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™.

All Rights Reserved Worldwide. 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 **HiSoft Devpac80**.

It is an infringement of the copyright pertaining to **HiSoft Devpac80** and its associated documentation to copy, by any means whatsoever, any part of **HiSoft Devpac80** for any reason other than for the purposes of making a security back-up copy of the object code.

Introduction

How to Use this Manual

This manual is set out in five chapters, this introduction, a chapter on **HDE**, the interactive editor, one on **GEN80**, the macro assembler, then a chapter on the debuggers **ProMON** and **MON80** and, finally, a **Full Tutorial**.

Each chapter is separated from the previous one by a light blue card with the chapter title on it.

Beginners

If you are a newcomer to assembly language then we recommend that you read one of the books in the Bibliography alongside this manual.

You should start by working through the **HDE** chapter, **Section 1**, **Getting Started and Tutorial**, to familiarise yourself with the editor.

Then, go through the **Full Tutorial** (the fifth chapter in this manual) after which you can dip into the reference sections of **HDE**, **GEN80** and **ProMON** at your leisure.

If you are an Amstrad, Einstein or MSX owner then all the programs will have been installed for your computer's screen and you will only need to use the Installation programs if you wish to change the commands in **HDE**, **ProMON** or **MON80** or the default options in **GEN80**.

If you have an Amstrad PCW8256/8512/9512 then please note that the programs have been installed for a 24-row, 80-column screen. If you wish to make full use of the large screen size on the PCW computers then you should run the relevant installation program (HDEINST, MON80INS or PMONINS), choose option 2, select 31 rows and 90 columns and then use option 3 to save the configured version.

Remember to make a backup and working disc as described later.

Experienced Users

If you are experienced in the use of assembly language but have not used **Devpac80** before then we recommend that you treat yourself as a beginner and work through the sections given on the previous page.

If you have used **Devpac80** before then you will, in all probability, find it sufficient to read through the following sections:

HDE **Section 1.1** to **Section 1.4**.

GEN80 **Section 2.6**, to **Section 2.11**, **Section 4**.

ProMON **All Sections**.

If you are an Amstrad, Einstein or MSX owner then all the programs will have been installed for your computer's screen and you will only need to use the Installation programs if you wish to change the commands in **HDE**, **ProMON** or **MON80** or the default options in **GEN80**.

Otherwise, you should read the **Installation** sections of the editor and the debuggers carefully and run the installation programs to configure the programs for your screen.

If you have an Amstrad PCW8256/8512/9512 then please note that the programs have been installed for a 24-row, 80-column screen. If you wish to make full use of the large screen size on the PCW computers then you should run the relevant installation program (HDEINST, MON80INS or PMONINS), choose option 2, select 31 rows and 90 columns and then use option 3 to save the configured version.

Remember to make a backup and a working disc as described in the following section.

The Package

Devpac80 is a fast, interactive Z80 development system designed to run under the CP/M 2 and CP/M Plus (CP/M 3) disc operating systems. The package comprises, essentially, a macro assembler, a full-screen editor and a symbolic front-panel debugger and disassembler.

What to do First

Make a Backup

Discs are sensitive animals, prone to heat, cold, damp, magnetic fields and, above all, coffee. So the first thing you should do is to make a copy of your **Devpac80** disc and then store the original somewhere safe, away from things like the sun, televisions, telephones, radiators and the kids!

The way you make a copy will depend on your computer configuration; the space occupied by all the files on a **Devpac80** disc comes to about 160K (even though most of the individual programs are no larger than 14K) and there must, therefore, be this much space on the disc to which you are going to copy. Most CP/M systems come with a program called `BACKUP` or `DISCCOPY` or `DISCKIT` (don't use `DISCKIT` on an Amstrad 8256/8512/9512 - see later) which allows a whole disc to be copied in one go. Check with your system handbooks.

If you don't have such a program then you should format a new disc, write-protect it and then use our WP program, to copy all files i.e. put the master disc in drive A and type:

```
WP A: B: [ENTER]
```

This copies all files from the disc in drive A to the disc in drive B (it won't matter if you get them the wrong way round, you'll just get some error message, then you can swap the discs round and try again).

If you only have one drive then things get a little more complicated. If you have CP/M 3 (also called CP/M Plus) then you may be able to use the virtual disking feature. This means that you can use the WP program just as if you had two drives, A and B, and you will be asked to put in the disc for drive B whenever the copying program needs it i.e. put the master disc into your drive and type:

```
WP A: B: [ENTER]
```

You will be asked to swap discs twice for every file copy, this can get a little tedious and you will be much better off using a whole disc copying program.

If you have CP/M 2 and only one disc then you must use a disc copying program since the only other way involves using the programs DDT (or **ProMON**) and SAVE and this is rather like cracking a nut with a sponge!

Amstrad Owners

CPC 464/664/6128

Owners of the Amstrad CPC machines can use DISCKIT2 or DISCKIT3 to make a backup copy of the **Devpac80** disc.

PCW 8256/8512/9512

Amstrad PCW8256/8512/9512 owners must go about making a copy a little differently since we supply **Devpac80** on a CPC6128 format disc which the PCW DISCKIT program will not recognise.

If you have two drives then:

1. Format a new disc in drive A using DISCKIT.
2. Put the **Devpac80** master in drive B and type:

```
B: [ENTER]
WP B: A: [ENTER]
```

You now have a backed-up copy in drive A.

If you only have one drive then:

1. Format a new disc in drive A using DISCKIT.
2. Put the **Devpac80** master in drive A and type:

```
THINCOPY [ENTER]
```

and then follow instructions to change discs. This will copy all the programs on the master disc onto your newly-formatted disc via the RAM disc.

Make a Working Disc

Although all the programs on your **Devpac80** disc occupy a total space of nearly 160K you do not need all of them to develop programs. The programs that are essential for interactive development are:

Programs needed for interactive Development

HDE.COM, HDE.HLP, GEN80.COM, PMON.COM, PMON.MON

HDE.COM is the screen editor and menu system, HDE.HLP is its help file, GEN80 is the assembler and PMON is the debugger/disassembler.

If you don't want to work interactively you can use the assembler and debugger from within CP/M and use ED80.COM as your screen editor, again from within CP/M normally.

There are some useful utility programs on the master disc that you may wish to put on your working disc in addition to the above.

Useful Utilities

WP.COM, WD.COM, SD.COM, COMTOBIN.COM, BINTOCOM.COM, GTOG.COM, UNLOAD2.COM

WP is a file copying program, WD deletes files, SD gives a directory with sizes and COMTOBIN/BINTOCOM convert between CP/M COM files and the Amstrad Amsdos BIN files. All the utilities are explained later.

To make a work disc simply format a new disc, put the CP/M system on it and then copy the files you need to it using PIP or our WP program.

Amstrad PCW owners may find it useful to copy the necessary files onto their RAM disc (drive M:) for speedy development, to this end we provide a PROFILE.SUB file on the master disc that does this.

Copy this program to your working disc together with the boot program (JxxCPM3.EMS) to automatically copy all the **Devpac80** files on start-up.

Some Useful Utilities

Some useful (and small!) file utilities are provided with **Devpac80** to make file management and conversion more straightforward. The utilities WP, WD, SD, COMTOBIN, BINTOCOM, GTOG and UNLOAD2 are described below.

WP

WP.COM copies files from one disc to another. It is invoked by typing its name followed by its parameters at the CP/M prompt. The general form of the command line is

```
wp <source afn> <destination afn> [-q]
```

The items specified as <source afn> and <destination afn> above are standard CP/M *ambiguous file specifications*, with optionally a drive name at the front. An ambiguous file specification is a filename which can match more than one file; this is done using *wildcards*, which are described in great detail in your CP/M documentation. WP extends the definition slightly in line with CP/M's built-in DIR command, such that a drive name alone (such as B:) is equivalent to *.* on the specified drive. If this item is left out altogether, it is taken as *.* on the current (default) drive.

The item specified by [-q] is optional and will be described later.

Typical WP invocations, then, would be

```
wp a: m:    [RETURN]
```

which copies all files on drive A onto drive M,

```
wp m:    [RETURN]
```

which copies all files on the default drive to drive M and

```
wp b:*.com a:*.bak    [RETURN]
```

which copies all files on drive B with an extension of .COM to drive A with an extension of .BAK. If the source and destination files are the same, then WP prints an error message and returns to CP/M.

When a valid command line has been typed, WP collects the names of the matching files and displays each one in turn, followed by a prompt:

```
Copy (Y/N/A/Q/P/B/W)?
```

You may type Y to copy this file, N not to copy this file, P to go back to the previous selection, A to copy *this and all subsequent matching* files, Q to quit now without copying this or subsequent files, B to copy with a backup (extension .BAK) of the original file or W to copy this file without a backup (useful if you have specified option B on the command line).

If -Q is present as the *last* item on the command line, WP does not prompt and copies each matching file without asking.

WD

WD.COM deletes files from a disc. It is invoked by typing its name followed by its parameters at the CP/M prompt. The general form of the command line is

```
wd <afn>
```

The item specified as <afn> above is a standard CP/M *ambiguous file specification*, with optionally a drive name at the front. An ambiguous file specification is a filename which can match more than one file; this is done using *wildcards*, which are described in great detail in your CP/M documentation. WD extends the definition slightly in line with CP/M's built-in DIR command, such that a drive name alone (such as B:) is equivalent to *.* on the specified drive. If this item is left out altogether, it is taken as *.* on the current (default) drive.

Typical WD invocations, then, would be

```
wd a:    [RETURN]
```

which deletes all files on drive A and

```
wd b:*.com    [RETURN]
```

which deletes all files on drive B with an extension of .COM.

When a valid command line has been typed, WD collects the names of the matching files and displays each one in turn, followed by a prompt:

```
Delete (Y/N/A/Q)?
```

You may type Y to delete this file, N not to delete this file, A to delete *this and all subsequent matching* files or Q to quit now without deleting this or subsequent files.

SD

SD.COM is a utility to display a detailed directory listing and the disc free space. It takes exactly the same parameter types as CP/M's built-in DIR command: an *ambiguous file specification*, a drive name or no parameter at all. The files matching the given specification are listed on the screen along with their vital statistics. These include the length in CP/M *records* (128-byte units) and the size of the file rounded up to the nearest 1K boundary. If a file it set to *Read-Only*, an R is printed by its name; if a file is set to *System*, an S appears. Both can appear together for the same file.

The final part of the display is the bytes free on the disc, in 1K units.

COMTOBIN and BINTOCOM

COMTOBIN and BINTOCOM are provided to covert CP/M .COM files to Amstrad CPC .BIN files and vice versa. You will need COMTOBIN if you are writing programs using **Devpac80** that are designed to run on the native CPC machines and not under CP/M. Likewise BINTOCOM will be useful for debugging purposes.

COMTOBIN for converting CP/M COM files to Amstrad AMSDOS BIN files

Usage:

```
COMTOBIN destination[.BIN] source[.COM] [S] [&]Load [&Run]
```

[] means optional.

If you omit the drive then it will default to the currently logged-in drive.

If you omit the destination filetype then it will default to BIN and if you omit the source filename it will default to COM.

If there is an S after the source filename, all keyboard input will be suppressed, allowing COMTOBIN to be SUBMITted.

Load is the address at which the BIN file is to be loaded.

Run is optional and is the address at which to start executing the BIN file. If it is omitted then the value defaults to that of Load.

Both Load and Run will be read as decimal numbers unless they are preceded by an ampersand (&) in which case they will be read as hexadecimal numbers.

If you type just COMTOBIN and then [ENTER], you will be prompted with the word COMTOBIN) and you can then execute a sequence of commands. The command is entered as normal, but on completion you will be reprompted. To quit press [ENTER] alone.

Some example use of COMTOBIN:-

```
COMTOBIN
COMTOBIN amsfile cpmfile &1000
COMTOBIN AMSfile.bin CPMfile.com/S , &2000 , &5200
COMTOBIN amsfile=cpmfile.com 1024
COMTOBIN A:AMSFILE.BIN = A:CPMFILE;s;1024;&400
```

BINTOCOM for converting Amstrad AMSDOS BIN files to CP/M COM files

Usage:

```
BINTOCOM destination.[COM] source.[BIN] [S]
```

[] means optional.

If you omit the drive then it will default to the currently logged-in drive.

If you omit the destination filetype then it will default to COM and if you omit the source filename it will default to BIN.

If there is an S after the source filename, all keyboard input will be suppressed allowing BINTOCOM to be SUBMITed.

If you type just BINTOCOM and then [ENTER], you will be prompted (with the word BINTOCOM) and you can then execute a sequence of commands. The command is entered as normal, but on completion you will be reprompted. To quit press [ENTER] alone.

Some example uses of BINTOCOM:-

```
BINTOCOM
BINTOCOM cpmfile amsfile
BINTOCOM a:cpmfile.com amsfile.bin , S
BINTOCOM cpmfile=amsfile,bin
BINTOCOM A:COMFILE.COM = A:AMSFILE
```

GTOG AMSDOS DEVPAC TO DEVPAC80 SOURCE FILE CONVERSION

Syntax:

```
GTOG GEN80file GENA31file <ENTER>
```

The program GTOG.COM supplied on your **Devpac80** master disc allows you to convert source files created by our AMSDOS assembler (GENA31) to GEN80 source format.

The GENA31 source file should have been saved to disc using the editor's P command e.g.

```
P1,400,TEST.AMS <ENTER>
```

To convert this file to a GEN80 source file run GTOG from within CP/M like so:

```
GTOG TEST.GEN TEST.AMS <ENTER>
```

Notes:

Please note that GENA3.1 and GEN80 differ in some aspects regarding the syntax of source files:

1. GEN80 contains full operator-precedence arithmietic, whereas GENA3.1 does not. You may need to rewrite some of your expressions if they use multiplication or division *inter alia*.

2. GEN80 does not recognise the ENT directive. You must remove all references to ENT yourself.

UNLOAD2

UNLOAD2 is a public-domain utility that converts a .COM file into an Intel .HEX file. If you don't know what this means then you probably don't need UNLOAD2!

UNLOAD2 is useful if you are blowing ROMs or generally communicating with the outside world.

The usage of UNLOAD2 is:

```
UNLOAD2 filename [RETURN]
```

where filename.COM is a regular binary file. filename.HEX will be produced by the utility.

HiSoft HDE

Fast Interactive CP/M Editor

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™.

All Rights Reserved Worldwide. 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 **HiSoft Devpac80**.

It is an infringement of the copyright pertaining to **HiSoft Devpac80** and its associated documentation to copy, by any means whatsoever, any part of **HiSoft Devpac80** for any reason other than for the purposes of making a security back-up copy of the object code.

Contents

SECTION 1
Getting Started & Tutorial

On your **Devpac80** disc are two screen editors, HDE.COM and ED80.COM.

ED80 is simply an editor which can be used to produce text for whatever application you need, you can even write letters with it.

HDE is also a screen editor but is especially tailored to work efficiently with **Devpac80**, it has a menu system from which you can edit, assemble and debug your program. **HDE** also remembers all the assembler errors (if you had any) and allows you to go to the erroneous lines immediately, with the error message on the screen.

Both **HDE** and **ED80** are fully configurable, you can tailor the commands to suit your taste and you can also install the editor to suit your computer's particular screen codes.

To enable you to get the best out of these packages, we shall describe the menu system of **HDE** first and then how to use the screen editor, which is the same in both **HDE** and **ED80** except for the goto-error features of **HDE**. Then the installation program will be detailed. Before all that, let's see what should be on your disc:

The supplied disc contains the following files connected with the editor package:

1) HDE.COM (The interactive menu system/editor)
2) HDE.HLP (Help file for **HDE**)
3) ED80.COM (The non-interactive editor)
4) ED80.HLP (Help file for **ED80**)
5) JABBER.WOK (Example text file)
6) EXTRA.WOK (Example text file)
7) HDEINST.COM (The installation program for **HDE** & **ED80**)
8) ED80INST.MSG (Needed by **HDEINST**)

1.2 Using HDE

You enter **HDE** from CP/M by typing:

```
HDE [RETURN]
```

or

```
HDE filename [RETURN]
```

After some loading time a menu will appear on your screen:

```
      HiSoft Devpac80 Menu Selection

Start editing
Assemble
Run
Debug
Quit

Edit file:
Main file:
```

If you used `HDE filename` then the filename (with a default extension of `.GEN`) will appear after `Edit file:` and `Main file:`.

You can now press `S`, `A`, `R`, `D`, `Q`, `E` or `M`:

Start editing

This takes you into the editor to edit the filename given after `Edit file:`, if no name is present after `Edit file:` the cursor will be moved there and you can type in a filename, followed by `[RETURN]`. After this you will enter the editor.

Assemble

Assemble the file specified after `Main file:`. If no filename is present then the cursor will be moved after `Main file:` and you can type one in, followed by `[RETURN]`.

Once you have done this, **GEN80** will be loaded and the file assembled. Note that you must have **GEN80** on your disc to do this. After **GEN80** has finished, you will be returned to the menu, once you have pressed a key.

Run

Execute the file given by the `Main file:` with an extension of `.COM`. If no file is specified after `Main file:` you will be asked to type one in.

You will normally use `R` after assembling a program correctly. Once the program you have run finishes, you are returned to the menu.

Debug

`D` loads **ProMON** which then loads the `.COM` file specified after `Main file:`, if no file is given here you will be prompted to type one in.

Assuming **ProMON** finds the `.COM` file on your disc, it will load it in at address `#100` and look for a corresponding `.SYM` file. It will then automatically load the program's symbols and display `Symbols loaded`, now press a key for the debugger.

If no relevant `.SYM` file is found then the debugger is entered immediately after loading the file.

When you quit **ProMON** after invoking it like this, you will be returned to the menu after a message.

If `D` cannot find **ProMON**, it trys to load the compact debugger, **MON80**; if it can't find **MON80** either then it returns to the menu.

MON80 will not automatically load the `Main File` in, you must do it manually with the `R` command. Also **MON80** does not return to **HDE** after invocation, but to CP/M.

`PMON.COM` and `PMON.MON` or `MON80.COM` and `MON80.MON` must be on your disc for `D` to work.

Quit

Simply returns you to CP/M.

Edit file:

Pressing E allows you to type in the name of a new file to be edited. Once you have done this and pressed [RETURN], the program will be loaded and the editor will take charge.

When you leave the editor, you will return to the menu.

If you type in only the name with no extension (the 3 letters after the .) then an extension of .GEN is assumed.

Main file:

M allows you to enter a new Main file: name, followed by [RETURN].

The name here is used by the menu options A, D and R. A uses the name as the program to assemble while D and R look for the name with an extension of .COM as the file to debug or run respectively.

If you don't type in an extension, then .GEN is assumed.

1.3 Using ED80

You enter **ED80** from CP/M by typing

```
ED80 [RETURN]
```

You may, however, add a filename preceded by a space e.g.

```
ED80 MYFILE.TXT [RETURN]
```

This will cause that filename to become the Current File. If the Current File exists then it is loaded from the disc otherwise you simply start from scratch with an empty file.

When you leave **ED80**, you are returned to CP/M.

1.4 The Editor

The following description is of the editor which is common to both **HDE** and **ED80**.

1.4.1 The Status Lines

The upper status-line (the top line of the screen) has six sections and appears like this:-

```
JABBER.WOK      LINE:1 COL:1          INSERT
```

On the left is the name of the Current File. To the right is the line and column at which the cursor is positioned. The space to the right of the column number is for the commands you give to the editor. To the right of this is the mode (see next section) and finally there is a large space for messages.

The lower status-line (the bottom line of the screen) has three sections and appears like this:-

```
FREE:XXXXX      $                     $
```

On the left is the amount of free space you have left in memory which will vary depending on the size of the current text. The value is approximately equal to the number of characters you can type before the editor becomes full of text. The two $ signs mark the start of the Find and Substitute strings which are currently undefined.

1.4.2 The Text Window

The screen is best looked upon as a window onto the current text. This window may be moved in all four directions so that you can view any part of the file. If the window moves downwards then the text appears to move upwards and this is called upward scrolling. In the same way, if the window moves to the right then the text appears to move to the left (leftward scrolling). The scrolling is handled automatically and intelligently to give you the most convenient window onto the text.

1.4.3 Typing Modes

There are two basic typing modes: **INSERT** and **CHANGE. INSERT** mode is the more normal method of text entry. When you type a character in **INSERT** mode, all of the line to the right of the cursor is moved right one position before the character is entered. This means that the current line becomes longer by one character. In **CHANGE** mode, the character you type overwrites the current character (i.e. the one at the cursor) and thus the line remains the same length. You are not allowed to go over the end of the line in **CHANGE** mode.

In general, **INSERT** mode is used to build up a file and **CHANGE** mode is used to alter small sections within a line.

1.5 A Quick Tutorial

If you are quite familiar with the use of word-processors then a quick glance over this tutorial to clarify the points of difference with the one you are used to should be sufficient. If, however, you are in any doubt as to the full capabilities of a word-processor then it is strongly recommended that you work through this tutorial on your computer. Using a disc containing `ED80.COM` (or `HDE.COM`), `ED80.HLP` (or `HDE.HLP`), `JABBER.WOK` and `EXTRA.WOK`, enter CP/M and type:

```
ED80 JABBER.WOK [RETURN]    or
HDE JABBER.WOK [RETURN]     and, once the menu appears, press S
```

On the screen at the moment is the text of **JABBERWOCKY** from Lewis Carroll's *Alice Through the Looking-glass*. This is just a sample piece of text to enable you to become familiar with the editor. As a re-assuring start, verify that the help key (`[CTRL]-J`) will list the various one-key commands. After a quick look at the help page, pressing `[RETURN]` will get you back to the file.

An important factor in the efficient use of full-screen editors is gaining confidence in moving the cursor around the file. It is worthwhile trying out the two main cursor-moving commands which are:-

character left (`[CTRL]-S`) and
character right (`[CTRL]-D`)

If you keep pressing the character-right key until the cursor is just past the Y of JABBERWOCKY in the title and then press it a further time, you'll notice that the cursor *wraps around* to the start of the next line. Also notice how the line and column numbers change in the upper status-line.

If character spaces were the only way you could move the cursor then it would take a very long time to get to the end of some files and so, of course, there are many other ways to position the cursor. You can move the cursor one word to the left ([CTRL]-A) or one to the right ([CTRL]-F) or even straight to the beginning ([CTRL]-Q S) or end ([CTRL]-Q D) of the line.

Now move the cursor down ([CTRL]-X) and place it at line 7 column 1 just below All mimsy etc. As you may have realised, there is a line missing from the poem and the fourth line of the first stanza should read:

```
And the mome raths outgrabe
```

Type this line in preceded by the correct number of spaces and press [RETURN] at the end. You may have noticed that the second line of this verse is also incorrect and should read:

```
Did gyre and gimble in the wabe;
```

Move the cursor up to where the word the has been missed out i.e. the space after in and simply type the word the.

Note that the rest of the line is moved to the right after each letter. This is because we are in **INSERT** mode (see the upper status-line). You can change (toggle) the typing mode by pressing ([CTRL]-V). Now the word CHANGE appears in the upper status-line.

Move the cursor to the letter o of wobe which should be an a and simply type the letter a. Note that in **CHANGE** mode, the text is not moved to the right and the characters typed simply overwrite the existing ones.

If you finish off the line by typing be; then you can also see that you are not allowed to overwrite the end-of-line in **CHANGE** mode.

A word of explanation is in order here about end-of-line. At the end of each line is an invisible character (its value is 13) which can be cursored-over, deleted, found and substituted as can any other character. In fact the installation program gives you the option of displaying all end-of-line characters so you can see where you are more easily. The most usual way of displaying them is with <.

The second verse has been omitted from the poem and this is a good excuse to test the deleting commands and the auto indent feature.

Firstly go into **INSERT** mode ([CTRL]-V) and then move the cursor to the start of the blank line below at the end of the first verse. Now press [RETURN] twice to give a good separation between the stanzas. You can toggle auto-indent by pressing ([CTRL]-O I) and the message I/AUTO will appear on the upper status-line. Now type the first line preceded by the correct number of spaces so that the line starts directly under those of the first verse:

```
Beware the Jabberwock, my son!
```

If you make a mistake then you can delete the character before the cursor ([DEL]) or delete the character in front of the cursor ([CTRL]-G). Press [RETURN] at the end of the line and note that the new line starts immediately under the one above. This is due to the auto-indent and is an extremely useful feature when writing programs to improve legibility. Now type in the other three lines of the stanza ending all with [RETURN]:

```
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!
```

Typing mistakes are usually very common when entering programs and the editor has been designed to minimise the effects of the more common errors. Thus, whenever you delete a line, it is stored *until you start to edit another line* and can be recovered. To illustrate this, move the cursor until it rests on the line starting:

```
Beware the Jabberwock
```

and give the command to delete the line ([CTRL]-Y). As you can see,the line disappears from the text, but the restore line command ([CTRL]-O R) can be used to get the line back and in fact you may move the cursor to an entirely different place and *restore* the line again as many times as required. This manoeuvre can be quite useful for moving a line or copying it to some other place in the text.

If you start editing a line i.e. type in a few characters to an existing line and then use the restore line command, the line is restored to what it was originally.

In a large program using the Find and Substitute facilities is often the best way of getting to a known point in a program. To illustrate this, position the cursor on the first line of the poem, and give the find-first command ([CTRL]-Q F). This puts the cursor just after the first dollar sign on the bottom status-line. Now type in `Jabberwock.` (you can use the delete-character-left ([DEL]) if you make a mistake) and then [RETURN]. Just press [RETURN] for the second or Substitute string.

The cursor will now be positioned on line 34. Note that the two previous occurrences of `Jabberwock` are not followed by a full-stop. If you now put the cursor to the top of the file ([CTRL]-Q R) and redefine the Find string as `Jabberwock` (*without* the full-stop) then the cursor will first rest on line 10. Now the find next command ([CTRL]-L) will go to line 23.

The next part of the tutorial illustrates the way you can manipulate blocks of text, rather than just characters and lines.

To define a block of text you have to mark its start and end. The original Lewis Carroll poem duplicates the first verse at the end and so the aim is to mark the whole of the first verse as a block and then copy it to the end. However, this will be done in rather an unusual way to illustrate the block buffer in the editor. Put the cursor to the start of line 4 i.e. the start of the first line of the first verse and mark this point as the start of a block ([CTRL]-K B).

Now position the cursor at the end of the last line of the poem (the space after `outgrabe.`) and mark this as the end of the block ([CTRL]-K K). Now the unusual part: delete this block ([CTRL]-K Y)!!

You can now see that there is a star (*) after the figure showing the amount of free space left on the bottom status-line. This star means that there is a block in the block buffer.

You can see the size of the block by operating the free-space toggle ([CTRL]-O F). You get the block back by using the paste block command ([CTRL]-O P). Do this *now*. Note that the block is still there and can be pasted as many times as required.

Now move the cursor to the end of the file ([CTRL]-Q C) and paste the block again and lo! the last verse is duplicated. Exactly the same effect would have been achieved by marking the block and then copying it ([CTRL]-K C) except that the block would not have been put in the buffer. Both block delete and block move (which is exactly equivalent to delete followed by paste) put the block in the buffer if there is space.

It is possible to write a block to the disc ([CTRL]-K W) and read a block from the disc. As an exercise, move the cursor right to the end of the file and then issue the command to read a block ([CTRL]-K R). You are now prompted for the filename of the file you wish to read in. Type:-

```
EXTRA.WOK [RETURN]
```

(you can use [DEL] if you make a mistake typing in the name).

Naturally Alice did not understand the explicit and obscure sexual connotations of the poem as we do today and the poem stands as an interesting if rather distressing insight into Dodgson's dark, tulgey mind.

Finally, it is obviously very important to be able to save an edited file to the disc. There are three ways to quit. Firstly, you can abandon the file ([CTRL]-K Q). Here nothing is saved and the current text is lost.

Or you can save with no backup ([CTRL]-O Q). This will save the current text on the disc deleting a file of the same name if it existed. This method is generally used when space is low on the disc.

The most normal method of leaving the editor is to save with a backup ([CTRL]-K X). Here, if there is a file with the same name it is converted to type .BAK and thus is preserved.

Do the save-with-backup command ([CTRL]-K X). You are prompted for a filename, with the Current Filename already given for convenience. The Current Filename may be deleted if you require and the text saved under another name. In this case, however, just press [RETURN].

When silence and the A> prompt rules again, a look at the disc directory with DIR will show that the original file is now called JABBER.BAK and there is a new JABBER.WOK that is the file we have just edited.

It should now be perfectly possible (with frequent forays into the help-pages) to edit your own files. Before doing so it is advisable to cast a quick glance over the reference section as there are some very useful features documented there that have not been covered in this tutorial.

SECTION 2
Installing the Editor

The process of installing the editor (either **ED80** or **HDE**) involves three phases. The editor is first read in from the disc. Then, sections of the program are modified and finally it is written back out to the disc (as a .COM file) together with a help (.HLP) file. Thus the process involves a permanent change to the editor.

There are two reasons that you might want to run the installation program. Primarily, it may be that there are problems with the screen layout and the editor seems not to work at all. This will be due to incorrect terminal codes and in this case you should read the section on **Terminal Installation**. Alternatively, you may wish to modify some of the commands or options to suit either keyboard or taste. This procedure is covered in the section **Re-Defining the Editor Commands** and **Re-Defining User Options**. In either case you should first read the next section.

2.1 Starting up the Install Program

To run the installing program in order to install either **HDE** or **ED80**, insert your back-up disc and type:-

```
HDEINST [RETURN]
```

You will now see the copyright message. The purpose of the installation process is to alter the copy of your editor on the disc. To this end, some copy of the program (called the working copy) is read in from the disc into the machine. The first question is thus:-

```
Normally the working copy of HDE is
read in from a file called HDE.COM
Use another file instead
                          (Y/N)  ?
```

o install **HDE**, the reply will normally be N, the exception being when you have renamed a version of **HDE**. To install **ED80** you should answer Y to this question which will produce the prompt:-

```
[ESC] to abort
Omit file type (.COM assumed)
Enter filename
```

to which a filename should be typed in (omitting the filetype e.g. to use ED80.COM as the image, type ED80 [RETURN]). If you typed Y by mistake and really do want to use HDE.COM as the working copy then just type HDE. Whether you replied N to the opening question or Y and then specified a filename, the working copy will now be read in to the machine from the disc and the Installation Menu will appear.

There is now a copy of the editor in the memory of your machine ready to be altered and the Installation Menu on the screen.

```
HDE INSTALLATION MENU
---------------------

1. Return to CP/M
2. Alter screen codes
3. Save HDE as <working copy filename> (normally HDE.COM)
4. Save HDE as another file
5. Alter command codes
6. Alter user options
7. Load installation from .E80 file
8. Save installation to .E80 file

Type desired number:
```

If you are a first-timer using the installation program because the screen codes in the editor were wrong then turn first to the section **Terminal Installation** and then to **Leaving the Install Program**. The other sections in this chapter are **Re-Defining the Editor Commands**, **Re-Defining User Options** and **Use of Installation Files**.

2.2 Terminal Installation

Select option 2 from the main menu to alter the screen codes. You will be asked:

```
How many screen columns (80) ?
```

and then

```
How many screen rows (24) ?
```

In answer to each of these questions you should type in the correct number followed by [RETURN]. Pressing [RETURN] alone is equivalent to giving the answer in brackets.

The rest of the questions concern how the screen controller works on your machine. If you are in doubt about any of the questions, consult the manual for your machine. You are now asked for the:-

```
Cursor position lead-in sequence
 (    )   (    ) -
```

When the editor is in operation it has to be able to tell the screen controller to put the cursor at a certain position on the screen. To do this, it tells the controller the row and the column required. Most screen controllers require a special sequence of codes to indicate that the values to follow represent a row and a column. Thus inside the first set of brackets there will be the sequence as it is currently defined with the decimal values of the codes in that sequence in the second set of brackets. If the sequence is correctly set up then just press [RETURN] and move on to the next question. If the sequence is incorrect then it must be changed.

Assuming your screen controller does have a Cursor Position lead-in sequence (on Amstrad CP/M Plus computers it is [ESC] Y) then you should enter it now code by code (up to a maximum of four codes) terminated by [RETURN]. Each code may either be entered as a single keypress or as its decimal value terminated by [RETURN].

As an example, if the correct sequence for your controller was [CTRL]-K =. You could enter this either by typing

[CTRL]-K = [RETURN] or by typing

11 [RETURN] 61 [RETURN] [RETURN]

note the two [RETURN]s at the end. The first is to terminate the 61 and the second is to terminate the whole sequence.)

The next question asked is:

```
Is the row sent before the column (  )
                  (Y/N/ENTER) ?
```

The screen controller may require the row before the column, or the column before the row. As above, pressing [RETURN] is equivalent to giving the answer in brackets.

You are now asked:

Offset for column ( ) ? and then

Offset for row ( ) ?

When the values for the row and the column are sent, many screen controllers require an offset to be added to each. The values required for the offsets are those required to position the cursor at the top left of the screen (i.e. if the correct offsets for your machine were both 32 then sending the Cursor Position lead-in sequence, then 32, then 32 will put the cursor at the top left of your screen). If the value in brackets is correct then just press [RETURN] otherwise type in the correct value terminated by [RETURN]. You should consult the manual for your machine if in any doubt.

The next text to appear is:

```
Clear Screen sequence
 (  ) (  ) -
```

The layout is identical with that for the cursor positioning sequence detailed above. Press [RETURN] alone if the sequence for clearing the screen is correct or enter the correct code terminated by [RETURN] as above. If your controller does not recognize a sequence to clear the screen (possible but unlikely) then press D.

Next:

```
Clear to End of Line Sequence
(   )  (   ) -
```

prompts you for the sequence to clear to the end of the current line. Respond to the prompt exactly as above for the clear-screen sequence. It is quite possible that your screen controller does not recognize a sequence for clearing to the end of the current line. If this is so then press D to delete the sequence and the editor will perform the function by software (although more slowly than the controller would do it).

```
Inverse video on sequence
 (   )  (   ) -

Inverse video off sequence
 (   )  (   ) -
```

These codes will only be used if you are installing **HDE**. If your terminal supports, in some way, inverse video then enter the sequences to turn this feature on and off. If inverse video flipping is not supported, press D in answer to both questions to disable the facility in the editor.

```
Which RST 1-7  (   ) -
```

This question will appear only when installing **HDE**. You should enter the number (1 to 7 inclusive) of a restart that it is safe for **HDE** to use.

HDE will use this restart to allow running programs to return to the menu. This will involve lowering the top of the TPA while in **HDE** by a few bytes.

Normally, RST 6 (on an Amstrad) or 7 is a good choice. The restart number can be the same as used by **ProMON**.

```
Use lead-in (  )
            (Y/N/ENTER) ?

Use lead-out (  )
            (Y/N/ENTER) ?
```

These final questions concern the use of lead-in and lead-out sequences. These options allow you to send a command to the screen controller or run a small program at the start and end of an editing session. For example, this facility might be used to put your machine into 80 column mode for editing and reset back to 40 column mode on exit. However, unless you have an important reason for wanting to use this facility, it is advisable to answer `N` to both questions. If you answer `Y` to either you will be asked to specify a code sequence to send to the screen controller which you should enter as described above.

You will now be returned to the installation menu.

2.3 Redefining the Editor Commands

Pressing 5 from the main menu will allow you to alter the command definitions.

All of the commands will be shown and you have the opportunity to change the definition or accept it and pass on to the next command. After the last command you are returned to the main menu. For each of the commands the display format is:

```
Command name [keystroke definition | decimal definition]
```

where the `keystroke definition` is the sequence of keys the user presses to give the command and the `decimal definition` is the decimal ASCII value of those keys. These are alternatives.

At any stage you have the option to go back to consider the previous command, to retain the current definition or to change the current definition.

1) To backtrack to the previous command, press B

2) To retain the current definition press [RETURN]. The process then repeats for the next command. At the end you are returned to the main menu.

3) To change the current definition the new definition should be typed in element by element (up to a maximum of four elements) and terminated by [RETURN] after which the redefined command appears. If you are now sure that the definition is correct then press [RETURN] to pass to the next command, otherwise type in another definition and the whole process is repeated.

4) Definition elements are of two types. The first type is simply a keystroke and the second type is a sequence of digits terminated by [RETURN]. For example, the two ways to include a [CTRL]-Y (which has an ASCII value of 25) in the definition are:-

a) hold down the [CTRL] key and press Y
b) press 2 then 5 then [RETURN]

The two modes of entry of elements may not be mixed within the same definition. Thus if the first character of a definition was a number then all subsequent numbers are treated as their numerical value. However, if the first character was not a number then all subsequent numbers are treated as ASCII characters. This feature is included so that command definitions such as [CTRL]-K 0 can be entered directly (i.e. hold down [CTRL] and press K, then press 0)

If the definition given is the same as that of a previous command or a prefix to a previous command then this message will appear:

```
WARNING : There is a conflict between the
and                commands.
Do you wish to continue anyway (Y/N) ?
```

A response of Y will ignore the duplication and N will allow the current command to be re-defined. Note that if the editor is saved to the disc with two commands identical, the use of one of the commands will be lost.

It is recommended that you consult the reference section of the manual if in any doubt as to the meaning of some of the commands.

After the last command, you are returned to the main menu.

2.4 Re-Defining User Options

You can change the user options by selecting 6 from the main menu. There are four user options. They are:

```
Size of tabs (   ) ?
```

to which you should type in the tab size required followed by [RETURN] or [RETURN] alone to retain the value in brackets.

```
Tabs per scroll (  ) ?
```

When the cursor in the editor moves off the right-hand edge of the screen, the text window moves to the right (i.e. the text appears to scroll to the left). This left/right scroll works in units of one tab. On most screens, a value of one or two is best for this parameter. Enter the value as above.

```
End of line display (    )  (    ) ?

End of file display (    )  (    ) ?
```

A single key response is needed for both of these. If you don't wish the end of lines or the end of file to be displayed then press D to delete the current value, otherwise type the character you would like to be used for each (< is a common end-of-line marker with maybe | for end-of-file).

Although not normally used in word-processors, the markers can be useful in a program editor for distinguishing spaces and tabs at the end of lines and end of file.

After responding to these options you are returned to the main menu.

2.5 Use of Installation Files

There are many features of the editor that are alterable by the user.

Every copy of the editor naturally contains one set of these options. There is a type of file, however, called an Installation File that consists solely of the set of the alterable options. An Installation File is of type `.E80`.

To save the current installation information in a file, select option 7 from the main menu. You will then be prompted for a filename which you should type in terminated by `[RETURN]`.

It is possible that you will see the error message

```
Too many characters in commands
```

If so, you should decrease the number of characters used to define the commands.

To load an installation file, select option 8 from the main menu. As above, you will be prompted for a filename. If the file you give does not exist then the prompt will be repeated. You can press `[ESC]` to quit.

When the installation file is loaded into memory, it will overwrite the alterable options already present in the copy of the editor in memory.

The main use of Installation Files is when you are in the long-term process of tailoring your version of the editor to suite your own preferences. If you save each successive change you make to the installation then any changes you find undesirable can be overwritten by using the last installation file rather than going all the way through the commands. If you have an earlier version of **ED80** which you have modified you can use the new **HDEINST** program to save out a `.E80` from your old version. Then run it again to update your new versions of **HDE** and **ED80**. Note that when using an installation file from an old **ED80** then you will have to re-enter the inverse video codes.

2.6 Leaving the Installation Program

You can leave the install program by selecting option 1 from the main menu, but **BEWARE!** If you select option 1 then nothing will be changed on the disc. Thus if you are satisfied with the changes you have made in the last installation session, you should first use either option 3 or option 4. Both will save a copy of the editor (as a `.COM` file) and a help file (as a `.HLP` file) on the disc.

Option 3 will save both files under the name you specified at the beginning of the session (normally **ED80** or **HDE**) whereas option 4 allows you to change the name by which you will invoke the editor.

You may have more than one copy of the editor on the disc at the same time (under different names, of course) without a clash of help files.

Thus the normal method of leaving the install program will be first to select option 3 and then option 1. If you don't wish to save the results of your installing labours then select option 1 alone.

You may, when saving, get the error message

```
Too many characters in commands
```

in which case you should decrease the size of your command definitions.

It is well worth spending some time deciding on the design of the command definitions. A well-designed and succinct set will be easier to use and will also lead to quicker and more efficient editing.

SECTION 3
Command Reference Guide

This Section is intended as a short reference guide to the commands and features of the editor. In all cases, the default command is given in brackets followed by some space so that you may fill in the keypresses that you may have chosen using the Installation program. Where possible, the default command is the same as the Wordstar™ command. [CTRL]- means that the control key is to be held down, [RETURN] indicates that you should press [RETURN] or [ENTER] on your keyboard, and <CR> indicates a byte of the value #0D (ASCII 13).

3.1 Cursor-Moving Commands

Character left/right [CTRL]-S : [CTRL]-D

Move the cursor one character position left/right. Moving past the end of a line positions the cursor at the beginning of the next line. Likewise moving past the beginning of a line puts the cursor at the end of the previous line. This feature is hereafter called *wraparound*).

Word left/right [CTRL]-A : [CTRL]-F

Move the cursor to the beginning of the last/next word. Characters that constitute the boundaries between words are :-

" ( ) [ ] { } = + - * / < > ^ - ; : , # $ & \ [TAB]

and wraparound operates.

Tab left/right [CTRL]-OS : [CTRL]-OD

Move the cursor to the last/next tab position. Wraparound operates.

Start/End of line [CTRL]-QS :[CTRL]-QD

Move the cursor to the start/end of the current line. Wraparound does not operate.

Line up/down [CTRL]-E :[CTRL]-X

Move the cursor up/down one line. After moving up or down one line, the cursor column is always the same. Thus it may appear that the cursor is positioned beyond the end of a line. If another line up/down or page up/down command is issued then the cursor will move as described. However, if any other key is pressed, the editor will behave as though the cursor was at the end of the current line (ambiguous cursor).

Top/Bottom of screen

[CTRL]-OE :[CTRL]-OX

Move the cursor to the top/bottom of the screen.

Page up/down [CTRL]-R :[CTRL]-C

Move the text window down/up by one less than the number of non-status lines displayed on the screen. Thus a page up command on a 32-line screen will move the text window up by 29 lines (32 screen lines-2 status lines-1) and the old top line becomes the new bottom line. Ambiguous cursor operates.

Start/End of file [CTRL]-QR :[CTRL]-QC

Move the cursor to the start/end of the file.

3.2 Text Deleting Commands

Delete line [CTRL]-Y

Delete the current line. Note that the line is placed into the editing buffer and can be recalled into the text by use of the restore line command. The deleted line will be overwritten when the user next makes a change to any line.

Delete last character [DEL]

Delete the character to the left of the cursor. Wraparound operates.

Delete this character [CTRL]-G

Delete the character under the cursor. Wraparound operates.

Delete word left/right

[CTRL]-OT : [CTRL]-T

Delete from the cursor to the beginning of the last/next word. The characters that constitute the boundaries between are given under the Word left/right command above. Wraparound operates.

Delete to start of line [CTRL]-Q[DEL]

Delete from the cursor to the beginning of the current line.

Delete to end of line [CTRL]Q-Y

Delete from the cursor to the end of the current line.

3.3 Block Commands

Mark start/end of block

[CTRL]-KB : [CTRL]-KK

Place the block markers. A marker will be positioned at the cursor position. The markers are lost if the line containing the marker is altered subsequently.

Move block [CTRL]-KV

Delete the currently marked block from the text and place in the block buffer, then insert the block at the cursor position. If there is enough space the block will be retained in the buffer, but the less space there is, the longer the command will take.

Copy block [CTRL]-KC

Copy the currently marked block from the text to the cursor position.

Delete block [CTRL]-KY

Delete the currently marked block from the text and place in the block buffer. The less space there is, the longer this command will take. This is due to the procedure required to place the block in the buffer rather than abandoning it altogether.

Thus, if the amount of free space is very small (less than 256) you are asked whether to abandon the block. If you press Y then the block will be deleted from the text and not placed in the block buffer. If you don't want to completely abandon the block then N should be pressed, the block should be written out to the disc (from where it may later be read back in if desired) and then deleted.

Paste block [CTRL]-OP

Insert at the cursor the block currently in the block buffer. The block remains in the buffer if there is sufficient space.

Read block [CTRL]-KR

The user is asked for a filename. [RETURN] alone aborts the command. A filename followed by [RETURN] will search the disc for the filename given and insert it at the cursor. The response RDR: will read the block from the logical reader device.

Write block [CTRL]-KW

The user is asked for a filename. [RETURN] alone aborts the command. A filename followed by [RETURN] will write the currently marked block to the disc with the filename given.

Printing a Block

In response to the filename prompt, LST: will send the block to the current logical list device and may thus be used to print a block of text. The response PUN: will send the block to the current logical punch device. The whole file may thus be printed by setting the block markers to the start and end of the file and writing the block to LST: (but see **Printing the File** below).

3.4 Quick Cursor Movement

Goto line [CTRL]-OG

User will be prompted for a line number. This should be entered digit by digit (the **DELETE CHAR LEFT** command may be used as a destructive backspace) and after [RETURN] the cursor will be positioned at the start of the line given. This command is extremely convenient for quick access to an error reported by a compiler or assembler.

Goto start/end of block

[CTRL]-QB : [CTRL]-QK

Move the cursor to the start/end block marker.

Remember position [CTRL]-KO

The current cursor position is stored. The marker is lost if the line in which it lies is subsequently changed.

Return to position [CTRL]-QO

The cursor is positioned at the stored position.

3.5 Find and Substitute

Find first [CTRL]-QF

The current Find string is displayed. [RETURN] will retain the current string, otherwise you should type in the required Find string (up to a maximum of 32 characters) and then press [RETURN].

[DEL] may be used as a destructive backspace, [CTRL]-R will redisplay the previous string and [CTRL]-U will abort the operation leaving the strings as they were.

A control character may be entered by pressing the control meta-key ([CTRL]-P (see **Miscellaneous**) and then the control character (e.g. [CTRL]-P then [RETURN] enters a <CR> or [CTRL]-M into the string). Pressing the meta-key and then ? will return a value which is displayed as ? and counts as a wild-character when in the Find string.

After [RETURN] is pressed the operation is repeated for the Substitute string, and then the cursor is positioned at the start of the first occurrence of the Find string in the file.

Find next [CTRL]-L

The file is searched for the next occurrence of the Find string starting from one character after the cursor. A wild- character in the Find string will match with any character at all in the file.

Substitute and find [CTRL]-OL

The file is searched for the next occurrence of the Find string starting from the cursor. A wild-character in the Find string will match with any character at all in the file. When the string is found, it is replaced by the Substitute string and the cursor is positioned after the last character of the Substitute string. Finally the file is searched for the next occurrence of the Find string starting from the cursor.

Substitute all [CTRL]-OA

Starting from the cursor, all occurrences of the Find string in the file are replaced by the Substitute string. A wild-character in the Find string will match with any character at all in the file. The cursor is then placed after the last string substituted.

3.6 Leaving the Editor

Quit and Exit [CTRL]-KQ

You are asked whether to abandon the file. Pressing Y will cause a return to CP/M or to the **Devpac80** menu if the editor was invoked from **HDE** and the current text will be abandoned. Any other response will abort the command.

Exit without backup [CTRL]-OQ

The Current Filename is displayed after the prompt Filename:. This may be deleted using the **DELETE CHAR LEFT** command ([DEL]) and altered. When you are satisfied with the filename, [RETURN] will cause the current text to be saved on the disc under the filename given. You are then returned to CP/M or to the **Devpac80** menu.

A file already on the disc with the same name will be lost.

Exit with a backup [CTRL]-KX

Identical with above except that a file already on the disc with the same name as the Current Filename will be renamed as a .BAK file and any .BAK file with the same name will be lost.

Printing the File

With the Exit without backup command in response to the filename prompt, LST: will cause the current text to be written to the current logical list device and may thus be used to send a file to the printer. PUN: will write the text to the current logical punch device.

Note that after both these responses you will abandon the current text and the disc copy of the Current Filename will be unaltered. A better way of printing the whole file is to set the block markers at the start and end of the file and then write the block to `LST:` (see **Printing a Block** above).

3.7 Toggles

Toggle insert mode `[CTRL]-V`

Switch between **INSERT** and **CHANGE** mode. A character typed in **INSERT** mode will only be entered after the characters to the right of the cursor on the same line have been moved right one character position. A character typed in **CHANGE** mode will overwrite the current character. A `[CR]` may not be overwritten in **CHANGE** mode.

Toggle auto indent `[CTRL]-OI`

Auto indent will only operate in **INSERT** mode. The message `INSERT` will become `I/AUTO`. When indent is on and `[RETURN]` is pressed in **INSERT** mode, the next line will be indented so that it starts at the same column as the line above.

Toggle free space `[CTRL]-OF`

A star following the amount of free space indicates that there is a block in the block buffer. The free space toggle is used to check the size of the block.

3.8 Miscellaneous

Deliver tab `[CTRL]-I`

Will return a tab character (`[CTRL]-I` or ASCII 9).

The size of tabs may be defined. Tabs will be entered as a ASCII 9 in the file and will not be changed to spaces. They are treated in the main like any other character in the file.

Restore line [CTRL]-OR

If you are in the process of editing a line then this command will restore the line to what it was when you first positioned the cursor on it. If you are not in the process of editing a line then this command will insert in front of the current line, the last edited line.

This aspect of the command is useful because the **DELETE LINE** command places the deleted line into the line-buffer exactly as though it had just been edited. You may thus move a line from one place to another by deleting it, moving the cursor to the desired place and then issuing a **RESTORE LINE** command.

Disc directory [CTRL]-KF

The prompt `Filename:` is given. See **Rules for Filenames**. A reply of [RETURN] or [SPACE] alone will abort the command.

After the filename is terminated by [RETURN] or [SPACE], the screen is cleared and a directory is printed (in fact the directory given will be the same as that seen after the equivalent `DIR` command). Any key will then return you to the current text.

Erase file [CTRL]-KJ

The prompt `Filename:` is given. See **Rules for Filenames**. A reply of [RETURN] or [SPACE] alone will abort the command. After the filename is terminated by [RETURN] or [SPACE] the named file or files will be deleted from the disc.

Control meta-key [CTRL]-P

Any key pressed after the meta-key will be entered into the file as its literal value. This may thus be used to enter control characters into the file that are normally commands or prefixes to commands (e.g. [CTRL]-P then backspace enters a [CTRL]-H).

The meta-key can also be used in the same way to enter control characters in the Find and Substitute strings. In this case if ? is pressed after the meta-key a character is returned that is displayed as ? but acts as a wild-character in the find string.

Help key [CTRL]-J

Pressing the help key will display help pages giving information on the commands available from the editor and how to access them.

The last two commands are only available if the editor has been called from **HDE** and is being used interactively with the assembler, **GEN80**.

Goto Next Error [CTRL]-ON

Assuming you have just performed an assembly that has produced some errors, the assembler will have, by default, produced a file with the same name as your `Main file:` but with an extension of `.ERR` that contains information about the errors found in this assembly.

You can step through these errors one by one using the **Goto Next Error** command. You will be taken to the start of the line on which an error was found and the error message will appear on the top status line, to the right. You can then correct the error and issue another **Goto Next Error** command to find the next error.

If the next error is in another file (perhaps one that was included at assembly time) then you will be told:

```
Next error is in <filename>
```

on the bottom status line and the **Exit with a Backup** command will be entered. You can abort this with `[CTRL]-U` if you wish or press `[RETURN]` to save the current file, load the one with the next error in it and go to the error. You then repeat the **Goto Next Error** command as many times as you like. If there are no more errors then you will remain on the current line.

Goto First Error [CTRL]-OM

This takes you back to the first error reported and you may then use **Goto Next Error** to step through the errors from the beginning again.

SECTION 4
Prompts and Messages

There are four prompts produced by the editor. They appear on the upper status line. Two require a single key response and the other two require a string of characters terminated by [RETURN].

```
Abandon block: Sure?
```

This prompt requires a single character response. It appears if the user has issued the **DELETE-BLOCK** command and there are less than 256 bytes free. If you respond Y then the block will be deleted and lost (note that the block is normally saved in the block buffer and thus not lost) while any other response will abort the command.

The prompt also appears if you have issued any command the execution of which would overwrite the block in the block buffer. If you respond Y then the block in the block buffer will be lost and the command executed while any other response will abort the command.

```
Abandon file: Sure?
```

This prompt requires a single character response. It is produced after the **QUIT** command. Respond Y and the current text will be lost and you will be returned to CP/M or the **Devpac80** menu. Any other response will abort the command.

```
Filename:
```

This prompt requires a string of characters terminated by [RETURN]. It is produced after any command that requires reading from or writing to the disc. The response is interpreted as a filename and the maximum allowed length is 14 characters (See **Rules for Filenames**). In building up the filename, the currently defined **DELETE CHARACTER LEFT** can be used as a destructive backspace. When [RETURN] is pressed, the Current Filename is displayed for convenience although it may be deleted if desired and another name substituted.

If the name returned is null i.e. [RETURN] alone then the command is aborted.

There are three responses to this prompt that are not interpreted as a filename, viz LST: PUN: RDR: . These address the logical list device, the logical punch device and the logical reader device respectively.

Go to line:

This prompt requires a string of numbers terminated by [RETURN]. It is produced after the **GO TO LINE** command. The response is interpreted as a line number and the maximum allowed length is 4 characters (only numbers are accepted). In building up the number the currently defined **DELETE CHARACTER LEFT** can be used as a destructive backspace. [RETURN] alone aborts the command.

Rules for Filenames

A filename consists of three fields. The drivename, the filename and the filetype e.g. B: MYFILE .GEN

When giving a filename:-

1) The drivename is optional and if not given the current logged-in drive is assumed.

2) In a command where ambiguous filenames are allowed (i.e. **ERASE FILE** and **DISC DIRECTORY**) a ? may be used to represent any single character and a * may be used as if the remainder of the field in which it occurs (barring the drivename field) were filled out with ?s.

3) In the ambiguous filename commands a response of the drivename field alone is interpreted as though the filename and filetype were *.

For example:

B:MYFILE.*
Addresses files of any filetype on disc B called MYFILE

FILE*.GEN
Addresses files of type .GEN on the current disc whose filename starts with the letters FILE

FILE?.GEN
Addresses files of type .GEN on the current disc whose filename contains 5 letters and starts with the letters FILE

B: or B:*.*
The first form (drivename alone) can be used only for the **DISC DIRECTORY** command. Addresses all files on drive B

Error Messages

There are fourteen messages produced by the editor and they appear mainly on the upper status-line as do the prompts.

Out of memory
Indicates that there is not enough space in the machine to carry out the proposed command.

Line is too long
Produced when the length of the line would exceed the maximum allowed length (255 characters) if the proposed action was taken. This might either be simply the press of a key, or the deletion of a <CR>.

Undefined command
Indicates that the initial key of a command is correct, but the second or subsequent keys do not form a valid command.

Block start unmarked/Block end unmarked
Produced after any block operation if the start/end of the block has not been marked or the mark has been lost (i.e. the line containing the mark has been edited).

Block marks reversed
Produced after any block operation if the start of the block occurs after the end.

Invalid destination

Produced after a **MOVE BLOCK** or **COPY BLOCK** and indicates that the destination (cursor) lies between the start and end of the block.

Block too big

Produced after a **READ BLOCK** command and indicates that the file on the disc is too large to fit into memory.

No block in buffer

Produced after a **PASTE BLOCK** operation and is self-explanatory.

Marker lost

Produced after a **RETURN TO POSITION** command and indicates that the position marker has not been placed or has been lost.

No file/Bad filename

Produced after any command which prompts the user for a filename. The command indicates either that the filename is badly formed or inappropriate or the file does not exist.

Disc full

Produced after any command that tries to write to the disc. Indicates that either the disc or the disc directory is full. The user should consider using the **ERASE FILE** command.

No such line

Produced after the **GO TO LINE** command and indicates that the line number given is greater than the number of lines in the file.

Next error is in <filename>

Produced on the lower status line only when the editor is used from the **Devpac80** menu and when you have used the **Goto Next Error** command. Indicates that the next assembly error is in file `<filename>`; press `[CTRL]-U` to abort and stay in this file or any other key to save this file, load `<filename>` and goto the next error.

SECTION 5
Technical Details

5.1 Internal File Format

Text is held simply as a string of ASCII characters. The end-of-line sequence is <CR> (ASCII 13) rather than <CR><LF>, allowing the user greater text space. The end-of-file is marked by a <NULL> (ASCII 0).

When the text is written to the disc, however, <CR> is replaced with <CR><LF> and the <NULL> is replaced by [CTRL]-Z (ASCII 26) thus making the disc file written by the editor compatible with normal CP/M text files.

The maximum line-length is 255 characters. Note that the cursor column number may exceed 255 due to tab and control characters (in which case the column number displayed on the status-line is 255).

The maximum number of lines in the file is limited only by memory considerations, but note that the line number display on the status-line is only of four digits (due to space considerations) and the **GO TO LINE** command can only reach line 9999 and no further. All other commands will work as normal if the number of lines exceeds 9999 (although note also that on most systems the average number of characters per line would have to be about three for the line numbers to exceed 9999).

5.2 Non-Printing Characters

Characters of ASCII value less than 32 decimal (Control characters) are treated as any other character. They may be entered into the file by pressing the control meta-key ([CTRL]-P) and then the control character desired. If the terminal is capable of producing characters of value greater than 127 decimal, then these characters are entered as any other into the file and are displayed as ?.

The meta-key may also be used to specify control characters in the find and substitute strings in the same way as above. An obvious use for this feature is to find the end-of-line character (reached by [meta-key] [RETURN] or [meta-key] [CTRL]-M). The ? character when pressed after the meta-key returns #80. This character is displayed as ? in the find and substitute strings, but is treated as a wild-character in the find string i.e. it will match with any character at all in the file. (Note that if a terminal has a key that can return #80 then this will be in all respects identical to [meta-key] followed by ?).

5.3 Data and External Devices

Whenever you give a command that would normally access the disc (i.e. **READ BLOCK, WRITE BLOCK, EXIT WITHOUT BACKUP**) there are three responses to the prompt Filename: that are interpreted as logical external devices.

1) LST: if used for a write operation will send the data to the current logical list device which is normally a printer (but may, of course, be set from CP/M using STAT). When the data is sent a <LF> character (ASCII 10) is sent after every <CR> as is usual for CP/M files.

2) PUN: if used for a write operation will send the data to the current logical punch device. As above, every <CR> is sent as <CR><LF>, but unlike the use of LST: a [CTRL]-Z (ASCII 26) is sent after the data to mark the end-of-file. [CTRL]-Z is the standard CP/M end-of-file character.

3) RDR: when used for a read operation is designed to be compatible with PUN: or indeed any standard CP/M data transfer operation. The top bit of every character is reset (thus masking out any parity bits sent by the transmitting hardware) and all <LF> characters are ignored to produce the standard editor internal format. RDR: requires a [CTRL]-Z character to mark the end-of- data. Files may thus easily be transferred from one machine to another from inside the editor by use of PUN: and RDR:.

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.

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™.

All Rights Reserved Worldwide. 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 **HiSoft Devpac80**.

It is an infringement of the copyright pertaining to **HiSoft Devpac80** and its associated documentation to copy, by any means whatsoever, any part of **HiSoft Devpac80** for any reason other than for the purposes of making a security back-up copy of the object code.

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.

1.1 For 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 EXTRNal) 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.
```

DEFW expression{,expression,expression etc.}
DW expression{,expression,expression etc.}

DEFW or DW 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 will be evaluated as 16 bits. For each expression, the appropriate word, or two bytes (starting from the Location Counter and incrementing by 2 after each expression) is set to hold the value of the expression. The least significant byte is placed at the lower address while the most significant byte is placed after it i.e. normal Z80/Intel format. Examples:

```
defw    10000,1000,100,10,1
dw      label+2,label+4
```

DEFL expression
ASET expression

DEFL must be preceded by a label and 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. DEFL is very similar to EQU except that if a label that has already been defined is redefined using EQU, an error results. The use of DEFL to redefine a label overrides this error, and means that a label may act as a variable during the time of assembly. Thus the code:-

```
label1    EQU  0
label1    EQU  1
```

will produce an error, whereas the code:-

```
label1    EQU  0
label1    DEFL 1
```

is legal. The DEFL directive may be used on the same label as many times as desired. This feature is often used in conjunction with macros to avoid the same piece of code being produced repeatedly. See **Section 2.12** on macros below.

ASET is an alternative for DEFL.

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    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   equ    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 to 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.

HiSoft ProMON

Fast Interactive CP/M Debugger

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™.

All Rights Reserved Worldwide. 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 **HiSoft Devpac80**.

It is an infringement of the copyright pertaining to **HiSoft Devpac80** and its associated documentation to copy, by any means whatsoever, any part of **HiSoft Devpac80** for any reason other than for the purposes of making a security back-up copy of the object code.

Contents

SECTION 1
ProMON Debugger

This Section describes the professional version of the debugger (called **ProMON** from now on) which is provided for those who want the maximum facilities; the pro version allows advanced features such as symbolic debugging, conditional breakpoints, watchpoints, page swapping (under CP/M Plus) etc. **ProMON** is obviously larger than the compact version and if you wish to debug very large programs then you are advised to use the compact package. However **ProMON** is still only just over 12K in length. We recommend that you use **ProMON** unless you have particular need for the smallest possible debugger. **ProMON** needs a screen with at least 80 columns to work correctly.

The command set of the professional model is very different from that of the compact version although the underlying concepts are very similar.

1.1 Getting Started

Your supplied disc will hold the two programs required to run the professional model: `PMON.COM` and `PMON.MON`. To activate simply type:-

```
PMON {command line} [RETURN]
```

ProMON will now load, tell you what type of CP/M system it has found and ask you:

```
Filename:
```

If you wish to load a program to debug into memory now then type a valid CP/M filename here (an extension of `.COM` will be assumed if you don't type one), otherwise hit `[RETURN]`. If you asked to load a program and **ProMON** finds a `.SYM` file for this program on the disc then it will say:

```
Load symbols?
```

Answer Y or N to this question to load up the symbols or not.

If you say Y then the symbols for the program will be loaded and the message `Symbols loaded` displayed; now hit any key. After this the Front Panel (see below) will appear and you are ready to debug your program.

If you reply N then the Front Panel will appear immediately.

Another way of getting into **ProMON** is from the menu in **HDE**; simply type D from the **HDE** menu and **ProMON** will load (assuming it is on the disc). The `Main file` on the menu will be loaded automatically (if present) together with any symbols (if a `.SYM` file is present for the `Main file`).

If you invoked **ProMON** from CP/M and you type a command line after **ProMON** then this command line will be treated normally i.e. it will be used to set up the two default FCBs at #5C and #6C and the command line will be copied into location #81 with its length in #80. This is so you can debug programs that need a command line, example:

```
PMON file 1 file2,options [RETURN]
```

will load **ProMON**, put `FILE1 FILE2 ,OPTIONS` at address #81, `FILE1` at #5D and `FILE2` at #6D. CP/M upper-cases the command line.

Now, to get the most out of **ProMON**, please read through the rest of this Section. If, after that, you are not sure how to use **ProMON**, work through the **Devpac80** tutorial.

ProMON loads itself into high CP/M memory, just under the top of the TPA, and then adjusts the address at locations 6 and 7 so that CP/M thinks that the top of the TPA is just under where **ProMON** loaded itself. Thus, in some ways, it behaves like an RSX does under CP/M Plus.

When a `.SYM` file is loaded, **ProMON** creates a symbol table and a hash table for the symbols and again lowers the TPA accordingly.

ProMON is roughly 12K long and 6K of the code must always be in common RAM (above #C000) on CP/M Plus systems.

This means effectively that you must have at least 54K of available TPA on CP/M Plus systems before loading **ProMON**. If there is not enough room above `#C000` for **ProMON**'s common memory code, you will see the message `Low TPA!` at load time. **ProMON** will still run if you see this message but you will not be able to bank-switch.

1.2 The Front Panel

After loading **ProMON** as above a *Front Panel* appears. The name Front Panel stems from the type of panels that are mounted on mainframe and mini computers to provide information on the state of the machine at a particular moment, usually through the use of flashing lights. These lights represent whether or not particular flip-flops (electronic switches) within the computer are open or closed; the flip-flops that are chosen to be shown on this panel are normally those that make up the internal registers and flags of the computer thus enabling programmers and engineers to observe what the computer is doing when running a program.

So these are hardware front panel displays; what **ProMON** provides you with is a software front panel - the code within **ProMON** works out the state of your computer and then displays this information on the screen.

Let's have a look at **ProMON**'s front panel; make sure you've made a working copy of **Devpac80** (see the front of this manual) and that you have also copied the program `FILES.GEN` onto your working disc. Now load CP/M, insert your **Devpac80** working disc and type:

`gen80 files [RETURN]` to assemble `FILES.GEN`

when the assembly has finished (with no errors, hopefully!) type:

`pmon [RETURN]`

In response to `Filename:` type:

`files [RETURN]`

and then press `Y` when asked `Load symbols?` and `[RETURN]` once you have the `Symbols loaded` message.

The **ProMON** front panel is now on the screen, it should look something like this:

Example ProMON Front Panel

```
}0100            LD   SP,(#0006)      PC 0100   00E0 66 20 6D 65 6D 6F 72 79 f memory
 0104            CALL TITLE            SP F5FE   00E8 20 74 68 61 74 20 43 50  that CP
 0107 ONE_TIME   LD   C,#11            IY 0000   00F0 2F 4D 20 75 73 65 73 20 /M uses
 0109            JR   TIME1            IX 0000   00F8 61 73 20 61 20 62 75 66 as a buf
 010B NEXT_TIME  LD   C,#12            HL 0000 > 0100 ED 7B 06 00 CD 30 01 0E m{..M0..
 010D TIME1      CALL CPMFCB           DE 0000   0108 11 18 02 0E 12 CD C9 01 .....MI.
 0110            CP   #FF              BC 0000   0110 FE FF 28 18 CD 4B 01 CD ~.(.MK.M
 0112            JR   Z,FINISH         AF 0000   0118 64 01 CD 80 01 30 08 CD d.M..0.M
 0114            CALL GET_TO_NAM                 0120 3C 01 CD AC 01 18 E0 CD <.M,..'M
 0117            CALL PRINT_FCB         Alts     0128 36 01 18 DF CD 42 01 C7 6.._MB.G
 011A            CALL CHECK_NAME       HL'0000
 011D            JR   NC,NAME_OK       DE'0000        ProMON 2.7 (C) HiSoft 1987
 011F            CALL BAD_MESSAG       BC'0000   Break           Condition/Scale Count
 0122            CALL DELETE_BAD       AF'0000
 0125            JR   ONE_TIME
 0127 NAME_OK    CALL GOOD_MESSA       IR 0075
 012A            JR   NEXT_TIME        Ints ON
 012C FINISH     CALL CONCLUDE
 012F            RST  0                 Flags
 0130 TITLE      PUSH HL
 0131            LD   HL,TITLE_M
 0133            JR   OUT_MESS         Bank: 01
 0135 GOOD_MESSA PUSH HL
 0137            LD   HL,GOOD_M
 013A            JR   OUT_MESS
 013C BAD_MESSAG PUSH HL
 013D            LD   HL,BAD_M
 013E            JR   OUT_MESS
 0140 CONCLUDE   PUSH HL

Command:
```

This is a sample screen from the 31-line, 90-column version running on an Amstrad PCW8256. If you have installed **ProMON** for a smaller screen then you won't see so many lines of disassembly and the symbols will be shorter.

The Front Panel screen display is composed of three main sections:-

a) The Register Display
b) The List Display
c) The Memory Display

`PC 0100`	the program counter
`SP B906`	the stack pointer
`IX 0000`	the **IX** register
`IY 0000`	the **IY** register
`HL 0000`	the **HL** registe
`DE 0000`	the **DE** register
`BC 0000`	the **BC** register
`AF 00FF`	the **AF** register
`Alts`	the alternate register set
`HL'0000`	
`DE'0000`	
`BC'0000`	
`AF'0000`	
`IR 007A`	the interrupt and refresh registers
`Ints ON`	
`Flags`	
`SZ H VNC`	the Z80 status flags
`Bank: 01`	which CP/M Plus bank we are in

This display shows the values held by the various internal Z80 registers including the Program Counter (PC), Stack Pointer (SP) and flag register. Remember that the `HL`, `DE` and `BC` registers (plus the alternate equivalents) may each be regarded as one 16 bit or two 8 bit registers.

Also shown is the interrupt status (`Ints ON`/`Ints OFF`) and, for CP/M Plus computers, the current CP/M bank number. Bank 1 is the normal CP/M TPA bank, Amstrad CP/M Plus computers use Bank 0 for much of the BDOS code.

To the left of the register display is the list display:

```
}0100             LD   SP,(#0006)
 0104             CALL TITLE
 0107 ONE_TIME    LD   C,#11
 0109             JR   TIME1
 010B NEXT_TIME   LD   C,#12
 010D TIME1       CALL CPMFCB
 0110             CP   #FF
 0112             JR   Z,FINISH
 0114             CALL GET_TO_NAM
 0117             CALL PRINT_FCB
 011A             CALL CHECK_NAME
 011D             JR   NC,NAME_OK
 011F             CALL BAD_MESSAG
 0122             CALL DELETE_BAD
 0125             JR   ONE_TIME
 0127 NAME_OK     CALL GOOD_MESSA
 012A             JR   NEXT_TIME
 012C FINISH      CALL CONCLUDE
 012F             RST  0
 0130 TITLE       PUSH HL
 0131             LD   HL,TITLE_M
 0133             JR   OUT_MESS
 0135 GOOD_MESSA  PUSH HL
 0137             LD   HL,GOOD_M
 013A             JR   OUT_MESS
 013C BAD_MESSAG  PUSH HL
 013D             LD   HL,BAD_M
 013E             JR   OUT_MESS
 0140 CONCLUDE    PUSH HL
```

The list display consists of a disassembly of instructions initially starting from address #100. If any of the instructions disassembled is at the same address as held by the program counter (PC) then that instruction will be marked on the display with a right curly bracket }.

Symbols will be included in the disassembly if a .SYM file has been loaded for the program under inspection. You have the option of loading symbols when you first enter **ProMON** and when you read in a file using the FR command (see later).

On the top right of the screen is the memory display:

Memory Display

```
  00E0 66 20 6D 65 6D 6F 72 79 f memory
  00E8 20 74 68 61 74 20 43 50  that CP
  00F0 2F 4D 20 75 73 65 73 20 /M uses
  00F8 61 73 20 61 20 62 75 66 as a buf
> 0100 ED 7B 06 00 CD 30 01 0E m{..M0..
  0108 11 18 02 0E 12 CD C9 01 .....MI.
  0110 FE FF 28 18 CD 4B 01 CD ~.(.MK.M
  0118 64 01 CD 80 01 30 08 CD d.M..0.M
  0120 3C 01 CD AC 01 18 E0 CD <.M,..'M
  0128 36 01 18 DF CD 42 01 C7 6.._MB.G

       ProMON 2.7 (C) HiSoft 1987
Break            Condition/Scale Count
```

The display is a snapshot of an 80-byte area of memory, initially centred on address #100. The addresses are shown down the left- hand side with the contents of the next 8 bytes from the address shown to the right of it (in hexadecimal). Following this, to the right, is the ASCII representation of these 8 bytes with . being displayed if the code cannot be usefully interpreted.

You will notice that one of the addresses on the memory display has a > symbol to its left; this address is known as the Memory Pointer - this is a concept internal to **ProMON** and has nothing to do with the Z80. You may set the Memory Pointer independently by using the MA command - see below.

Underneath the display of memory addresses and values comes the copyright message and, under that, the various breakpoints, conditional breakpoints and watchpoints that you have set are listed. For full details, see the section on breakpoints.

1.3 ProMON Commands

There is a wide range of commands that may be entered and executed whenever the front panel is displayed and the command prompt `Command:` is present at the bottom left of the screen

Before proceeding to describe these commands in detail, we shall explain the powerful expression handler within **ProMON** since there are many times when you will want to enter an expression when using the debugger.

1.3.1 Expressions

There are many times when you will find yourself wanting to enter a expression when using **ProMON** e.g. modifying memory, searching for a string, updating a register etc.

The expression handling in **ProMON** allows a wide range of operators to be used, with full precedence which may be overridden by the use of brackets; the expression handler is the same as that used in **GEN80**.

Curly brackets (`{` and `}`) may be used to force indirection e.g. `{6}` returns the word held at memory locations `0006` and `0007`.

Items in expressions are either symbols, in which case their current value is used, registers known by their usual names (`PC`, `SP`, `IX`, `HL` etc.), Reserved Words (`MP` for Memory Pointer, `WP` for watchpoint, `HIGH` and `LOW`, described later) or numbers or characters.

Numbers are one of the following:

1) A decimal number. A sequence of decimal digits preceded by a backslash (`\`) character e.g. `\32768`.

2) A hex number. Any sequence of hex digits (`0-9`, `A-F`, `a-f`). You can precede the number with a hash character (# or ASCII 35) if you wish but this is not necessary. The default number-entry mode in **ProMON** is hexadecimal since this is more natural when debugging e.g. `4A2E #4A2E ae`

3) A binary number. The % character followed by binary digits e.g. %11011111

Literal characters and strings are represented by enclosing them in double or single quotes.

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)

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.

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

Curly brackets indicate indirection and the value of the expression within the curly brackets ({ }) will be considered an address so that the value returned by {x} will be the 16-bit word at location x and x+1, Intel format i.e. low-order byte first. If you require only the 8 bits at location x then you should use {x}.and.255 to return the low order byte fetched.

The Reserved Words HIGH and LOW are described under **Execute Single** below while the Reserved Word WP is described under **Watchpoints.**

Operator Precedence Table

1)	Unary plus (+) Unary minus (-) Logical NOT (.NOT.)
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 **ProMON**:-

```
#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"
'A string'
```

Notes on the operators

`.NOT.` is a unary operator. To produce the same effect as (`label≠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).

You may abort the entry of an expression by using `[ESC] [RETURN]`.

Now for the commands available within **ProMON**.

1.3.2 The Commands Available

All **ProMON** commands are two-character commands designed to be mnemonically significant and, hopefully, easy to remember.

After you press the first character of the command, a list of second characters available, with a description of each command, will be displayed to help your choice.

Having chosen the second character of the command, the command description remains on the screen as confirmation; you can abort the command at this stage by pressing `[ESC]` followed by `[RETURN]`.

If you wish to abort the command after entering part of an expression, use [DEL] to delete back to the start of your entry and then press [ESC] followed by [RETURN].

Throughout the following, messages displayed by **ProMON** will be shown in *italics* to distinguish them from your keyboard input.

Memory Commands

Set the Memory Pointer Address MA

Enter an expression. The Memory Pointer will be set to the value of the expression and the memory display updated accordingly. Examples:

```
Memory - Address: pc+100 [RETURN]
Memory - Address: \32768 [RETURN]
Memory - Address: {1}+3*10 [RETURN]
```

Set the Memory Bank MB

Enter an expression. **ProMON** will now work on the contents of the memory bank specified by the expression. Normally, values of 0, 1 and 2 are useful. *This only works in Banked CP/M Plus systems.* Example:

```
Memory - Bank: 0 [RETURN]
```

Compare Memory MC

Enter a memory address and you will then be prompted `With:` and then `Length:`. Enter an expression in each case.

The command then compares two blocks of memory given by the first two addresses you specified and of the length you asked for. If any byte does not compare, the addresses and contents in each block will be displayed.

You can pause the display of mismatched bytes by hitting any key, then hit to [ESC] abort and return to the front panel or any other key to continue the comparison. Example:

Memory - Compare: Start [RETURN]
With: 8000 [RETURN]
Length: 1000 [RETURN]

Memory Fill MF

You are prompted for First: and Last: values, enter expressions representing memory addresses between which you wish to fill.

Then you will be asked With:, enter the 8-bit value with which you want to fill memory. All locations between First and Last inclusive will then be set to this byte value. Example:

First: 1000 [RETURN]
Last: 1fff [RETURN]
With: 0 [RETURN]

Memory Move MM

You will be prompted to enter three addresses, First:, Last:, To:. Assuming that you give valid expressions, the command will move the block of memory given by First and Last inclusive to the address you enter after To.

The move is intelligent in that memory may be moved over itself, forwards or backwards. Example:

First: Start+20 [RETURN]
Last: Start+30 [RETURN]
To: 5800 [RETURN]

Memory View MV

Simply displays memory in the Memory Display rather than registers. Use **Register View** (RV) to display registers and indirections off them.

Memory Window MW

Moves the cursor into the Memory Display. You can now use your cursor keys (or, by default, Wordstar-style cursor keys) to move the cursor around the Memory Display. The keys available are shown below.

↑	([CTRL]-E)	Cursor Up
↓	([CTRL]-X)	Cursor Down
←	([CTRL]-S)	Cursor Left
→	([CTRL]-D)	Cursor Right
[CTRL]-↑	([CTRL]-R)	Page Up
[CTRL]-↓	([CTRL]-C)	Page Down
[TAB]		flip between ASCII and hex display
[ESC]		returns to command mode

You can instal,your cursor keys into **ProMON** using the installation program, PMONINS.COM (see the next section).

Wherever you are on the Memory Display you can modify the byte under the cursor by typing in either a hexadecimal digit or an ASCII value depending whether you are on the hex or ASCII display.

Be careful you don't corrupt **ProMON** or important system memory!

Press [ESC] to exit this mode and return to command mode.

Register Commands

Register Update RU

Enter a register name as displayed on the Register Display followed by a space and then the value you want to assign to the register. As usual this value can be any general expression involving literals, symbols and other register names if required. Examples:

```
Register - Update: pc 100 [RETURN]
Register - Update: hl de+100 [RETURN]
Register - Update: ix {Line_Number}+2 [RETURN]
```

Be very careful updating the Stack Pointer!

Register View RV

Displays the registers and what they point to in the Memory Display, use **Memory View** (MV) to flip back to memory addresses and their values. The values on the Stack Pointer are shown as words while the values on the other registers are shown as bytes with the ASCII equivalent alongside.

This display is useful when single-stepping code where a number of registers point to buffers whose contents you want to monitor. If there is just one area of memory you want to keep on the screen while debugging, then it is more natural to display memory rather than registers.

File Commands

File Read FR

This command will produce the prompt `File Read:` to which you should give the filename you wish to read in followed by `[RETURN]`.

The filetype will default to `.COM`. In response to `First:` give the address you wish to load the file to. Pressing just `[RETURN]` here will load the file to the standard CP/M base file address of `#100`. **ProMON** will inform you of the address of the end of the last block of the file loaded and then you should press a key.

If there is a corresponding `.SYM` file on the same disc as the file you have loaded the message `Load Symbols?` will appear; answer `Y` to load the file's symbols for symbolic debugging or `N` otherwise. If symbols are loaded successfully the message `Symbols loaded` will appear, press a key to return the Front Panel. Example:

```
File - Read: test [RETURN]
First: [RETURN]
Load Symbols? Y
```

will load the file `TEST.COM` from the disc into location `100h` onwards and then load the `TEST.SYM` file ready for symbolic debugging.

File - Write FW

Type the filename of the program/code you wish save from memory. The default filetype is .COM.

In response to First: and Last: you should give the start and end addresses (inclusive) of the block of memory you wish to write to the disc, each followed by [RETURN]. If you press [RETURN] by itself to these two questions then the relevant start and end addresses of the last file read (using FR) will be used.

File Command line FC

Allows you to type in a command line that will be placed at address 80h which is where CP/M places a file's command line. The length of the line is placed at 80h and the line itself from 81h onwards. For example suppose you are debugging a program called UNERA that un-erases a file and that this program expects to be run from CP/M like this:

```
UNERA MISTAKE.COM [RETURN]
```

When you run a program like this from CP/M, CP/M copies the command line (in this case MISTAKE.COM) into locations 81h onwards and puts the length (11) in location 80h. If you are debugging UNERA from within **ProMON** you can set up this command line by using the FC command, type:

```
FC
File - Command line:  MISTAKE.COM [RETURN]
```

Note that you should enter a space before MISTAKE.COM because this is judged to be part of the command line.

The other way of setting up this location is to include the command line when invoking **ProMON** e.g.

```
PMON MISTAKE.COM [RETURN]
```

this only works when **ProMON** is entered from CP/M & not from **HDE**.

Note that FC does *not* set up the default FCB at #5c but just the command line at #80.

File - Zap syms

Clears the symbol table. This is useful if you have loaded a new file with no symbols and you don't want the old symbols used.

Search Commands

Search Byte/string SB

Enter a series of expressions separated by spaces and then hit [RETURN]. This command will search memory (in the currently-selected bank) for the pattern so defined and then update the Memory Display to point to the found pattern. Example:

```
Search - Byte/string: 3e "a" c9 [RETURN]
```

will search from the current memory pointer for the byte sequence 3e 61 c9.

As usual, you can enter any expression involving symbols or even register names as part of your search string. Note, though, that this is a *byte* search so that each expression is evaluated to its bottom 8 bits i.e. modulo 256.

To search for the next occurrence of the pattern use **Search Next**.

Search Mnemonic SM

This command allows you to search memory for a pattern that you type in as a line of assembly code e.g.

```
Search - Mnemonic: A,(HL) [RETURN]
Search - Mnemonic: JP Start [RETURN]
```

The only restriction on what you type in is that it must obey the disassembler's syntax so that you must use capital letters and use a # before hex numbers e.g. `LD HL,#8000` rather than `LD HL,8000h` etc.

While the search is going on you may press any key to interrupt the search and return to the front panel. The search may take some time.

Search Next SN

Search for the next occurrence of the pattern you defined with the **Search Byte/string** command. As a short-cut, you can use just the character N (or n) from command mode to do this.

Breakpoints

Breakpoints allow you to stop the execution of your program at specified points within it. **ProMON** has a wide range of such commands and we'll spend a little time now explaining the different types of breakpoints within **ProMON** and how to use them.

What is a Breakpoint?

A breakpoint consists quite simply of a Z80 restart instruction; this restart instruction is one byte long and causes execution to be transferred to low memory. There are 8 restart instructions on the Z80, RST 0 to RST 7; RST 0 goes to address 0 (which is warm boot under CP/M), RST 1 to address 8, ..., RST 7 to address 38h. You can choose which restart is used for breakpoints by running the installation program; by default RST 7 is used except on Amstrads when RST 6 is used.

When your program executes the breakpoint restart, control is passed to **ProMON** because the debugger has patched the low memory address to which the restart goes. **ProMON** then decides whether or not to halt execution of your program or continue; this depends on the type of the breakpoint. There are four types of breakpoint:

Hard Breakpoints

If you have set a hard breakpoint (using the BS or EB commands) then execution will always halt at the breakpoint and the front panel will be displayed showing the current state of all your registers, the PC, flags etc.

When the breakpoint is encountered the message Hard Break will appear in the top left of the screen, now hit a key; Y will keep the breakpoint set, any other key will reset it before returning to the front panel.

Conditional Breakpoints

A conditional breakpoint will only cause an interruption if a particular condition is true at the time of the breakpoint.

When setting a conditional breakpoint (using `BC`) you define the condition to be tested; it can be any general expression involving registers, symbols, literals etc. E.g. you can set a breakpoint that will only break when the condition `{Count}=hl` i.e. the contents of the symbol `Count` in your program is equal to the contents of register `HL`.

Conditional breakpoints are very powerful but do slow down execution; note, though, that this type of conditional breakpoint only involves testing of the condition at the breakpoint itself; there is another tpye of conditional execution available that does not involve breakpoints, continuous conditional execution, more of this under **Execution Commands**.

If the condition tested at the conditional breakpoint is true then the message `Conditional Break` is displayed in the top left corner of your screen, now hit any key; `Y` will keep the breakpoint set, any other key will reset it. The front panel now appears with the state of the computer as it was when it hit the breakpoint.

Watchpoints

Watchpoints are not really breakpoints although they use the same restart instruction. A watchpoint simply keeps a watch over your code, it counts how many times a particular instruction has been executed.

When you set a watchpoint (using the `BW` command) you specify the location at which you need a watchpoint together with a *scale* for this watchpoint, the scale affects the counting rate; a scale of 2 increments the count every second time that you go through the watched instruction, a scale of 10 increases the count only every tenth time etc. The count cannot go above 65535 so it is often useful to apply a scale if an instruction is to be executed many times.

Of course, a watchpoint slows down the execution of your program but it can be an invaluable aid to profiling your code. A watchpoint never causes the front panel to appear and can only be reset by you, manually (using the `BR` or `BZ` command).

The Reserved Word WP holds the count of the most recent watchpoint and thus, if you set just one watchpoint at the place that you want to go through a number of times, and then either **Execute Conditional**, or set a conditional breakpoint, to stop when WP equals the relevant count, then your section of code will be executed just that number of times. Note that you cannot set two breakpoints on top of each other so that a conditional breakpoint cannot be set at the same address as a watchpoint.

Warm Boot

Most CP/M programs terminate by jumping to location 0 either with a `jp 0` or `rst 0` instruction, this reloads CP/M in a clean fashion.

When debugging, you do not usually want this to happen so we have placed a breakpoint at location 0 so that any attempt to go there will result in the message `Warm Boot Break` appearing in the top left of the screen, hit any key for the Front Panel.

If address 0 was reached by a `rst 0` instruction then the address of the instruction after that `rst 0` will be on the stack and you can investigate the code (use *Memory - Address:* `{sp} [RETURN]`). However, if 0 was jumped to, then you will have no idea where the jump occurred although you can try searching for it using:

Search - Byte/string: `c3 00 00 [RETURN].`

If you don't want this breakpoint at address 0 then change location 0 to `c3` which is what it is normally but remember that, if you do this and a program inadvertantly finishes, you will return to CP/M.

Breakpoint Display

To help you keep tabs on the various breakpoint/watchpoints that you may have set in your program, the Breakpoint Display underneath the Memory Display shows all the breakpoints and watchpoints that are set at the moment.

The breakpoint display looks like this:

```
        ProMON 2.0 (C) HiSoft 1987
Break              Condition/Scale Count

1 loop
0 0D12                  {Count}=hl
1 432A                   06  0000
```

This tells us the following about what's set:

There is a hard breakpoint at location `loop` in bank 1, a conditional breakpoint at address `d12h` in bank 0 with condition `{Count}=hl` and a watchpoint at `432ah` in bank 1 with a scale of 6 and a current count of 0.

Breakpoint Commands

Breakpoint Set BS

Enter an expression. A hard breakpoint will be set at the address to which the expression evaluates. The Breakpoint/Watchpoint display will be updated to show that a breakpoint has been set.

When you execute your code, subsequently, and this breakpoint is encountered, the message `Hard Break` will appear at the top of the screen, hit a key; press `Y` to keep the breakpoint or any other key to reset it. Control returns to the Front Panel.

Example:

Breakpoint - Set: `loop [RETURN]`

sets a hard breakpoint at the address of `loop`.

Breakpoint Reset BR

Enter an expression. The breakpoint at the address given by the expression will be reset so that it will not cause a program break.

You can reset any type of breakpoint with this command.

Example:

```
Breakpoint - Reset: loop [RETURN]
```

resets the breakpoint at location `loop`.

If you press `[RETURN]` only then the **Execute Conditional** condition will be reset, see the `EC` command.

Breakpoint Conditional BC

Firstly, you should enter the address of the breakpoint, as usual you can use a generalised expression here. After this you will be prompted to enter the `Breakpoint condition:`, type in an expression that you want evaluated every time this breakpoint is encountered.

On execution, when this expression is true, the message `Conditional Break` will appear, top left, and you should hit a key; `Y` will keep this conditional breakpoint set whereas any other key will reset it. The Front Panel will now appear.

Obviously, conditional breakpoints slow down the execution speed of your program but, since the condition is only evaluated at the marked instruction, this decrease is normally acceptable. If you wish to run *every* instruction under a condition then you should use the **Execute Condition** instruction (`EC`), see below.

```
Breakpoint - Conditional: 402a [RETURN]
Breakpoint condition: hl=de+1 [RETURN]
```

will set a breakpoint at address `402ah` and only break when register `HL` is one more than register `DE` at that address.

Breakpoint Watchpoint BW

Firstly enter the address for the watchpoint as usual. Now you will be prompted to enter the `Watchpoint scale:`, here you enter a number which is used to scale the watchpoint count.

Whenever a watchpoint breakpoint is encountered the watchpoint count will be incremented according to the scale; if the scale is 1 then the count is incremented by one every time that the watchpoint is encountered, if the scale is 4 then the count will be incremented by one only every fourth time that the watchpoint is met etc. Thus, the scale allows you to scale down the count, useful if you are going to execute an instruction more than 65535 times, which is the limit to the value of count.

Watchpoints can only be reset manually with the `BR` or `BZ` commands.

Breakpoint - Watchpoint: `Label20` [RETURN]
Watchpoint scale: 2 [RETURN]

Sets a watchpoint at address `Label20` with a scale of 2.

Breakpoint Zap BZ

Simply clears *all* breakpoints except for the continuous conditional condition. Use with care!

Execute Commands

You can execute your program in a variety of ways, at full speed up to a breakpoint, at reduced speed so that you can interrupt it with a key press, at reduced speed checking a condition on every instruction or by single-stepping each instruction (optionally skipping `call` instructions and the like).

In every case, you initiate the execution using one of the following commands. Execution will start from the address in the Program Counter (`PC`) and with all the registers set up as displayed on the Register Display.

Execute Breakpoint EB

Allows you to set a hard breakpoint before beginning full-speed execution, useful if you simply want to execute up to a particular point. Example:

```
Execute - Breakpoint: Finish [RETURN]
```

will set a hard breakpoint at location `Finish` and then execute from the address held in the `PC`. Execution will only return to the Front Panel when a breakpoint is encountered.

Execute Miss EM

Allows you to single-step a `call` instruction in one move. Normally, if you single-step a `call` instruction you will be transferred into the called subroutine to single-step that. However, if you don't want to do this but want to execute the whole of the subroutine instead then you should use this command.

In fact, the command effectively places a breakpoint after the current instruction and then executes the instruction. So it is not restricted to executing `call`s but can be used at any time. Be careful when using it on jumps that may never return to the instruction after the jump.

Execute Long EL

Begins execution from the address held in the Program Counter (`PC`) but at reduced speed, allowing you to press any key to interrupt the execution and return to the front panel.

The most common use for `EL` is when you suspect that your code is entering an infinite loop; use of this instruction will allow you to break out of the loop by pressing any key.

The execution speed of your program is substantially reduced when running like this which can, at times, be useful especially with fast-running games, using `EL` you can see exactly what happens at any particular time.

All breakpoints are recognised by this instruction and will be obeyed.

Execute Quick EQ

Simply start execution from the address held in the Program Counter (PC) and continue until the program finishes or a breakpoint is encountered.

This command runs your code at full speed and normally should be used only in conjunction with breakpoints.

Execute Single ES

Single-step the current instruction i.e. the one at the address in the Program Counter (PC).

Normally an instruction will be executed exactly as the Z80 chip would execute it so that `call`-type instructions will transfer execution to the called subroutine for continued single-stepping; if you do not want this then use the EM instruction above.

However there are two Reserved Words, HIGH and LOW, which affect single-stepping. If execution is to be transferred (e.g. by a `call` or a `jp` instruction) to an address above HIGH or below LOW, then the code will not be single-stepped, instead the instruction will be executed in one go just as if you had performed an **Execute Miss** (EM) instruction.

The reason for this is that it is unwise to single-step the CP/M BIOS or BDOS or **ProMON** itself because of unpredictable keyboard interaction and other non-re-entrant code within these areas.

The Reserved Words HIGH and LOW may be used in any expression and may be assigned to using the **Register Update** command. Be careful single-stepping if you do change HIGH and LOW, unpredictable and potentially disastrous results may follow.

Single-step is used so often that it is also available by typing Z (or z) from the `Command:` prompt.

Execute Conditional EC

With this command you can execute you code, at reduced speed, until a particular condition becomes true. You are prompted to enter the `Breakpoint condition:`, type in an expression that specifies the condition under which you wish execution halted and then press `[RETURN]`.

Execution will then take place, from the Program Counter, at an interpreted speed (rather as if you had used the **Execute Long** command) and, after the interpretation of each instruction, the condition will be evaluated to see if it is true. If the condition becomes true then the message `Continuous Break` will appear, top left; hit any key to enter the front panel with all registers etc. updated to reflect their state when the condition became true.

Example:

Execute - Condition:
Breakpoint condition: `hl=de+2 [RETURN]`

will begin execution from the `PC` and continue, at a reduced speed, until the value in register `HL` is two more than the value in register `DE`. Execution will also be halted if any other breakpoints are encountered or if any key is pressed.

If, instead of entering a condition, you just hit `[RETURN]` then the current condition (displayed on the Breakpoint Display with no breakpoint address) will be used. To abort this command, enter `[ESC]` as the first character of the condition.

If you wish to erase the condition use the `BR` command and simply press `[RETURN]`.

Disassembly Commands

Disassemble Address DA

Prompts you to enter an address, as usual you can type in any general expression involving symbols, registers and numbers. The List Display will then be updated with the disassembled instructions starting from the address you entered. Example:

```
Disassemble - Address: hl+offset [RETURN]
```

shows a page of disassembly from the address specified by adding the value in register `HL` to the value of the symbol `offset`.

Disassemble File DF

Disassemble a block of memory to screen, printer or disc. The command first prompts you (`First: Last:`) to enter the start and end (inclusive) addresses of the block of memory that you wish to disassemble; these may be entered in hexadecimal or decimal and if the start address exceeds the end address then the command is aborted.

Then the question `Disc?` appears; answer `Y` if you wish to produce a disc file of the disassembly - this disc file may be loaded by our editor (**ED80** or **HDE**) and assembled by our assembler (**GEN80**) as a normal text file.

If you answer this question in the affirmative then you will be prompted for the filename that you wish the file to have on disc - this should be of normal CP/M format i.e. 8 character filename, then a dot and a three character filetype, although the dot and the filetype may be omitted and will then default to `.GEN`.

Now (whatever you answered to the last question) you will be asked whether you want the output to go to your `Printer?`; answer `Y` to direct the listing to the printer or any other key for screen output.

`Workspace:` appears next - the disassembler needs some workspace for its primitive symbol table and its disc buffer (if you have told it to produce a file on disc).

Simply replying [RETURN] to this question will make a workspace of 2K immediately under **ProMON**. If this response produces an error or you know in advance that more than 2K is required then you should specify the workspace to be some other area.

Finally, you are repeatedly asked to specify the First: and Last: addresses of any areas of memory within the disassembly that you wish to be treated as data areas. Data areas are blocks that you do not wish to be treated as Z80 instructions, they might be text messages among other things. Any memory contents within a data area will be disassembled as a sequence of DEFB xxx where xxx is the relevant store contents. xxx will be displayed in ASCII (as a character between quotes) if its value is between 32 and 127 or otherwise in hexadecimal (as two hex digits preceded by a hash). To terminate your list of data areas simply press [RETURN] in answer to both the First: and Last: questions.

Having answered, or defaulted, all the above questions, the screen will be cleared and there will be a pause while the first pass of the disassembly builds up the symbol table of labels.

You may pause the listing at any stage by hitting a key ; then hit [ESC] to go back to the front panel or any other key to continue the listing.

Labels are generated, where relevant (e.g. in #C3 #00 #98), in the form LXXXX where XXXX is the absolute hex address of the label; if this address lies outside the limits of the disassembly then the assembler pseudo-mnemonic EQU is generated to define the label - this is for compatibility with our assembler **GEN80**.

Example block disassembly:

```
DF [RETURN]
First:0 [RETURN]          (Disassemble from #0000)
Last:10 [RETURN]          (to #0010)
Disc: [RETURN]            (Don't make a disc file)
Printer:Y [RETURN]        (Do send to the printer)
Workspace: [RETURN]       (Use the default workspace)
First:3 [RETURN]          (Data area starting at #0003)
Last:4 [RETURN]           (and ending at #0004)
First: [RETURN]
Last: [RETURN]
```

the above might produce the following output on the printer:

```
        JP      LE203
        DEFB    #D5,#00
        JP      LB906
        JP      L0545
        LD      A,#01
        OUT     (#E4),A
        LD      A,B
        OUT     (#E2),A
L0545   EQU     #0545
LB906   EQU     #B906
LE203   EQU     #E203
```

Disassemble Memory DM

This command produces a disassembly on the List Display whose start address is taken from the current value of the Memory Pointer - useful if you are grubbing about memory looking at code.

Disassemble Program DP

Produces a disassembly on the List Display whose start address is taken from the current value of the Program Counter (PC) - useful if you have been grubbing about memory looking at code and then want to return to single-stepping. You can achieve the same effect with the DA PC command i.e. **Disassemble Address** PC.

Disassemble Next DN

Produces a disassembly of the next block of instructions following on from the current page of disassembly displayed on the panel.

Disassemble Window DW

Takes the cursor into the middle of the List Display. You can now use your cursor keys, if you have installed them using the installation program or, if you haven't installed your cursor keys, the Wordstar-like cursor keys to move around in the List Display. This is most useful for looking at the previous or next page of disassembly.

In fact, this is so useful that it is available from `Command:` mode using the Page-Up and Page-Down commands.

`[ESC]` takes you out of the List Display and back into command mode.

The keys available under this command are:

↑	(`[CTRL]-E`)	Cursor Up
↓	(`[CTRL]-X`)	Cursor Down
`[CTRL]-`↑	(`[CTRL]-R`)	Page Up
`[CTRL]-`↓	(`[CTRL]-C`)	Page Down
`[ESC]`		returns to command mode

Miscellaneous Commands

Print Expression PE

Displays the hexadecimal and decimal values of the expression that you type in. Very useful as a quick calculator. Example:

Print - Expression: `hl+de*2 [RETURN]` = *a5ed 42477*

Print Screen PS

Simply clears the whole screen and redraws all the displays. This is useful if output from the program being debugged has corrupted parts of the front panel display.

Quit Q

Asks you `Quit - Yes,No?`; type `Y` (or `y`) to return to CP/M or **HDE** (depending which **ProMON** was called from) or `N/n` to abort the command. `[ESC]` performs the same function.

ProMON Command Summary

MA	Go to memory address
MB	Change memory banks (CP/M Plus only)
MC	Compare memory blocks
MF	Fill memory with byte
MM	Move a block of memory
MV	View the hex/ASCII display
MW	Enter the Memory Display to edit memory
RU	Change the value of a register/Reserved Word
RV	View the registers and memory addressed by them
FR	Read a file into memory
FW	Write a file from memory contents
FC	Enter a CP/M command line
FZ	Clear the symbol table
SB	Search for a byte sequence or a string
SM	Search for a Z80 mnemonic
SN	Search for next occurrence
BS	Set a breakpoint
BR	Reset a breakpoint
BC	Set a conditional breakpoint
BW	Set a watchpoint
BZ	Reset all breakpoints
EB	Execute up to a certain address
EM	Skip this instruction
EL	Execute slowly with keyboard checks
EQ	Execute at full-speed
ES	Single-step
EC	Execute until condition is met
DA	Show disassembly from address
DF	Disassemble a block of memory
DM	Show disassembly from the memory pointer
DP	Show disassembly from the program counter
DN	Show next page of disassembly
DW	Enter disassembly window
PE	Print an expression in hex and decimal
PS	Clear and redraw screen
Q	Quit to CP/M or **HDE**
N	Search for next occurrence
Z	Single-step

SECTION 2
Installing ProMON

The process of installing **ProMON** involves three phases, **ProMON** is first read in from the disc. Then, sections of the program are modified and final **ProMON** is written back out to the disc (as a **.MON** file). Thus, the process involves a permanent change to **ProMON.**

There are two reasons that you might want to install **ProMON**. Primarily, it may be that there are problems with the screen layout and **ProMON** seems not to work at all. This will be due to incorrect terminal codes and in this case you should read the section on **Terminal Installation** Alternatively, you may wish to modify the cursor commands to suit your keyboard. This procedure is covered in the section **Redefining ProMON Commands.** In either case you should first read the next section.

2.1 Starting up the Install Program

To run the installing program, insert the suplied disc and type:-

```
PMONINS [RETURN]
```

You will now see the **PMONINS** copyright message and some general information. When you're ready, press any key. The purpose of the installation process is to alter the copy of **ProMON** on the disc. To this end, some copy of the program (called the working copy) is read in from the disc into the machine. The first question is thus:-

```
Normally the working copy of PMON  is
read in from a file called PMON.MON
Use another file instead (Y/N)
```

The reply will normally be N, the exception being when you have renamed a version of **ProMON**. A reply of Y will produce the prompt:

```
[ESC] to abort
Omit file type (.MON assumed)
Enter filename
```

to which a filename should be typed in (omitting the filetype). Whether you replied N to the opening question or Y and then specified a filename the working copy will now be read in to the machine from the disc and the **ProMON** installation Menu will appear.

There is now a copy of **ProMON** in the memory of your machine ready to be altered. The **ProMON** Installation Menu is on the screen.

```
ProMON INSTALLATION MENU

1.  Return to CP/M
2.  Alter screen codes
3.  Save ProMON as <working copy filename> (normally PMON.MON)
4.  Saved ProMON as another file
5.  Alter command codes
6.  Load installation from .P80 file
7.  Save installation to .P80 file

Type desired number:
```

If you are a first-timer using the installation program because the screen codes in **ProMON** wre wrong then turn first to the section **Terminal Installation** and then to **Leaving the Install Program**. The other sections in this chapter are **Redefining ProMON Commands** and **Use of Installation Files.**

2.2 Terminal Installation

Select options 2 from the main menu to alter the screen codes. You will be asked:

```
Screen at least 90 columns wide (  )
(Y/N/[RETURN])?
```

in anwser to the question you should type either `Y` (if your screen is 90 columns wide or greater) or `N` (if your screen is less than 90 columns wide). Your screen must be at least 80 columns wide for **ProMON** to work effectively. Pressing `[RETURN]` alone is equivalent to giving the answer shown in brackets.

Next, you are asked:

```
Number of screen lines (  ) ?
```

Enter the number of lines on your display and then press `[RETURN]`, pressing `[RETURN]` by itself will be the same as entering the number in brackets. Your screen should have at least 24 lines for **ProMON** to work well.

The rest of the questions concern how the screen controller works on your computer. If you are in doubt about any of the questions, consult the manual for your computer. You are now asked for the:-

```
Cursor position lead-in sequence
(    )    (    ) -
```

When **ProMON** is in operation it has to be able to tell the screen controller to put the cursor at a certain postion on the screen. To do this, **ProMON** tells the controller the row and the column required.

Most screen controllers require a special sequence of codes to indicate that the values to follow represent a row and a column. Thus, inside the firest set of brackets there will be the sequence as it is currently defined with the decimal values of the codes in that sequence in the second set of brackets. If the sequence is correctly set up then just press [RETURN] and move on to the next question. If the sequence is incorrect then it must be changed.

You should enter the cursor positioning sequence code by code (up to a maximum of four codes) terminated by [RETURN]. Each code may either be entered as a single keypress or as its decmal value terminated by [RETURN]. As an example, if the correct sequence for your contoller was [CTRL]-K=. You could enter this either by typing:

```
[CTRL]-K=[RETURN]                          or by typing
1 1 [RETURN] 6 1 [RETURN] [RETURN]
```

([CTRL]-K is ASCII 11 and = is ASCII 61

note the two [RETURN]s at the end; the first is to terminate the 61 and the second is to terminate the whole sequence.)

The next question asked is:

```
Is the row sent before the column (  )
(Y/N/[RETURN])?
```

The screen controller may require the row before the column, or the column before the row. As above, pressing [RETURN] is equivalent to giving the answer in brackets.

You are now asked:

```
Offset for column (    )?      and then
Offset for row (  )?
```

When the values for the row and the column are sent, many screen controllers require an offset to be added to each.

The values required for the offsets are those required to position the cursor at the top left of the screen (i.e. if the correct offsets for your machine were both 32 then sending the Cursor Position lead-in sequence, then 32, then 32 will put the cursor at the top left of your screen).

If the value in brackets is correct thn just press [RETURN] otherwise type in the correct value terminated by [RETURN]. As above, you should consult the manual for your machine if in any doubt.

The next text to appear is:-

```
Clear Screen sequence
(  )    (  )  -
```

The layout is identical with that for the cursor positioning sequence detailed above. Press [RETURN] alone if the sequence for clearing the screen is correct or type the correct code terminated by [RETURN] as above. If your controller does not recognize a sequence to clear the screen (possible but unlikely) then press D.

```
Use lead-in (  )
Use lead-out (  )

(Y/N/[RETURN])?
```

These questions concern the use of lead-in and lead-out sequences. These options allow you to use **ProMON** to send a command to the screen controller or run a small program at the start and end of a session.

For example, this facility might be used to put your machine into 80 column mode on entry to **ProMON** and reset back to 40 column mode on exit. However, unless you have an important reason for wanting to use this facility, it is advisable to answer N to both questions. If you answer Y to either you will be asked to specify a code sequence to send to the screen controller which you should return as described above.

```
Which RST (1-7) (   )?
```

This question concerns which Z80 restart instruction **ProMON** should use as a breakpoint. This will, on entirely standard CP/M systems, be `RST 7`, but many modern systems use this restart for interrupts and in that case another restart should be used. (See the manual for your computer).

You will now be returned to the main menu.

2.3 Redefining ProMON Commands

Pressing 5 from the main menu will allow you to assign your own particular keys to some important **ProMON** commands. The commands will be shown and you have the opportunity to change the definition or accept it and pass on to the next command. After the last command you are returned to the main menu. For each of the commands the display format is:-

```
Command name
(keystroke definition) (decimal definition) -
```

where the keystroke definition is the key you press to give the command and the decimal definition is the decimal ASCII value of that key.

At any stage you have the option to retain the current definition or to change the current definition.

1) To retain the current definition press `[RETURN]`. The process then repeats for the next command. At the end you are returned to the main menu.

2) To change the current definition the new key should be pressed after which the new definition is deplayed. Then the whole process is repeated for the next command.

3) Definition elements are of two types. The first type is simply a keystroke and the second type is a sequence of digits terminated by [RETURN]. For example, the two ways to define cursor-left as [CTRL]-Q (ASCII value 17) are:-

a) Simply press [CTRL]-Q
b) press 1 then 7 then [RETURN]

If the definition given is the same as that of a previous command then this message will appear:-

```
WARNING: There is a conflict between
this and another command.
Do you wish to continue anyway (Y/N)?
```

A response of Y will ignore the duplication and N will allow the current command to be re-defined. Note that if **ProMON** is saved to the disc with two commands identical, the use of one of the commands will be lost.

It is recommended that you consult the reference section of the manual if in any doubt as to the meaning of some of the commands. After the last command, you are returned to the main menu.

2.4 Use of Installation Files

There are many features of **ProMON** that are alterable by the user. Every copy of **ProMON** naturally contains one set of these options. There is a type of file, however, called an Installation File that consists solely of the set of the alterable options. An Installation FIle is of type .P80. Note that installation files for **ProMON** are *not* compatable with those of **MON80**.

To save the current installation information in a file, select option 6 from the main menu. You will then be prompted for a filename which you should type in terminated by [RETURN].

To load an installation file, select option 7 from the main menu. As above, you will be prompted for a filename. If the file you give does not exist then the prompt will be repeated.

You can press [ESC] to quit. When the installation file is loaded into memory, it will overwrite the alterable options already present in the copy of **ProMON** in memory.

The main use of Installation Files is when you are in the long-term process of tailoring your version of **ProMON** to suit your own preferences. If you save each successive change you make to the installation of **ProMON** then any changes you find undesirable can be overwritten by using the last installation file rather than going all the way through the commands. You may also find it useful to save your final installation in a file as a reminder of how your commands etc. are defined.

2.5 Leaving the Install Program

You can leave the install program by selecting option 1 from the main menu, but **BEWARE!** If you select option 1 then nothing will be changed on the disc. Thus, if you are satisfied with the changes you have made in the last installation session, you should first use either option 3 or option 4.

Both will save a copy of **ProMON** (as a .MON file) on the disc. Option 3 will save **ProMON** under the name you specified at the beginning of the session (normally PMON) whereas option 4 allows you to change the name by which you will invoke **ProMON**. You may have more than one copy of **ProMON** on the disc at the same time (under different names, of course).

Thus, the normal method of leaving the install program will be first to select option 3 and then option 1. If you don't wish to save the results of your installing labours then select option 1 alone.

SECTION 3
MON80 Debugger

This section describes the compact version of the debugger which is the same package as was released with **Devpac80** version 1. The purpose of including this version is to provide facilities for debugging very large programs; the compact model is just over 7K in length, compared with a little over 12K for the professional version, **ProMON**. Thus, an extra 5K of the TPA is available with the compact model and this can be crucial in some cases.

The command set of the compact model is very different from that of the professional version although the underlying concepts are very similar. If you intend to use the compact model then please study the rest of this section carefully before proceeding. Of course, if you have upgraded from **Devpac80** version 1, you will be able to use this version immediately.

Throughout this section we shall refer to the compact version simply as **MON80**.

3.1 Getting Started

Your supplied disc will hold the two programs required to run the compact model: `MON80.COM` and `MON80.MON`. To activate simply type:-

```
MON80 [RETURN]
```

MON80 will now be loaded into the memory of your computer and it will then run itself and produce what is called a Front Panel on the screen.

3.2 The Front Panel

What is now on your computer's screen is called a *Front Panel.* The name Front Panel stems from the type of panels that are mounted on mainframe and mini computers to provide information on the state of the machine at a particular moment, usually through the use of flashing lights. These lights represent whether or not particular flip-flops (electronic switches) within the computer are open or closed; the flip-flops that are chosen to be shown on this panel are normally those that make up the internal registers and flags of the computer thus enabling programmers and engineers to observe what the computer is doing when running a program.

So these are hardware front panel displays; what **MON80** provides you with is a software front panel - the code within **MON80** works out the state of your computer and then displays this information on the screen. **MON80** can operate in either 40 or 80 column mode: [CTRL]-E flips between the two modes. Here's a sample screen from **MON80** in 40 column mode:-

```
}0100 LD   HL,(#0006)          PC 0100
 0103 LD   DE,#1B00            SP B906
 0106 OR   A                   IY 0000
 0107 SBC  HL,DE               IX 0000
 0109 LD   SP,HL          0000 HL 0000
 010A PUSH HL             0000 DE 0000
 010B LD   A,(#0004)      0000 BC 0000
 010E LD   E,A            0000 AF 00FF
 010F LD   D,#00               .MR 0100
 0111 LD   C,#0E               IR 007A
 0113 CALL #0005          I SZ H VNC

 00E0 FF 00 FF 00 FF 00 FF 00 ........
 00E8 FF 00 FF 00 FD 00 FF 00 ....}...
 00F0 FF 00 FF 00 FF 80 FF 00 ........
 00F8 FF 00 FF 00 FF 20 FF 00 ..... ..
 0100>2A<06 00 11 00 1B B7 ED *.....7m
 0108 52 F9 E5 3A 04 00 5F 16 Rye:.._.
 0110 00 0E 0E CD 05 00 11 8E ...M....
 0118 01 0E 0F CD 05 00 3C 20 ...M..<
 0120 19 21 DD 01 7E B7 28 0B .!]..7(.
 0128 5F 0E 02 E5 CD 05 00 E1 _..eM..a

>
```

As can be seen, the screen display is composed of three main sections:-

a) The Register Display
b) The List Display
c) The Memory Display

The Register Display

PC 0100	the program counter
SP B906	the stack pointer
IX 0000	the **IX** register
IY 0000	the **IY** register
0000 HL 0000	the **HL'** and **HL** registers
0000 DE 0000	the **DE'** and **DE** registers
0000 BC 0000	the **BC'** and **BC** registers
0000 AF 00FF	the **AF'** and **AF** registers
.MR 0100	the pseudo memory register
IR 007A	the interrupt and refresh registers
I SZ H VNC	the interrupt status and flags

This display shows the values held by the various internal Z80 registers including the flag register. Remember that the HL, DE and BC registers (plus the alternate equivalents) may each be regarded as one 16 bit or two 8 bit registers. A pseudo-register (i.e. register that does not exist within the Z80 chip) has been included; the memory register (MR). This is a useful place for storing addresses that you may wish to go back to later, think of it as being rather like a pocket calculator's memory.

To the left of the register display is the list display:

List Display

```
}0100 LD   HL,(#0006)
 0103 LD   DE,#1B00
 0106 OR   A
0107 SBC   HL,DE
 0109 LD   SP,HL
 010A PUSH HL
 010B LD   A,(#0004)
 010E LD   E,A
 010F LD   D,#00
 0111 LD   C,#0E
 0113 CALL #0005
```

In 40 column mode the list display consists of a disassembly of 11 instructions initially starting from address #100 (in 80 column mode the disassembly is 22 instructions long). This disassembly may be updated at any time to start at the address held by the program counter (PC) by using the L command. If any of the instructions disassembled is at the same address as held by the program counter (PC) then that instruction will be marked on the display with a right curly bracket }.

Below the list display is the memory display:

Memory Display

```
00E0 FF 00 FF 00 FF 00 FF 00 ........
00E8 FF 00 FF 00 FD 00 FF 00 ....}...
00F0 FF 00 FF 00 FF 80 FF 00 ........
00F8 FF 00 FF 00 FF 20 FF 00 ..... ..
0100>2A<06 00 11 00 1B B7 ED *.....7m
0108 52 F9 E5 3A 04 00 5F 16 Rye:.._.
0110 00 0E 0E CD 05 00 11 8E ...M....
0118 01 0E 0F CD 05 00 3C 20 ...M..<
0120 19 21 DD 01 7E B7 28 0B .!]._7(.
0128 5F 0E 02 E5 CD 05 00 E1 _..eM..a
```

The display is a snapshot of an 80 byte area of memory, initially centred on address #100. The addresses are shown down the left- hand side with the contents of the next 8 bytes from the address shown to the right of it (in hexadecimal). Following this, to the right, is the ASCII representation of these 8 bytes with . being displayed if the code cannot be usefully interpreted.

You will notice that one of the bytes on the memory display is enclosed in angle brackets (> <); the address of this byte is known as the Memory Pointer - this is a concept internal to **MON80** and has nothing to do with the Z80. The Memory Display (unless set independently) is tied to the address held in the currently- addressed register (the one with a . to its left) and if the value in this register changes or you change the pointer to point to another register then the Memory Pointer (remember the angle brackets) will change to the new address held by the register. You may set the Memory Pointer independently by using the M command - see below.

The following two sub-sections describe the commands available to you within **MON80**; there are 2 sets, the standard set and the advanced set. The standard set of commands is described in **Section 3.3** while the advanced set is detailed in **Section 3.4**. You should read **Section 3.3** first and may find that you never need to dip into **Section 3.4**, you can drive **MON80** adequately by using the standard command set.

3.3 Standard MON80 Commands

There is a wide range of commands that may be entered and executed whenever the front panel is displayed and the command prompt > is present at the bottom left of the screen. Before proceeding to describe these commands in detail, we shall give details of how numbers are entered when using **MON80**.

3.3.1 Entering Numbers

There are many times when you will find yourself wanting to enter a number when using **MON80** e.g. a memory address. Numbers may be either decimal or hexadecimal (base 16) and you distinguish between the number base by starting a decimal number with a \ symbol.

A decimal number may have a maximum of 5 digits whilst a hexadecimal number has a maximum of 4 digits; if you enter more than the relevant number of digits then the most significant digit will be lost e.g. if you enter `\123456` then the display will show `\23456` and this is the number that will be accepted.

To terminate the entry of a number simply press any character that is not valid within the context of the number i.e. any character other than `0-9` when entering a decimal number or any character other than `0-9` and `A-F` (or `a-f`) when specifying a hex number. If the character pressed is a valid command then that command will then be executed.

You may specify negative numbers by using a minus sign before the number and you should note that all numbers entered will be taken as modulo 65536 or modulo 256 depending whether the context allows for words or bytes respectively.

You may abort the entry of a number by using `[ESC]`.

Now for the commands available:

3.3.2 The Commands Available

The key used to abort all operations is [ESC] and the key used to delete the last character entered is [DEL]. The keys used to invoke the commands can be changed by running the installation program and we have left a blank space before each command for you to fill in your own choice of command key.

Set the Memory Pointer M

This prompts you with a colon to enter a memory address. Once you have entered the address the Memory Display on the screen is updated so that it is centred around the specified address. You may abort the command by pressing [ESC] at any time.

Increase the Memory Pointer by 1 :

Add 1 to the Memory Pointer so that the Memory Display advances.

Decrease the Memory Pointer by 1 ;

Subtract 1 from the Memory Pointer so that the centre of the Memory Display is decreased by 1.

Increase the Memory Pointer by 8 >

Add 8 to the Memory Pointer so that the centre of the Memory Display is increased by one line.

Decrease the Memory Pointer by 8 <

Subtract 8 from the Memory Pointer so that the centre of the Memory Display is decreased by one line.

Memory Pointer address to Register T

Copies the current Memory Pointer address into the register that is pointed to by the register pointer. This is a second way of changing the contents of a register, the other being to enter the required value followed by a period (see next command).

Advance Register Pointer full stop

Advance the pointer (.) on the register display by one from top to bottom (i.e. PC to MR). When the pointer reaches the bottom of the display (MR) then on the next execution of this command it will step back to the top of the display (PC).

Note that, if this command is used to terminate a number, then the number will be entered into the register currently addressed by the Register Pointer. Remember that the Memory Display is tied to the address held in the currently-selected register and therefore the Memory Display will change as you use this command.

Read in a file R

This command will produce the prompt `Name:` to which you should give the filename you wish to read in followed by `[RETURN]`. The filetype will default to `.COM`. In response to `First:` give the address you wish to load the file to. Pressing just `[RETURN]` here will load the file to the standard CP/M base file address of `#100`. **MON80** will inform you of the address of the end of the last block of the file loaded and then pressing a key will return to the front panel.

Write out a file W

This command will produce the prompt `Name:` to which you should give the name you wish to call the file. The default filetype is `.COM`. In response to `First:` and `Last:` you should give the start and end of the block of memory you wish to write to the disc, each followed by `[RETURN]`.

Single Step a Program Z

Executes instructions one at a time from the current state of the computer as shown by the Register Display. After each instruction is obeyed, the Register Display and Memory Display are updated to reflect the new state of the machine.

Jump to PC J

Continue execution of a program from the register state currently displayed on the register display (top right). Execution will start from the address held in the PC and continue indefinitely or until a breakpoint is encountered; to set a breakpoint you may specify the address at which you want the breakpoint to be placed after the colon with which J prompts you - if you do not wish to set a breakpoint then simply hit [RETURN] after J e.g.

J:B800 [RETURN]	will execute until address #B800
J: [RETURN]	will execute indefinitely

To abort the command hit [ESC] before hitting [RETURN].

If you wish to set more than one breakpoint then use the Set Breakpoint routine [CTRL]-B detailed in the next section.

List from PC L

Update the List Display with a disassembly starting from the address currently held in the program counter (PC). The list display marker (}) will point to the first instruction on the display and will move as the PC address changes.

Show Alternate Display S

This displays an alternative Memory Display of the form:

```
 SP   IY   IX   HL   DE   BC   AF
DB85 0000 0000 0000 0000 0000 00FF

3297 F3 s F3 s F3 s F3 s F3 s OO .
F427 C3 C C3 C C3 C C3 C C3 C 2A *
4C73 D7 W D7 W D7 W D7 W D7 W 06 .
0006 BF ? B7 ? B7 ? B7 ? B7 ? 00 .
4891 02 . 02 . 02 . 02 . 02 . 11 .
08AA 1B . 1B . 1B . 1B . 1B . 00 .
8006 98 . 98 . 98 . 98 . 98 . 1B .
4601 98 . 98 . 98 . 98 . 98 . B7 7
```

This display shows the contents of the registers (at the top) and, underneath these, either the contents of the 8 words (for the SP) or the 8 bytes (for IY to AF) that are addressed by the register.

For registers IY to AF the ASCII equivalent of the byte addressed is also shown. This display is useful if you know that one of the registers is addressing a table or if you want to inspect the state of the stack.

This command is a toggle and thus repeated uses will flip between the normal Memory Display and the above alternative display.

Go to Relative Offset O

Go to the destination of a relative displacement. This command takes the byte currently addressed by the Memory Pointer, treats it as a relative displacement and updates the Memory Display accordingly. Example:

Assume the Memory Pointer is set to #8800 and that the contents of locations #87FF and #8800 are #20 and #16 respectively - these 2 bytes 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 #8817, the required destination of the branch.

Remember that relative displacements of greater than #7F (127) are treated as negative by the Z80 processor; The command takes this into account.

To return to the branch instruction address, see the following command.

Reverse Relative Offset U

Remembering that O updates the Memory Display according to a relative displacement i.e. it shows the effect of a JR or DJNZ instruction, then this command is used to update the Memory Display back to the address at which the last O command was issued.

Go to Extended Address X

This command takes the 16 bit address specified at the current position of the Memory Pointer and then updates the Memory Pointer so that it is centred round this address. Remember that the low order half of the address is specified first, followed by the high order half. Example:

Assume you wish to look at the routine that the code #CD #05 #83 calls; set the Memory pointer using M so that it addresses the #05 within the CALL (#CD) instruction and then press X. The Memory Display will be updated so that it is centred around location #8305.

To return to the address at which the X was used see the following command.

Reverse Extended Address V

This updates the Memory Display to where it was before the last X command was issued.

Get a Pattern G

Search memory from the Memory Pointer for a sequence of bytes or characters or a mnemonic. This command prompts you with a colon to enter the pattern for which you want to search. This pattern may be:

a) A sequence of bytes, specified in decimal (start with a \) or hexadecimal and separated with spaces.

b) A string of characters, in this case you should start the string with a double-quote character (") and then simply type the string, character by character.

c) A mnemonic (e.g. `LDIR`) or operand from a disassembly (e.g. `A,C`), you should start by typing a `%` sign followed by the mnemonic or operand that you want. Letters will be converted to upper case as you type them and you must obey the syntax of the disassembler e.g. `HL,#A000` and not `HL,A000` or `HL,A000H`.

While **MON80** looks for the requested disassembled instruction it displays the message `Wait...` on the screen; you can abort the search by pressing `[ESC]`.

Examples:

```
G:21 00 00 7E FE 55 [RETURN]
G:"SYNTAX ERR [RETURN]
G:%LD A,(HL) [RETURN]
```

Next Pattern N

Searches from the Memory Pointer for the next occurrence of the pattern that you previously specified using the `G` command. When the pattern is found the Memory Display is updated to centre around the start of the pattern.

Fill (Put) Memory P

Fill memory between specified addresses with a specified byte. This command prompts for the start and end addresses (`First: Last:`), inclusive, of the block that you wish to fill and then for the byte with which you wish to fill the block (`With:`) e.g.

```
>P
First:A000 [RETURN]
Last:A5FF [RETURN]
With:55 [RETURN]
```

will fill memory between `#A000` and `#A5FF` (inclusive) with the byte `#55` (The character `U`).

If the start address is greater than the end address or if you press `[ESC]` then the command will be aborted and no action taken.

Intelligent Copy I

Copy a block of memory. You are prompted for the start address and end address (`First: Last:`) of the block that you wish to copy; these addresses are inclusive and if the last address is smaller than the first address then the command is aborted. You are then prompted for the destination address (`To:`) after which the block will be moved. The copy is intelligent in that the destination address may be anywhere in memory, even within the block that you are moving. You can abort the command at any stage by pressing `[ESC]`.

Toggle Extended mode [CTRL]-E

This command changes the screen layout so that the memory display appears top right and the list display is extended to 22 instructions forming a very useful long dissassembly with no loss of other information. This command is a switch or toggle and thus using it repeatedly switches between 80 and 40 column mode. Note that in either mode, this command may be used to clear the screen.

Return to CP/M [CTRL]-C

Simply returns you to CP/M.

This concludes the section on the standard commands available under **MON80**; in the next section you will find details of other commands which enhance the power and flexibility of the package but which are not essential for simple debugging purposes. For most purposes you should find that the standard commands will serve perfectly adequately.

Here is a summary of the standard commands and their default values:

Modifying Memory

M to set the Memory Pointer
: to step the Memory Pointer forward 1
; to step the Memory Pointer back 1
> to step the Memory Pointer forward 8
< to step the Memory Pointer back 8.

Any hexadecimal or decimal number (preceded by \) terminated with an invalid digit (often [RETURN] or the cursor keys) to modify the RAM memory at the current Memory Pointer.

A string of ASCII characters beginning with " and terminated with [RETURN].

Modifying Registers

. (full stop) to move the Register Pointer round the registers. Any hexadecimal or decimal number (preceded by \) terminated by a full stop will be entered into the currently addressed register.
T to enter the Memory Pointer address into the currently addressed register.

Program debugging

R to read in a file from the disc.
W to write out a file to the disc.
Z to single-step with all register values as displayed.
J to continue from current machine state up to optionally specified breakpoint.
L to produce a page of disassembly from the PC.
S to obtain an alternative Memory Display of the register values and what they address.
O
U to inspect relative and absolute jump and call
> instructions and then return to the source instruction.
<

Memory Search

G to define a pattern (bytes, ASCII (") or mnemonic (%)) and search for the first occurrence.
N to search for subsequent occurrences after G.

Memory Copy

P to fill an area of memory with a specified byte.
I to intelligent copy a block of memory.

3.4 Advanced MON80 Commands

In this sub-section we detail 9 further commands which extend the power and flexibility of **MON80**; you can debug your programs adequately using the standard set of commands given in **Section 3.3** but the following so-called advanced commands allow you more control over the debugging process - you may feel that you do not need this extra power and in that case you need never read this section again. However, we recommend that you at least browse through this section once.

3.4.1 Disassembly Commands

Disassemble from Memory Pointer [CTRL]-L

This command produces a disassembly on the List Display whose start address is taken from the current value of the Memory Pointer - useful if you are grubbing about memory looking at code. Use command L to restore the List display to disassembly from PC.

Disassemble Next Page [CTRL]-N

Produces a disassembly of the next block of instructions following on from the current page of disassembly displayed on the panel. Use the commands [CTRL]-L and L to recover the original page of disassembly at any time.

Block Disassembly [CTRL]-D

Disassemble a block of memory to screen, printer or disc. The command first prompts you (First: Last:) to enter the start and end (inclusive) addresses of the block of memory that you wish to disassemble; these may be entered in hexadecimal or decimal and if the start address exceeds the end address then the command is aborted.

Then the question Disc? appears; answer Y if you wish to produce a disc file of the disassembly - this disc file may be loaded by our editors (**ED80** and **HDE**) or assembled by our assembler (**GEN80**) as a normal text file.

If you answer this question in the affirmative then you will be prompted for the filename that you wish the file to have on disc - this should be of normal CP/M format i.e. 8 character filename, then a dot and a three character filetype, although the dot and the filetype may be omitted and will then default to .GEN.

Now (whatever you answered to the last question) you will be asked whether you want the output to go to your Printer?; answer Y to direct the listing to the printer or any other key for screen output.

`Workspace:` appears next - the disassembler needs some workspace for its primitive symbol table and its disc buffer (if you have told it to produce a file on disc).

Simply replying `[RETURN]` to this question will make a workspace of 2K immediately under **MON80**. If this response produces an error or you know in advance that more than 2K is required then you should specify the workspace to be some other area.

Finally, you are repeatedly asked to specify the `First:` and `Last:` addresses of any areas of memory within the disassembly that you wish to be treated as data areas. Data areas are blocks that you do not wish to be treated as Z80 instructions, they might be text messages among other things. Any memory contents within a data area will be disassembled as a sequence of `DEFB xxx` where `xxx` is the relevant store contents. `xxx` will be displayed in ASCII (as a character between quotes) if its value is between 32 and 127 or otherwise in hexadecimal (as two hex digits preceded by a hash). To terminate your list of data areas simply press `[RETURN]` in answer to both the `First:` and `Last:` questions.

Having answered, or defaulted, all the above questions, the screen will be cleared and there will be a pause while the first pass of the disassembly builds up the symbol table of labels.

You may pause the listing at any stage by hitting `[ESC]` ; then hit `[ESC]` again to go back to the front panel or any other key to continue the listing.

Labels are generated, where relevant (e.g. in `#C3 #00 #98`), in the form `LXXXX` where `XXXX` is the absolute hex address of the label; if this address lies outside the limits of the disassembly then the assembler pseudo-mnemonic `EQU` is generated to define the label - this is for compatibility with our assembler **GEN80**.

Example block disassembly:

```
>[CTRL]-D
First:0 [RETURN]        (Disassemble from #0000)
Last:10 [RETURN]        (to #0010)
Disc: [RETURN]          (Don't make a disc file)
Printer:Y [RETURN]      (Do send to the printer)
Workspace: [RETURN]     (Use the default workspace)
First:3 [RETURN]        (Data area starting at #0003)
Last:4 [RETURN]         (and ending at #0004)
First: [RETURN]
Last: [RETURN]
```

might produce the following output on the printer:

```
          JP      LE203
          DEFB    #D5,#00
          JP      LB906
          JP      L0545
          LD      A,#01
          OUT     (#E4),A
          LD      A,B
          OUT     (#E2),A
L0545     EQU     #0545
LB906     EQU     #B906
LE203     EQU     #E203
```

3.4.2 Breakpoint Commands

Set Breakpoint [CTRL]-B

Set a breakpoint at the current address held in the Memory Pointer. The byte at this address is saved and replaced with a RST instruction which, when executed, restores the byte to its orginal value, waits for you to hit a key and then enters the front panel displaying the current state of the machine. The RST normally used is RST #38 (equivalent to RST 7 or #FF). This can be changed with the installation program.

You will only use this command when you wish to set multiple breakpoints within a program; the standard command J, which allows you to set one breakpoint, will be sufficient.

The number of breakpoints that may be set at any one time is very large and limited only by memory considerations.

Reset All Breakpoints [CTRL]-R

This resets all the breakpoints previously specified and restores your code to its virgin state.

3.4.3 Execution Commands

Three commands exist to add flexibility while single-stepping your program; one of these uses interpretation of the Z80 code to perform their task. During interpretation, you may interrupt the execution by pressing [ESC] - this will immediately break back to the front panel display, showing the state of the machine when you stopped it.

This powerful feature has the disadvantage that your code will run slowly while being interpreted and you must therefore compromise between the advantages of being able to break out of your program and the disadvantage of a slow, unrealistic execution - the standard command J and the advanced commands [CTRL]-S and [CTRL]-Z do not interpret and thus your code will run at normal speed.

Interpret to Breakpoint [CTRL]-J

Interprets your program from the current PC value up to any breakpoint that may have previously been set. You may also set a breakpoint with this command by specifying the breakpoint address following the colon prompt that [CTRL]-J produces. If you do not wish to set a breakpoint here then simply press [RETURN] in response to the colon.

This command interprets your program and thus runs fairly slowly; you may, however, hit [ESC] at any time to enter the front panel.

Execute Loop Repeatedly [CTRL]-Z

This command prompts you with a hash to enter a number (in hexadecimal or decimal). Once you have entered this number, terminated by [RETURN], execution of the program starting from the address held in the PC takes place. Execution will halt only when the address at which execution was initiated has been encountered the number of times specified by the number that you entered after the hash. For example, consider the following code:

```
844F CD4690   CALL     L9046
8452 2B       DEC      HL
8453 10FA     DJNZ     L844F
8455 21FFFF   LD       HL,#FFFF
```

Say that the PC contains 8452 and that register B contains #20 and you wish to see what happens when you go round the DJNZ loop until register B equals #10 i.e. another 16 (decimal) times. Use the [CTRL]-Z command and type 10 [RETURN] or \16 [RETURN] in response to the hash. The front panel will reappear after the loop has been repeated 16 times.

Skipover Instruction [CTRL]-S

Sets a breakpoint at the end of the Z80 instruction currently addressed by the Program Counter (PC) and executes the instruction, non-interpretatively. This is useful for skipping over CALL instructions if you do not wish to step through the whole of the subroutine.

This command is equivalent to J:X+N where X is the address of the current instruction and N is the length of the instruction - and as such is provided merely for convenience.

SECTION 4 INSTALLING MON80

The process of installing **MON80** involves three phases. **MON80** is first read in from the disc. Then, sections of the program are modified and finally **MON80** is written back out to the disc (as a **.MON** file). Thus the process involves a permanent change to **MON80**.

There are two reasons that you might want to install **MON80**. Primarily, it may be that there are problems with the screen layout and **MON80** seems not to work at all. This will be due to incorrect terminal codes and in this case you should read the section on **TERMINAL INSTALLATION**. Alternatively, you may wish to modify some of the commands or options to suit either keyboard or taste. This procedure is covered in the section **RE-DEFINING MON80 COMMANDS**. In either case you should first read the next section.

4.1 STARTING UP THE INSTALL PROGRAM

To run the installing program, insert the supplied disc and type:–

MON80INS [ENTER]

You will now see the **MON80INS** copyright message and some general information. When you're ready, press any key. The purpose of the installation process is to alter the copy of **MON80** on the disc. To this end, some copy of the program (called the working copy) is read in from the disc into the machine. The first question is thus:–

```
Normally the working copy of MON80 is
read in from a file called MON80.MON
Use another file instead (Y/N)  ?
```

The reply will normally be **N**, the exception being when you have renamed a version of **MON80**. A reply of **Y** will produce the prompt:–

```
[ESC] to abort
Omit file type (.MON assumed)
Enter filename
```

to which a filename should be typed in (omitting the filetype). Whether you replied **N** to the opening question or **Y** and then specified a filename, the working copy will now be read in to the machine from the disc and the **MON80** Installation Menu will appear.

There is now a copy of **MON80** in the memory of your machine ready to be altered and the **MON80** Installation Menu on the screen.

```
MON80 INSTALLATION MENU
-----------------------
1. Return to CP/M
2. Alter screen codes
3. Save MON80 as <working copy filename> (normally
   MON80.MON)
4. Save MON80 as another file
5. Alter command codes
6. Load installation from .M80 file
7. Save installation to .M80 file

Type desired number:
```

If you are a first-timer using the installation program because the screen codes in **MON80** were wrong then turn first to the section **TERMINAL INSTALLATION** and then to **LEAVING THE INSTALL PROGRAM**. The other sections in this chapter are **USER PATCHES**. **REDEFINING MON80 COMMANDS**, and **USE OF INSTALLATION FILES**.

4.2 TERMINAL INSTALLATION

Select option 2 from the main menu to alter the screen codes. You will be asked

```
Screen at least 80 columns wide (      )
                          ( Y/N/[ENTER] ) ?
```

in answer to the question you should type either **Y** (if your screen is 80 columns wide or greater) or **N** (if your screen is less than 80 columns wide). Pressing [ENTER] alone is equivalent to giving the answer shown in brackets. If you answered **N** to the last question then the question below does not appear, otherwise the next question is:–

```
Enter MON80 in extended mode (      )
                          ( Y/N/[ENTER] ) ?
```

If you wish **MON80** to start in extended mode (ie.. with long disassembly utilising the full width of an 80 column screen) then you should press **Y**, otherwise press **N**. As before, pressing [ENTER] is equivalent to giving the answer in brackets.

The rest of the questions concern how the screen controller works on your computer. If you are in doubt about any of the questions, consult the manual for your computer. You are now asked for the:–

Cursor position lead-in sequence
( ) ( )-

When **MON80** is in operation it has to be able to tell the screen controller to put the cursor at a certain position on the screen. To do this, **MON80** tells the controller the row and the column required. Most screen controllers require a special sequence of codes to indicate that the values to follow represent a row and a column. Thus inside the first set of brackets there will be the sequence as it is currently defined with the decimal values of the codes in that sequence in the second set of brackets. If the sequence is correctly set up then just press [ENTER] and move on to the next question. If the sequence is incorrect then it must be changed and there are two ways to change it.

1) If your screen controller does not have a special sequence of codes then you will have to write a program in assembly language to do it instead. In this extremely unlikely event, press D to delete the current sequence.

2) If, as is much more likely, your screen controller does have a Cursor Position lead-in sequence then you should enter it now code by code (up to a maximum of four codes) terminated by [ENTER]. Each code may either be entered as a single keypress or as its decimal value terminated by [ENTER]. As an example, if the correct sequence for your controller was [CTRL]-K =. You could enter this either by typing

[CTRL]-K = [ENTER] (ie.. 4 keypresses) or by typing
1 1 [ENTER] 6 1 [ENTER] [ENTER]
([CTRL]-K is ASCII 11 and = is ASCII 61 and note the two [ENTER]s at the end. The first is to terminate the 61 and the second is to terminate the whole sequence.)

The next question asked is

Is the row sent before the column ( )
(Y/N/[ENTER]) ?

The screen controller may require the row before the column, or the column before the row. As above, pressing [ENTER] is equivalent to giving the answer in brackets.

You are now asked

Offset for column ( ) ? and then
Offset for row ( ) ?

When the values for the row and the column are sent, many screen controllers require an offset to be added to each. The values required for the

offsets are those required to position the cursor at the top left of the screen (ie.. if the correct offsets for your machine were both 32 then sending the Cursor Position lead-in sequence, then 32, then 32 will put the cursor at the top left of your screen). If the value in brackets is correct then just press [ENTER] otherwise type in the correct value terminated by [ENTER]. As above, you should consult the manual for your machine if in any doubt.

The next text to appear is:-

Clear Screen sequence
( ) ( )-

The layout is identical with that for the cursor positioning sequence detailed above. Press [ENTER] alone if the sequence for clearing the screen is correct or type the correct code terminated by [ENTER] as above. If your controller does not recognize a sequence to clear the screen (possible but unlikely) then press D.

Bell character
( ) ( )-

If your computer has a bell or beep then you should press the key that normally rings it. This is almost always [CTRL]-G. If your computer doesn't have a bell then press D to delete the code. As before, pressing [ENTER] is equivalent to giving the answer in brackets.

Which RST (1-7) ( ) ?

This question concerns which Z80 restart instruction **MON80** should use as a breakpoint. This will, on entirely standard CP/M systems, be RST 7, but many modern systems use this restart for interrupts and in that case another restart should be used. (See the manual for your computer).

Use lead-in ( )
Use lead-out ( )
(Y/N/[ENTER]) ?

The final questions concern the use of lead-in and lead-out sequences. These options allow you to use **MON80** to send a command to the screen controller or run a small program at the start and end of a session. For example, this facility might be used to put your machine into 80 column mode on entry to **MON80** and reset back to 40 column mode on exit. However, unless you have an important reason for wanting to use this facility, it is advisable to answer **N** to both questions. If you answer **Y** to either you will be asked to specify a code sequence to send to the screen controller which you should enter as described above. If however, you wish to do something more complicated than just send a sequence

then you should press D to delete the current sequence and prepare to write a program for the patch file!

Normally you will now be returned to the main menu. If, however, you do not see the main menu now then read on!

4.3 USER PATCHES

A user-patch is a program written by you in assembly language to perform a function not within the capabilities of your screen controller. They will be needed if your response to certain questions from option 2 of the main menu has been to press D ie.. the screen controller cannot perform certain functions. The functions which may need user patches are:- Cursor Position, Clear Screen, Lead-in, and Lead-out. If you have answered D to any of these then after the last question of option 2, you will see the message

Please read the manual (Section 4) ! (which you are. Good)
Read in a new Patch file
(Y/N) ?

The normal process by which you can write a user-patch is to use an assembler (GEN80 I hope) to write and assemble the program and create a .COM file, which is the form needed for the patch. Included in this manual is an assembly language source file that is extensively commented to illustrate the general format for a patch-file. If you need to write a patch-file then reply N to the question for the moment and study the example patch-file closely. If you have already written and assembled the patch-file then reply Y to the question above and then in response to the prompt type in the filename. The machine-code in the patch-file will then be incorporated from the disc into the memory copy of MON80 and you will then be returned to the main menu.

4.4 RE-DEFINING MON80 COMMANDS

Pressing 5 from the main menu will allow you to alter the command definitions. All of the commands will be shown and you have the opportunity to change the definition or accept it and pass on to the next command. After the last command you are returned to the main menu. For each of the commands the display format is:-

Command name
(keystroke definition) (decimal definition) -

where the keystroke definition is the key the user presses to give the command and the decimal definition is the decimal ASCII value of that key.

At any stage you have the option to go back to consider the previous command, to retain the current definition or to change the current definition.

1) To backtrack to the previous command, press B

2) To retain the current definition press [ENTER]. The process then repeats for the next command. At the end you are returned to the main menu.

3) To change the current definition the new key should be pressed after which the new definition is displayed. Then the whole process is repeated for the next command.

4) Definition elements are of two types. The first type is simply a key-stroke and the second type is a sequence of digits terminated by [ENTER]. For example, the two ways to define a command as I (which has an ASCII value of 73) are:–

a) Simply press I
b) press 7 then 3 then [ENTER]

If the definition given is the same as that of a previous command then this message will appear:–

WARNING: There is a conflict between this and another command.
Do you wish to continue anyway (Y/N) ?

A response of Y will ignore the duplication and N will allow the current command to be re-defined. Note that if **MON80** is saved to the disc with two commands identical, the use of one of the commands will be lost.

It is recommended that you consult the reference section of the manual if in any doubt as to the meaning of some of the commands. After the last command, you are returned to the main menu.

4.5 USE OF INSTALLATION FILES

There are many features of **MON80** that are alterable by the user. Every copy of **MON80** naturally contains one set of these options. There is a type of file, however, called an Installation File that consists solely of the set of the alterable options. An Installation File is of type **.M80**.

To save the current installation information in a file, select option 6 from the main menu. You will then be prompted for a filename which you should type in terminated by [ENTER].

To load an installation file, select option 7 from the main menu. As above, you will be prompted for a filename. If the file you give does not exist then the prompt will be repeated. You can press [ESC] to quit. When the installation file is loaded into memory, it will overwrite the alterable options already present in the copy of **MON80** in memory.

The main use of Installation Files is when you are in the long-term process of tailoring your version of **MON80** to suit your own preferences. If you save each successive change you make to the installation of **MON80** then any changes you find undesirable can be overwritten by using the last installation file rather than going all the way through the commands. You may also find it useful to save your final installation in a file as a reminder of how your commands are defined.

4.6 LEAVING THE INSTALL PROGRAM

You can leave the install program by selecting option 1 from the main menu, but **BEWARE!** If you select option 1 then nothing will be changed on the disc. Thus if you are satisfied with the changes you have made in the last installation session, you should first use either option 3 or option 4. Both will save a copy of **MON80** (as a .MON file) on the disc. Option 3 will save **MON80** under the name you specified at the beginning of the session (normally **MON80**) whereas option 4 allows you to change the name by which you will invoke **MON80**. You may have more than one copy of **MON80** on the disc at the same time (under different names, of course).

Thus the normal method of leaving the install program will be first to select option 3 and then option 1. If you don't wish to save the results of your installing labours then select option 1 alone.

APPENDIX EXAMPLE PATCH-FILE

```
;This is an example patch-file for use with ED80 or MON80.
;The patch-file for MON80 must be position independent, whereas that
;for ED80 need not be.  This example is written to be position independent
;so that it can be used for both programs.

;*****************************************************************************
;If this is a patch-file for ED80 (which does not have to be position-
;independent) and the file includes some position-dependent code eg., a
;CALL to within itself or a LD  instruction addressing an area within
;itself then there should be an ORG statement here at the head of the file.
;The value for the ORG is that given by the installing program in the line
;          User Patch Area starts at 8XXXX
;so assuming the 8XXXX was 82438 the statement should be
;          ORG  82438
;Note that the value given is adjusted for the initial length byte
;*****************************************************************************


LENGTH    DEFB FINISH-START
;The first byte of the file must be the total length
;of the patch-file excluding itself (255 maximum).


START
          JR   CLEAR_SCREEN
;Jump relative to the routine to clear the screen

          JR   CURSOR_POSITION
;Jump relative to the routine to position the cursor

          DEFS 2
;The vector to your lead-in routine would go here, but in this example
;none is required.  The two bytes should be filled however.

          DEFS 2
;A lead-out routine is not used in this example.  See above.

;The file should start with four two-byte areas that are either vectors
;to the routines concerned or uninitialised space (DEFS 2).  The way ED80
;and MON80 uses the vectors is simply to CALL the requisite location
;ie., where the vector to the routine is positioned.  If the user has
;specified in the installing program a certain patch is not required
;then the vector will never be called.  Common sense may be used; as an
;example, if the only patch required was to clear the screen, then the patch
;file need only contain the length byte followed by the routine itself as
;the vectors for CURSOR_POSITION, LEAD_IN and LEAD_OUT are never called.
```

```
CLEAR_SCREEN
;This routine must clear the whole screen, but need not adjust the
;cursor position, or do anything else normally associated with a
;screen-clearing routine.  It takes no parameters and may corrupt all
;registers except IX.  Use of IY and the alternate register set is not
;advisable as the BIOS of some machines corrupt these.

        CALL 0F06C      ;This is an example
        LD   HL,0FFC8   ;of a possible screen
        SET  3,(HL)     ;clearing routine.  Note that
        RET             ;it should finish with a simple RET
CURSOR_POSITION
;This routine must set the cursor position on the screen.
;The routine takes two parameters, the column and the row thus:-

;B holds the desired row     + offset for row
;C holds the desired column  + offset for column

;The routine may corrupt all registers except IX.  Use of IY and the
;alternate set is not advisable for the reason given above.

        LD   A,B
        LD   (0FFD0),A  ;An example routine
        LD   A,C        ;to position the cursor
        LD   (0FFD2),A  ;Note that the
        PUSH IX         ;routine saves the IX
        CALL 0F07A      ;register which is
        POP  IX         ;destroyed by the CALL
        RET             ;and then does a simple RET


;Lead-in and lead-out are not required in this example.  Code need
;not, of course, be included.  The only requirement is that there be
;two bytes where the vector ought to be.   See above

;NB the code for the lead-in and lead-out may corrupt all registers
;and should end with a simple RET

FINISH  EQU  $
```

HiSoft Devpac80

Full Tutorial

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™.

All Rights Reserved Worldwide. 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 **HiSoft Devpac80**.

It is an infringement of the copyright pertaining to **HiSoft Devpac80** and its associated documentation to copy, by any means whatsoever, any part of **HiSoft Devpac80** for any reason other than for the purposes of making a security back-up copy of the object code.

Devpac80 Tutorial

Introduction

This chapter is a tutorial in the use of **Devpac80**. It assumes that you have a working relationship with the editor (you should have worked through the editor tutorial) and a passing knowledge of Z80 assembly language, but requires very little knowledge of CP/M, and introduces you both to use of **Devpac80** and programming under CP/M in general. Although the tutorial looks long, it is broken up into logical sections so that it doesn't have to be taken in one dose. At the end of the tutorial you will have a useful utility program that will clean up your discs, ridding them of files with illegal filenames.

Printing on the screen

First things first. This section will build up a flexible routine that will print messages on the screen. The best place to start is to write a routine that will print a single character. This routine can then be called repeatedly to print a message.

Activate **HDE** by using your **Devpac80** work disc and typing:

```
HDE PRINT1 [RETURN]          and then type
S
```

now type in the routine at the top of the next page exactly as written (the space at the start of the lines is a `[TAB]` character). It will often be useful to go immediately into **AUTO INDENT** mode at the start of a session (see the editor manual). Also, you should normally leave a blank line at the front of your program, do this by pressing `[RETURN]` alone.

```
;Write the character in A to the screen
; with all registers preserved
PUTCHAR
          PUSH HL
          PUSH DE
          PUSH BC
          PUSH AF
          LD   E,A
          LD   C,2
          CALL 5         ;Use the BDOS
          POP  AF
          POP  BC
          POP  DE
          POP  HL
          RET
```

As a word of explanation, there is a set way of getting CP/M to perform functions for you. You first load the C register with the number of the function you want (see any CP/M technical manual) and then call CP/M. CP/M is set up so that there is a jump to it at location #0005, so that you only ever need CALL 5 to use the BDOS functions in CP/M.

The call number used above is 2, which is the standard function for writing a character to the screen.

The last complication in the routine above is the LD E,A instruction. This is because function 2 requires that the character to be written is present in the E register and we shall always be calling PUTCHAR with the character to print in the accumulator (i.e. A).

Having typed in the primitive routine above, we need some way of testing it. Move the cursor to the top of the screen and type the following three lines:-

```
LD   A,"Z"
CALL PUTCHAR
RST  0
```

This routine loads A with a character to print and calls the routine we wrote to output the character. The final instruction is the standard way of returning to CP/M safely. A RST (short for restart) instruction performs exactly the same as the equivalent CALL (e.g. CALL 0), but only occupies one byte.

Now save the program and assemble it. Do this by typing

`[CTRL]-K X [RETURN]` and then, from the menu
A

Of course, if you have installed the editor commands to suit your tastes, the save command may not be `[CTRL]-K X`; so from now on we will give the name of the editor command rather than specifying how to get it.

The assembly should work correctly and create a file `PRINT1.COM`. If there is some mistake then the assembler will report the error and finish by saying `Error(s) found, hit a key for the editor:`, hit a key and then use the **Goto Next Error** command to find the error, correct it and then re-save and re-assemble `PRINT1`. When all is well, and you are back at the **HDE** menu type:

```
R
```

to execute your program. Lo and behold the letter Z appears on the screen and an orderly return to the menu is made.

Now type `S` to resume editing and delete the three line test program. We'll now develop an all-purpose message-printing routine. Type in the following text:-

```
;Print the message pointed to by HL
; and terminated by an ASCII 0
MOUT
          LD   A,(HL)
          INC  HL
          OR   A
          RET  NZ
          CALL PUTCHAR
          JR   MOUT
```

`MOUT` stands for *message out.*

As you can see the routine should be entered with `HL` pointing to the desired message and puts the message out character by character until it encounters a zero and then returns.

To test this routine, type in the following short test routine. Note of course that the main routine (in this case called the test routine) must appear at the start of the file. CP/M works by loading in the specified program (assuming it is of type .COM) to location #100 and then jumping to location #100. Thus execution is transferred to the first instruction in your file.

```
        LD   HL,MESSAGE
        CALL MOUT
        RST  0

MESSAGE
        DEFM "Hello world"
        DEFB 0
```

Thus HL points to the start of the message, the message-printing routine is called and the return to CP/M is made. Note the zero marking the end of the message.

Save this program and assemble it from the menu. Now run it using the R command from the menu.

Oops, it does not work but returns to the menu having done nothing.

The first of the deliberate errors has occurred! This is a good opportunity to use the debugger, type

D from the menu

ProMON (assuming you have PMON.COM and PMON.MON on your work disc) will now load in PRINT1.COM, automatically load its .SYM file and then say Symbols loaded. Now press a key for the front panel.

PRINT1 has now been loaded into memory starting at #100. Note also that the PC (program counter) is at #100.

You can see on the left of the screen the instructions making up your program with a curly bracket at the top left marking the position of the PC. Let's start to step through the program instruction by instruction to try and see where things are going wrong.

Press Z to execute one instruction. Note that the value in the PC becomes #103, the curly bracket addresses the next instruction, and the value in HL becomes #107 which is the start of the message. All this is to say that the instruction LD HL,MESSAGE has been executed.

Now press Z again. This does not execute the CALLed subroutine but follows the code into the subroutine. Thus the PC jumps to the start of the subroutine.

You are now at the start of MOUT. Single-step the next instruction (Z) and notice that the accumulator (A register) becomes loaded with #48 which is the ASCII value of H, the first byte of the message. The next instruction increments HL and then A is tested for zero with the OR A instruction. Single-step both of these and note that the flags do not show zero (there would be a Z if so).

When you single-step the next instruction you can easily see the mistake. The return has been made already, even though A does not hold zero. This instruction should be RET Z not RET NZ. This is why none of the message was printed.

ProMON has done its work for the minute so quit (Q and Y).

Edit the text (press S at the menu) to change the RET NZ to a RET Z and save it, then assemble and run from the menu; this should now print your message on the screen and then return to the menu.

Now we can return to the editor and edit PRINT1.GEN to make a useful library file of it. Delete the three line test program and the message, so that you're just left with MOUT and PUTCHAR. Now we'll make the assembler do a bit of work and make the program easier to understand.

Change PUTCHAR so that the instruction LD C,2 becomes LD C,WRITE and the CALL 5 becomes CALL BDOS . In the main file we will include the two instructions BDOS EQU 5 and WRITE EQU 2. These last two types of instructions are called equates and while they don't change the code generated, they do make the source code easier to read and understand. Having made those changes, save the new file as MOUT.LIB by editing the name as you save it.

Accessing the disc directory

It may seem like rather a large jump from printing on the screen to accessing the disc, but the way CP/M is designed makes handling the disc reasonably easy. One just calls CP/M in exactly the same way as for writing characters as we've seen above. This section will be concerned with designing a routine to access the disc directory to get at the names of files on the disc.

Obviously when dealing with files on the disc, there must be some method of knowing which file or files are being dealt with and a place to store file information such as:- the disc the file is on, the filename, the position on the disc where the file is stored, and which section of the file has just been read or written. The way CP/M handles this problem is by means of a 33 byte data area called a *File Control Block* (or FCB). An FCB for a file can reside in memory and when operations on the file in question are finished, the act of closing the file writes the memory FCB into the disc directory.

An FCB contains all the relevant information for a file, but all that need concern us here is that bytes 2 to 12 contain the filename and filetype. If the filename is less than 8 characters or the filetype is less than 3 then the remaining bytes of the name or the type are padded out with spaces (ASCII 32).

The two functions we are going to be mainly concerned with here are the CP/M functions 17 and 18, which are `FIND FIRST` and `FIND NEXT` respectively. You use them by making `DE` point at an FCB and they search the disc directory for a file whose name matches the one in that FCB. These functions are going to be useful to us because the memory FCB is allowed to be ambiguous. This means that the filename and filetype can have `?`s (ASCII 63) in them. This character is a wild-card, meaning that it will match with any character. The upshot of all this is that if the filename and filetype in the memory FCB are filled with `?`s, `FIND FIRST` and then `FIND NEXT` used repeatedly will find all the files on the disc.

Right, after all that theory, let's get down to using **Devpac80**. From the menu, type `E` to edit a new file and type in the name of this new `Edit file` as `FILES [RETURN]`.

This program is going to be the main file, so let's put some useful equates at the top (maybe you want to put a couple of comments in concerning function and date etc).

Type in the lines:-

```
;Some CP/M addresses
BDOS EQU   5
DMA        EQU  #80

;CP/M function numbers
WRITE      EQU  2
FINDFIRST  EQU  17
FINDNEXT   EQU  18

;ASCII values
NULL EQU   0    ;Message terminator
LF         EQU  10       ;Line feed
CR         EQU  13       ;Carriage return
BLANK      EQU  32       ;Space
```

Now we'll skip the main program for a minute and write a couple of FCBs. So type in the following at the end of the program:

```
ambigFCB
           DEFB NULL              ;Byte 1 is the drive number
           DEFM "???????????  ;11 chars.Ambiguous filename
           DEFS 20,0              ;The rest

workFCB
           DEFB NULL
           DEFM "              "
           DEFS 20,0
```

There are two FCBs: one to be used for the matching process and the other to be used as a working area.

Note the use of the pseudo-operands allowed by **GEN80**. `DEFB` is used for single bytes, `DEFM` for a string of ASCII characters, and `DEFS` for space reservation (the space is actually filled with zero's).

Now we'll write a general-purpose routine to use the ambiguous FCB. Type the following text in between the equates and the two FCBs.

```
CPMFCB

;C holds the desired CP/M function number on entry
;which will be either FIND FIRST (17) or FIND NEXT (18)

          PUSH HL
          PUSH DE
          PUSH BC
          LD   DE,ambigFCB  ;Set up DE
          CALL BDOS
          POP  BC
          POP  DE
          POP  HL
          RET
```

Note that we haven't saved AF in this routine. This is because A holds valuable information after using CP/M calls 17 and 18. Let's now write a simple test routine to see if we're doing things half-way correctly. Immediately after the equates include the following:-

```
          LD   C,FINDFIRST
          CALL CPMFCB
          RST  0
```

As you may gather, this routine provides no output, but first things first. Save the program and then assemble from the menu.

The deliberate mistake strikes again (be of good cheer! It's the penultimate one) and the assembler reports a `String not terminated` error on the first pass. Press a key to get the editor up and then issue the **Goto Next Error** command. You will see that the question marks begin with a " to signal a string but there is no finishing ". Go to the end of the question marks and type a " so that the line looks like:

```
          DEFM "???????????"  ;11 chars.Ambiguous filename
```

then save and re-assemble.

All should be well, so go right ahead and execute the program from the menu. The disc should have been accessed, and the return to the menu is made, but nothing very exciting happens. At least the machine didn't fall over, so let's use **ProMON** to try and see what happened.

Type D from the menu and press a key once the symbols have been loaded.

Single-step (Z) the program up to but not including the CALL #0005 instruction. At this point, type RV and note that DE points to the ambiguous FCB. The first byte is zero and the next eleven are all question marks.

Execute the whole of the subroutine, which goes into the heart of CP/M, at one go by pressing EM (to *miss* the subroutine) or simply Z again (since location 5 is below LOW and therefore will not be single-stepped unless you change LOW).

The disc drives should spring into action as CP/M function 17 is executed and the curly bracket and the PC value will change. Now just step through the POPs and the RET at the end of the subroutine.

The important register at this stage is A. After a call to FIND FIRST or FIND NEXT, A can hold one of five values: #FF indicates that no match has been found (i.e. no file in the disc directory matches the filename in the FCB addressed by DE), and 0, 1, 2 or 3 give information as to where you can find the disc FCB.

As a word of explanation here, if FIND FIRST or FIND NEXT are successful in the search, they copy the matching FCB from the disc into memory and specifically into the Direct Memory Access area or DMA. Suffice it to say for the purposes of this program, the DMA is 128 bytes long and the default DMA used by CP/M is at address #80 and thus extends to #FF or just below where programs sit.

Now, to find out where the matching FCB is, one has to multiply the value in A by 32 (the length of an FCB) and add this to the address of the current DMA. Do this in your head and you'll come up with an address #80 (for A=0), #A0 (for A=1), #C0 (for A=2) or #E0 (for A=3). Look at the A register on the front panel and work out this address now.

There are a few methods of grubbing around in memory, but perhaps it is best at the moment to use the simplest, the MA command. Having worked out the address you want to look at (#80, #A0, #C0 or #E0) type:

MV	to recover the hex and ASCII display
MA	and then the address you want, you don't need the #

If you now look to the right of the memory block where the ASCII values of the bytes in memory are displayed, you will see that there is a zero and then a filename. This is the first part of a disc FCB. So FIND FIRST works! The use of FIND NEXT is identical in operation, except that you can use FIND NEXT repeatedly to get all subsequent files (i.e. until it returns a value of #FF in A).

Printing out filenames

Now we can get down to the meat of the program. Delete the 3 line test program and type in the main structure of the program as below:-

```
            LD   SP,(6)          ;See below
            CALL TITLE           ;Print a title
ONE_TIME
            LD   C,FINDFIRST     ;The first time
            JR   TIME_1

NEXT_TIME
            LD   C,FINDNEXT      ;Other times
TIME_1      CALL CPMFCB          ;Do the search
            CP   #FF
            JR   Z,FINISH            ;If no more

            CALL GET_TO_NAME     ;Locate the FCB
            CALL PRINT_FCB       ;Print the filename
            CALL CHECK_NAME      ;Is the name legal
            JR   NC,NAME_OK      ;If legal

            CALL BAD_MESSAGE     ;Show its bad
            CALL DELETE_BAD      ;Delete the offender
            JR   ONE_TIME            ;Back and start again

NAME_OK
            CALL GOOD_MESSAGE    ;Show its good
            JR   NEXT_TIME       ;Back for the next

FINISH      CALL CONCLUDE            ;Print an end message
            RST  0               ;Back to CP/M
```

This could be written in a much more compact way, but the listing given has the benefit of being sufficiently modular to be able to develop the program easily section by section. The first instruction in the program deserves some explanation. What it does is to set up the stack immediately under CP/M (i.e. in a safe place).

Before writing the first of the subroutines, read in the message-printing library file (MOUT.LIB) from the disc. Make sure it sits below the main program. Now type in the following text (in between the main program and the printing routines) to take care of all the messages:-

```
TITLE           PUSH    HL
                LD      HL,TITLE_M
                JR      OUT_MESS
GOOD_MESSAGE
                PUSH    HL
                LD      HL,GOOD_M
                JR      OUT_MESS
BAD_MESSAGE
                PUSH    HL
                LD      HL,BAD_M
                JR      OUT_MESS
CONCLUDE
                PUSH    HL
                LD      HL,END_M
OUT_MESS
                CALL    MOUT
                POP     HL
                RET

TITLE_M         DEFB    CR,LF
                DEFM    "DELETE ILLEGAL FILES (C) HiSoft"
                DEFB    CR,LF
                DEFM    "Written by R. Teller 14/2/85"
                DEFB    CR,LF,CR,LF,NULL

GOOD_M          DEFB    "Ok",CR,LF,NULL

BAD_M           DEFM    "Erased"
                DEFB    CR,LF,CR,LF,NULL

END_M           DEFB    CR,LF
                DEFM    "Program finished. Disc files Ok."
                DEFB    CR,LF,NULL
```

Note that to save space, all the message-printing routines jump to the same point to print the message and exit. Note also that it is quite possible to print carriage returns and line-feeds as any other character.

Now, as the final task in this section, the two routines GET_TO_NAME and PRINT_FCB will be developed, so that the filenames got by FIND FIRST and FIND NEXT can be located and printed out.

Type in the following routine that will take the value in A returned by FIND FIRST and FIND NEXT and point DE and HL at the found FCB.

```
GET_TO_NAME
                LD      HL,DMA
                ADD     A,A
                ADD     A,A
                ADD     A,A
                ADD     A,A
                ADD     A,A         ;A=A*32
                ADD     A,L
                LD      L,A         ;HL=DMA+(A*32): the FCB
                INC     HL          ;Filename starts at 2
                LD      DE,workFCB
                PUSH    DE
                INC     DE          ;Skip the first byte
                LD      BC,11       ;Filename(8)+filetype(3)
                LDIR                ;Copy filename from the
                                    ;DMA to the working FCB
                POP     HL
                LD      D,H
                LD      E,L
;DE and HL now point to the working FCB
                RET
```

Now to address the problem of printing the filename. It can be done in two parts. First print the filename, remembering that this is always 8 characters long and padded with spaces. Then print a full-stop, followed by the filetype printed in exactly the same way as the filename.

Having inwardly digested the strategy, type in the following subroutine to implement it.

```
;Print the filename from FCB addressed by HL

PRINT_FCB
                LD      B,8  ;Length of the filename
PFCBNAME
                INC     HL
                LD      A,(HL)
                CALL    PUTCHAR
                DJNZ    PFCBNAME

                LD      A,"."
                CALL    PUTCHAR

                LD      B,3 PFCBTYPE
                INC     HL
                LD      A,(HL)
                CALL    PUTCHAR
                DJNZ    PFCBTYPE

                LD      A,BLANK
                JP      PUTCHAR
```

Note that a space is printed at the end and an exit is made through the character printing subroutine.

Now, that would seem to complete bulk of the program. The final problem of checking the filename for illegal characters will be tackled in the last section. For the minute, therefore, make the two remaining subroutines null i.e. just RET thus:-

```
CHECK_NAME
DELETE_BAD
                RET
```

A lot has been typed in this session, and mistakes of one sort or another are quite likely. Thus, before assembling, type the following command line as the very first line in your program and then save it (still as FILES.GEN):

```
*F,N,W [RETURN]
```

This says to **GEN80**, assemble the source file FILES.GEN: if there are any mistakes in the first pass then press on to second pass (F stands for Force pass 2): send all output to a file called FILES.PRN (W stands for Write a print file) and do not generate the object file FILES.COM (N stands for No object file).

This is a fairly typical set of options for checking syntax of a source file. Assemble using A from the menu, any errors occurring will appear on the screen; at the end of the assembly get to the menu and correct any errors using the **Goto Next Error** command repeatedly.

When all errors have been corrected, assemble FILES properly (i.e. generate a .COM file, remove the N from the first line).

When you execute the program all should go well and what amounts to a disc directory will be given with a message printed at the beginning, the message Ok printed after every file and a message printed at the end.

Parsing the filename

The final important routine is CHECK_NAME. We are going to enter this routine with DE pointing to an FCB and want it to parse the filename and return with carry flag set if the name is illegal in some way. It is necessary, of course, to define what is an illegal filename. For present purposes, any of the following characters is illegal:-

a) Control characters (i.e. values less than #20)
b) Characters in lower-case. These should never be found in filenames anyway.
c) Any character in the following list:-

```
<>.,;:=?*[] %|()/\
```

Characters with their top bit set (i.e. value more than #7F) will have it reset and will not be treated as a bad character, as CP/M sets the top bit of some characters in the filename to indicate the status of the file.

The following subroutine will perform the function so insert it between the labels CHECK_NAME and DELETE_BAD that were typed in before:-

```
;Parse the filename and return C if illegal

CHECK_NAME
          LD   H,D
          LD   L,E       ;Reset HL to point to the FCB
          LD   B,11      ;Check all 11 characters

DO_NEXT
          INC  HL        ;1 is not checked (drive number)
          LD   A,(HL)
          RES  7,A       ;Ignore the top bit

          CALL CHECK_CHAR  ;Is it in the table
          RET  C         ;If so
          CP   BLANK
          RET  C         ;If a control character
          CP   "z"+1
          JR   NC,NEXT_CHAR
          CP   "a"
          RET  C         ;If between a and z inclusive

NEXT_CHAR
          DJNZ DO_NEXT ;For all 11 characters

          OR   A         ;Give no carry=OK
          RET
```

As you can see the most difficult section of the program i.e. *is the character in the table of illegal characters* has been turned into a subroutine which is generally good practice in cases like this. From the context, the subroutine CHECK_CHAR must preserve HL, preserve A, and return carry or not as appropriate. Type in the following subroutine:-

```
;Preserves HL and A. Gives carry if the character in A is
;present in the table of illegal characters.

CHECK_CHAR
          PUSH HL
          LD   HL,BAD_CHARS ;The table
          LD   C,A          ;Keep the original
                            ;character in C

CH_LOOP   LD   A,(HL)       ;An illegal character
          INC  HL
          OR   A
          JR   Z,END_CHECK  ;If at end of table return NC
          CP   C
          JR   NZ,CH_LOOP   ;If no match
          SCF               ;If it does match
END_CHECK
          LD   A,C          ;Restore A
          POP  HL           ;Restore HL
          RET

;Table of illegal characters in filenames
BAD_CHARS
          DEFM "<>.,;:=?*[] %|()/\"
          DEFB NULL         ;At end of table
```

The program is very nearly finished now. As a safety precaution, we'll still leave the actual deleting of the files until the parsing is seen to be correct.

Save this version and assemble and run it. As you can see, things do not quite go according to plan. All the files have the message `Erased` after them! This bug is actually quite obscure, and a final opportunity to use **ProMON** in this tutorial.

Activate **ProMON** (type D from the menu) and step the first instruction. This sets up the stack and need not concern us.

There was no problem with the opening message so execute the whole subroutine using `EM`. Use `PS` to clear the message from the screen. The next three instructions load `C` with `#11` and then jump to the subroutine. The searching also appeared to be all right so execute the whole subroutine again here (`EM`).

The drive should be activated and then the return to **ProMON** made. Note the value in A which should not be #FF. Step the comparison and the jump (which will not occur). The next subroutine points HL and DE to the found FCB and should be all right as should the next one which prints out the filename (remember the filenames appeared OK when the program was executed).

We want to execute both subroutines GET_TO_NAME and PRINT_FCB. We can do this most easily by using EM twice. However, another way is to set a breakpoint after the second routine and we'll do this now to see how to set breakpoints.

Enter BS (**Breakpoint Set**) and type 11A, the address at which we want a breakpoint. Note that the breakpoint display is now updated to show that we have set one. Now, to execute up to this breakpoint press EQ (**Execute Quick**). When the breakpoint is reached (which will be instantaneous), press a key to return to the front panel, the breakpoint will have been reset.

It is almost certainly the next subroutine which is wrong and so single-step into it. Before executing any instructions, verify that DE points to an FCB, use the RV command.

The first two instructions set HL pointing to the FCB and then the first character is loaded into A and the top bit stripped. Now the routine CHECK_CHAR is called. Just as a preliminary test, execute it in one go, EM. It returns no carry as it should and both A and HL are preserved so press on with the main routine.

The next comparison is for control characters and this gives no carry as expected. Then the test is made for lower-case characters. The value in A will be lower than z+1 so a carry will be generated here. The comparison is then made with a and, as expected, the value in A is lower and thus a carry is generated again. This is where the problem is. We must return with carry if that last comparison gave no carry and vice versa. The answer is surely to insert a CCF instruction here.

Quit **ProMON** and change the source to include the instruction CCF (complement the carry flag) between CP "a" and RET C. Save, assemble and run. This time there should be no problems with executing the program and all the filenames should have Ok after them.

The final touch is thus to write the routine DELETE_BAD. Use the label already put into the file and type simply:-

```
DELETE_BAD
          LD   C,ERASE
          JP   CPM
```

and then add the equate

```
ERASE     EQU  19
```

after the others at the top of the file.

As a word of explanation, CP/M function 19 requires that DE points at the FCB of the file to be deleted. This it already does (from GET_TO_NAME) and, as no registers need be preserved at this stage, the two lines above will suffice.

Save and assemble this final version. On testing it you will find a slight problem in that after a file is deleted, the routine terminates. This is because FIND NEXT will return #FF in A if the directory is altered (i.e. a file deleted). There are many solutions to this problem. None is particularly the right one, but one suggestion would be that after a file is deleted, the jump is to the LD C,FINDF instruction which would keep running through the directory until all files were legal.

A further suggestion is to write routines that instead of deleting a file with an illegal name, ask the user for a name (get characters from the keyboard using CP/M function 1) and then rename the file (CP/M function 23).

Hopefully, this tutorial has done the three things it set out to do.

Firstly it introduced the use of **GEN80** and **ProMON** in the development of assembly language programs.

Secondly it illustrated (although by no means exhaustively) how to program under CP/M.

Lastly, it has provided a useful utility program, together with some ideas on how to improve it.

Making an RSX under CP/M Plus

An RSX is a special type of program available only under CP/M Plus (e.g. on the Amstrad CPC6128 and PCW 8256/8512/9512 computers).

You can install an RSX into your CP/M by using GEN80, LINK and GENCOM (the last two utilities will be supplied with your CP/M Plus system). When an RSX is installed, it sits in high memory just under the BDOS and lowers the top of your TPA. If you install more than one RSX then each can be chained to the next RSX up.

The most-recently installed RSX intercepts all BDOS calls and has the chance of processing them or ignoring them and passing them on to the next RSX in the chain and, ultimately, to the BDOS.

Thus, RSXs, once installed and until removed, sit permanently in your CP/M system intercepting and possibly processing all BDOS calls. So, common uses for RSXs are printer buffers, key conversion routines, file loaders and the like.

RSXs can be attached to ordinary programs so that they are loaded automatically whenever the program is executed. Alternatively they can be attached to null programs so that just the RSX is loaded. Both these functions (of attaching RSXs to programs) are performed by the GENCOM utility.

GENCOM takes a command line involving one .COM file followed by a number of .RSX files (page relocatable files with .RSX extensions), each being an RSX, and then attaches the RSXs to the .COM file so that the RSXs are loaded automatically when the .COM file is invoked.

To GENCOM only RSXs (i.e. produce a file that just loads the RSXs and does nothing else) you include the option [NULL] (including the square brackets) after the files. Some examples of GENCOM:

```
GENCOM LANG,FRENCH,SPANISH,GERMAN [RETURN]
```

takes the files LANG.COM, FRENCH.RSX, SPANISH.RSX and GERMAN.RSX and produces LANG.COM with the three RSXs attached to it.

```
GENCOM PBUFFER [NULL] [RETURN]
```

takes PBUFFER.RSX and produces a dummy file PBUFFER.COM which simply loads the RSX.

So how do you produce the .PRL files that GENCOM uses? The answer is, with GEN80 and LINK; you use GEN80 to produce a relocatable file (a .REL file) and then use LINK with the [op] option to convert this to a page relocatable file (a .PRL file) and then rename this file to have an extension of .RSX, ready for GENCOM.

There is a further complication in that RSXs must start in a particular way so that CP/M can identify them and link them together; the first 27 bytes of every RSX should look like this:

The RSX Header

```
serial      ds      6,0         ;serial number
start       jp      begin       ;start of RSX
next        jp      0           ;the next RSX
prev        dw      0           ;the previous RSX
remove      db      -1          ;the remove flag
nonbank     db      0           ;a bank flag
name        defm    "RSX NAME"  ;8-character name
loader      db      0           ;the loader flag
            dw      0           ;reserved
```

`serial` The CP/M loader loads the operating system's serial number into `serial` when the RSX is loaded, so you just need to reserve 6 bytes here.

`start` This location is executed when the RSX is called so, normally, this should go to the beginning of the RSXcode.

`next` The jump to the next RSX in the chain, this will be filled in by the loader when this RSX is loaded. This RSX should normally finish by jumping to `next`, also all its BDOS calls must go through `next` so that the other RSXs have a chance of intercepting them.

`prev` The address of the previous RSX, if any. Again, this is filled in by the loader and will not normally be useful.

`remove` If this is set to `#ff` (-1) then, on the next call to the CP/M loader, this RSX will be removed. A warm start (`RST 0` or jump to location 0) always calls the loader so that, if `remove` is -1, then this RSX will be removed after the program that calls it finishes. The only exception to this is when an empty file with an attached RSX has been loaded; this is so that you can install an RSX in the system without running a program. Note that removing the RSX lowest in memory frees the memory occupied by that RSX but that removing an RSX that is not lowest in memory only removes it from the chain and does not free the memory it occupies until all the RSXs beneath it have been removed.

`nonbank` If `#ff` (-1) then this RSX will only be loaded on non-banked CP/M Plus systems, otherwise the RSX will always be loaded. You would set this flag to -1 if the RSX code is not capable of running in a banked system.

`loader` Should always be 0 since a value of -1 identifies the last RSX in the chain which is always the loader RSX.

Right, how about an example of producing and installing an RSX?

Type in the following two programs using **HDE**, the first one is an RSX to print out the time and the second one is a small program to call the RSX.

Program 1 `TIMERSX.GEN`

```
;produce a .REL file
*r+
                ds      6,0             ;serial
                jp      start           ;RSX beginning
next            jp      0               ;loader will fill in
prev            dw      0               ;loader will fill in
remove          db      -1              ;remove
```

```
nonbnk      db      0           ;all systems
            db      "PRTIME  "  ;RSX name
            db      0,0,0       ;not last RSX

;the beginning of this RSX
start       ld      a,c         ;BDOS number
            cp      #3c         ;call RSX function?
            jr      nz,next     ;no, finished
            ld      a,(de)      ;get required RSX #
            cp      43          ;is it us?
            jr      nz,next     ;no, finished

;the code to print out the time
            ld      c,105       ;get-time function
            ld      de,time     ;buffer for time
            call    next        ;call the BDOS via any
                                ;other RSXs
            ld      a,(time+2)  ;get hours
            call    byteout     ;and print them
            ld      a,":"       ;print a colon
            call    charout
            ld      a,(time+3)  ;get minutes
                                ;and print them

;subroutine to print out BCD number in a
byteout     push    af
            rra                 ;divide by 16
            rra                 ;to get low digit
            rra
            rra
            call    digout      ;put out low digit
            pop     af          ;and then high digit

digout      and     %1111       ;only 4 low bits
            add     a,"0"       ;get ASCII number

;subroutine to print character in a to screen
            ld      e,a         ;BDOS needs it in e
            ld      c,2         ;BDOS function #
            jp      next        ;go through other
                                ;RSXs and return

;storage buffer for time (actually date and time)
time        ds      4
```

Program 2 CALLTIME.GEN

```
;A small program to call the TIMER RSX. Normally, the RSX
;will be bound to this program.

            ld      sp,(6)      ;safe place for stack

            ld      c,#3c       ;call RSX function #
            ld      de,rsxpb    ;parameter block for RSX
            call    5           ;call the BDOS
            rst     0           ;warm boot, removes the
                                ;RSX if remove=-1

rsxpb       db      43          ;the RSX number
; db npara   the number of parameters
; dw param1,param2,etc.   the actual parameters
```

You can assemble the above programs either from within the **HDE** menu or by using:

```
GEN80 TIMERSX [RETURN]          to produce TIMERSX.REL
GEN80 CALLTIME [RETURN]         to produce CALLTIME.COM
```

Now link TIMERSX.REL to produce TIMERSX.RSX by using:

```
LINK TIMERSX [op] [RETURN]
ERA TIMERSX.RSX [RETURN]
REN TIMERSX.RSX=TIMERSX.PRL [RETURN]
```

and, finally, bind the RSX to the CALLTIME program with:

```
GENCOM CALLTIME TIMERSX [RETURN]
```

That's it. You can now get the time by typing CALLTIME [RETURN] which will load the RSX, call it and then remove it.

We hope that the above introduction to the production of RSXs has been useful; you can get more information from any good book on the CP/M Plus operating system.

Bibliography

The following books are recommended for CP/M assembly-language programming:

Programming the Z80	Rodney Zaks	Sybex 1982
Z80 Assembly Language Programming Manual	Zilog	Zilog, UK (0628) 39200
CP/M The Software Bus	Clarke, Eaton & Powys Lybbe	Sigma 1983
CP/M User Guide CP/M Plus Handbook	Digital Research	Digital Research (0635) 35304
The Amstrad CP/M Plus	Powys-Lybbe, Clarke	MML Systems (01) 247 0691