by default as)
`A_Label_no2`	(first 10 chars not unique)
`LDIR`	(`LDIR` is not a Reserved Word)
`.label1:`	(These 2 labels are identical)
`.label1`	(as trailing colons are lost)

2.6 Location Counter

The assembler maintains Location Counters so that symbols in the label field can be provided with addresses and entered into the Symbol Table.

2.6.1 .COM file Mode

Only one location counter is used when the assembler is generating `.COM` files directly (this is the default mode or `R-`), this location counter is initially set to the value `#100` which is the start address of any file loaded by CP/M. The location counter is increased as instructions are generated.

You may set the location counter to any absolute value through use of the `ORG` directive; note, though, that this will simply change the value of the counter so that the next code will be generated *as if* it loaded at the new address, no padding code will be created to actually force the code to load at that address, that is your responsibility e.g.

```
        ld    de,start
        ld    hl,code
        ld    bc,length
        ldir
        jp    start

message db    "Hello World!$"
```

```
code
         org   #8000

start
         ld    de,message
         ld    c,9
         call  5
         rst   0

length   equ   $-start
```

This code will load at #100 since this is the default. The code moves the 4 instructions at the end to location #8000 and then jumps there to print out a message. These 4 instructions are actually held immediately after the message but are assembled as if they were to run at #8000 (the purpose of the ORG) and thus, when they are moved to #8000, they will execute correctly there.

If you wish to pad out your code so that code after an ORG is generated in the place it is going to run, then you can use the DEFS directive like this:

```
address  equ   #8000

         jp    far
         defs  address-$

         org   address
far
         call  routine
         jp    more
```

Remember, though, that this will create a program on your disc that is nearly #8000 (32K) long, mostly full of zeroes! Normally this would not be a sensible thing to do.

The above example demonstrates the use of the $ symbol to mean the *current value of the location counter*; $ gives the value of the counter at the beginning of this instruction.

2.6.2 .REL file Mode

When using the R+ command to generate .REL files the assembler keeps separate location counters for the ASEG, CSEG and DSEG segments. This is so you can mix up the different segment types without confusing the assembler. All location counters are set initially to 0.

The Location Counter value within any segment may be set by use of the ORG directive but this has different effects depending on which type of segment you are currently using.

Within ASEG (the Absolute SEGment), an ORG will behave as described above for .COM files except that, at link time, the code following the ORG will be loaded at the address of the ORG i.e. there is no need for you to move it or to pad out using DEFS, the linker does the padding for you. Also the intial location counter is 0 not #100.

For CSEG (Code SEGment) and DSEG (Data SEGment) an ORG sets the location counter *relative* to the start of this segment e.g.

```
*r+
         DSEG

         ORG   256
message  defm  "Devpac80$"
```

will generate the message at 256 bytes from the start of this data segment, not at absolute location 256. The linker will decide where it is going to load this segment and will generate 256 nulls within the segment, before the message.

If you've been following the above discussion closely you might now be thinking, *What if I want to generate some code that is to be moved by me and executed at a different address, like for the .COM file example above?* The answer is to use the directives .PHASE and .DEPHASE; .PHASE exp says to the assembler: generate the following code as if it were to execute at address exp but leave it here. The linker also leaves it where it was generated and does not move it. It is up to you, the programmer, to move it to its execution address when appropriate. .DEPHASE turns off this mode and reverts to the previous mode.

For example:

```
*r+
         jp      more

         .PHASE  C000h
         or      a
         jp      p,Not_Scr
         call    Get_Screen
         or      a
         ret
Not_Scr  call  Get_Key
         scf
         ret
         .DEPHASE

more ex    de,hl
```

The code between `.PHASE` and `.DEPHASE` is left where it is by both the assembler and the linker but the location counter is changed to be at hex `C000` after the `.PHASE` so that the code is generated as though it was at that address. You can then move it when you want. The expression after the `.PHASE` must be absolute and the mode after a `.PHASE` is `ASEG`.

In `.REL` mode, the symbol `$` works as you would expect, returning the value of the location counter, *within this segment*, at the beginning of this instruction.

2.7 Symbol Table

In the following discussion, the words symbol and label are used to mean largely the same thing. In general, the text that is found in the label field is called a symbol. Every time a symbol is encountered for the first time (either in the label field or in the operand field) it is entered into a table.

```
LABEL    LD HL,3    ;LABEL in the label field
         LD HL,LABEL;LABEL in the operand field
```

If the first occurrence of the label occurs in the label field then its value (the value of the Location Counter at this point) is also entered into the table. Otherwise the value is entered later whenever the symbol is found in the label field. In `.REL` file mode, any symbol listed after the `EXTRN` directive is also included in the symbol table.

If, at the end of the first pass, any symbol in the table does not have a value associated with it (apart from those declared as `EXTRN`al) then the message:

```
*WARNING* label absent
```

will be generated for each symbol without a value.

If, during the first pass, a symbol is defined more than once in the label field then the error:

```
Re-defined symbol in <source filename>
```

will be generated since the assembler does not know which value should be associated with the label.

Note that, by default, only the first 10 characters of a label (see **Section 2.5** above) are entered into the Symbol Table in order to keep down its size, this may be changed using the command `S`. The space allocated to the Symbol Table may also be set (using the command `B`, see **Sections 2.2** and **2.3**) and the default space allocated is 38% of the available memory. As a rough guide as to how much space to allow for the Symbol Table (in files producing less than about 8k of object code, the default value should be sufficient) assume that each symbol occupies `7+S` bytes within the table, where `S` is the significant size of your symbols. If you have a great number of macro definitions then you may need to increase the size of the table since macro definitions are also stored in the symbol table.

At the end of each assembly you will be given a message stating how much memory was used by the Symbol Table during this assembly. It is possible, however, to obtain a complete alphabetic list of the Symbol Table at the end of the second pass (use the command `T`, see **Section 2.3**). Again, only the first `S` characters of any symbol will appear in this list.

2.8 Relative & Absolute Values

In `.COM` mode, all symbols are deemed to be absolute and can be added, subtracted, multiplied etc. together at will, see **Section 2.9** below.

In `.REL` mode symbols can be absolute or relative and there are restrictions as to how these different types of symbols can be combined together. A relative symbol arises within the `CSEG` or `DSEG` segments where it is effectively relative to the start of this particular section e.g

```
*r+,l+
            CSEG

Absolute    equ     5

Relative    call    Absolute
            jp      m,Relative
```

`Absolute` is an absolute symbol since it has one, unchanging value. `Relative`, on the other hand, is a relative type of symbol since its value will depend on where this `CSEG` is loaded by the linker.

All symbols defined within `ASEG` are absolute whilst symbols defined in `CSEG` and `DSEG` segments are absolute or relative depending on their definition as in the above example. The rules for combining absolute and relative symbols are given in **Section 2.9** below.

If an expression evaluates to a relative type then the letter `R` is included after the machine code representation in the assembler listing, this `R` does not get generated in the code!

2.9 Expressions

The expression handling in **GEN80** allows a wide range of operators to be used, with full precedence which may be overridden by the use of brackets. The items in expressions are either labels, in which case their current value is used, or numbers or characters.

Numbers are one of the following:

1) A decimal number. Just a sequence of decimal digits.

2) A hex number. The hash character (# or ASCII 35 decimal) followed by hexadecimal digits *or* a decimal digit (to distinguish it from a label) followed by hex digits terminated by `H`.

   e.g. `#4A2E #AF 5AF0H 0AFH`

   Note that `C050H` is a label but `0C050H` is a number.

3) A binary number. The `%` character followed by binary digits *or* binary digits terminated by `B`.

   e.g. `%11011111 1101B %1111 10101010B`

Literal characters are represented by enclosing them in double or single quotes. Thus all the following lines will produce the same object code:-

```
LD   A,65
LD   A,#41
LD   A,41H
LD   A,%1000001
LD   A,01000001B
LD   A,"A"
ld   a,'A'
```

A single quote character can be represented by `"'"` and double quote by `'"'`.Arithmetic operations generally use signed 16-bit arithmetic, but the result is given modulus 65536 and overflow is ignored. What this means in real terms is that the result will almost always be what is expected. As an example:- `3 * #4000 = #C000`. Strictly speaking this operation should lead to an overflow using signed arithmetic (`#C000` is really `-#4000`), but the result returned is what would be expected (i.e. `3*4=12=#C`).

The only exception to this rule is during division where the operand is a negative number (i.e. greater than `#7FFF`). As an example:-

`#C000 / 2 = #E000` (i.e. `-4/2=-2`)

When used with operators other than the + and - operators the operands used must be absolute and not relative since, for example, multiplying two relative values together is meaningless because you have no idea what the end result is going to be since the linker decides relative values. The results of combining absolute and relative values with addition and subtraction are given below:

Operation	1st operand	2nd operand	Result type
+	absolute	absolute	absolute
+	absolute	relative	relative
+	relative	absolute	realtive
+	relative	relative	*illegal
-	absolute	absolute	absolute
-	absolute	relative	*illegal
-	relative	absolute	relative
-	relative	relative	absolute

* these operations are illegal and give an assembly-time error.

Relative values may be defined in the CSEG or DSEG segments but a relative value defined in CSEG cannot be combined in any way with a relative value defined in DSEG. Symbols defined in ASEG or when generating a .COM file are absolute in type and can be combined in any way.

External symbols (defined with the EXTRN directive) may also be used in expressions. Any expression (absolute or relative) can be **added** to an external, but you may not have more than one external in an expression. Thus if we have

```
            EXTRN  ext,ext2
offset      equ    4
label       ld     hl,10
.......
```

then the following are all valid:

```
      ld     hl,ext+2
      ld     de,label+ext
      ld     de,ext +offset*2
```

But the following are illegal:

```
    ld      hl,2-ext          ; can't have -external
    ld      de,ext*2
    ld      bc,ext+ext2       ;two externals in expression
```

Logical false is represented by 0 and logical true by -1 (or #FFFF), although other non-zero values will be treated as true in most cases. The logical operators are performed bitwise.

Note: The symbol $ returns the current value of the Location Counter.

Strings may also be used in expressions with comparison operators only. This may not sound very useful but can be used to great advantage in complex macro definitions.

A list follows of all the operators with their priority (1 is the highest priority). Brackets () may be used to override the normal priority.

Operator Precedence Table

1)	Unary plus (+) Unary minus (-) Logical NOT (.NOT.) Get high 4 bits (.HIGH.) Get low 4 bits (.LOW.)
2)	Exponential (.EXP.)
3)	Multiplication (*) Division (/) Remainder (.MOD.) Shift left logical (.SHL.) Shift right logical (.SHR.)
4)	Binary plus (+) Binary minus (-)
5)	Logical AND (&) or (.AND.)
6)	Logical OR (.OR.) Logical EXCLUSIVE OR (.XOR.)
7)	Equals (=) or (.EQ.) Signed less than (<) or (.LT.) Signed greater than (>) or (.GT.) Unsigned less than (.ULT.) Unsigned greater than (.UGT.)

The following are allowable expressions in **GEN80**:-

`#5000-label`	
`16-%1110001`	
`label1-label2+label3`	These two expressions
`label1-(label2+label3)`	are not the same
`2.EXP.label`	
`label1+(2*.NOT.(label2=-1))`	
`"A"+128`	
`"A"-"a"`	
`label-$`	
`$+(label2-label1)`	

Notes on the operators

`.NOT.` is a unary operator. To produce the same effect as (`label1≠label2`) use the construct `.NOT.(label1=label2)`

`.EXP.` is used to raise a number to a power. Thus `3.EXP.4=81`. The expression following `.EXP.`is treated as unsigned and the result will be modulus 65536 (i.e. overflow is ignored).

`.SHL.` and `.SHR.` shift the first argument left or right by the number of bit positions specified in the second argument. Zeros are shifted into the low-order or high-order bits. Either operator may have a second argument that is negative. Thus `label1.SHL.-2` is equivalent to `label1.SHR.2`

The five comparison operators (`.EQ. .LT. .GT. .ULT. .UGT.`) will evaluate to logical true (-1 or #FFFF) if the comparison is true and to logical false (0) otherwise. Thus (`1.EQ.1`) will return the value -1 and (`1>2`) will return the value 0. The operators `.GT.` and `.LT.` deal with signed numbers whereas `.UGT.` and `.ULT.` assume unsigned arguments. Thus (`1.UGT.-1`) is false (i.e. 1 is not greater than 65535) but (`1.GT.-1`) is true (i.e. 1 is greater than -1).

`.HIGH.` and `.LOW.` are monadic operators returning the top and bottom 8 bits of their arguments respectively. For example:

`.HIGH.#1234`	returns `#12`
`.LOW.#1234`	returns `#34`

2.10 Assembler Directives

Certain pseudo-mnemonics are recognised by **GEN80**. These assembler directives, as they are called, have no effect on the Z80 processor 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.

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

ORG expression

Sets the Location Counter to the value of the expression. In CSEG and DSEG modes the location counter is set relative to the start of the section whilst in ASEG and .COM mode it is set to an absolute value.

EQU expression

Must be preceded by a label. Sets the value of the label to the value of the expression. The expression cannot contain a symbol which has not yet been assigned a value.

DEFB expression{,expression,expression etc.}
DB expression{,expression,expression etc.}

DEFB or DB may be followed by as many expressions as can fit onto a line. Each should be separated from the next by a comma and each must evaluate to 8 bits or be a string. For each expression, the appropriate byte is set to hold the value of the expression. Examples:

```
    DEFB    "A message",CR
    db      1,2,3,4,5
    defb    CR,LF,'Press a key',0
```

Strings are enclosed with single or double quotes. To include the quote character in a string type it twice. E.g.

```
    defb    "double "" single '"         gives
    double " single '                    in the object code.
```

DEFS expression{,expression}
DS expression{,expression}

Reserves a number of bytes equal to the value of expression at the current Location Counter and fills that memory with the value of the second expression or zero if there is no second expression.

DEFM "string"

Defines the contents of N bytes of memory to be equal to the ASCII representation of the string, where N is the length of the string and may be between 0 and 255 inclusive. The first character in the operand field can be either ' or " and acts as the string delimiter.

DC "string"

DC works like DEFM except that the top bit of the last character in the string is set. This is sometimes useful for messages where the message printing routine detects the end of the message by checking the top bit.

```
dc     "Hello there"   gives

48 65 6C 6C 6F 20 74 68 65 72 E5
```

MACRO {parameters}

This directive must be preceded by a label and marks that label as identifying a macro. The parameters for the macro follow. Each parameter must start with the character @ and is separated from the next by a comma. The actual macro definition follows and must be terminated by the directive ENDM (see below).

ENDM

This directive is used to signal the end of a macro definition.

IF expression
COND expression

This is the first of the three conditional directives. The other two are ELSE and ENDC. IF will evaluate the expression. If the result is false (zero) then assembly of subsequent lines is turned off until either an ELSE or an ENDC pseudo-mnemonic is encountered. If the result is non-zero then assembly is left in its current state. IFs are nestable to a depth of 8.

ELSE

This pseudo-mnemonic normally 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. However, if ELSE occurs in a nested IF then assembly will only be flipped if assembly was on before the previous IF. If assembly was off then the ELSE has no effect.

ENDC
ENDIF

This pseudo-mnemonic returns assembly to the state it was in before the previous IF.

The use of these conditional directives lies in the ability to control whether certain sections of code are compiled or not. They are often used in conjunction with labels and may be used, say, to control whether a certain block of code used for debugging purposes is assembled using the lines:-

```
IF    DEBUG
--       ;the debugging code sits here and will only
-        ;be assembled if the value of DEBUG is not 0
ENDC
```

The feature may also be used if the same code is being used on several different machines and then generation of the various machine-specific sections of code may be controlled by the lines:-

IF expression
COND expression

This is the first of the three conditional directives. The other two are ELSE and ENDC. IF will evaluate the expression. If the result is false (zero) then assembly of subsequent lines is turned off until either an ELSE or an ENDC pseudo-mnemonic is encountered. If the result is non-zero then assembly is left in its current state. IFs are nestable to a depth of 8.

ELSE

This pseudo-mnemonic normally 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. However, if ELSE occurs in a nested IF then assembly will only be flipped if assembly was on before the previous IF. If assembly was off then the ELSE has no effect.

ENDC
ENDIF

This pseudo-mnemonic returns assembly to the state it was in before the previous IF.

The use of these conditional directives lies in the ability to control whether certain sections of code are compiled or not. They are often used in conjunction with labels and may be used, say, to control whether a certain block of code used for debugging purposes is assembled using the lines:-

```
IF   DEBUG
--      ;the debugging code sits here and will only
-       ;be assembled if the value of DEBUG is not 0
ENDC
```

The feature may also be used if the same code is being used on several different machines and then generation of the various machine-specific sections of code may be controlled by the lines:-

```
IF    CPC
-       ;this code will be assembled if the
-       ;value of CPC is not 0
ENDC

IF    PCW
-       ;this code will be assembled if the
-       ;value of PCW is not 0
ENDC
```

END

This directive signals that no more text is to be examined on this pass. It might, for example, be used in a macro in conjunction with the IF directive to abort assembly if the parameters used are inconsistent with the proper operation of the macro, or potentially disastrous to the system. As an example, assume a macro:-

```
MOVE MACRO      @BYTES,@FROM,@TO
         IF     @BYTES<1 ;if the number to move is zero
*Zzzzz                   ;make the listing stop here to
         END             ;see what's happening and quit
         ENDC
         LD     BC,@BYTES
         LD     HL,@FROM
         LD     DE,@TO
         LDIR
         ENDM
```

This would stop a disastrous piece of code being produced by the line:-

```
         MOVE   L2-L1,L1,L3
```

when L2 is the same as L1.

.COMMENT delimited string

This directive allows multi-line comments: .COMMENT must start the line (*not* in the mnemonic field) and you should follow the .COMMENT by a space, a delimiter of your choosing followed by your comment text. This comment text may then flow over as many subsequent lines as you like and the assembler will treat everything as a comment until it finds another occurrence of your chosen delimiter, for example:

```
.COMMENT / This is a long comment that flows over
           a number of lines and this feature allows
           your source to be commented more neatly. /

         ld    hl,label
         bit   7,(hl)       ;etc.
```

.Z80

Does nothing, this directive is included for compatibility with other assemblers, specifically the Microsoft M80™ assembler. .Z80 must appear at the start of the line, not in the mnemonic field.

.PHASE expression

This is used in .REL file mode to allow code to be assembled to run at a different address, given by expression, from where it is placed. .PHASE can be used in ASEG, CSEG and DSEG modes but the mode is absolute while .PHASE is in effect, use .DEPHASE to end this mode.

For example, say you are writing some code that needs to run at address #C000 but your main program is designed to execute at #100. This might be the case on an Amstrad CP/M Plus computer if you are trying to access the video screen by the extended BIOS call SCR_RUN. So you need some code ORGed at #C000 but you don't want it loaded there by the linker. If you use ORG, the linker will load the code at the ORG address, you don't want this because this would result in a very large (approx. 48K) .COM file. So use .PHASE like this:

```
;some code to move a block of screen memory
;must go in common memory

MoveScr
         .PHASE #C000
         push   ix
         pop    bc          ;because SCR_RUN corrupts BC
         ldir               ;move screen RAM about
         ret
         .DEPHASE
MSLen    equ   $-MoveScr

;some time later

SCR_RUN  equu   0e9h

         ld     hl,MoveScr
         ld     de,#C000
         ld     bc,MSLen
         ldir
         ld     hl,ScrStart
         ld     de,ScrDest
         ld     ix,ScrLen
         ld     bc,#C000
         call   Call_USERF
         defw   SCR_RUN
                            ;etc
;some time later

.COMMENT / routine to call extended BIOS routine USERF
           which takes extended routine address inline /
Call_USERF
         push   hl
         push   de
         ld     hl,(1)
         ld     de,87       ;to give USERF
         add    hl,de
         pop    de
         ex     (sp),hl
         ret

;rest of your code
```

The above is a fairly complex example of the use of `.PHASE`, included for those people who have an interest in hacking the screen environment on an Amstrad CPC6128 or PCW8256/8512/9512. In general, you use `.PHASE` in `.REL` mode when you would use `ORG` in `.COM` mode.

.DEPHASE

Simply turns off the `.PHASE` mode and reverts to the mode that was in force prior to the previous `.PHASE`.

PUBLIC symbol, symbol, ...

Used to export symbols from this file that are to be used by other assembly modules. This directive is only available when in `.REL` file mode and tells the linker that the symbols have been defined values in this assembly. Labels may also be declared PUBLIC by following the label with 2 colons e.g. `Message::   defm   "Hello"`

EXTRN symbol, symbol, ...
EXTERNAL symbol, symbol, ...

The symbols listed here are not defined in this source file but in some other file. **GEN80** accepts them as valueless and the linker will resolve these references and fix-up the right values which will have been declared `PUBLIC` is some other assembly. See Section 2.9 for the rules regarding the use of externals in expressions. `EXTRN` can be used only when generating `.REL` files.

2.11 Assembler Commands

Assembler commands, with one important exception (`*Include`) do not affect the code produced by **GEN80**. They are used for producing and formatting the assembly listing. They are entered on lines that begin with a * and may appear anywhere in the file. Two or more may appear on the same line and they should be separated by a comma, tab or space character. Only the first character of the command is significant (and may appear in upper or lower case) and the rest of the command up to the next space, tab, or comma is ignored. The following commands are available:-

*Eject

Causes a new page to be produced on the printer; carriage returns/ linefeeds are sent to the printer until a new page is reached. The number of lines per page on your printer may be installed into **GEN80**, see **GEN80 Installation**.

*Zzzzz

Causes the listing to be stopped at this point. The listing may be reactivated by pressing any key on the keyboard. Useful for reading addresses in the middle of listing. Note: `*Z` is still recognised after a `*L-`; see below. `*Z` does not halt printing.

*Heading string

Causes the first 32 characters of the specified string to be taken as the heading which is printed on the top of every new page. An automatic `*E` is done after `*H`. The heading is only sent to the printer or print file. The end-of-line character is taken as the terminator of the string and white space may appear as desired in the string. No other text may appear on the same line as a `*H` command.

*Include filename
*Maclib filename

This powerful assembler command causes source code to be taken from another file and assembled exactly as though it were explicitly present in the file. `*Include` must be followed by a filename (separated from it by white space). If the filetype is not specified it will be assumed to be `.GEN`. `*Include` commands may be nested up to 4 levels (i.e. an included file may contain a `*Include` etc.). The `*Include` command encourages and facilitates the modular approach to programming as it becomes possible to develop and test modules one by one and finally assemble the whole program from an extremely small main *including* file thus:-

```
*LIST ON PRINTER ON  TABLEPRINT  WRITE_PRN_FILE
;Main linking file

DEBUG    EQU 0 ;The real thing

*Include MODULE1
*Include MODULE2
*I       MODULE3
```

Alternatively you can use the .REL file mode of the assembler to gether with the **EXTRN** and **PUBLIC** directives and a linker to assemble modules separately and then link them together. INCLUDE can also be present as a mnemonic for Macro80 compatibility.

*List, *Printer, *MacList

These are the three assembler commands that are also top-of-file options. They may thus appear on the command line and in the options list. They are switches and details of their actions are described in the section on top-of-file options, **Section 2.3**.

*Generate, *Quick, *Relocate

Top-of-file options, see **Section 2.3**.

2.12 Macros

In **GEN80** macros are a powerful tool that let you greatly simplify assembly language programming. When using macros in some less sophisticated assemblers it is easy to generate huge code files, but using the DEFL, conditional assembly and textual parameter facilities of **GEN80**, files may be kept extremely readable and yet compact.

A macro may be defined thus:-

```
BC_DE    MACRO @PARAM1, @PARAM2
         LD    BC,@PARAM1   ;The text of the
         LD    DE,@PARAM2   ;macro
         ENDM
```

The pseudo-mnemonic MACRO is used to introduce a macro, and causes the label (which must precede it) to be entered as a macro name into the symbol table. The label thus becomes the name by which the macro will be called. An optional list of parameters follow. Each must be preceded by the character @ and may contain any of the characters legal in a label (See **Section 2.5**). Parameters are separated from each other by space, tab, or comma characters, but these characters may appear in a macro parameter if enclosed in *single* quotes (when the single quote is repeated to stand for itself) e.g.

```
PRINT    MACRO @P1
         PUSH  HL
         LD    HL,M@$YM
         CALL  MOUT      ;A message printing routine
         POP   HL
         JR    L@$YM     ;See below for use of @$YM
M@$YM    DEFM  "@P1"
         DEFB  0
L@$YM
         ENDM
         PRINT 'It''s a message'  ;note, single quotes and
                                  ;It''s to give It's
```

The number of parameters allowed will rarely if ever be a practical limit.

The parameters declared on the first line of the macro are addressed in the body of the macro by using the name with which they were declared.

The macro definition is terminated using the pseudo-mnemonic ENDM. All of the text between the MACRO line and the ENDM line is the macro definition. The statements in the macro definition are not assembled when they are encountered so they will not define labels, cause errors or generate code. A macro may not be defined inside another macro definition (nested definitions are not allowed), but a macro may be called from inside a macro (recursion is allowed) and a macro may thus call itself.

A macro is called thus:-

```
         BC_DE #4424, -1
```

i.e. the name occurs in the mnemonic field. It is then followed by any actual parameters separated by delimiters. Delimiters are either space, tab or comma characters. A parameter may optionally be enclosed in single quotes and these will be stripped when the macro is expanded. If the parameter contains space, tab, or comma characters then the single quotes are obligatory. The quote character itself is represented by two successive single quotes.

Parameters are substituted textually. When the macro is invoked, each parameter in the definition is replaced for the *text* that is in the corresponding position in the definition. Thus in the example above, the call to the macro will produce exactly the same code as if the following text had been typed explicitly:-

```
        LD    BC,#4424
        LD    DE,-1
```

The following example will illustrate the real power of true textual substitution as opposed to evaluation before substitution used in some other assemblers:-

```
EXCH    MACRO @REG1, @REG2
        PUSH  @REG1
        PUSH  @REG2    ;The body of the
        POP   @REG1    ;macro definition
        POP   @REG2
        ENDM
        EXCH  DE,BC    ;calling the macro
```

The calling of the macro in the statement on the previous page will be expanded to produce the code:-

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

As can be seen, a new and highly useful pseudo-instruction has been created which can be used exactly as a normal assembler mnemonic which allows the user to swap the value of any of the register pairs (excepting SP), providing an extension to the standard `EX DE,HL` instruction.

In addition to the parameters declared by the user every macro has an extra implicit parameter `@$YM`. This returns a 4 digit hexadecimal number which increases each time any macro is called. Its main use is in generating labels which occur in macros. As an example:-

```
ABS       MACRO
          OR     A
          JP     P,ABS@$YM
          NEG
ABS@$YM
          ENDM
```

then assuming that this was the only macro in a program it would generate

```
          OR     A
          JP     P,ABS0001
          NEG
ABS0001
```

when it is first called, and then

```
          OR     A
          JP     P,ABS0002
          NEG
ABS0002
```

when next called. If `@$YM` had not been used then the same label would have been produced twice resulting in an error. There is an example of `@$YM` on your disc, called `FACT.GEN`. Here is a listing of it, bend your brain to fathom out how it works!

```
.COMMENT * A macro to generate factorial n and assign it
           to  result. Does up to factorial 6 (6!) *
fact:     macro  @result,@n
          if     @n=1
@result   defl   1
          else
          fact   t@$YM,@n-1
@result   defl   t@$YM*(@n)
          endc
          endm
```

```
;a sample call

         fact   test,5
         ld     hl,test      ;loads HL with 5 factorial
```

Another method of inhibiting the possible error above is given below to stimulate the imagination. The method above runs faster but may generate large amounts of code. The method below is extremely compact.

```
ABSLAB   EQU   0
ABS      MACRO
         CALL  ABSUB
         IF    .NOT.ABSLAB
         JR    ABSEND
ABSUB    OR    A
         RET   P
         NEG
         RET
ABSLAB   DEFL  .NOT.ABSLAB
ABSEND
         ENDC
         ENDM
```

Macros may be called recursively i.e. a macro may call itself, but macro definitions may not be nested.

String comparisons may be used in macros to give optional parameters or default values.

e.g.

```
CPM      MACRO @FUN,@FCB
         IF    "@FCB">""
         LD    DE,FCB       ;we have a 2nd parameter
         ENDC
         LD    C,@FUN
         CALL  5
         ENDM
```

then

```
        LD    E,A
        CPM   2
        CPM   26,80h       ;set dma
```

will expand to

```
        LD    E,A
        LD    C,2
        CALL  5
        LD    DE,80h
        LD    C,26
        CALL  5
```

This is an easy way of detecting missing macro parameters thus adding considerable flexibility to the use of macros.

2.13 Assembly Listing

Each line of the assembler listing generated during the second pass of **GEN80** has the following format:

```
6000 210100       25    label   LD   HL,1     ;set HL to 1
```

The first entry in a line is the value of the Location Counter at the start of processing this line, unless the mnemonic or macro in this line is the pseudo-mnemonic EQU or DEFL (see **Section 2.10**) in which case the first entry will represent the value in the Operand field of the instruction.

The next entry, from column 6, is up to 8 characters (representing up to 4 bytes) in length and is the object code produced by the current instruction. This will be followed by the letter R if any operand expression is found to be relative when assembling a .REL file.

Then comes the line number. Line numbers are integers in the range 1 to 65535. The line numbers corespond to lines in a particular file rather than lines in the assembly; thus after a *I compiler command the number becomes 1 and when listing the expansion of macros no line numbers are output.

Columns 21 to 20+S (where S is the length of labels defined by the S command with default S=10) contain the characters of any labels that may be present. If the line contains no labels then the field is left blank.

Next, in column 32 (assuming 10 character labels) is the mnemonic, macro, pseudo-mnemonic or assembler directive.

Finally, from column 37 onwards (assuming 10 character labels), the operands are output followed by any comment present. Comments start at column 50 unless specified otherwise using the command C for comment format.

SECTION 3
Installing GEN80

GEN80 does not require a correct installation to make it work proper,but for maximum flexibilty you can change three aspects of **GEN80**:-

a) The printer page length
b) The printer page width
c) The defaults for the top-of-file options

Type:

```
GEN80INS [RETURN]
```

then hit any key and N to the next question, this will read in the working copy of **GEN80** and then show the following menu:

```
GEN80 Installation Menu

1. Return to CP/M
2. Make changes
3. Save GEN80 as <working copy filename> (normally GEN80  .COM)
4. Save GEN80 as another file
```

Press 2 to make changes. You are now asked:-

```
Enter Printer Page Length (       )
```

The current value is given in brackets (and will be 66 which is the normal value for most printers). If the printer page length is different for your printer then type in the number (in decimal) and then press [RETURN]. Pressing [RETURN] alone acts like typing the number in brackets. Next you are asked to:-

```
Enter Printer Page Width (        )
```

Enter this value in exactly the same way as above. Both of these values are *built-in* to **GEN80**, and are only alterable by using the installation program. Finally, a menu explaining the meaning of the various top-of-file options is given, together with the current default settings and you are asked:-

```
Do you wish to change this ? (Y/N) ?
```

The current default options are those that are *built in* to **GEN80.** The effect of them is exactly as though they had been typed in (preceded by a `*`) on the top line of every file assembled by **GEN80**. As an example, if you always like macros expanded in the listing, and only use the first six characters of labels then you should include `M+,S 6` (or perhaps `Macrolist ON, SymbolLength 6`) in the line. You can use either `[CTRL]-H` (backspace) or `[DEL]` as a destructive backspace when typing in the new default string. Press `[RETURN]` when you are satisfied. Pressing `[RETURN]` alone will accept the current default settings.

Note that the options are of two types. The first is followed by a parameter (either `+/-/ON/OFF` or a number or string) and the other is not. The first type can be overriden by an explicit command on the command line or the top line of the file e.g. a default of `Comment 50` can be overridden by the line at the top of the file `*Comment 40`. The second type, however, is only alterable by re-using the install program e.g. if `N` is a default then **GEN80** will never generate an object file unless reconfigured by the install program.

When back at the main menu, you can save **GEN80** as `GEN80.COM` (option 3) or as another file (option 4). Finally you can quit the install program with option 1 (this does not save anything on the disc). It may prove desirable to save two versions of **GEN80** on the disc. One may be a version configured for syntax checking:-

```
N,F,L-,M-,P-,W
```

so that no object file is created, but the second pass is forced and all errors are sent to a disc file for easy inspection by the editor. If you normally produce linkable code then you can build the `R+` option into your file.

The two options `S` and `B` (for controlling the significant length of labels and size of symbol table) are parameters which are likely to be file-specific. That is, they are dependent upon the particular file being assembled (i.e. does it use long or short labels and does it have an abnormal length symbol table). Thus common-sense might dictate that these options should appear on the first line of a file, if required, rather than being *built- in* to **GEN80** (or having to be remembered on the command line each time the file is assembled).

SECTION 4
Quick Reference Guide

4.1 Error Messages

The following is a list of the error messages generated by **GEN80**.

`Label missing`
One of the assembler directives `EQU DEFL MACRO` occurs on a line that does not have an entry in the label field.

`Illegal symbol`
This message indicates that a label is badly formed and contains illegal characters. Note that mnemonics and assembler directives are acceptable as labels.

`Symbol is Reserved Word`
A label is declared which is a reserved word. Note that a reserved word may constitute *part* of a label. Thus `HL` is an illegal label but `HL1` is not.

`Redefined symbol`
This occurs if a label appears twice in the label field (if `DEFL` has not been used the second and subsequent times). This may be caused when seemingly different labels are present, if the label length (`S`) is such that the first `S` characters of the labels are identical.

`Bad mnemonic`
Indicates that the mnemonic (or opcode) is illegal. This error will occur if a macro is called without (or before) having been declared.

`Bad expression`
An expression is badly formed. This generally means that an operator is missing or unrecognisable.

`Expression syntax`
The operand field of a line is badly formed. e.g. `LD A,DE`

`Illegal Digit after # or %`
A character which is not a valid hex digit is present after a `#` or a character which is not a valid binary digit is present after a `%`.

`Expression too complex`
The expression evaluator has been called upon to do too much. Three levels of brackets are the approximate maximum. Split the expression into simpler units.

`Division by zero`
Self evident

`Bad dot operator`
An invalid dot operator has been used in an expression. This means that a dot operator is badly formed eg `.LT` or `.NOTT`.

`Numeric expected`
This occurs when an expression contains a register where a number or a label is expected. e.g. `LD A,-HL`

`Missing )`
This error indicates that an expression is missing a closing bracket. The expression may be one containing an indirection off a register e.g. `LD HL,(32*LABEL` or `LD A,(HL`

`Illegal index`
There are no brackets around an expression `(IX+n)` or `(IY+n)`.

`JP (IX+n), JP (IY+n) illegal`
Self-evident

`Mismatch of registers`
Two of the register pairs `HL`, `IX`, `IY` occur in the same line, for example `ADD HL,IX`

`Bad command`
This error indicates that the initial letter used for a command is incorrect or the syntax of a command is bad e.g. `*A` or `*L`

`Bad filename`
The name of a file to be `*Included` is badly formed or does not exist.

`Too many includes`
Includes may be nested up to four deep.

`Bad directive`
This error occurs if an assembler directive has the wrong number of parameters :
e.g. `IF LABEL,6`

`Forward reference`
This error indicates that the expression after one of the directives `ORG EQU DEFL` contains a label whose value is not yet declared.

`Macro parameter stack overflow`
The total number of characters generated during the expansion of a macro is too great. The maximum is 255. This error will generally occur when a macro is recursive, but may also occur if macros are nested i.e. a macro uses a macro etc.

`Bad Macro parameter`
Macro parameters must be preceded by `@` when the macro is declared.

`Nested macro definition`
A macro cannot be defined within another macro definition.

`Bad ENDM`
The directive `ENDM` occurs without a preceding directive `MACRO`.

`Re-defined Macro`
You have attempted to re-define an existing macro name.

`Illegal for COM file`
The directives `ASEG`, `CSEG`, `DSEG`, `PUBLIC`, `EXTRN`, `.PHASE` and `.DEPHASE` can only be used when generating a `.REL` file (having used `R+`).

`Expression must be absolute`
The type of this expression cannot be relative, it must be absolute. (e.g. after `IF`.)

`String not terminated`
A string has not been closed with either " or '. Version 1 users please note that this wasd not previously enforced.

`Illegal DEFM`
The structure of this `DEFM` statement is incorrect.

`Error in Conditional`
The nesting of your conditional statements has gone awry.

`Out of range`
This is the only error that can occur during the second pass. It most frequently indicates a relative jump or `DJNZ` out of range. In general it indicates that the value of an expression is too large to be held in one byte e.g. `LD A,256` or `DJNZ $-300` etc.

The following error messages arise from fatal errors. A fatal error is one that will terminate the assembly process immediately and return to CP/M.

`No Source File:`
The source file specified on the command line does not exist. This error is suppressed if the `D` option is specified, allowing the assembly of small files without the use of an editor. This is a fatal error.

`Symbol Table too big!`
The size assigned tu the Symbol Table by the `B` option is too large for the system. There is not enough space for the source and object buffers. This is a fatal error.

`Used all #XXXX bytes of Symbol Table!`
The Symbol Table has grown too large to fit into the space assigned to it. This is a fatal error.

`Disc full!`
Self-evident. This is a fatal error.

`Directory Full!`
Self-evident. This is a fatal error.

4.2 Reserved Words

The following is a list of Reserved Words within **GEN80.** These symbols may not be used as labels although they may form part of any label.

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

Reserved words may appear in upper or lower case.

4.3 Valid Mnemonics

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	INCLUDE		

Mnemonics may appear in upper or lower case.

4.4 Assembler Directives

.COMMENT	.DEPHASE	.PHASE	.Z80		
ASEG	ASET	COND	CSEG	DB	DEFB
DEFL	DEFM	DEFS	DEFW	DS	DSEG
DW	ELSE	END	ENDC	ENDIF	ENDM
EQU	EXTERNAL	EXTRN	IF	MACLIB	MACRO
ORG	PUBLIC				

Assembler directives may appear in upper or lower case.

4.5 Top-of-File Options

```
BufferSymbols           CommentPosition
DirectInput             ForceSecond
GenerateSYMfile         KillObject
List                    Maclist
NoObject                Printer
Quick                   Relocate
SizeOfLabels            TablePrint
Upper case              VirtualDisking
WritePRNfile
```

Top-of-file options may appear in upper and/or lower case.

4.6 Assembler Commands

```
*Eject
*Heading
*Include
*List
*Maclist
*Printer
*Zzzzz
```

Assembler commands may appear in lower and/or upper case.

4.7 Operators

All the operators are listed in order of precedence.

```
1)   +     -     .NOT. .HIGH. .LOW.
2)   .EXP.
3)   *     /     ?     .MOD. .SHL. .SHR.
4)   +     -
5)   &     .AND.
6)   .OR.  .XOR.
7)   =     .EQ.  >     .GT.  <     .LT.  .UGT. .ULT.
```

4.8 .REL File Format

A **GEN80** .REL file contains information encoded in a bit stream. In the unlikely event that you should want to interpret this bit stream, we give its structure below:

If the first bit is a 0, then the following 8 bits are loaded at the current value load of the location counter.

If the first bit is a 1, then the following 2 bits mean:

00	Special link item, these items are described below.
01	Program relative item. The next 16 bits are loaded after being added to the program segment origin.
10	Data relative item. The next 16 bits are loaded after being added to the data segment origin.

A *special item* consists of the following:

1. A 4 bit control field that specifies one of the 16 special link items described in Table 4.8.1.

2. An optional value field that is a 2-bit address-type field and a 16-bit address field. The address-type field is one of:

00	absolute
01	program relative
10	data relative

3. an optional name field which is a 3-bit count followed by the name in 8-bit ASCII.

Table 4.8.1 Special Link Items

Field	Meaning
These link items are followed by a name field only;	
`0000`	This symbol is declared `PUBLIC` in this module.
`0010`	The name of this program.
These link items are followed by a value field and a name field:	
`0110`	Chain external. The value field contains the head of a chain that ends with an absolute 0. Each element in the chain contains the previous occurrence of the symbol given in the name field so that the linker can patch-up all references to this external.
`0111`	Define entry point. The value field gives the value of the symbol in the name field.
These link items are followed by a value field only:	
`1001`	External plus offset. The value in the value field after all chains are processed must offset the following two bytes in the current segment.
`1010`	Define data size. The value field contains the number of bytes in the data segment of this module.
`1011`	Set location counter. Set the location counter to the value indicated in the value field.
`1101`	Define program size. The value field contains the number of bytes in the code segment of this module.
`1110`	End module. Defines the end of this module. If the value field contains a value other than absolute, the value is the start address for the linking program. The next item in the file will start at the next byte boundary.
This item has no value field or name field:	
`1111`	End file. Follows the end module item for the last module in the file.