HiSoft Pascal80

HiSoft Pascal80 consists of a fast Pascal compiler and a full screen editor together with several example programs.

How to use this manual

This manual is divided into 3 parts. The first tells you how to get started and describes the free example programs. The second is about the particular features of Pascal80 and the third describes the full screen editor ED80.

You should read the **'What to do first'** section on the next page, then familarise yourself with ED80 and then read the Pascal80 section.

If you get stuck read **'The questions we are asked most often'** section at the very back of this manual first.

David Nutkins.
February 1986.

HiSoft
180 High Street North
Dunstable
Beds
LU6 1AT
U.K.

Tel (0582) 696421

HiSoft Pascal80

First Edition Published 1981
Second Edition Published 1985
Third Edition Published February 1986

ISBN 0 948 517 03 4

What to do First

Before you use HiSoft Pascal80 please make a backup copy; if you destroy your master disc we will have to charge you for updating it. It is a good idea to write-protect the disc if is not already write-protected.

Please read the relevent section below depending upon which computer you have on the following pages and then read the section '**For all computers**'. It is essential to read this section before using the compiler.

1. Amstrad CPC 464/664
2. Amstrad CPC 6128 or 464/664 with expanded memory and CP/M Plus
3. Amstrad PCW 8256
4. MSX computers
5. Other computers

1. Amstrad 464/664

Use `|CPM [ENTER]` to load CP/M from your Amsoft CP/M System/utilities disc or a copy of it. When CP/M has loaded use `FORMAT` to format a new disc and then use `DISCCOPY` (or `COPYDISC` if you are lucky enough to have 2 drives) to copy the Pascal80 disc Side A onto your newly formatted disc. Side B is of no use to you. Next you should use `BOOTGEN` and `SYSGEN` to copy the system onto your disc. You will then have a working copy of Pascal80.

Reset your computer using `[CTRL][SHIFT][ESCAPE]` and type `|CPM [ENTER]` with your working copy in the drive and CP/M should load.

Remember when using CP/M 2.2 that whenever you change discs you should type [CTRL] C at the A> prompt otherwise you will get '`BDOS ERROR ON A:R/O`' errors when you write to a disc.
The version of ED80 supplied is configured for the CPC 464 screen and has special code to give a flashing cursor; thus it will only work on CPC machines under CP/M 2.2.
Remember *not* to use the `SETUP` command to set the mode to 'FAST'. As the Amstrad Disc Firmware Supplement states, 'FAST' is a misnomer and should not be used.

2. Amstrad CPC 6128 or 464/664 with expanded memory & CP/M Plus

Use `|CPM` to load CP/M from your Amsoft CP/M System/utilities disc Side 1 or a copy of it. When CP/M has loaded use `DISCKIT3` to copy the Pascal80 disc **Side B** onto a blank disc. Side A is only for use under CP/M 2 and we recommend that you use CP/M Plus. The version of ED80 has been configured for the CPC6128 CP/M Plus terminal emulator.

3. Amstrad PCW 8256/8512

Load CP/M by booting with your Amsoft System/Utilities/Basic disc Side 2 and then use `DISCKIT` to FORMAT a new disc. Then type `PIP` at the A> prompt to load the PIP program. Place your master Pascal80 disc **Side B** in the drive and type

```
M:=A:*.*
```

to the * prompt. This will copy the Pascal disc to the memory drive.

Next place your newly formatted disc in the drive and type

```
A:=M:*.*
```

to the * prompt. This will give you a backup copy of the Pascal80 disc. Note that you cannot use `DISCKIT` to copy the master disc as `DISCKIT` does not copy Vendor format discs.

To use the Pascal, it is best to follow the procedure described above to copy the Pascal backup onto the memory drive and then log into drive M by typing `M:[ENTER]` at the A> prompt. As supplied the editor ED80 works on a **24x80** screen; you should run the Amstrad `SET24X80` program (on the System/Utilities/Basic disc Side 2) before running this or alternatively use the `ED80INST` program as described in the ED80 section of this manual to change the screen size to **31x90**.

4. MSX computers

Boot MSX-DOS by switching on your disc drive first. Then insert the Pascal80 master and switch on your computer. Hit [RETURN] when MSX-DOS has booted and asked for the date. Type

```
FORMAT [RETURN]
```

and follow the instructions to format the disc you are going to use for your working copy.

When you have returned to MSX-DOS and the A> prompt place the Pascal80 disc in the drive and type `COPY *.* B:[RETURN]` and follow the instructions.

You will then have a backup copy from which you can boot.

The version of ED80 supplied has been configured for the MSX screen with 37 columns although this can be changed with the ED80 installation program. This version of ED80 will work only on MSX computers.

5. Other computers

For most machines (and nearly all with two disc drives) you should format a new disc and, if using CP/M 2.2, copy the system tracks using SYSGEN. If your system accepts a number of formats note that the exact format should be written on our master disc. Then use

```
PIP B:=A:*.*[ENTER]
```

to copy all the files onto the backup copy.

In most cases you will have to install ED80 for your screen using ED80INST. Your manufacturer's documentation should give you details of the screen codes.

Remember when using CP/M 2.2 that whenever you change discs you should type [CTRL] C at the A> prompt otherwise you will get 'BDOS' ERROR ON A:R/O' errors when you write to a disc.

In some cases, for disc formats with low capacities, Pascal80 may be supplied on two discs and you should backup them both.

For all computers

You should now have a backed-up copy of Pascal80 in your drive. Type `DIR [ENTER]` and at least the following file-names should be listed

```
HP80.COM
ED80INST.COM
ED80INST.MSG
ED80.COM
JABBER.WOK
EXTRA.WOK
PRIMES.PAS
DIS.PAS
HEX.PAS
RANDREC.PAS
BIRTHDAY.PAS
```

There may also be other files on disc. If you don't get a directory with the files above then you should try again to make a back up copy ; if you have no success please return your disc to your supplier or HiSoft.
Amongst the directory there may be a file called READ.ME. If there is you should use

```
TYPE READ.ME
```

to read it. If there are any files with an extension of `.DOC` it is a good idea to `TYPE` these as well.

To make sure that the compiler is working properly you should type

```
HP80 PRIMES
```

This will then create the file `PRIMES.COM`

To run this type

`PRIMES[ENTER]` at the A> prompt. This will then produce a list of prime numbers less than 20499; there are quite a few of these!

`ED80INST.COM` and `ED80INST.MSG` comprise the ED80 installation program which is described in the ED80 manual. If you don't have one of the computers listed above then you will probably need to install ED80.
Once you have a copy of ED80 installed for your system it is a good idea to create a working disc with a `HP80.COM`, `ED80.COM`, `ED80.HLP` and in addition the program that you use for transferring files from disc-to-disc (normally `PIP.COM`). If you are using CP/M 2 remember to use a disc with CP/M on it.

There now follows information on the various other files on the disc and a section on the problems that people most frequently encounter.

`JABBER.WOK` and `EXTRA.WOK` are example files used in the ED80 tutorial.

`DIS.PAS` is an interactive Z80 disassembler intended primarily as an example of "systems" programming in Pascal80 and as a tool for assembly language programs. This is described in a later section.

`HEX.PAS` is used by `DIS.PAS` and contains 3 functions/procedures; it cannot be compiled separately. The 3 functions/procedures in `HEX.PAS` are

1 `upper`

has one CHAR parameter and returns the character in upper case.

2. `DUMP`

The first parameter is a file to which a hexadecimal and ASCII dump of memory between the second and third integer parameters is sent.
e.g. `DUMP(OUTPUT,0,128)` dumps the operating system's base page to the screen.

3. `READHEX`

has one file parameter and reads a hexadecimal number returning it as an INTEGER. The file buffer variable is left on the terminating character.

RANDOM ACCESS

RANDREC.PAS contains procedures and functions for random access disc files.

You should be familar with Pascal before attempting to use them.

When using READRAND and WRITERAND, files are considered to be an array of fixed sized records known as File Components. The first record in the file is File Component 0. Before using these routines, the file to be read/written to should be declared as type TEXT and opened using RESET or REWRITE. If you write to a file using WRITERAND then you must close it by calling the procedure CLOSE. This has one parameter of type TEXT.

READRAND is a function which returns TRUE if a record can not be found on the disc. It returns FALSE if the data was read successfully. The first parameter is the Text File variable you wish to read. The second parameter is the File Component you wish to read starting from 0. The third parameter is the address of the variable you wish to read into. The fourth parameter is the size (in bytes) of the file components.

WRITERAND is a procedure with 4 parameters: the first parameter is the Text File variable to which you wish to write The second parameter is the number of the File Component you wish to write. The third parameter is the address of the variable you wish to write. The fourth parameter is the size of the components.

Note that when using these routines EOF will not give sensible results. If you wish to use variable sized records then it is best to call the lower level routines READR and WRITER. These have three parameters. The first is the text file to use, the second the 128-byte record to read and the third is the address of the 128-byte record to which the read/write takes place. They return the result of the CP/M random access call; this is 0 if there is no error.

For example: The following program creates a file TESTDATA.DAT and writes 3 records and then reads in the second record and displays it.

```
PROGRAM A;
VAR F:TEXT;
    S:ARRAY[1..8] OF CHAR;
    DUMMY:BOOLEAN;
{$F RANDREC.PAS }
BEGIN
  REWRITE(F,'  TESTDATA.DAT');
  S:='RECORD 1';
  WRITERAND(F,0,ADDR(S),8);
  S:='RECORD 2';
  WRITERAND(F,1,ADDR(S),8);
  S:='RECORD 3';
  WRITERAND(F,2,ADDR(S),8);

{ We have written the file - now let's read in the second
  record }

  DUMMY:=READRAND(F,1,ADDR(S),8);
  WRITELN(S);
  CLOSE(F)   {We must explicitly close the file }
END.
```

`BIRTHDAY.PAS` is an example of the use of the routines in `RANDREC.PAS` and manipulates the file `BIRTHDAY.DAT`. It can be used as the basis of your own programs.

The following commands can be used when the program is running.

I	inserts a record
D	deletes a record
P	prints a record
E	exits

You should always use E to exit rather than CTRLC so that the file is properly closed.

Amstrad-Specific Programs

The files `FROMAMS.COM` and `TOAMS.COM` convert files from and to the format used by the HiSoft Pascal compiler that runs under AMSDOS.
Their syntax is

```
FROMAMS destfile sourcefile
TOAMS destfile sourcefile
```

e.g.

```
FROMAMS TEST.PAS TEST.AMS
TOAMS TEST.AMS TEST.PAS
```

These programs can be used to convert programs developed under Amsdos for use under CP/M, or to edit programs being written under Amsdos with ED80 (or heaven forbid another editor).

HISOFT PASCAL80 TURTLE GRAPHICS FOR THE AMSTRAD CPC SERIES

Amstrad CPC 464/664 owners are supplied with a program called `TURTLE2.PAS` on their disc; this is a Turtle Graphics example program which runs under CP/M 2 and uses the Amstrad firmware. For CPC 6128 owners we have `TURTLE3.PAS` which is the same as `TURTLE2.PAS` except for the method used to call the firmware. The Amstrad PCW 8256 does not contain these firmware routines so they will not run.

To compile the programs as they stand, use

```
HP80 TURTLE2;T,Y[ENTER]
```

or

```
HP80 TURTLE3;T,Y[ENTER]
```

depending on your machine.

The demonstration can then be run by typing the program's name at the A> prompt.

To write your own programs you should make a copy of the demonstration program, delete the parts that you do not require and then add your own program. Note that you must use the 'T' option in the command line as in the example above.

As in the majority of Turtle Graphics implementations, HiSoft Pascal's TURTLE creates an imaginary creature on the screen which the user can move around via some very simple commands. This 'turtle' can be made to leave a trail (in varying colours) or can be made invisible. The turtle's heading and position are held in global variables which are updated when the creature is moved or turned; obviously these variables may be inspected or changed at any time.

The facilities available are as follows:

GLOBAL VARIABLES

heading

this is used to hold the angular value of the direction in which the turtle is currently facing. It takes any REAL value, in degrees, and may be initialised to 0 with the procedure TURTLE (see below). The value 0 corresponds to an EASTerly direction so that after a call to the procedure TURTLE the turtle is facing left to right. As the heading increases from zero then the turtle turns in an anti-clockwise direction.

Xcor, Ycor

these are the current (x,y) REAL co-ordinates of the turtle on the screen. The Amstrad graphics screen has a logical size of 640x200 pixels and the turtle may be positioned on any point within this area; however the resolution is not one pixel - it varies according to the screen mode as in Locomotive BASIC.

Initially XCOR and YCOR are undefined; use of the procedure TURTLE initialises them to 300 and 200 respectively, thus placing the turtle near the middle of his 'pool'.

penstatus

a BOOLEAN variable holding the current status of the 'pen' (i.e. the trail left by the turtle). TRUE means the pen is down, FALSE means the pen is up.

PROCEDURES

The first few procedures are general graphic utilities that you may find useful in your own non-Turtle graphic programs. Most of the graphics features of Locomotive BASIC are accessible in a similar manner using the firmware. See the Amsoft Firmware Manual or equivalent documentation. The procedures available are:

`mode( m : INTEGER)`

this takes an integer between 0 and 2 and puts the screen into the corresponding mode.

`ink( i,c1,c2 : INTEGER)`

sets the ink **`i`** to have the colour values specified by **`c1`** and **`c2`**. If **`c1=c2`** then the ink will be a steady colour, otherwise the ink will be flashing.

E.g.

`ink(1,12,12);` sets ink 1 to steady yellow
`ink(0,16,21);` sets ink 0 to flashing pink and lime!

`paper ( i:INTEGER )`

sets the background (paper) colour of the screen to the colour(s) associated with the ink **`i`** which is an integer.

`pen ( i : INTEGER)`

sets the turtle's pen colour to the colour(s) associated with the parameter **`i`**.

`plot ( x,y : INTEGER)`

plots the point **`(x,y)`** on the screen.

line (x,y : INTEGER)

plots a line from the last point plotted to the absolute location **(x,y)** .

pendown(i:INTEGER)

sets the turtle state so that it will leave a trail in the ink colour associated with the parameter **i.** This procedure assigns **TRUE** to **penstatus .**

penup

subsequent to a call to this procedure the turtle will not leave a trail. Useful for moving from one graphic section to another. **penup** assigns the value **FALSE** to **penstatus.**

sethd (angle:REAL)

takes a **REAL** parameter which is assigned to the global variable **heading** thus setting the direction in which the turtle is pointing. Remember that a heading of 0 corresponds to EAST, 90 to NORTH, 180 to WEST and 270 to SOUTH.

setxy (x,y : REAL)

sets the absolute position of the turtle within the graphics area to the value (x,y). No check is made within this procedure to ascertain if **(x,y)** is out of bounds; the firmware checks for this.

fwd (len : REAL)

moves the turtle forward **len** units in the direction of its current heading. A unit corresponds to a graphics pixel, rounded up or down where necessary.

back (len:REAL)

moves the turtle **len** units in the directly opposite direction to that in which it is currently heading (i.e. -180) - the heading is left unchanged.

right(angle : REAL)

changes the turtle's heading by **angle** degrees without moving it. The heading is increased in the clockwise direction.

left (angle : REAL)

changes the turtle's direction by **angle** degrees anti-clockwise.

arcr (r:REAL; deg:INTEGER)

the turtle moves through an arc of a circle whose size is set by **r**. The length of the arc is determined by **deg**, the angle turned through (subtended at the centre of the circle) in a clockwise direction. Typically **r** may be set to 0.5.

turtle

this procedure simply sets the initial state of the turtle; it is placed in the middle of the screen, facing EAST (heading of 0), on a bright yellow backgound paper and leaving a blue trail. Remember that the state of the turtle is not initially defined so that this procedure is often used at the beginning of a program.

This concludes the list of facilities available with **TURTLE**; although simple in implementation and use you will find that Turtle Graphics are capable of producing very complex designs at high speed. To give you a taste of this we present some example programs below. Remember that you must edit a copy of the example program before entering these.

Example Programs

In all the example programs given below we assume that you have already loaded `TURTLE2.PAS` and `TURTLE3.PAS` into the editor and deleted the text from `polyspi` onwards. Now proceed with the examples:

1. CIRCLES

Add the following variable declaration:

```
I:INTEGER;
```

and add the following main program:

```
BEGIN
 TURTLE;
 FOR I:=1 TO 9 DO
  BEGIN
   ARCR(0.5,360);
   RIGHT(40)
  END
END.
```

2. SPIRALS

Add the following main program:

```
PROCEDURE SPIRALS ( L,A:REAL );
  BEGIN
   FWD(L);
   RIGHT(A);
   SPIRALS(L+1,A)
  END;
BEGIN
 TURTLE;
 SPIRALS(9,95)          {or (9,90)  or  (9,121)  ... }
END.
```

3. FLOWER

Add the following main program:

```
PROCEDURE PETAL ( S:REAL );
BEGIN
 ARCR(S,60);
 LEFT(120);
 ARCR(S,60);
 LEFT(120)
END;
PROCEDURE FLOWER ( S:REAL );
VAR I:INTEGER;
BEGIN
 FOR I:=1 TO 6 DO
  BEGIN
   PETAL(S);
   RIGHT(60)
  END
END;
BEGIN  TURTLE;
 SETXY(127,60);
 LEFT(90); FWD(10);
 RIGHT(60); PETAL(0.2);
 LEFT(60);  PETAL(0.2);
 SETHD(90); FWD(40);
 FLOWER(0.4)
END.
```

For further, extended study of Turtle Graphics we highly recommend the excellent (if expensive) book 'Turtle Geometry' by Harold Abelson and Andrea di Sessa, published by MIT Press, ISBN 0-262-01063-1.

INTERACTIVE Z80 DISASSEMBLER

To compile this use

```
HP80 DIS;R,L-,C-,S-,A-[ENTER]
```

Note that the file `HEX.PAS` is 'included' by this program and so must be on the disc when compiling.

This program lets you disassemble a .COM and interactively set up data areas which may be saved to disc (in a .DIS file) so they do not have to be re-entered on subsequent sessions. Disassembly may be of all or part of the program and sent to disc (as a .GEN file) possibly for re-assembly using HiSoft's Devpac80 or to the screen.

When compiled use

```
DIS filename.COM[ENTER]
```

to disassemble `filename.COM`. If you haven't disassembled this program before then you will be prompted for:

'`Run Address?`' and '`End Address ?`'. These values should be entered in hexadecimal.

You can then enter commands. These consist of a single letter followed by up to two hex numbers depending on the command.

Data Area Commands:

There are 4 sorts of data areas that may be specified; they are followed by the start and end addresses of the data area.

`M`	Messages. Disassembled as DEFM and DEFB
`B`	Byte Data. Disassembled as DEFB
`W`	Word data. Disassembled as DEFW - useful for jump tables etc
`S`	Space. Disassembled as DEFS - avoids disassembling junk.

Dissassembly commands:

`L`	list to screen optionally followed by start and end address -if they are not specified the whole of the file is disassembled.
`G`	dissassemble to disc again optionally followed by a range.

Other commands:

`D`	display the currently defined data areas
`K`	kill (remove) a data area. Followed by the number as displayed by the `D` command. Note that this is in decimal rather than hex.
`P`	save the data areas to disc in a .DIS file so you don't have to type them in again.
`R`	read the .DIS file from disc. Useful if you mess things up.
`E,X`	return to CP/M. This must be used rather than [CTRL] C to ensure that all files are closed.
`H`	Help. Prints a short list of the commands available.

Some of the techniques used in this program are very machine specific and it is not an example of how to program in Pascal in general but it does show how to use the HiSoft Pascal extentions to provide a systems programming tool of practical use.

If there is a feature that you dislike or think should be added, then please feel free to improve it, and tell us about it.

CONTENTS

IMPORTANT NOTE FOR AMSTRAD USERS

We have packed so many files onto the Pascal80 disc now that it is not possible to make a back-up on a PCW8256 by `PIP`ing all the files onto the RAM (M:) disc and from there onto your 3" back-up disc. This now has to be done in two stages. We are supplying a free utility called WP.COM to facilitate this process. To make a back-up of the Pascal80 on a PCW8256 we recommend that you proceed as follows:

1. Format, using DISCKIT, a new disc in A to be used for the back-up.

2. Put the Pascal80 master in drive A (B side uppermost) and type:

```
WP A: M: [ENTER]
```

you will be prompted, one file at a time, with all the files on the Pascal80 master disc; answer Y to all of them except the ones ending in `.PAS`, answer `N` to these. This has copied the files onto the RAM disc.

3. Now put your backup disc in A and type:

```
M:WP M: -Q [ENTER]
WD M: [ENTER]
```

and then

```
A
```

when asked which files to delete.

4. Next, put the Pascal80 disc back in drive A (B side uppermost again) and type:

```
WP A:*.PAS M: -Q [ENTER]
```

5. Finally, put your backup disc in drive A and type:

```
WP M:*.PAS A: -Q [ENTER]
```

and you have now backed up the Pascal80 master disc. We apologise for any inconvenience this process may cause but it is simpler for us (and ultimately you) to supply our software in 6128 format rather than 8256/8512 format because we do not always know which machine you have; this necessitates the above procedure.

Pascal80 Version of 30 May 1986

A new procedure has been added to Pascal80 (version of 30 May 1986) to allow the chaining of programs (Pascal or otherwise) from within Pascal80 object code. The new procedure `CHAIN(filename)` is described below:

CHAIN(filename)

Invoking the procedure CHAIN causes the current object code to be discarded and a new program with the name `filename` to be loaded into the TPA. The Pascal80 program's global data are maintained but the runtime stack and heap are lost.

Filename must be a string of 14 characters (`ARRAY[1..14] OF CHAR`) of the form:

```
'A:TESTFILE.COM'
```

padded with spaces where necessary to keep a length of 14 characters e.g.

```
'  TEST    .COM'
```

to load `TEST.COM` from the default drive.

The procedure is very simple but potentially very useful; imagine a menu object module invoking the relevant files on the menu with each file either invoking another file or bringing back the menu, all through the use of CHAIN.

You can use a Pascal global variable in the program that `CHAIN`s and the program that is `CHAIN`ed as long as the *definition of the global and all the previous definitions are the same in both programs*. The best way to ensure that these declarations are the same is to store them in a file and then use the `$F` compiler include option when compiling. You must also always use the same `V` option (the one that sets the top of the runtime stack) when compiling if you want more than one program to share global variables.

Remember, CHAIN keeps the global data of the program that uses it but corrupts the local variables and the heap. Use it carefully!

MSX DISC PASCAL80

Pascal80 running under MSXDOS now comes complete with a Turtle Graphics package (`TURTLE.PAS`) and two example programs (`GRAPH1.PAS` and `GRAPH2.PAS`). The documentation for the turtle graphics package is the same as that for the Amstrad Turtle Graphics routines which can be found in the Pascal80 manual. To compile the example programs type, for example:

```
HP80 GRAPH1;L-,T [ENTER]
```

New features of the Pedigree version of HiSoft Pascal80

This document describes the extra features of the latest (from 1st September 1986) version of HiSoft Pascal80. Perhaps the biggest improvement for users who are learning Pascal or writing 'quick and dirty' programs is the Menu facility, which is described in a separate section.

The other improvements are as follows:

1. English Errors messages:

If the file HP80.ERR from your master disc is present on the logged-in drive then the compiler will display the reason for the error rather than the error number. Hitting `'E'` will now invoke the editor as described in the Menu section, `[CTRL]-C` returns to CP/M immediately whilst hitting any other key continues the compilation. In addition the line numbers displayed in the compilation listing now refer to the line number in the file rather than the compilation. Thus the line number displayed can be used to find a source line with the editor's `Go to line` command.

2. Variant records:

The variant part of records is now supported as in Jensen and Wirth. However the procedure `NEW` can only have one parameter.

3. Non-text files:

Files other than type `TEXT` are now supported. The operations `RESET`, `REWRITE`, `GET`, `PUT`, `READ`, `WRITE` and `EOF` are all supported. The non-standard buffer size parameter is not supported.

4. Bigger programs can be compiled.

The compiler now saves the object code to disc while it is compiling so the size of progam you can produce is bigger. The compiler's symbol table can now be about twice as large as before. Of course, you may not be able to run the large program you produce because there is insufficient memory to load your program or to store your variables.

The maximum size of each procedure/function and the main program is the size of your TPA -24K. Thus if you have a TPA of 38K (as on the Amstrad CPC 464/664) then the maximum size of code for each procedure is 38-24 =14K. On a 61K TPA machine this maximum is 61K-24K= 37K. If you write in a modular fashion then this will not be a problem. If you are writing procedures this large then split them up into smaller ones. As a result your code should be much easier to understand.

5. Lower case reserved words:

Reserved words and predefined identifiers (like `WRITE`) can now be entered in lower case. As a result case is not significant in all identifiers, so that `FRED`, `fred` and `Fred` are now treated as the same identifier. If you have an existing program that uses the same name in different case as two different identifiers then either change one name or use the new '`U`' option described below.

6. New compiler options:

Compiler options may now be entered in lower-case and those that previously had to appear on the command line can now appear at the start of your program so long as they appear before the word `PROGRAM`. So if you normally want to delete the old object file and have the compilation listing switched off, start your programs with:

```
{$y,l-}
PROGRAM myfile;
```

The following compiler options have been added:

U if this option is used then upper and lower case are treated as different by the compiler; otherwise upper and lower case are treated as the same in reserved words and identifiers.

X if this option is used then after a successful compilation the compiler will load and run the code you have just compiled. This only works if you have used if you have not specified different object and source file names.

Dxxx This option is followed by a four digit hexadecimal number and is used to generate programs that can run in different size TPAs whilst using all of the available memory for the heap and run-time stack. This is useful when developing programs that will be run on more than one machine or for running under CP/M plus in conjunction with RSXs such as `SUBMIT` and `GSX`. At the end of each compilation the compiler prints a message of the form:

```
Minimum value for the D option: XXXX
```

This is the value that should be used when generating production programs. Whilst programs are under-development it is easiest to embed a D option at the front of your program with a larger value.

Use of this option moves the programs global variables from the top of memory to after the end of the program code and before the heap. In general it is best to use the D option in place of the V option.

Gxxxx This option finds a run-time error in a program. This is normally used by the menu system; however it can be useful if a run-time error occurs when the compiler and editor are not on the current disc. Say the run-time program stops with the message

```
Overflow at PC=1234
```

and the editor and compiler are not available; then subsequently running the compiler with the same options as it was originally compiled with the addition of `G1234` will cause compilation to stop with the message

```
Runtime error
```

just after relevant piece of code is reached. Hitting 'E' will then load the editor and place the cursor accordingly.

7. DISPOSE and MemAvail

`DISPOSE` is now implemented as in Jensen & Wirth so that indivual dynamic variables may be deleted. `MARK` and `RELEASE` may still be used. These procedures may be mixed. However,~~it~~ a call to `RELEASE` may not remove all the variables allocated since the corresponding `MARK` if `DISPOSE` was called before `MARK`. `MemAvail` is a parameterless function which returns the free space between the top of the heap and the stack. It returns an integer and can be used to detect if a program is about to run out of memory and so take corrective action. Note that for small programs the ammount of free space may be 32K or more. This is represented as a negative number when returned from `MemAvail`. Also it may be possible to create a dynamic variable that is larger than MemAvail would suggest since there may be sufficient free space within the heap where space has been freed using `DISPOSE`.

SECTION 0 PRELIMINARIES

0.0 Introduction

HiSoft Pascal is a fast, easy-to-use and powerful version of the Pascal language as specified in the Pascal User Manual and Report (Jenson/Wirth Second Edition). The only important omission from this standard is that PROCEDUREs and FUNCTIONs may not take procedures and functions as parameters.

Many extra functions and procedures are included to reflect the changing environment in which compilers are used; among these are POKE, PEEK, CPM and ADDR together with register variables e.g RHL, RIY etc.

The compiler occupies approximately 14K of storage while the runtimes take up roughly 5K. Both are supplied on disk in one package.

0.1 Scope of this manual

This manual is *not* intended to teach you Pascal; you are referred to the excellent books given in the Bibliography if you are a newcomer to programming in Pascal.

This manual *is* a reference document, detailing the particular features of HiSoft Pascal.

Section 1 gives the syntax and the semantics expected by the compiler.

Section 2 details the various predefined identifiers that are available within HiSoft Pascal, from CONSTants to FUNCTIONs.

Section 3 contains information on the various compiler options available and also on the format of comments.

The above Sections should be read carefully by all users.

Appendix 1 details the error messages generated both by the compiler and the runtimes.

Appendix 2 lists the predefined identifiers and reserved words.

Appendix 3 gives details on the internal representation of data within HiSoft Pascal - useful for programmers who wish to get their hands dirty.

Appendix 4 gives some example Pascal programs - study this if you experience any problems in writing HiSoft Pascal programs.

0.2 A Quick Program

When writing a Pascal program for use with HiSoft Pascal you can type in your source text using a text editor, like Ed80 or a wordprocessor that produces ASCII files or you can use the built-in, interactive editor supplied with Pascal80. This latter approach will speed-up program development because of the ease with which errors can be detected and corrected. To whet your appetite, do the following.

First, make a back-up of your Pascal80 disc as described at the front of the manual and put the backed-up disc in drive A of your computer.

Make sure the editor (`HPE.COM`) is configured for your terminal by running `ED80INST.COM` as described in the INSTALLING ED80 section of this manual. Answer `Y` to the first question and then type `HPE` when prompted to `Enter Filename`. Amstrad owners need not bother to run `ED80INST.COM` since the editor is already configured for their system.

Once you have configured `HPE.COM` for your computer type:

```
HPE [ENTER]
```

from the CP/M A> prompt. A menu will appear that looks like the one on the next page ...

```
      HiSoft Pascal80 Menu Selection

 Start editing
 Compile
 Run
 eXecute
 Quit

 Edit file:

 Main file:
```

Now type `E` to Edit a file, watch the cursor move down to the `Edit file:` line and type `FACT.PAS [ENTER]`. After some disc access the editor will appear; you can now type your program in directly. Let's enter a program with a deliberate mistake. Type the following ending each line by hitting the `[ENTER]` key:

```
PROGRAM FACT (INPUT,OUTPUT);
VAR I:INTEGER;
FUNCTION FACT(N:INTEGER):REAL;
 BEGIN
  IF M>1 THEN FACT:=N*FACT(N-1) ELSE FACT:=1
 END;
BEGIN
 REPEAT
  WRITE ('FACTORIAL: ');
  READLN;READ(I);
  WRITELN('IS',FACT(I));
  WRITELN
 UNTIL I=0
END.<ENTER>
```

Now type: `<CTRL>K X <ENTER>` to save the text to disc from the editor and the menu will reappear. Press `C` to Compile your program; the compiler will be loaded and the program compiled. This will cause a compiler error because on line 5 of the program we deliberately used the variable `M` instead of `N` (look on line 5, `M>1` should be `N>1`). The compiler has spotted this, placed an arrow at the point where it realised there was an error (this is always just *after* the real cause of error) and told you that there is an `Undeclared identifier` here.

Now press E; the editor is loaded again and the cursor is placed at the point in your program where the error was spotted. To correct it in this case type:

```
[CTRL]-S [DEL] N
```

and then:

```
[CTRL]-K X [ENTER]
```

to save your program and go back to the menu. This time type `X` from the menu to compile and run the program; the compilation should be successful and the program will automatically run asking:

```
FACTORIAL:
```

Type a number to find its factorial or `0` to end and return to the menu.

Programs can be created, modified, compiled and run in this interactive mode or from within CP/M. The next two sub-sections describe these 2 ways of development.

0.3 Interactive Editing, Compiling & Running

When you type:

`HPE [ENTER]` or `HPE filename [ENTER]`

then the first thing that appears on the screen is the Pascal80 menu which looks like:

```
      HiSoft Pascal80 Menu Selection

Start editing
Compile
Run
eXecute
Quit

Edit file:

Main file:
```

From here you can type `S`, `C`, `R`, `X`, `Q`, `E` or `M` i.e. the non-highlighted letters on the menu. These have the following effects:

Edit file

Having pressed E you should now type in the name of the file that you wish to edit. You can omit the .PAS extension since this will be supplied automatically. Having typed in the filename press [ENTER]; the file will be loaded and the editor will take control. The editor behaves exactly like ED80 and you should consult the ED80 manual for details of the editing process.

Main file

Press M to specify the filename for compiling and running; filename.PAS will be compiled when you press C or X and filename.COM will be executed when you press R or X. The Edit file command will automatically set up the Main file if there was no Main file to start with, otherwise, the two filenames and the two commands (E and M) are regarded as independent so that you can be editing one file (say an include file) while compiling and running another file.

Start editing

Press S to begin editing the current Edit file; if there is no Edit file then you will be prompted to enter a filename.

Compiling

Compile the Main file. Pressing C loads the compiler (HP80.COM) into memory and compiles the file shown by Main file:. Compiler options may appear at the start of the program; see below for full details of the options available.

Compiler errors

If the compiler detects an error during compilation then it will pause, display the line in error with an up-arrow (↑) pointing at the symbol just *after* the error occurred and either print an error message or an error number. Error messages will be shown if the file HP80.ERR is on the logged-in disc which is normally the same disc as the compiler, otherwise error numbers will be displayed; refer to Appendix 1 for an explanation of these numbers. Once the display has paused like this, you may press E to return to the editor with the cursor positioned near the error, [CTRL-C] to return to CP/M or any other key to continue the compilation.

Compiler Options

There are many one-letter options that can be used to control the compilation process; some of them can be used throughout your program and these are described in Section 3 of this manual, others can only appear at the start of the program and these are described here. Options described in Section 3 may be used anywhere in the program.

A list of options must begin with a $ and must be enclosed within comment delimiters (i.e. (* *) or { }). The options that are available only at the start of your program are:

option	effect
N	- don't produce any object code.
Y	- delete any existing object code before compiling.
T	- include trig routines and EXP,LN, RANDOM and FRAC in the runtimes.
R	- do not include REAL routines in the run-times.
X	- compile & then run the object after a successful compilation.
U	- do not treat upper and lower case as equivalent.
Vnnnn	- set the runtime stack to nnnn (hexadecimal).
Gnnnn	- Goto runtime error close to runtime address nnnn.
Dnnnn	- move global variables to beneath nnnn.

The T and R options can be used to keep the runtimes fairly small if REALs or trigonometric functions are not used. If the latter are used without including them in the runtimes then an Undeclared identifier error is given. Please use the R option with care since the compiler makes no checks.

U makes the compiler accept upper and lower case letters as being different so that the identifiers Loop, LOOP, LooP are 3 separate identifiers; without the U option these 3 would be converted to one identifier LOOP by the compiler. When using `U` you must make sure that all reserved words and pre-defined identifiers are in upper case.

The `G` option allows you to go back into the editor on the source line at which a runtime error was detected. The interactive editor makes automatic use of this feature; you can use it manually by using the `G` option with the address at which the runtime error occurred immediately after the `G` in hexadecimal.

The `D` option lets you move your global variables to grow down towards the code from address `nnnn` with the heap growing above `nnnn` and the stack growing down from the top of the computer's TPA. This is useful if you wish to run the compiled code on machines with different-sized TPAs. It is also needed if you are running compiled code from within SUBMIT files under CP/M Plus.

The compiler tells you, at the end of a compilation, the minimum value that `nnnn` can take without the variables hitting the compiler by displaying the message:

```
Minimum D option value:nnnn.
```

The `V` option allocates the global variables downwards from `nnnn` with the stack growing down beneath the variables and the heap growing up from the end of the code. This is useful if you wish to place machine code or other data in the space between `nnnn` and the top of the TPA.

The following diagrams illustrate the effects of the `D` and `V` options.

Compiler Listing

The compiler generates a listing of the form:

```
xxxx nnnn text of source line
```

where `xxxx` is the address where the code generated by this line begins.
`nnnn` is the line number with leading zeroes suppressed.

Line numbers are shown relative to the start of the file being compiled so that when a file is included (with the `$F` option) then the line number is set to 1 at the beginning of the compilation of the included file.

The listing may be directed to a printer, if required, by the use of option `P+`(see Section 3).

You may pause the listing at any stage by holding the `[CONTROL]` key down and the `S` key down at the same time; use `[CONTROL]` and `C` to return to CP/M or any other key to restart the listing.

If the program terminates incorrectly (e.g. without `END.`) then the message `No more text` will be displayed and control returned to the menu.

If no errors are detected a `COM` file is generated (unless you use option `N`) ready for direct execution from CP/M or from within the menu. The `COM` file contains the object code produced by the compilation and, automatically, the runtime support routines. Remember that you can cut down the size of these runtime routines by using options in the command line - see above.

Run

Typing `R` from the menu display will run the `COM` file corresponding to the filename shown as the Main file on the menu. If no such `COM` file exists then the menu will reappear immediately. After the execution is complete control returns to the menu through the prompt:

```
Hit any key for menu
```

During a run of the object code various runtime error messages may be generated (see Appendix 1); you can return to the source file at the position of the runtime error by pressing `E` or `e` when the error is reported. Otherwise, pressing `[CTRL]-C` will return you to CP/M while hitting any other key will return you to the menu.

You may suspend a run while it is outputting to the screen or printer by using `[CONTROL]` and `S`; subsequently use `[CONTROL]` and `C` to abort the run and return to CP/M or any other key to resume the run. If you have keyboard checks on (compiler option `$C+`) then you can abort the run by typing `[CONTROL]-C` at any time; this produces a `Halt` message, press any key to return to the menu or CP/M.

eXecute

The `X` command from the menu simply compiles the current `Main file` and, if there were no compilation errors, runs it immediately.

Quit

The `Q` command quits to CP/M.

0.4 Non-interactive Edit / Compile / Run

There may be occasions when you don't want to use the interactive editor when compiling a program. For example, you may want the object program to have a different filename from the source program or perhaps you are compiling many programs one after the other and don't want to load the interactive editor after every compilation. To help you with this we supply an alternative, non-interactive, version of the editor on your master disc, it is called `ED80.COM`. You should configure this for your terminal as described in the section **INSTALLING ED80** (it comes pre-installed on Amstrad computers). The development cycle using this editor is described below.

Editing

To create or change a program simply type:

```
ED80 filename.PAS [ENTER]
```

You will now be able to type in new lines, amend existing text etc. as described in the ED80 manual within this binder. Remember that you can change the command keys by using the `ED80INST` program; see **INSTALLING ED80**.

When you have finished editing leave the editor, initially the command to do this is set to `[CTRL]-K X`, and you will be back in CP/M with the familiar `A>` prompt. You can now compile your program.

Compiling

To compile a program from within CP/M type:

```
HP <objectfile> sourcefile <;option<,option...> > [ENTER]
```

The items enclosed in < > are optional, *do not type the < and >*!

`objectfile` is the name that you want the object code to have, an extension of `.COM` is assumed.

`sourcefile` is the name of the Pascal program that you want to compile, an extension of `.PAS` is assumed. This filename must be present, if it isn't the compiler will tell you what the usage is and return to CP/M.

`option` is any of the compiler options described under **Compiler Options** in the interactive section above, both the options that can only appear at the start of your program and the options that may appear anywhere in the program are allowed here, separated by commas.

So, some examples of compiling non-interactively are:

```
HP TEST PRIMES;L-,R [ENTER]
```

creates the file `TEST.COM` by compiling `PRIMES.PAS` with no compiler listing (`L-`) and not including the REAL routines in the runtimes (`R`).

```
HP PRIMES;L-,R,Y,O-,C-,S-,A- [ENTER]
```

creates the file `PRIMES.COM` by compiling `PRIMES.PAS` with all runtime checks off (`O-, C-, S-, A-`), not including the REALs in the runtimes (`R`), automatically deleting any pre-existing object code (`Y`) and with no compiler listing (`L-`). You might use these options for compiling a benchmark program.

`HP DIS;P+,N [ENTER]`

compiler `DIS.PAS`, with a compiler listing to the printer (`P+`) but not producing any object code (`N`).

When compiling non-interactively everything happens just as described in the interactive section above; you can get a **Compiler Listing**, which may be paused, and **Compiler Errors** will be detected and displayed as detailed in the previous section. In fact, if you have the interactive editor (`HPE.COM`) present on your disc and you press `E` or `e` after an error has been detected, then you will enter interactive mode and continue in it until you Quit. However, if `HPE.COM` is not on your disc, then pressing `E` or `e` after an error will return you straight to CP/M as will `[CTRL]-C`. Hitting any other key will continue the compilation. Remember to have `HP80.ERR` on your disc if you want error messages rather than error numbers.

Running

If a compilation ends successfully with no errors then the compiler will have created you object code on the disc under the name you asked for or, by default, `sourcefile.COM`. You can run this from CP/M simply by typing the name of the file. You can pause runtime output as described in **Run** above and you may *goto runtime error* by pressing `E` or `e` after an error has been detected, *but only if* you have `HPE.COM` on your disc and then you will be in interactive mode again.

HP80.COM

You will notice that, in addition to `HPE.COM` and `HP.COM`, you also have `HP80.COM` on your disc. This is supplied for people that want extra fast compilation and minimum run-time overheads. HP80 is compatible with previous versions of **Pascal80** and it does not support the new features like variant records, files of any type etc. In particular, the compiler options `U`, `G`, `X` and `D` are not supported, reserved words must be in upper case and procedures are not automatically dumped to disc as they are compiled; this gives a quicker compilation. Also the runtimes are more compact because they do not include code for the new features present in `HP.COM`.

In general, we do not recommend the use of HP80 and it is supplied mainly for the convenience of existing users of **Pascal80**.

SECTION 0 PRELIMINARIES.

0.0 Introduction.

HiSoft Pascal is a fast, easy-to-use and powerful version of the Pascal language as specified in the Pascal User Manual and Report (Jensen/ Wirth Second Edition). Omissions from this specification are as follows:

Only FILEs of CHAR are allowed.
A RECORD type may not have a VARIANT part.
PROCEDUREs and FUNCTIONs are not valid as parameters.

Some extra functions and procedures are included to reflect the changing environment in which compilers are used; among these are POKE, PEEK, CPM and ADDR.

The compiler occupies approximately 12K of storage while the runtimes take up roughly 4K. Both are supplied on disk in one package.

0.1 Scope of this manual.

This manual is not intended to teach you Pascal; you are referred to the excellent books given in the Bibliography if you are a newcomer to programming in Pascal.

This manual is a reference document, detailing the particular features of HiSoft Pascal.

Section 1 gives the syntax and the semantics expected by the compiler.

Section 2 details the various predefined identifiers that are available within HiSoft Pascal, from CONSTants to FUNCTIONs.

Section 3 contains information on the various compiler options available and also on the format of comments.

The above Sections should be read carefully by all users.

Appendix 1 details the error messages generated both by the compiler and the runtimes.

Appendix 2 lists the predefined identifiers and reserved words.

Appendix 3 gives details on the internal representation of data within HiSoft Pascal – useful for programmers who wish to get their hands dirty.

Appendix 4 gives some example Pascal programs - study this if you experience any problems in writing HiSoft Pascal programs.

0.2 Compiling and Running.

When writing a Pascal program for use with HiSoft Pascal you can type in your source text using a text editor, like ED80 or a wordprocessor that produces ASCII files.

For example type the following:

```
ED80 FACT.PAS <ENTER>

PROGRAM FACT;
VAR I:INTEGER;
  FUNCTION FACT(N:INTEGER):REAL;
  BEGIN
    IF N>1 THEN FACT:=N*FACT(N-1) ELSE FACT:=1
  END;
BEGIN
 REPEAT
   WRITE('FACTORIAL:');
   READLN; READ(I);
    WRITELN('IS',FACT(I));
   WRITELN
 UNTIL FALSE
END.<ENTER>
```

Now type: <CTRL>K X <ENTER> to save the text to disc from ED80 and then, when you are back in CP/M:

```
HP80 FACT<ENTER>
```

The source code will be compiled and assuming no errors are detected an object file named **FACT.COM** will be created. This can be run by simply typing **FACT** to the CP/M prompt. If there are any errors you should hit 'E' to exit the compiler. Then use ED80 to edit the program and correct the typing mistake.
When this program is run it will prompt you to enter numbers and give the factorial of each number input until you type <CTRL/C> which will return you to CP/M.
In the example above we used the simplest form of command line.

The full form looks like this:

```
HP80 <file1> file2 <;option<,option,....>>
```

where:

file2	is the source file which must have extension of PAS. If this file does not exist then an error message is given.
file1	is an optionally different obect file (with extension COM).
option	is an ordinary compiler option (see Section 3.2) or one of the following:

N	– don't produce any object code.
Y	– delete any existing object code.
T	– include trig routines and EXP,LN and FRAC in the runtimes.
R	– do not include REAL routines in the run-times.
Vnnnn	– set the runtime stack to nnnn (hexadecimal).

The textfile should be standard CP/M ASCII text. That is lines are terminated by CR,LF and the file is terminated with a CTRL/Z. If you have a wordprocessor that normally uses the top bit of characters you should make sure that this feature is disabled.

The T and R options can used to keep the runtimes fairly small if REALs or trigonometric functions are not used. If the latter are used without including them in the runtimes then an '*ERROR* 3' is given. Please use the R option with care since the compiler makes no checks.

Options within the source program override those specified in the command line. The N, Y, T, R, and V options are not available within the program.

Some example command lines are given below:

```
HP80 HILBERT;P+,N
```

get a compilation listing (to printer) of a program.

```
HP80 BYTE;R,Y,O–,C–,S–,A–
```

turn off all checks; say for a benchmark.

```
HP80 OBJ1 SCE1;R,L–
```

compile the program SCE1.PAS to produce the file OBJ1.COM with no compiler listing (to make compilation faster) and not including REAL routines with the object file.

The compiler generates a listing of the form:

xxxx nnnn text of source line

where: xxxx is the address where the code generated by this line begins.
nnnn is the line number with leading zeroes suppressed.

If a line contains more than 80 characters then the compiler inserts new-line characters so that the length of a line is never more than 80 characters.

The listing may be directed to a printer, if required, by the use of option P+ (see Section 3).

You may pause the listing at any stage by holding the CONTROL key down and the 'S' key down at the same time; use CONTROL and 'C' to return to CP/M or any other key to restart the listing.

If an error is detected during the compilation then the message '*ERROR*' will be displayed, followed by an up-arrow ('↑'), which points after the symbol which generated the error, and an error number (see Appendix 1). The listing will pause; hit 'E' to return to CP/M tidily (deleting the object file), CONTROL and 'C' to abort, or any other key to continue the compilation.

If the program terminates incorrectly (e.g. without 'END.') then the message 'No more text' will be displayed and control returned to CP/M.

If a compilation contains any errors then the number of errors detected will be given and any object code produced will be deleted. Otherwise a COM file will be generated (unless you use option N) ready for direct execution from CP/M. The COM file contains the object code produced by the compilation and, automatically, the runtime support routines. Remember that you can cut down the size of these runtime routines by using options in the command line – see above.

During a run of the object code various runtime error messages may be generated (see Appendix 1). You may suspend a run while it is out-putting to the screen or printer by using CONTROL and 'S'; subsequently use CONTROL and 'C' to abort the run and return to CP/M or any other key to resume the run.

0.3 Strong TYPEing.

Different languages have different ways of ensuring that the user does not use an element of data in a manner which is inconsistent with its definition.

At one end of the scale there is machine code where no checks whatever are made on the TYPE of variable being referenced. Next we have a language like the Byte 'Tiny Pascal' in which character, integer and Boolean data may be freely mixed without generating errors. Further up the scale comes BASIC which distinguishes between integers and strings and, sometimes, between integers and reals (perhaps using the '%' sign to denote integers). Then comes Pascal which goes as far as allowing distinct user-enumerated types. At the top of the scale (at present) is a language like ADA in which one can define different, incompatible numeric types.

There are basically two approaches used by Pascal implementations to strength of typing; structural equivalence or name equivalence. HiSoft Pascal uses name equivalence for RECORDs and ARRAYs. The consequences of this are clarified in Section 1 – let it suffice to give an example here; say two variables are defined as follows:

```
VAR A : ARRAY['A'..'C'] OF INTEGER;
    B : ARRAY['A'..'C'] OF INTEGER;
```

then one might be tempted to think that one could write A:=B; but this would generate an error (*ERROR* 10) under HiSoft Pascal since two separate 'TYPE records' have been created by the above definitions. In other words, the user has not taken the decision that A and B should represent the same type of data. She/He could do this by:

```
VAR A,B : ARRAY['A'..'C'] OF INTEGER;
```

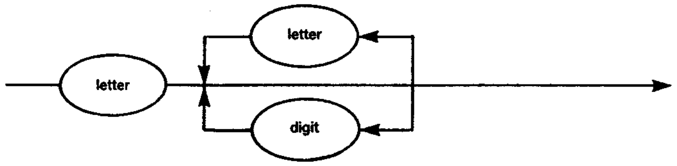
and now the user can freely assign A to B and vice versa since only one 'TYPE record' has been created.

Although on the surface this name equivalence approach may seem a little complicated, in general it leads to fewer programming errors since it requires more initial thought from the programmer.

SECTION 1 SYNTAX AND SEMANTICS.

This section details the syntax and the semantics of HiSoft Pascal – unless otherwise stated the implementation is as specified in the Pascal User Manual and Report Second Edition (Jensen/Wirth).

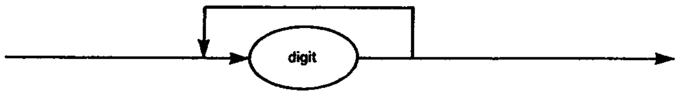
1.1 IDENTIFIER.



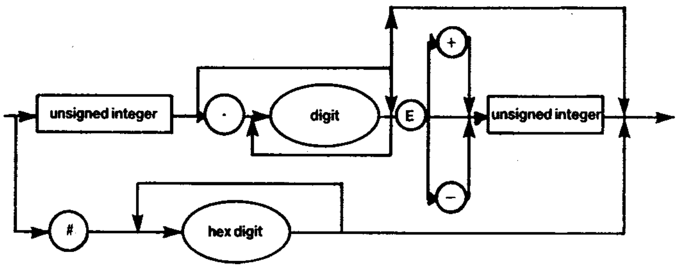
Only the first 8 characters of an identifier are treated as significant. These first 8 characters must not constitute a reserved word (see Appendix 2).

Identifiers may contain lower or upper case letters. Lower case is not converted to upper case so that the identifiers HELLO, HELlo and hello are all different. Reserved words and predefined identifiers may only be entered in upper case.

1.2 UNSIGNED INTEGER.



1.3 UNSIGNED NUMBER.



Integers have an absolute value less than or equal to 32767 in HiSoft Pascal. Larger whole numbers are treated as reals.

The mantissa of reals is 23 bits in length. The accuracy attained using reals is therefore about 7 significant figures. Note that accuracy is lost if the result of a calculation is much less than the absolute values of its arguments e.g. 2.00002 – 2 does not yield 0.00002. This is due to the inaccuracy involved in representing decimal fractions as binary fractions. It does not occur when integers of moderate size are represented as reals e.g. 200002 – 200000 = 2 exactly.

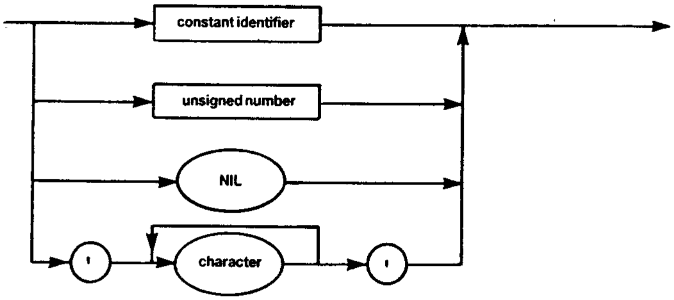
The largest real available is 3.4E38 while thesmallest is 5.9E–39.

There is no point in using more than 7 digits in the mantissa when specifying reals since extra digits are ignored except for their place value.

When accuracy is important avoid leading zeroes since these count as one of the digits. Thus 0.000123456 is represented less accurately than 1.23456E–4.

Hexadecimal numbers are available for programmers to specify memory addresses for assembly language linkage inter alia. Note that there must be at least one hexadecimal digit present after the '#', otherwise an error (*ERROR* 51) will be generated.

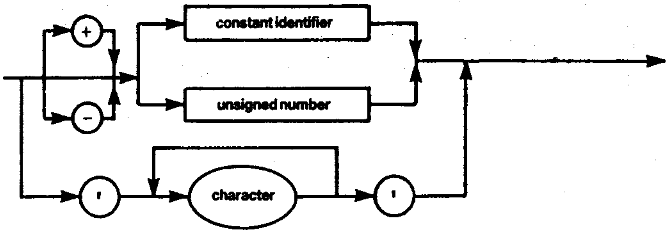
1.4 UNSIGNED CONSTANT.



Note that strings may not contain more than 255 characters. String types are ARRAY [1..N] OF CHAR where N is an integer between 1 and 255 inclusive. Literal strings should not contain end-of-line characters (CHR(13)) – if they do then an '*ERROR* 68' is generated.

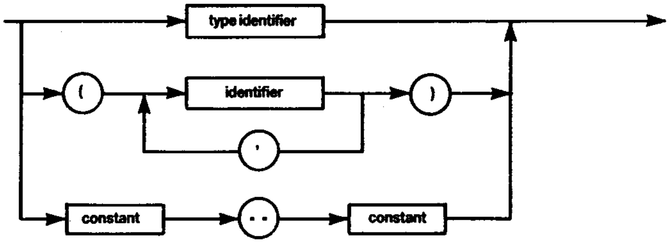
The characters available are the full expanded set of ASCII values with 256 elements. To maintain compatibility with Standard Pascal the null character is not represented as '' ; instead CHR(0) should be used.

1.5 CONSTANT.



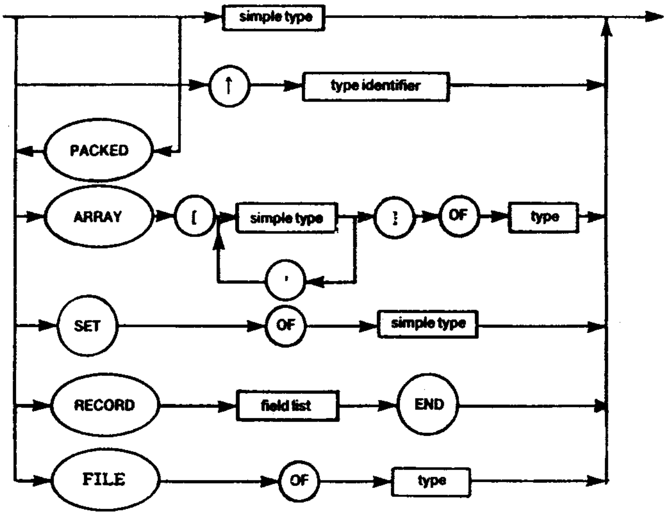
The comments made in Section 1.4 concerning strings apply here.

1.6 SIMPLE TYPE.



Scalar enumerated types (identifier, identifier, ...) may not have more than 256 elements.

1.7 TYPE.



The reserved word PACKED is accepted but ignored since packing already takes place for arrays of characters etc. The only case in which the packing of arrays would be advantageous is with an array of Booleans – but this ismore naturally expressed as a set when packing is required.

1.7.1 ARRAYs and SETs.

The base type of a set may have up to 256 elements. This enables SETs of CHAR to be declared together with SETs of any user enumerated type. Note, however, that only subranges of integers can be used as base types. All subsets of integers are treated as sets of 0..255.

Full arrays of arrays, arrays of sets, records of sets etc. are supported.

Two ARRAY types are only treated as equivalent if their definition stems from the same use of the reserved word ARRAY. Thus the following types are not equivalent:

```
TYPE
              tablea = ARRAY[1..100] OF INTEGER;
              tableb = ARRAY[1..100] OF INTEGER;
```

So a variable of type tablea may not be assigned to a variable of type tableb. This enables mistakes to be detected such as assigning two tables representing different data. The above restriction does not hold for the special case of arrays of a string type, since arrays of this type are always used to represent similar data.

1.7.2 Pointers.

HiSoft Pascal allows the creation of dynamic variables through the use of the Standard Procedure NEW (see Section 2). A dynamic variable, unlike a static variable which has memory space allocated for it throughout the block in which it is declared, cannot be referenced directly through an identifier since it does not have an identifier; instead a pointer variable is used. This pointer variable, which is a static variable, contains the address of the dynamic variable and the dynamic variable itself is accessed by including a '↑' after the pointer variable. Examples of the use of pointer types can be studied in Appendix 4.

There are some restrictions on the use of pointers within HiSoft Pascal. These are as follows:
Pointers to types that have not been declared are not allowed. This does not prevent the construction of linked list structures since type definitions may contain pointers to themselves e.g.

```
TYPE
        item = RECORD
                    value : INTEGER;
                    next : ↑item
                 END;
        link = ↑item;
```

Pointers to pointers and pointers to files are both not allowed.

Pointers to the same type are regarded as equivalent e.g.

```
VAR
        first : link;
        current : ↑item;
```

The variables first and current are equivalent (i.e. structural equivalence is used) and may be assigned to each other or compared.

The predefined constant NIL is supported and when this is assigned to a pointer variable then the pointer variable is deemed to contain no address.

1.7.3 FILEs.

HiSoft Pascal supports FILEs OF CHAR. Files are sequential and are thought to be made up of lines, separated by a line separator; the lines are physically separated by the codes CRLF (i.e. CHR(13), CHR(10)) but only the CR is treated as a line separator, the LF is ignored and skipped. The end of a file is detected by the presence of a CTRL/Z (CHR(26)) character.

The buffer variable construct is supported – see Section 2 for more details.

The Standard Type TEXT is defined as FILE OF CHAR so that the declarations

```
        file1 : FILE OF CHAR;
        file2 : TEXT;
```

are equivalent.

Two files are predefined viz. INPUT and OUTPUT. These are textfiles which represent the standard I/O media of the computer i.e. the keyboard (via CP/M routine 10) and the CRT (via CP/M routine 2). They are the default values in any textfile operations e.g. WRITE(value); is equivalent to WRITE(OUTPUT,value);. Note that INPUT is considered to begin with a blank line – see Section 2.

The procedures RESET and REWRITE are provided to open files while the procedures GET and PUT allow primitive operations on files. More details of these procedures and file handling in general may be found in Section 2. An example of the use of files may be found in Appendix 4.

Restrictions on the use of files follow:

Files may not be components of structured types.

Files cannot be declared as local variables, only global.

Pointers to files are not allowed.

Files may be used as variable parameters but not as value parameters – this is Standard Pascal.

1.7.4 RECORDs.

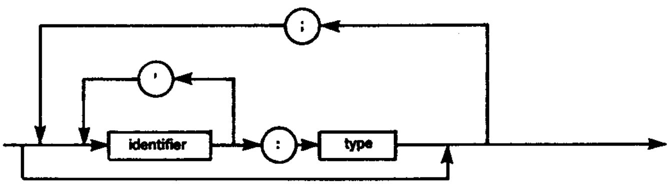
The implementation of RECORDs, structured variables composed of a fixed number of constituents called fields, within HiSoft Pascal is as Standard Pascal except that the variant part of the field list is not supported.

Two RECORD types are only treated as equivalent if their declaration stems from the same occurrence of the reserved word RECORD see Section 1.7.1 above.

The WITH statement may be used to access the different fields within a record in a more compact form.

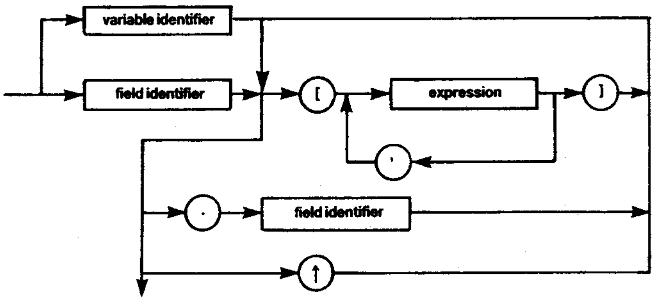
See Appendix 4 for an example of the use of WITH and RECORDs in general.

1.8 FIELD LIST.



Used in conjunction with RECORDs see Section 1.7.4 above and Appendix 4 for an example.

1.9 VARIABLE.



Two kinds of variables are supported within HiSoft Pascal; static and dynamic variables. Static variables are explicitly declared through VAR and memory is allocated for them during the entire execution of the block in which they were declared.

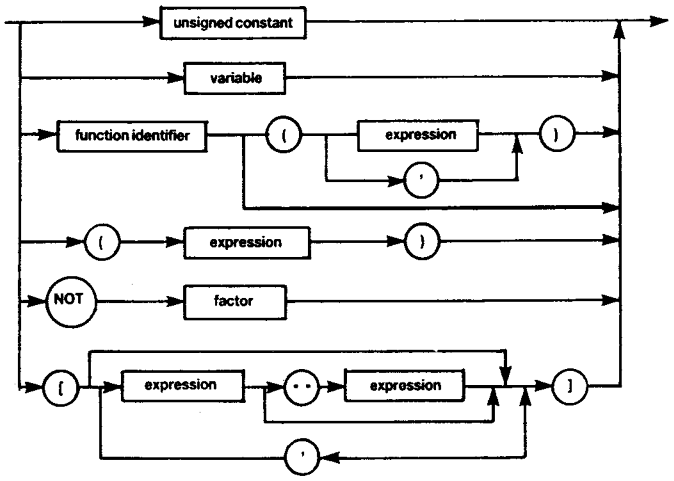
Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type pointer, which contains the address of the dynamic variable.

See Section 1.7.2 and Section 2 for more details of the use of dynamic variables and Appendix 4 for an example.

When specifying elements of multi-dimensional arrays the programmer is not forced to use the same form of index specification in the reference as was used in the declaration.

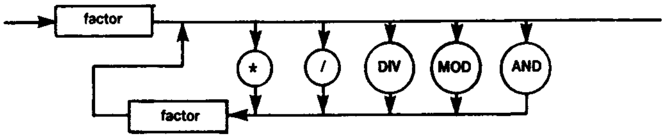
e.g. if variable a is declared as ARRAY[1..10] OF ARRAY[1..10] OF INTEGER then either a[1][1] or a[1,1] may be used to access element (1,1) of the array.

1.10 FACTOR.



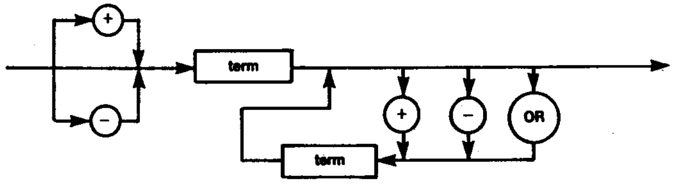
See EXPRESSION in Section 1.13 and FUNCTIONs in Section 3 for more details.

1.11 TERM.



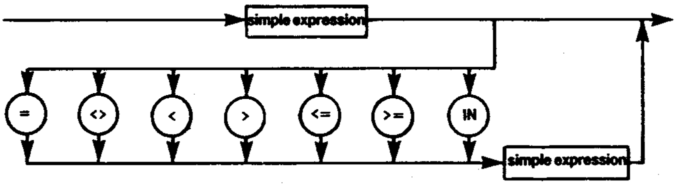
The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements – one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

1.12 SIMPLE EXPRESSION.



The same comments made in Section 1.11 concerning sets apply to simple expressions.

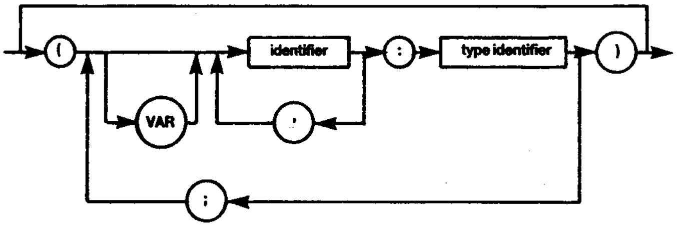
1.13 EXPRESSION.



When using IN, the set attributes are the full range of the type of the simple expression with the exception of integer arguments for which the attributes are taken as if [0..255] had been encountered.

The above syntax applies when comparing strings of the same length, pointers and all scalar types. Sets may be compared using >=, <=, < > or =. Pointers may only be compared using = and < >.

1.14 PARAMETER LIST.



A type identifier must be used following the colon – otherwise *ERROR* 44 will result.

Variable parameters as well as value parameters are fully supported.

Procedures and functions are not valid as parameters.

Files can only be designated variable parameters not value parameters.

1.15 STATEMENT

Refer to the syntax diagram on Page 14.

Assignments:

See Section 1.7 for information on which assignment statements are illegal.

CASE statements:

An entirely null case list is not allowed. i.e. `CASE OF END;` will generate an error (*ERROR * 13) and you should **not** have a semicolon before the `END` of the `CASE` statement.

The `ELSE` clause, which is an alternative to `END`, is executed if the selector ('expression' overleaf) is not found in one of the case lists ('constant' overleaf).

If the `END` terminator is used and the selector is not found then control is passed to the statement following the `END`.

FOR statements:

Control variables must either be defined in the same procedure/function that they are used or in the outer level of the program. This is half way between the ISO and Jensen and Wirth definitions.

GOTO statements:

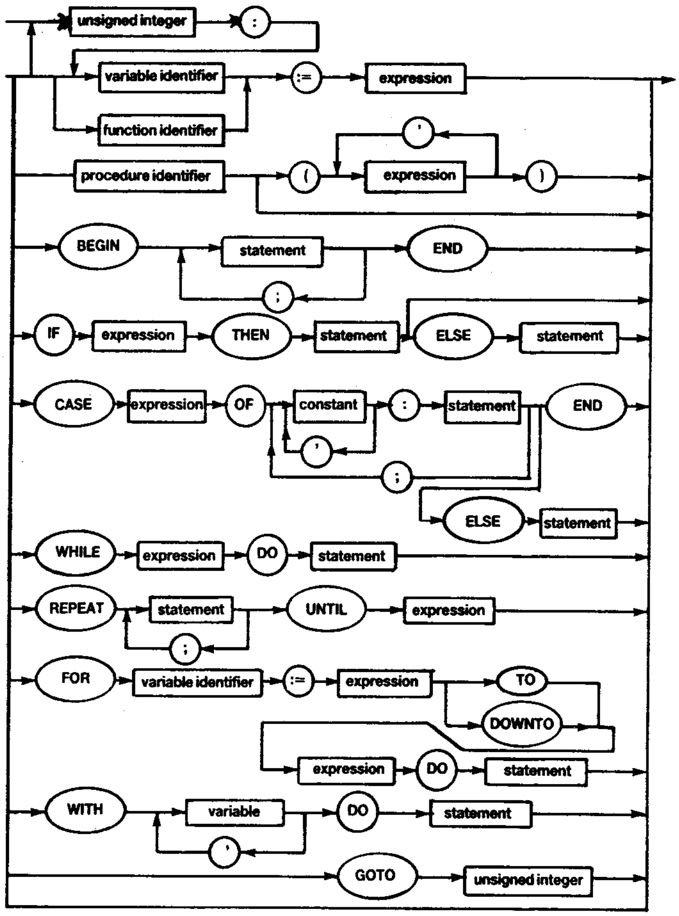
It is only possible to `GOTO` a label which is present in the same block as the `GOTO` statement and at the same level. You must **not** jump out of `FOR` loops.

Labels must be declared (using the reserved word `LABEL`) in the block in which they are used; a label consists of a least one and up to four digits. When a label ia used to mark a statement it must appear at the beginning of the statement and be followed by a colon ':'.

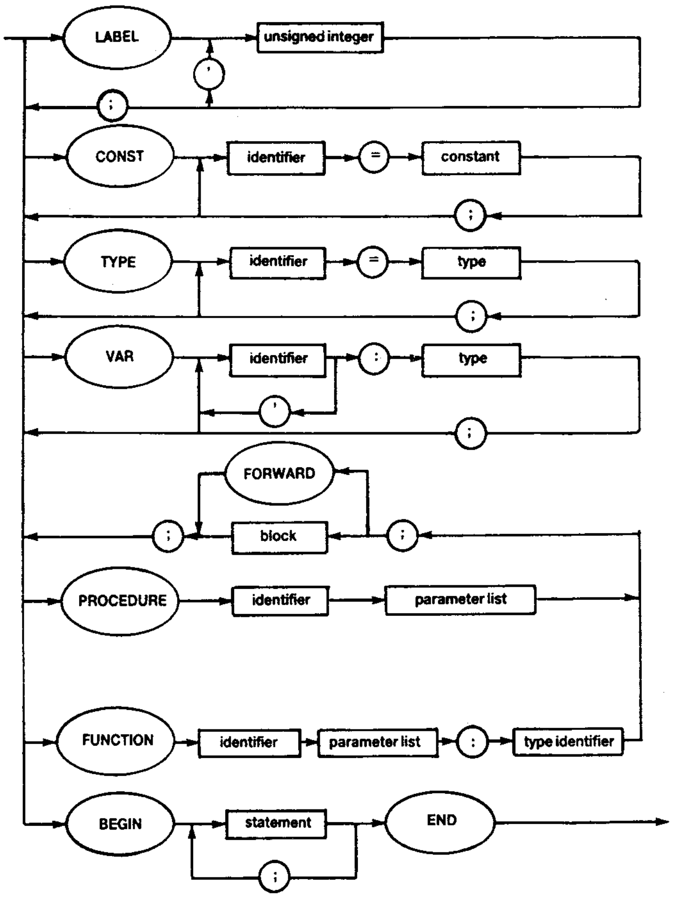
WITH statements:

`WITH` statements must not be used recursively.

STATEMENT.


unsigned integer
:
variable identifier
:=
expression
function identifier
procedure identifier
(
,
expression
)
BEGIN
statement
;
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
,
:
statement
;
END
ELSE
statement
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement
WITH
variable
,
DO
statement
GOTO
unsigned integer

1.16 BLOCK



Note that, when a file variable is declared, then it may be followed, optionally, by a constant with a value between 1 and 255 inclusive enclosed in square brackets. This constant specifies the buffer size to be used for this file, in 128 character units. For example if you require the file file1 to have a buffer size of 2K (2048 characters) then the declaration should look like:

```
VAR       file1 : FILE OF CHAR [16];
```

or

```
CONST   filesize = 16;
VAR       file1 : TEXT[filesize];
```

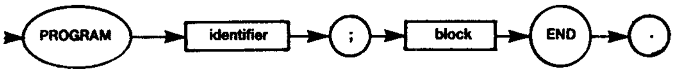
Forward References.

As in the Pascal User Manual and Report (Section 11.C.1) procedures and functions may be referenced before they declared through use of the Reserved Word FORWARD e.g.

```
PROCEDURE a(y:t);
FORWARD;           (* procedure a declared to be *)
PROCEDURE b(x:t);  (* forward of this statement *)
  BEGIN
  ....
  a(p);            (* prodedure a referenced. *)
  ....
  END;
PROCEDURE a;       (* actual declaration of procedure a. *)
  BEGIN
  ....
  b(q);
  ....
  END;
```

Note that the parameters and result type of the procedure a are declared along with FORWARD and are not repeated in the main declaration of the procedure. Remember, FORWARD is a Reserved Word.

1.17 PROGRAM.



Note that the 'microcomputer standard' form is used i.e. there are no formal parameters of the program and any file operations, on default of the file name, refer to the predefined file identifiers INPUT and OUTPUT depending on the type of file operation. See Section 1.7.3 and Section 2.

SECTION 2 PREDEFINED IDENTIFIERS.

2.1 CONSTANTS.

MAXINT	The largest integer available i.e. 32767.
TRUE, FALSE	The constants of type Boolean.

2.2 TYPES.

INTEGER	See Section 1.3.
REAL	See Section 1.3.
CHAR	The full extended ASCII character set of 256 elements.
BOOLEAN	(TRUE,FALSE). This type is used in logical operations including the results of comparisons.
TEXT	This equivalent to the type FILE OF CHAR;.

2.3 VARIABLES.

INPUT, OUTPUT.

The default file identifiers used in all file operations. INPUT is set to read data through CP/M routine 10 – read console buffer.

OUTPUT is set to write data through CP/M routine 2 – normally the CRT.

2.4 PROCEDURES AND FUNCTIONS.

2.4.1 File Handling Procedures.

2.4.1.1 Preamble – the Buffer Variable f↑.

Whenever a textfile f is declared then a variable f↑, the buffer variable, is also created. f↑, of type CHAR, can be thought of as a window through which we can access a component of the file – all data transfers to or from the file occur through f↑. The buffer variable may be assigned or appear in an expression provided type compatibility is maintained.

The standard procedures GET and PUT are provided to handle the buffer variable on a primitive level while the procedures READ(LN) and WRITE(LN) allow more sophisticated input and output such as conversion of numbers to strings etc.

Before data is transferred between the file and the buffer variable, in either direction, the file must have been opened through the use of RESET (for reading) or REWRITE (for writing). These procedures provide the only method of opening a file.

The buffer size is, by default, 128 characters; this may be increased, in units of 128 characters, by enclosing a constant (giving the number of 128 character units) in square brackets after the declaration of the file f – see Section 1.16.

2.4.1.2 PUT(f)

PUT(f) appends the value of f↑ to the file f provided that EOF is TRUE. After completion of the operation EOF remains TRUE and the buffer variable f↑ becomes undefined. Note, therefore, that it is possible only to append a value to a textfile within HiSoft Pascal files must be written sequentially.

Before PUT is used on a file, the file must have been opened for writing by the use of REWRITE – the exception to this is if the file is the default file OUTPUT which is considered already open on entry to the program; a REWRITE must not be issued on the file OUTPUT.

2.4.1.3 REWRITE(f,fn)

REWRITE is used to open a textfile for writing. f specifies the file variable that is to be used within the program – this should have been declared in the outer block e.g. VAR file1 : FILE OF CHAR;
remember that file variables may only be global. fn is ARRAY[1..14] OF CHAR and denotes the actual CP/M file specification under which the file is to occur on the diskette. Note that fn must contain 14 characters i.e. 1 character for the drive name, then a colon, then 8 characters for the filename, then a period and finally 3 characters for the file extension; the drive name and colon may be replaced by 2 spaces in which case the default drive name is used. Examples:

```
REWRITE(file1,'A:TESTFILE.DAT');
```
creates ('makes') the file TESTFILE.DAT on drive A of the disk system.

```
REWRITE(file2,'  LETTER  .TXT');
```
creates the file LETTER.TXT on the default drive.

REWRITE causes any existing file fn to be deleted from the diskette's directory – EOF(f) becomes TRUE and the buffer variable f↑ is undefined.

2.4.1.4 WRITE

The procedure WRITE is used to append structured data to a textfile. The default textfile is the predefined file OUTPUT which is normally the standard output device of the computer. When the variable e to be written is simply of type character then WRITE(f,e) is exactly equivalent to: BEGIN f↑ := e; PUT(f) END;.

Generally though, if f is a textfile:

WRITE({f,}P1,P2,........Pn); is equivalent to:

BEGIN WRITE({f,}P1); WRITE({f,}P2); ....; WRITE({f,}Pn);

The write parameters P1,P2,......Pn can have one of the following forms:

<e> or <e:m> or <e:m:n> or <e:m:H>

where e, m and n are expressions and H is a literal constant.

We have 5 cases to examine:

1] e is of type integer: and either <e> or <e:m> is used.

The value of the integer expression e is converted to a character string with a trailing space. The length of the string can be increased (with leading spaces) by the use of m which specifies the total number of characters to be output. If m is not sufficient for e to be written or m is not present then e is written out in full, with a trailing space, and m is ignored. Note that, if m is specified to be the length of e without the trailing space then no trailing space will be output.

2] e is of type integer and the form <e:m:H> is used.

In this case e is output in hexadecimal. If m=1 or m=2 then the value (e MOD 16↑m) is output in a width of exactly m characters. If m=3 or m=4 then the full value of e is output in hexadecimal in a width of 4 characters. If m>4 then leading spaces are inserted before the full hexadecimal value of e as necessary. Leading zeroes will be inserted where applicable. Examples:

```
WRITE(1025:m:H);

m=1  outputs: 1
m=2  outputs: 01
m=3  outputs: 0401
m=4  outputs: 0401
m=5  outputs:  0401
```

3] e is of type real. The forms <e>, <e:m> or <e:m:n> may be used.

The value of e is converted to a character string representing a real number. The format of the representation is determined by n.

If n is not present then the number is output in scientific notation, with a mantissa and an exponent. If the number is negative then a minus sign is output prior to the mantissa, otherwise a space is output. The number is always output to at least one decimal place up to a maximum of 5 decimal places and the exponent is always signed (either with a plus or minus sign). This means that the minimum width of the scientific representation is 8 characters; if the field width m is less than 8 then the full width of 12 characters will always be output. If m>=8 then one or more decimal places will be output up to a maximum of 5 decimal places (m=12). For m>12 leading spaces are inserted before the number. Examples:

```
WRITE(-1.23E 10:m);
```

```
m=7   gives: -1.23000E+10
m=8   gives: -1.2E+10
m=9   gives: -1.23E+10
m=10  gives: -1.230E+10
m=11  gives: -1.2300E+10
m=12  gives: -.23000E+10
m=13  gives:   -1.23000E+10
```

If the form <e:m:n> is used then a fixed-point representation of the number e will be written with n specifying the number of decimal places to be output. No leading spaces will be output unless the field width m is sufficiently large. If n is zero then e is output as an integer. If e is too large to be output in the specified field width then it is output in scientific format with a field width of m (see above). Examples:

```
WRITE(1E2:6:2)        gives:    100.00
WRITE(1E2:8:2)        gives:      100.00
WRITE(23.455:6:1)     gives:      23.5
WRITE(23.455:4:2)     gives:    2.34550E+01
WRITE(23.455:4:0)     gives:      23
```

4] e is of type character or type string.

(If e is of type character then `WRITE(f,e);` is exactly equivalent to `BEGIN f↑ := e; PUT(f) END;`.)

Either <e> or <e:m> may be used and the character or string of characters will be output in a minimum field width of 1 (for characters) or the length of the string (for string types). Leading spaces are inserted if m is sufficiently large.

5] e is of type Boolean.

Either <e> or <e:m> may be used and 'TRUE' or 'FALSE' will be output depending on the Boolean value of e , using a minimum field width of 4 or 5 respectively.

2.4.1.5 WRITELN

WRITELN{(f)} appends an end of line marker to the textfile f.

WRITELN({f,}P1,P2,........P3); is equivalent to:

```
   BEGIN WRITE({f,}P1,P2,.......P3); WRITELN{(f)} END;
```

The default file identifier f is the file OUTPUT. Remember that, for any WRITE or WRITELN on a textfile (other than OUTPUT) to be successful, then the file must first have been opened for writing through the use of REWRITE(f).

2.4.1.6 PAGE{(f)}

The procedure PAGE{(f)} causes an ASCII form feed character (#0C) to be written to the file f. If f is absent then the file OUTPUT is assumed; this will normally cause the video screen to be cleared.

2.4.1.7 GET(f)

GET(f) advances the file window for textfile f by one position and then transfers one element of data from the textfile f to the associated buffer variable f↑.

If EOF(f) is TRUE after the window has been advanced then the value of f↑ after the GET(f) is CHR(26) and EOF remains TRUE.

Before GET(f) is used on a file the file must have been opened for reading through the use of RESET(f); the only exception is if the file is the default file INPUT which is considered already open on entry to the program – a RESET should not be issued on the file INPUT.

2.4.1.8 RESET(f,fn)

RESET(f,fn) is used to open a file for reading – the file identifier f, which should have been declared as a global variable, is associated with the CP/M diskfile fn which must already exist. See Section 2.4.1.3 for details of the syntax of fn.

RESET(f,fn) opens the file f for reading and, if the file is not empty the first component of the textfile is assigned to the buffer variable f↑ and EOF(f) becomes FALSE. If the file f is empty before the RESET(f) is issued then f↑ is left undefined and EOF(f) becomes TRUE.

If the file is already open when RESET(f) is issued then the file is first closed before being opened for read operations.

RESET must not be used on the default file INPUT – this file is already open on entry to the program.

2.4.1.9 READ

The procedure READ is used to access data from textfiles. If the variable V to be read from the textfile f is simply of type character then READ(f,V); is exactly equivalent to: BEGIN V := f↑; GET(f) END;. In general though:

READ({f,}V1,V2,........Vn); is equivalent to:

BEGIN READ({f,}V1); READ({f,}V2); ....; READ({f,}Vn) END;

where V1, V2 etc. may be of type character, string, integer or real.

The default value of f is INPUT which is normally assigned to the system's keyboard.

The statement READ({f,}V); has different effects depending on the type of V. There are 4 cases to consider:

1] V is of type character.

In this case READ({f,}V); is exactly equivalent to: BEGIN V := f↑; GET(f) END; and a character is effectively read from the buffer and assigned to V. If the text window on the file is positioned on a line marker (a CHR(13) character) then the function EOLN(f) will return the value TRUE and the buffer variable f↑ will contain the value CHR(13). When a read operation is subsequently performed on the file the file window will be positioned at the start of a new line.

Important note: the default file INPUT initially contains a blank line so that EOLN is TRUE at the beginning of the program. This means that if the first read on the file INPUT is of type character then a CHR(13) value will be returned followed by the reading in of a new line from the keyboard; a subsequent read of type character will return the first character from this new line, assuming it is not blank. See also the procedure READLN below.

2] V is of type string.

A string of characters may be read using READ and in this case a series of characters will be read from the file until the number of characters defined by the string has been read or (EOLN) OR (EOF) = TRUE. If the string is not filled by the read (i.e. if end-of-line or end-of-file is reached before the whole string has been assigned) then the end of the string is filled with null (CHR(0)) characters – this enables the programmer to evaluate the length of the string that was read.

The note concerning the file INPUT in 1] above also applies here.

3] V is of type integer.

In this case a series of characters which represent an integer as defined in Section 1.3 is read. All preceding blanks and end-of-line markers are skipped (this means that integers may be read immediately from the default file INPUT cf. the note in 1] above).

If the integer read has an absolute value greater than MAXINT (32767) then the runtime error 'Number too large' will be issued and execution terminated.

If the first character read, after spaces and end-of-line characters have been skipped, is not a digit or a sign ('+' or '–') then the runtime error 'Number expected' will be reported and the program aborted.

4] V is of type real.

Here, a series of characters representing a real number according to the syntax of Section 1.3 will be read.

All leading spaces and end-of-line markers are skipped and, as for integers above, the first character afterwards must be a digit or a sign. If the number read is too large or too small (see Section 1.3) then an 'Overflow' error will be reported, if 'E' is present without a following sign or digit then 'Exponent expected' error will be generated and if a decimal point is present without a subsequent digit then a 'Number expected' error will be given.

Reals, like integers, may be read immediately from the default file INPUT; see 1] and 3] above.

2.4.1.10 READLN

READLN({f,}V1,V2,.......Vn); is equivalent to:

BEGIN READ({f,}V1,V2,.......Vn); READLN END;

READLN{(f)} positions the file window over the component past the next end-of-line marker and assigns to f↑ the value of this next component. Thus EOLN{(f)} becomes FALSE after the execution of READLN{(f)} unless the next line is blank.

The default value of f is INPUT.

READLN may be used to skip the blank line which is present at the beginning of the file INPUT i.e. for the file INPUT only, READLN has the effect of reading in a new buffer. This will be useful if you wish to read a component of type character from the beginning of INPUT – it is not necessary if you are reading an integer or a real from the first line in INPUT (since end-of-line markers are skipped) or if you are reading characters from subsequent lines.

2.4.2 File Handling Functions.

2.4.2.1 EOLN{(f)}

The function EOLN is a Boolean function which returns the value TRUE if the file window of file f is currently positioned over an end-of-line character (CHR(13)). Otherwise the function returns the value FALSE.

The default value of f is the file INPUT.

When EOLN{(f)} is TRUE then the value of the associated buffer variable f↑ is CHR(13).

2.4.2.2 EOF{(f)}

This is a Boolean function which returns the value TRUE when an end-of-file character (CHR(26)) appears under the file window of the file f. Otherwise EOF{(f)} returns the value FALSE.

The default value of f is the file INPUT.

Note: EOF will become TRUE and remain TRUE when a CHR(26) character appears under the text window of any file but this will not prevent the action of any subsequent GET requests; this enables any spurious end-of-file markers within a file to be skipped.

EOF can also be used to detect abnormal conditions during the writing of a file. During the writing of a file f then EOF(f) is normally TRUE. However if an abnormal condition, such as a full disk or directory, occurs then EOF(f) will become FALSE and the user can detect this.

2.4.2.3 INCH

The function INCH causes the standard input device (normally the keyboard) using CP/M call 6 of the computer to be scanned and, if a key has been pressed, returns the character represented by the key pressed. If no key has been pressed then CHR(0) is returned. The function therefore returns a result of type character. You should use the compiler option C– with this function. (See Section 3).

2.4.3 Transfer Functions.

2.4.3.1 TRUNC(X)

The parameter X must be of type real or integer and the value returned by TRUNC is the greatest integer less than or equal to X if X is positive or the least integer greater than or equal to X if X is negative. Examples:

TRUNC(–1.5) returns –1   TRUNC(1.9) returns 1

2.4.3.2 ROUND(X)

X must be of type real or integer and the function returns the 'nearest' integer to X (according to standard rounding rules). Examples:

ROUND(–6.5) returns –6   ROUND(11.7) returns 12
ROUND(–6.51) returns –7   ROUND(23.5) returns 24

2.4.3.3 ENTIER(X)

X must be of type real or integer – ENTIER returns the greatest integer less than or equal to X, for all X. Examples:

ENTIER(–6.5) returns –7   ENTIER(11.7) returns 11

Note: ENTIER is not a Standard Pascal function but is the equivalent of BASIC's INT. It is useful when writing fast routines for many mathematical applications.

2.4.3.4 ORD(X)

X may be of any scalar type except real. The value returned is an integer representing the ordinal number of the value of X within the set defining the type of X.

If X is of type integer then ORD(X) = X ; this should normally be avoided.

Examples:

ORD('a') returns 97   ORD('   ') returns 64

2.4.3.5 CHR(X)

Xmust be of type integer. CHR returns a character value corresponding to the ASCII value of X. Examples:

CHR(49) returns '1'   CHR(91) returns '['

2.4.4 Arithmetic Functions.

In all the functions within this sub-section the parameter X must be of type real or integer.

2.4.4.1 ABS(X)

Returns the absolute value of X (e.g. ABS(–4.5) gives 4.5). The result is of the same type as X.

2.4.4.2 SQR(X)

Returns the value X*X i.e. the square of X. The result is of the same type as X.

2.4.4.3 SQRT(X)

Returns the square root of X – the returned value is always of type real. A 'Maths Call Error' is generated if the argument X is negative.

2.4.4.4 FRAC(X)

Returns the fractional part of X: FRAC(X) = X – ENTIER(X).

As with ENTIER this function is useful for writing many fast mathematical routines. Examples:

FRAC(1.5) returns 0.5   FRAC(–12.56) returns 0.44

2.4.4.5 SIN(X)

Returns the sine of X where X is in radians. The result is always of type real.

2.4.4.6 COS(X)

Returns the cosine of X where X is in radians. The result is always of type real.

2.4.4.7 TAN(X)

Returns the tangent of X where X is in radians. The result is always of type real.

2.4.4.8 ARCTAN(X)

Returns the angle, in radians, whose tangent is equal to the number X. The result is of type real.

2.4.4.9 EXP(X)

Returns the value e↑X where e = 2.71828. The result is always of type real.

2.4.4.10 LN(X)

Returns the natural logarithm (i.e. to thebase e) of X. The result is of type real. If X <= 0 then a 'Maths Call Error' will be generated.

2.4.5 Further Predefined Procedures.

2.4.5.1 NEW(p)

The procedure NEW(p) allocates space for a dynamic variable. The variable p is a pointer variable and after NEW(p) has been executed p contains the address of the newly allocated dynamic variable. The type of the dynamic variable is the same as the type of the pointer variable p and this can be of any type except FILE.

To access the dynamic variable p↑ is used – see Appendix 4 for an example of the use of pointers to reference dynamic variables.

To re-allocate space used for dynamic variables use the procedures MARK and RELEASE (see below).

2.4.5.2 MARK(v1)

This procedure saves the state of the dynamic variable heap to be saved in the pointer variable v1. The state of the heap may be restored to that when the procedure MARK was executed by using the procedure RELEASE (see below).

The type of variable to which v1 points is irrelevant, since v1 should only be used with MARK and RELEASE never NEW.

For an example program using MARK and RELEASE see Appendix 4.

2.4.5.3. RELEASE(v1)

This procedure frees space on the heap for use of dynamic variables. The state of the heap is restored to its state when MARK(v1) was executed – thus effectively destroying all dynamic variables created since the execution of the MARK procedure. As such it should be used with great care.

See above and Appendix 4 for more details.

2.4.5.4 INLINE(C1,C2,C3,.........)

This procedure allows Z80 machine code to be inserted within the Pascal program; the values (C1 MOD 256, C2 MOD 256, C3 MOD 256, ...) are inserted in the object program at the current location counter address held by the compiler. C1, C2, C3 etc. are integer constants of which there can be any number. Refer to Appendix 4 for an example of the use of INLINE.

2.4.5.5 USER(V)

USER is a procedure with one integer argument V. The procedure causes a call to be made to the memory address given by V. Since HiSoft Pascal holds integers in two's complement form (see Appendix 3) then in order to refer to addresses greater than #7FFF (32767) negative values ofV must be used. For example #C000 is -16384 and so USER(-16384); would invoke a a call to the memory address #C000. However, when using a constant to refer to a memory address, it is more convenient to use hexadecimal.

The routine called should finish with a Z80 RET instruction (#C9) and must preserve the IX register.

2.4.5.6 HALT

This procedure causes program execution to stop with the message 'Halt at PC=XXXX' where XXXX is the hexadecimal memory address of the location where the HALT was issued. Together with a compilation listing, HALT may be used to determine which of two or more paths through a program are taken. This will normally be used during de-bugging.

2.4.5.7 POKE(X,V)

POKE stores the expression V in the computer's memory starting from the memory address X. X is of type integer and V can be of any type except FILE or SET. See Section 2.4.5.5 above for a discussion of the use of integers to represent memory addresses. Examples:

POKE(#6000,'A') places #41 at location #6000.
POKE(-16384,3.6E3) places 00 0B 80 70 at #C000.

2.4.5.8 OUT(i,c)

The procedure OUT has one parameter of type integer, i, and one of type char, c. The character c is output directly to the Z80 port number i. Example:

OUT(23,'A') outputs the value 'A' to port 23.

2.4.5.9. PRON

The parameterless procedure PRON causes output to be directed to the printer instead of the console. This simply uses CP/M function 5 instead of 2.

2.4.5.10 PROFF

The parameterless procedure PROFF causes output to be directed to the console using CP/M call 2.

2.4.5.11 RANSEED(X,Y,Z)

This procedure has three INTEGER parameters which are used as seeds for the random numbers produced by RANDOM (See section 2.4.6.1). These integers must be between 1 and 30000.Normally the same random numbers are used for each run of a program; this procedure can be used to give different numbers.

To use RANSEED you must use the command line T option. See section 0.2 for details.

2.4.6. Further Predefined Procedures

2.4.6.1. RANDOM

This returns a pseudo-random REAL number between 0.0 and 1.0.
The algorithm used is based on that of B.A. Wichmann and I.D. Hill (NPL Report DITC 6/82). Normally the same random numbers are used for each run of a program.The seeds may be changed using the procedure RANSEED described above.

To use RANDOM you must use the command line T option. See section 0.2 for details.

2.4.6.2. SUCC(X)

X may be of any scalar type except real and SUCC(X) returns the successor of X. Examples:

SUCC('A') returns 'B'. SUCC('5') returns '6'.

2.4.6.3. PRED(X)

X may be of any scalar type except real; the result of the function is the predecessor of X.Examples:

PRED('j') returns 'i'. PRED(TRUE) returns FALSE.

2.4.6.4. ODD(X)

X must be of type integer. ODD returns a Boolean result which is **TRUE** if X is odd and **FALSE** if X is even.

2.4.6.5. CPM(V1,V2)

This useful function has two integer parameters (V1 and V2) and returns an integer result. The effect is as follows: the Z80 register C is loaded with the value (V1 MOD 256) and the Z80 register pair DE is loaded with the value V2 and then a CALL 5 is executed; this is a subroutine call to CP/M with the routine number contained in the register C and any extra information required by the routine in the register pair DE. Any result returned by the CP/M routine will be returned as the result of the function CPM. For example:

CP/M function 14 selects the disk drive whose number is contained in the register E (0= drive A,1= drive B etc). Thus to select drive B from within a HiSoft Pascal program you can use:

```
dummy:=CPM(14,1).
```

CP/M routine 25 returns the currently selected disk number so that to obtain this number in the Pascal variable disk you could use:

```
  disk := CPM(25,0);.
```

For further information on the CP/M routines see the relevant CP/M Interface Guide.

2.4.6.6 ADDR(V)

This function takes a variable identifier of any type as a parameter and returns an integer result which is the memory address of the variable identifier V. For information on how variables are held, at runtime, within HiSoft Pascal see Appendix 3. For an example of the use of ADDR see Appendix 4.

2.4.6.7 PEEK(X,T)

The first parameter of this function is of type integer and is used to specify a memory address (see Section 2.4.5.5). The second argument is a type; this is the result type of the function.

PEEK is used to retrieve data from the memory of the computer and the result may be of any type except FILE.

In all PEEK and POKE (the opposite of PEEK) operations data is moved in HiSoft Pascal's own internal representation detailed in Appendix 3. For example: if the memory from #5000 onwards contains the values: 50 61 73 63 61 6C (in hexadecimal) then:

`WRITE(PEEK(#5000,ARRAY[1..6] OF CHAR))` gives 'Pascal'
`WRITE(PEEK(#5000,CHAR))` gives 'P'

`WRITE(PEEK(#5000,INTEGER))` gives 24912
`WRITE(PEEK(#5000,REAL))` gives 2.46227E+29

see Appendix 3 for more details on the representation of types within HiSoft Pascal.

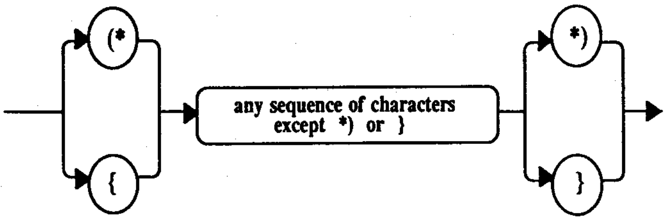
2.4.6.8 INP(p)

INP is a function with one integer parameter p. It returns as a character, the value that would be given by executing the Z80 instruction IN A,(p).

SECTION 3 COMMENTS AND COMPILER OPTIONS.

3.1 Comments

The following is the syntax for a comment:



A comment may occur between any two reserved words, numbers, identifiers or special symbols - see Appendix 2.

3.2. Compiler Options.

Compiler options are specified at the beginning of comments preceeded by a $. They consist of a letter followed by '+' or '-' except in the case of the 'F' option (see below). If more than one option is specified then they should be separated by commas.

Examples:

(*$L- turn the listing off *)
{$O-,C-,S-,A-,L- turn all checks off }

The following options are available:

Option L:

Controls the listing of the program text and object code address by the compiler.

If L+ then a full listing is given. If L- then lines are only listed when an error is detected.

DEFAULT: L+

Option O:

Controls whether certain overflow checks are made. Integer multiply and divide and all real arithmetic operations are always checked for overflow.

If O+ then checks are made on integer addition and subtraction routine calls are checked to ensure that their arguments are within range.

If O– then the above checks are not made.

DEFAULT: O+

Option C:

Controls whether or not keyboard checks are made during object code program execution. If C+ then if CTRLC is pressed then execution will return to CP/M with a HALT message – see Section 2.4.5.6.

This check is made at the beginning of all loops, procedures and functions. Thus the user may use this facility to detect which loop etc. is not terminating correctly during the debugging process.

If C– then the above check is not made.

DEFAULT: C+

Option S:

Controls whether or not stack shecks are made.

If S+ then, at the beginning of each procedure and function call, a check is made to see if the stack will probably overflow in this block. If the runtime stack overflows the dynamic variable heap or the program then the message 'Out of RAM at PC=XXXX' is displayed and execution aborted. Naturally this is not foolproof; ifa procedure has a large amount of stack usage within itself then the program may 'crash'. Alternatively, if a function contains very little stack usage while utilising recursion then it is possible for the function to be halted unnecessarily.

The address of the stack may be set at compilation time – see Section 1.2.

If S– then no stack checks are performed.

DEFAULT: S+

Option A:

Controls whether checks are made to ensure that array indices are within the bounds specified in the array's declaration.

If A+ and an array index is too high or too low then the message 'Index too high' or 'Index too low' will be displayed and the program execution halted.

If A– then no such checks are made.

DEFAULT: A+

Option I:

When using 16 bit 2's complement integer arithmetic, overflow occurs when performing a >, <, >=, or <= operation if the arguments differ by more than MAXINT (32767). If this occurs then the result of the comparison will be incorrect. This will not normally present any difficulties; however, should the user wish to compare such numbers, the use of I+ ensures that the results of the comparison will be correct. The equivalent situation may arise with real arithmetic in which case an overflow error will be issued if the arguments differ by more than approximately 3.4E38 ; this cannot be avoided.

If I– then no check for the result of the above comparisons is made.

DEFAULT: I–

Option P:

If P+ then the compiler listing is directed through the CP/M call number 5; normally to a printer.

P– stops listing through CP/M system call 5 and resumes normal listing through CP/M system call number 2; normally to CRT.

DEFAULT: P–

Option F:

This option letter must be followed by a space and then a valid CP/M file identifier of the form 'drive:filename.file extension' – if 'drive:' is omitted then the current default drive letter is assumed, the file extension must be PAS and this need not be specified.

The presence of this option causes inclusion of Pascal source text from the specified file from the end of the current statement – useful if the programmer wishes to build up a 'library' of much-used procedures and functions on the disk system and then include them in particular programs.

Example: {$F B:MATRIX include the text MATRIX.PAS from drive B};

This option may not be nested and may only appear within a program.

The compiler options may be used selectively. Thus debugged sections of code may be speeded up and compacted by turning the relevant checks off whilst retaining checks on untested pieces of code.

APPENDIX 1 ERRORS.

A.1.1 Error numbers generated by the compiler.

1. Number too large.
2. Semi-colon expected.
3. Undeclared identifier.
4. Identifier expected.
5. Use '=' not ':=' in a constant declaration.
6. '=' expected.
7. This identifier cannot begin a statement.
8. ':=' expected.
9. ')' expected.
10. Wrong type.
11. '.' expected.
12. Factor expected.
13. Constant expected.
14. This identifier is not a constant.
15. 'THEN' expected.
16. 'DO' expected.
17. 'TO' or 'DOWNTO' expected.
18. '(' expected.
19. Cannot write this type of expression.
20. 'OF' expected.
21. ',' expected.
22. ':' expected.
23. 'PROGRAM' expected.
24. Variable expected since parameter is a variable parameter.
25. 'BEGIN' expected.
26. Variable expected in call to READ.
27. Cannot compare expressions of this type.
28. Should be either type INTEGER or type REAL.
29. Cannot read this type of variable.
30. This identifier is not a type.
31. Exponent expected in real number.
32. Scalar expression (not numeric) expected.
33. Null strings not allowed (use CHR(0)).
34. '[' expected. 35. ']' expected.
36. Array index type must be scalar.
37. '..' expected.
38. ']' or ',' expected in ARRAY declaration.
39. Lowerbound greater than upperbound.
40. Set too large (more than 256 possible elements).
41. Function result must be type identifier.
42. ',' or ']' expected in set.
43. '..' or ',' or ']' expected in set.
44. Type of parameter must be a type identifier.

45. Null set cannot be the first factor in a non-assignment statement.
46. Scalar (including real) expected.
47. Scalar (not including real) expected.
48. Sets incompatible.
49. '<' and '>' cannot be used to compare sets.
50. 'FORWARD', 'LABEL', 'CONST', 'VAR', 'TYPE' or 'BEGIN' expected.
51. Hexadecimal digit expected.
52. Cannot POKE sets.
53. Array too large (> 64K).
54. 'END' or ';' expected in RECORD definition.
55. Field identifier expected.
56. Variable expected after 'WITH'.
57. Variable in WITH must be of RECORD type.
58. Field identifier has not had asociated WITH statement.
59. Unsigned integer expected after 'LABEL'.
60. Unsigned integer expected after 'GOTO'.
61. This label is at the wrong level.
62. Undeclared label.
63. Cannot assign or POKE files.
64. Can only use equality tests for pointers.
65. The parameter of this procedure/function should be of a FILE type.
66. File buffer too large (>=256 records i.e. 32K).
67. The only write parameter for integers with two ':'s is e:m:H.
68. Strings may not contain end of line characters.
69. The parameter of NEW, MARK or RELEASE should be a variable of pointer type.
70. The parameter of ADDR should be a variable.
71. All files must be FILEs OF CHAR or subrange thereof.
72. Files may only be used as global variables or variable parameters.
73. RESET and REWRITE may not be used on INPUT or OUTPUT.

A.1.2 Runtime Error Messages.

When a runtime error is detected then one of the following messages will be displayed, followed by ' at PC=XXXX' where XXXX is the memory location at which the error occurred. Consult the compilation listing to see where in the program the error occurred, using XXXX to cross reference.

1. Halt
2. Overflow
3. Out of RAM
4. / by zero   also generated by DIV.
5. Index too low
6. Index too high
7. Maths Call Error
8. Number too large

9. Number expected
10. Line too long
11. Exponent expected
12. File Error *

Runtime errors result in the program execution being halted.

*File Error is given if there is an attempt is made to access a file which has not been opened for the required type of access. e.g. if a read on a file occurs after a REWRITE without an intervening RESET. The address given in 'PC=XXXX' is the address of the file variable rather than the location in the program where the error occurred. If it is not obvious which file has generated the error use the ADDR function. See Section 2.4.6.6.

APPENDIX 2 RESERVED WORDS AND PREDEFINED IDENTIFIERS.

A 2.1 Reserved Words.

AND	ARRAY	BEGIN	CASE	CONST
DIV	DO	DOWNTO	ELSE	END
FILE	FOR	FORWARD	FUNCTION	GOTO
IF	IN	LABEL	MOD	NIL
NOT	OF	OR	PACKED	PROCEDURE
PROGRAM	RECORD	REPEAT	SET	THEN
TO	TYPE	UNTIL	VAR	WHILE
WITH				

A 2.2 Special Symbols.

The following symbols are used by HiSoft Pascal and have a reserved meaning:

+	–	*	/			
=	<>	<	<=	>=	>	
(	)	[	]	(*	*)	↑
:=	.	,	;	:	'	..

A 2.3 Predefined Identifiers.

The following entities may be thought of a declared in a block surrounding the whole program and they are therefore available throughout the program unless re-defined by the programmer within an inner block.

For further information see Section 2.

CONST	MAXINT=32767;

TYPE	BOOLEAN=(FALSE, TRUE);
	CHAR {The expanded ASCII character set};
	INTEGER=–MAXINT..MAXINT;
	REAL {A subset of the real numbers. See Section 1.3.}
	TEXT=FILE OF CHAR;

```
VAR         INPUT,OUTPUT : TEXT;

PROCEDURE   WRITE; WRITELN; READ; READLN; RESET;
            REWRITE; GET; PUT; PAGE; HALT; USER;
            POKE; INLINE; NEW; MARK; RELEASE; OUT;
            PRON; PROFF; RANSEED

FUNCTION    ABS; SQR; ODD; RANDOM; ORD; SUCC;
            PRED; INCH; EOLN; EOF; PEEK; CHR; SQRT;
            ENTIER; ROUND; TRUNC; FRAC; SIN; COS;
            TAN; ARCTAN; EXP; LN; ADDR; CPM; INP;
```

APPENDIX 3 DATA REPRESENTATION AND STORAGE.

A 3.1 Data Representation.

The following discussion details how data is represented internally by HiSoft Pascal.

The information on the amount of storage required in each case should be of use to most programmers; other details may be needed by those attempting to merge Pascal and machine code programs.

A 3.1.1 Integers.

Integers occupy 2 bytes of storage each, in 2's complement form. Examples:

```
   1  ≡ #0001
 256  ≡ #0100
-256  ≡ #FF00
```

The standard Z80 register used by the compiler to hold integers is HL.

A 3.1.2 Characters, Booleans and other Scalars.

These occupy 1 byte of storage each, in pure, unsigned binary.

Characters: 8 bit, extended ASCII is used.

```
'E'  ≡  #45
'['  ≡  #5B
```

Booleans:

ORD(TRUE)= 1	so TRUE is represented by 1.
ORD(FALSE) = 0	so FALSE is representd by 0.

The standard Z80 register used by the compiler for the above is A.

A 3.1.3 Reals.

The (mantissa, exponent) form is used similar to that used in standard scientific notation – only using binary instead of denary. Examples:

$2=2*10^0$ or $1.0_2 * 2^1$

$1=1*10^0$ or $1.0_2 * 2^0$

$$\begin{array}{llll} -12.5 & \equiv -1.25*10^1 & \text{or} & -25*2^{-1} \\ & & & -11001_2*2^{-1} \\ & & & \underline{-1.1001_2*2^3} \text{ when normalised.} \end{array}$$

$$0.1 \equiv 1.0*10^{-1} \quad \text{or} \quad \frac{1}{10} \equiv \frac{1_2}{1010_2} \equiv \frac{0.1_2}{101_2}$$

so now we need to do some binary long division..

```
          0.0001100
     101  0.1000000000000000
             101
              110
              101
                1000
                 101        at this point
                            we see that the
                            fraction recurs
```

$$= \quad \frac{0.1_2}{101_2} = 0.0001100_2$$

$$\underline{1.1001100_2 * 2^{-1}} \text{ answer.}$$

So how do we use the above results to represent these numbers in the computer? Well, firstly we reserve 4 bytes of storage for each real in the following format:

```
 [sign]  [normalised mantissa]   [exponent]     data

   23          22                 0    7    0   bit
  \___________v__________/      \_____v_____/
     H             L              E        D    register
```

sign:	the sign of the mantissa; 1 = negative, 0 = positive.
normalised mantissa:	the mantissa normalised to the form 1.xxxxxx with the top bit (bit 22) always 1 except when representing zero (HL=0, DE=0).
exponent:	the exponent in binary 2's complement form.

Thus:

```
2    ≡0 1000000 00000000 00000000 00000001 (#40, #00, #00, #01)
1    ≡0 1000000 00000000 00000000 00000000 (#40, #00, #00, #00)
-12.5≡1 1100100 00000000 00000000 00000011 (#E4, #00, #00, #03)
0.1  ≡0 1100110 01100110 01100110 11111100 (#66, #66, #66, #FC)
```

So, remembering that HL and DE are used to hold real numbers, then we would have to load the registers in the following way to represent each of the above numbers:

```
    2 ≡ LD HL,#4000
        LD DE,#0100
    1 ≡ LD HL,#4000
        LD DE,#0000
-12.5 ≡ LD HL,#E400
        LD DE,#0300
  0.1 ≡ LD HL,#6666
        LD DE,#FC66
```

The last example shows why calculations involving binary fractions can be inaccurate; 0.1 cannot be accurately represented as a binary fraction, to a finite number of decimal places.

N.B. Reals are stored in memory in the order ED LH.

A 3.1.4 Records and Arrays.

Records use the same amount of storage as the total of their components.

Arrays: if n=number of elements in the array and
s=size of each element then

the number of bytes occupied by the array is n.

e.g. an ARRAY[1..10] OF INTEGER requires 10*2 = 20 bytes
an ARRAY[2..12,1..10] OF CHAR has 11*10=110 elements and so requires 110 bytes.

A 3.1.5 Sets.

Sets are stored as bit strings and so if the base type has n elements then the number of bytes used is: (n–1) DIV 8 + 1. Examples:

a SET OF CHAR requires (256–1) DIV 8 + 1 = 32 bytes.
a SET OF (blue, green, yellow) requires (3–1) DIV 8 + 1 = 1 byte.

A 3.1.6 Files.

Files require (41 + 128*buffer size) bytes where buffer size is specified in square brackets ([]) after the file variable declaration (default is 1) – see Section 1.16.

Each file has associated with it a File Information Block (FIB); this is of the form:

0–1	Pointer to the data buffer; f↑ in Pascal terminology.
2–3	End of buffer address (set up on creation).
4	Status 0=unused (set up on creation). 1=read. –1=write.
5	EOF(f)
6	Drive number: 0=default 1=A,2=B,3=C etc., –1=terminal (i.e. INPUT and OUTPUT).

The following is the CP/M FCB entry – see the CP/M manual for details.

7	Zero.
8–15	Filename.
16–18	File type.
19–22	Zeroes.
23–38	CP/M disk map.
39	Record Count i.e. initialised to 0.
40–	Data Buffer.

For the file INPUT we have:

0–6	as above.
7	maximum linelength – 80 characters.
8	number of characters in the current line.
9–88	Data Buffer.

For the file OUTPUT we have

0–6	as above.
7	one byte buffer.

A 3.1.7 Pointers.

Pointers occupy 2 bytes which contain the address (in Intel format i.e. low byte first) of the variable to which they point.

A 3.2 Variable Storage at Runtime.

There are 3 cases where the user needs information on how variables are stored at runtime:

a. Global variables	– declared in the main program block.
b. Local variables	– declared in an inner block.
c. Parameters and returned values.	– passed to and from procedures and functions.

These individual cases are discussed below and an example of how to use this information may be found in Appendix 4.

Global variables

Global variables are allocated from the top of the runtime stack downwards e.g. if the runtime stack is at #B000 and the main program variables are:

```
VAR  i : INTEGER;
     ch : CHAR;
     x : REAL;
```

then:

i (which occupies 2 bytes – see the previous section) will be stored at locations #B000–2 and #B000–1 i.e. at #AFFE and #AFFF.

ch (1 byte) will be stored at location #AFFE–1 i.e. at #AFFD.

x (4 bytes) will be placed at #AFF9, #AFFA, #AFFB and #AFFC.

Local variables

Local variables cannot be accessed via the stack very easily so, instead, the IX register is set up at the beginning of each inner block so that (IX–4) points to the start of the block's local variables e.g.

```
PROCEDURE test;
VAR i,j : INTEGER;
```

then:

i (integer – so 2 bytes) will be placed at IX–4–2 and IX–4–1 i.e. IX–6 and IX–5. j will be placed at IX–8 and IX–7.

Parameters andreturned values

Value parameters are treated like local variables and, like these variables, the earlier a parameter is declared the higher address it has in memory. However, unlike variables, the lowest (not the highest) address is fixed and this is fixed at (IX+2) e.g.

```
PROCEDURE test(i : REAL; j : INTEGER);
```

then:

j (allocated first) is at IX+2 and IX+3.
i is at IX+4, IX+5, IX+6, and IX+7.

Variable parameters are treated just like value parameters except that they are always allocated 2 bytes and these 2 bytes contain the address of the variable e.g.

```
PROCEDURE test(i : INTEGER; VAR x : REAL);
```

then:

the reference to x is placed at IX+2 and IX+3; these locations contain the address where x is stored. The value of i is at IX+4 and IX+5.

Returned values of functions are placed above the first parameter in memory e.g.

```
FUNCTION test(i : INTEGER) : REAL;
```

then i is at IX+2 and IX+3 and space is reserved for the returned value at IX+4, IX+5, IX+6 and IX+7.

APPENDIX 4 SOME EXAMPLE HISOFT PASCAL PROGRAMS.

There follow some example programs written in HiSoft Pascal. All have been thoroughly tested and therefore can be typed in with confidence.

Some of the programs may be of practical use; certainly all of them demonstrate particular aspects of the implementation of Pascal within HiSoft Pascal, to facilitate the user's learning process.

```
(* A program to create a file 'TESTDATA.DAT' on disk drive A.
Shows the handling of strings versus individual characters
and use of FILE, REWRITE, EOLN, READLN *)

PROGRAM MAKEFILE;

CONST
   MAXFILE = 10;                           (* file entries *)
   MAXSTRING = 12;                         (* max. number length *)

VAR
   DATA : FILE OF CHAR;
   I : 1..MAXFILE;
   i : 1..MAXSTRING;
   CH : CHAR;
   CHSTRING : ARRAY[1..MAXSTRING] OF CHAR;

BEGIN
  REWRITE(DATA,'A:TESTDATA.DAT');          (* open file for writing *)
  FOR I := 1 TO MAXFILE DO
    BEGIN
      WRITE('Name please? ');
      READLN;
      WHILE NOT EOLN DO                    (* process input character *)
        BEGIN                              (* by character *)
          READ(CH);
          WRITE(DATA,CH)                   (* and output to file *)
        END;                               (* character by character *)
      WRITELN(DATA);                       (* new line to file *)

      WRITE('Number please? ');
      READLN;                              (* process input line as *)
      READ(CHSTRING);                      (* as whole string *)
      i:=1;
      WHILE CHSTRING[i] <> CHR(0) DO
       BEGIN
        WRITE(DATA,CHSTRING[i]);           (* write string to file, *)
        i:=i+1                             (* character by character. *)
       END;
      WRITELN(DATA)
    END
END.
```

```
(* Program to list the lines of a file in reverse order.
   Example of: Pointers (to create a linked-list), Files and strings. *)

PROGRAM FILEREVERSE;                          (* Reverses entries in a file. *)

CONST
   STRLEN=20;                                 (* Maximum string length *)

TYPE
   STRING = PACKED ARRAY[1..STRLEN] OF CHAR;  (* Note: PACKED ignored *)
   ID = RECORD
          NEXT : ^ID;                         (* Although illegal in  *)
          NAME,NUMBER : STRING                (* Standard Pascal, the recursive
                                                 reference to ID *)
       END;                                   (* allows a linked list *)
   LINK = ^ID;                                (* structure to be created. *)

VAR
  PREVIOUS, CURRENT : LINK;                   (* Pointers to ID. *)
  DATA : TEXT;

BEGIN
  RESET(DATA,'  TESTDATA.DAT');               (* Open 'TESTDATA.DAT' on the
                                               default drive, for reading. *)
  PREVIOUS := NIL;
   WHILE NOT EOF(DATA) DO
    BEGIN
      NEW(CURRENT);                           (* Creates a new dynamic variable
                                                of type ID. *)
      WITH CURRENT^ DO                        (* Get ready to assign to the new
                                                dynamic variable. *)
        BEGIN
         READLN(DATA,NAME);                   (* Read in the name and number *)
         READLN(DATA,NUMBER);                 (* from separate lines. *)
         NEXT := PREVIOUS                     (* Make link point to previous *)
        END;                                  (* entry. *)

      PREVIOUS := CURRENT                     (* Make this entry the entry
                                               for the next iteration. *)
    END;

(* Having initialised the linked list, now output it in reverse order. *)

  CURRENT := PREVIOUS;
  WHILE CURRENT<>NIL DO
    BEGIN
      WITH CURRENT^ DO
        WRITELN(NAME,'   ',NUMBER);
      CURRENT := CURRENT^.NEXT
    END
END.
```

```
(* Program to list lines of a file in reverse order.
 Shows use of pointers, records, EOF, MARK and RELEASE. *)

PROGRAM ReverseLine;

TYPE elem=RECORD                        (* create linked-list structure *)
           next:^elem;
           ch:CHAR
          END;
     link=^elem;

VAR prev,cur,heap:link;                 (* all pointers to 'elem' *)
    data : FILE OF CHAR;

BEGIN
  RESET(data,'A:TESTDATA.DAT');         (* open file for reading. *)
  REPEAT                                (* until end-of-file *)
    MARK(heap);                         (* assign top of heap to 'heap'. *)
    prev:=NIL;                          (* points to no variable yet. *)
    WHILE NOT EOLN(data) DO
     BEGIN
      NEW(cur);                         (* create a new dynamic record *)
      READ(data,cur^.ch);               (* and assign its field to one
                                         character from file. *)

      cur^.next:=prev;                  (* this field's pointer addresses *)
      prev:=cur                         (* previous record. *)
     END;

(* Write out the file entry backwards by scanning the
 records set up backwards. *)


    cur:=prev;
    WHILE cur <> NIL DO                 (* NIL is first. *)
     BEGIN
      WRITE(cur^.ch);                   (* WRITE this field i.e. character. *)
      cur:=cur^.next                    (* Address previous field. *)
     END;
    WRITELN;
    RELEASE(heap);                      (* Release dynamic variable space. *)
    READLN(data)
  UNTIL EOF(data)
END.
```

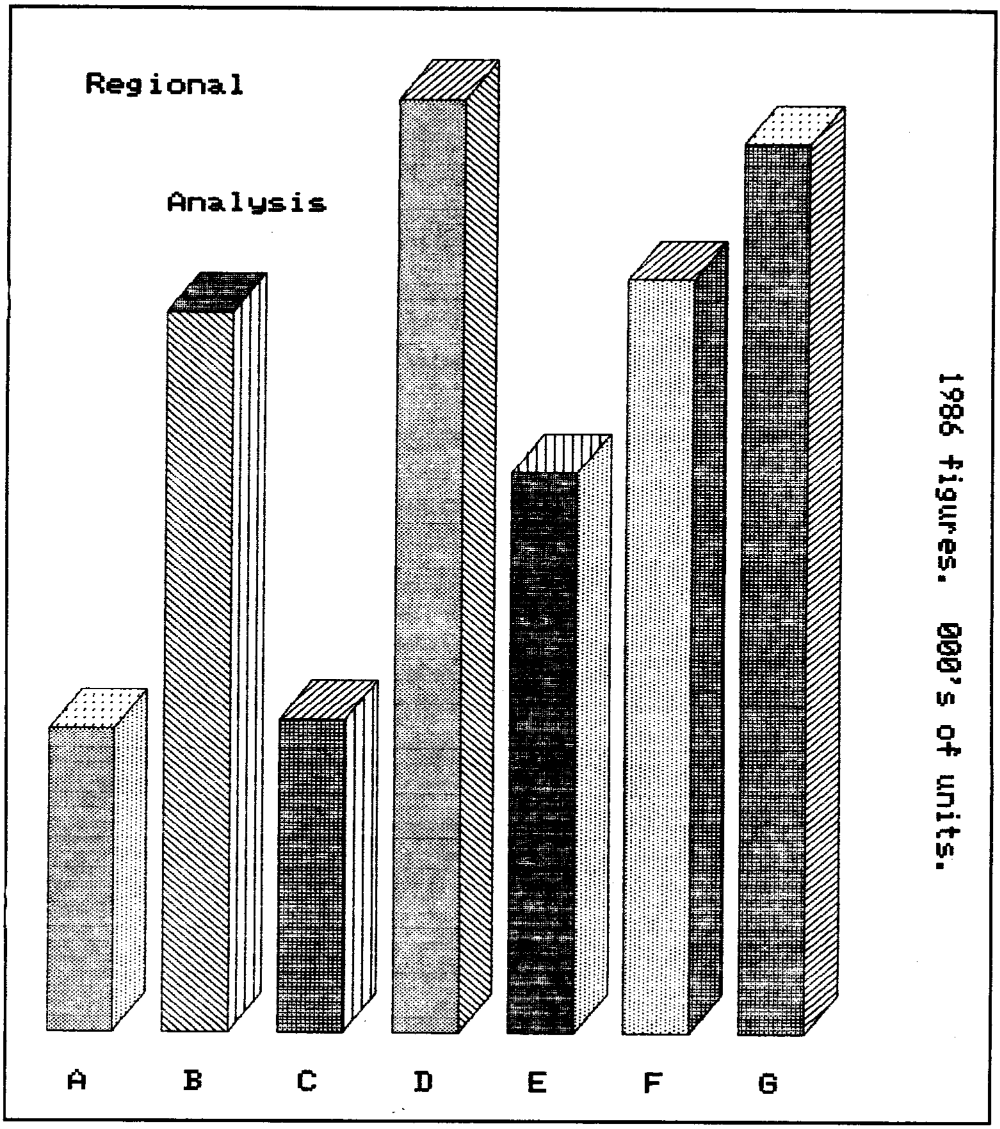
```
(* Program to show how to 'get your hands dirty'!
   i.e. how to modify Pascal variables using machine code.
   Demonstrates PEEK, POKE, ADDR and INLINE. *)

PROGRAM divmult2;

VAR r:REAL;

FUNCTION divby2(x:REAL):REAL;          (* Function to divide by 2 ..
                                        .. quickly. *)
VAR i:INTEGER;
BEGIN
  i:=ADDR(x)+1;                        (* Point to the exponent of x. *)
  POKE(i,PRED(PEEK(i,CHAR)));          (* Decrement the exponent of x.
                                        see Appendix 3.1.3. *)
  divby2:=x
END;

FUNCTION multby2(x:REAL):REAL;         (* Function to multiply by 2 ..
                                        .. quickly. *)
BEGIN
 INLINE(#DD,#34,3);                    (* INC (IX+3) - the exponent of x
                                        - see Appendix 3.2. *)
 multby2:=x
END;

BEGIN
 REPEAT
  WRITE('Enter the number r ');
  READ(r);                             (* No need for READLN - see
                                        Section 2.4.1.9. *)

  WRITELN('r divided by two is',divby2(r):7:2);
  WRITELN('r multiplied by two is',multby2(r):7:2)
 UNTIL r=0
END.
```

```
(* Program to show use of recursion. *)

 PROGRAM FACTOR;

(* This program calculates the factorial of a number input from the
 keyboard  1) using recursion  and 2) using an iterative method. *)

TYPE
  POSINT = 0..MAXINT;

VAR
  METHOD : CHAR;
  NUMBER : 0..MAXINT;

(* Recursive algorithm. *)

FUNCTION RFAC(N : POSINT) : INTEGER;

  VAR F : POSINT;

  BEGIN
    IF N>1 THEN F := N * RFAC(N-1)              (* RFAC invoked N times. *)
           ELSE F:=1;
    RFAC := F
  END;

(* Iterative solution. *)

FUNCTION IFAC(N : POSINT) : INTEGER;

  VAR I,F : POSINT;

  BEGIN
    F := 1;
    FOR I := 2 TO N DO F := F*I;                (* Simple loop. *)
    IFAC := F
  END;

BEGIN
  REPEAT
    WRITE('Give method (I OR R) and number  ');
    READLN;
    READ(METHOD,NUMBER);
    IF METHOD = 'R'
       THEN WRITELN(NUMBER,'! = ',RFAC(NUMBER))
       ELSE WRITELN(NUMBER,'! = ',IFAC(NUMBER))
  UNTIL NUMBER=0
END.
```

BIBLIOGRAPHY.

K. Jensen and N. Wirth	PASCAL USER MANUAL AND REPORT. Springer–Verlag 1975.
I.R. Wilson and A.M. Addyman	A PRACTICAL INTRODUCTION TO PASCAL WITH BS 6192. MacMillan 1982.
W. Findlay and D.A. Watt	PASCAL. AN INTRODUCTION TO SYSTEMATIC PROGRAMMING. Pitman Publishing 1978.
J. Tiberghien	THE PASCAL HANDBOOK. SYBEX 1981.
J. Welsh and J. Elder.	INTRODUCTION TO PASCAL.

HiSoft GSX Graphics



HISOFT GSX PASCAL80 LIBRARY

What is GSX ?

GSX is an extension to the CP/M Plus operating system that gives you a powerful graphics interface on your computer through the use of standard functions that are the same on all implementations. Both the Amstrad CPC 6128 and the Amstrad PCW8256/8512 have GSX supplied as standard under CP/M+ but the problem is that the calls to GSX are difficult to use and there is virtually no documentation available for them. This manual and the associated Pascal GSX procedures and functions attempt to help by supplying a high-level Pascal interface to GSX along with substantial documentation on using the routines.

Please read this manual all the way through at least twice and do try out the example programs before taking the plunge yourself.

What does GSX let you do?

With GSX you can:

1. Draw lines and any sort of polygon in 6 different styles.
2. Fill areas in 12 different fill styles.
3. Move a special graphics cursor and read the keyboard with one call.
4. Print text at any pixel position on the screen.
5. Write text in transparent or XOR mode.
6. Perform most of the CP/M Plus terminal emulation codes.
7. Display 5 sorts of 'markers' that are useful for drawing graphs.
8. Use colour on the CPC 6128.

If you are using a PCW or have an Epson compatible, Shinwa or Amstrad DMP-1 printer for your 6128 you also have the following features on the printer only:

1. Draw text in 12 different sizes, vertically and upside down on the printer.
2. Use 12 sizes of markers.

On Amstrad computers the CP/M operating system consists of just one file with extention `.EMS` but to program with GSX you need several files:

1. `GSX.SYS`
the machine independent part of GSX : about 2K long.
2. `GENGRAF.COM` the program that makes your programs load GSX before they run: about 2K long.
3. `ASSIGN.SYS` the file that tells GSX which driver to use : less than 1K long.
4. At least one device driver from the following:

`DDSCREEN.PRL`
The PCW series screen driver, 5K long: use the one on the disc with the GSX library on it; it will be better than the one on your system disc!

`DDMODE0.PRL`
The CPC mode 0 driver: 160 x 200 pixels with n colours. 20 characters per line. 4K long.

`DDMODE1.PRL`
The CPC mode 1 driver: 320 x 400 pixels with n colours. 40 characters per line. 4K long.

`DDMODE2.PRL`
The CPC mode 0 driver: 640 x 400 pixels with n colours. 80 characters per line. 4K long.

`DDFXHR8.PRL`
High resolution printer driver for PCW printers. 960 x 1368 resolution. 15K long.

`DDFXLR8.PRL`
Low resolution printer driver for PCW printers 480 x 672resolution. 12K long.

Or one of the printer drivers on your system disc. If you have a CPC 6128 see side 3 of your system disc. Type the file `DRIVERS.GSX`. This gives information on the various printer drivers.

5. `GSXLIB.PAS`
contains the Pascal procedures and functions for calling GSX.

6. `GSXVAR.PAS`
contains the Pascal variables that are used by `GSXLIB.PAS`.

The Format of ASSIGN.SYS

`ASSIGN.SYS` is an ordinary text file which *must* be present when using GSX; it is composed of lines with the following format:

```
nn d:fname
```

where `nn` is the number of the device. Conventionally

`01-09`	are used for screens.
`10-19`	are used for plotters.
`20-29`	are used for printers.

`d` is an extended drive name specifier. This can be just `A:` or `B:` or `M:` or alternatively `@:` which means the default drive. So to use the screen on the PCW it is normal to use

```
01 @:DDSCREEN
```

and to use the High resolution mode on the CPC

```
01 @:DDMODE2
```

or

```
21 @:DDFXHR8
```
to use the printer on the PCW.

You can use more than one device at a time, but this uses more memory and you can easily end up with a message 'file too large' because the device drivers must load in the top 16K of memory.

HOW TO COMPILE AND RUN PASCAL PROGRAMS USING THE GSX LIBRARIES

When compiling your program you need to use the compiler's `V` option. This is necessary because GSX runs at the top of memory and your program normally uses this area for its variables and so would corrupt GSX. You are generally safe if you use VBE00. The maximum value you can use can be determined using:

```
WRITE('Use V',PEEK(6,INTEGER):4:H)
```

in your program. Note that this value will vary depending on the device drivers you are using.

The format of your file must be of the form:

```
PROGRAM Example;
  { any constant or type definitions }
VAR

  {$F GSXVAR }

  { your global variables }

  {$F GSXLIB }

  { your procedures and functions }

BEGIN

    {your main program  }

END.
```

Then to compile a typical program like BAR.PAS (which is on your master disc) use:

```
HP80 BAR;Y,L-,VBE00 [ENTER]
```

Then when your program compiles correctly use:

```
GENGRAF BAR [ENTER]
```

to add the GSX loader to `BAR.COM`, then

```
BAR [ENTER]
```

will run your program. Try this now!

It is often a good idea to create a `.SUB` file to do the compilation and `GENGRAF` particularly if you are using the `M:` drive on the PCW.

GSX CONCEPTS

Before describing the function calls in detail its necessary to be aware of the concepts behind GSX.

At the lowest level your program 'talks' to GSX by passing the address of an array called `pblock`. This in turn contains the addresses of 5 further arrays : `contrl, intin, intout, ptsin, ptsout`. Normally you don't need to know about these, but they must be present in your program and are defined in the `GSXVAR.PAS`.

Normally a GSX program opens a '**workstation**', does some GSX calls and then closes the workstation. So that GSX can tell different workstations apart, opening the workstation returns a **'handle'** that must be used in all subsequent calls.

The **close workstation** call is used to clear the screen and put the cursor etc. back to normal for the screen and to actually do the printing on printers.

As far as the user is concerned the screen or printer page is considered to have (0,0) in the bottom left to (32767,32767) in the top right, regardless of the screen or printer physical resolution.

COLOURS

This section is mostly irrelevant when using a PCW.
Colours are refered to by colour indices. These are 0,1,2.. depending on how many colours can be on the screen at any one time.
In mode 0 this is 16 colours; in mode 1, 4 colours and in mode 0, 2 colours.

Colour index 0 is the normal background colour and Colour index 1 is the foreground colour.

The default colours are as follows:

0 black
1 red
2 green
3 blue
4 cyan
5 yellow
6 magenta
7 white

8-15 are also initially white.

Different 'inks' can be associated with a colour by specifying the proportions of red, green and blue. The device driver then does its best to match this colour from the available pallette on the CPC 6128. When using mode 2 it is generally a good idea to change the ink of index 1 as this is dark red and thus is difficult to read against a black background on a colour monitor and almost impossible to read on a monochrome one.

How and what GSX prints on the screen depends on several 'variables'. These are:

1. **Line type:**

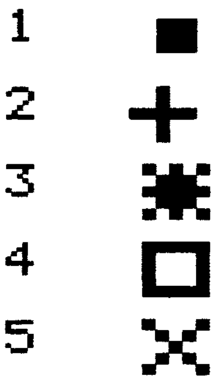
There are five different sorts of line type:

1. solid
2. dash
3. dot
4. dash,dot
5. long dash

There is also the **line colour index,** which gives the colour in which the lines are drawn.

2. **Marker type:**

Markers give you the ability to plot graphs in a more interesting way than just using dots. The different types are as follows:



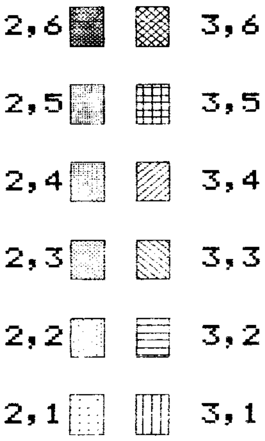
There is also the **marker colour index** which gives the colour in which the markers are drawn.

3. Fill interior style

0. hollow
1. solid
2. pattern
3. hatch

Associated with **fill interior styles** pattern and hatch there is a **'fill style index'.**

The following table shows the patterns generated for different styles and indices. `2,1` indicates an **fill interior style** of 2 and a **fill style index** of 1.



There is also the **fill colour index** which gives the colour that areas are filled with and the **text colour index** which is the colour in which graphics text is drawn.

For printers there is also the **text rotation angle.**

Writing mode

There are four different writing modes:

1	Replace	normal over- printing
2	Transparent	leaves any already printed pixels
3	XOR	reverses the bits
4	Reverse transparent	

The current **writing mode** is used in all the graph print routines.

THE GSX FUNCTIONS

1. OPEN WORKSTATION

```
PROCEDURE openwork(VAR handle:INTEGER;
          device,linetype,linecolor,marker,
          markercolor,textface,textcolor,fillstyle,
          fillindex,fillcolor:INTEGER);
```

This opens a workstation given the initial settings of the colours etc. given above. Normally 1 is used. The **handle** to use is returned in the variable `handle`.

e.g.
```
 openwork(handle,1,1,1,1,1,1,1,1,1,1);
```

Note that the device is the number given in the `ASSIGN.SYS` file and that textface will always be 1 since the Amstrad drivers do not support any other typefaces.
This function also sets up the array that is used when calling GSX.

The open workstation call also return the following in the `intout` array:

`intout[0]`	pixel width of device -1
`intout[1]`	pixel height of device -1
`intout[2]`	0= device is capable of producing a continously scaled image. 1= otherwise.
`intout[3]`	width of one pixel in microns
`intout[4]`	height of one pixel in microns
`intout[5]`	number of character heights
`intout[6]`	number of line types
`intout[7]`	number of line widths
`intout[8]`	number of marker types
`intout[9]`	number of marker sizes
`intout[10]`	number of typefaces
`intout[11]`	number of fill patterns
`intout[12]`	number of hatch styles

`intout[13]`	number of colours that can be displayed on the device at once.
`intout[14]`	number of Generalised Drawing Primitives (GDPs). Sadly for the Amstrad device drivers this is always 1.
`intout[15]-` `intout[24]`	list of the GDPs supported. This is always just the Bar.
`intout[25]-` `intout[34]`	list of attributes associated with each GDP. Not useful on Amstrads.
`intout[35]`	1= device supports colour. 0= No colour capability
`intout[36]`	1= text rotation available 0= no text rotation
`intout[37]`	1= fill area available 0= no area fill possible
`intout[38]`	1= cell array capability 0= no cell array capability
`intout[39]`	total number of colours in palette (2= monochrome)
`intout[40]`	number of locator devices (0= just keyboard)
`intout[41]`	number of valuator devices. Not applicable to Amstrads.
`intout[42]`	number of choice devices. n/a
`intout[43]`	number of string devices. n/a

e.g.

```
 openwork(handle,1,1,1,1,1,1,1,1,1,1);
```

CLOSE WORKSTATION

```
PROCEDURE closework(handle:INTEGER);
```

For screen devices clears the screen and re-displays the cursor etc. For printers this causes the current page to be output. Should be used before finishing your program. Because this clears the screen it is usual to place a `READLN` before it so that the screen can be viewed and then the screen cleared when `[RETURN]` is typed.

e.g.

```
 closework(handle);
```

CLEAR WORKSTATION

```
PROCEDURE clearwork(handle:INTEGER);
```

Clears the device. On printers all data output but not printed is 'forgotten'.

e.g.
```
 clearwork(handle);
```

UPDATE WORKSTATION

```
PROCEDURE updatework(handle:INTEGER);
```

On screens, does nothing as the screen is always up to date. On printers, causes the current page to be printed.

e.g.
```
 updatework(handle)
```

POLYLINE

```
PROCEDURE polyline(handle,count,pxyarray :INTEGER);
```

Draws a series of lines on the screen; the number of points is given by the parameter `count`. `pxyarray` is the *address* of an array that contains the points. The current **writing mode, line type** and **line colour** index are used.

e.g. Given the definitions:

```
TYPE point: RECORD
                  x,y:INTEGER
            END;

VAR p: ARRAY[1..3] OF point;
```

then

```
   WITH p[1] DO BEGIN x:=0; y:=0 END;
   WITH p[2] DO BEGIN x:=10000; y:=0 END;
   WITH p[3] DO BEGIN x:=0; y:=10000 END;

   polyline(handle,3,ADDR(p));
```

will then draw a line from (0,0) to (10000,0) and another one from (10000,0) to (0,10000).

The use of the record type here is optional but tends to lead to more readable programs.

The same effect can be obtained using

```
VAR p:ARRAY[1..6] OF INTEGER;
```

then

```
   p[1]:=0; p[2]:=0; p[3]:=10000; p[4]:=0; p[5]:=10000;
   polyline(handle,3,ADDR(p));
```

POLYMARKER

```
PROCEDURE polymarker(handle,count,pxyarray :INTEGER);
```

Draws a series of markers on the screen; the number of markers is given by the parameter `count`. `pxyarray` is the *address* of an array that contains the points on which the markers are based.The current **writing mode, marker colour, marker height** and **marker type** are used.

This is used in an exactly analagous way to `polyline`. Note that on the screen there is only one marker height.

WARNING: There appear to bugs in the printer drivers that cause markers not to be printed unless `gtext` (see below) is called previously.

GRAPHIC TEXT

```
PROCEDURE gtext(handle,x,y,string,len:INTEGER);
```

This procedure writes a string starting at graphics position `(x,y)`. `string` is the *address* of the string that is to be printed. `len` is the number of chars; normally the length of the string.

This uses the current **writing mode**, and **text colour**. On printers the current **rotation angle** is used.

e.g. given the declaration:

```
VAR HiSoft:ARRAY[1..6] OF CHAR;
```

then:

```
HiSoft:='HiSoft';

gtext(handle,16000,16000,ADDR(HiSoft),6);
```

writes `HiSoft` near the middle of the screen/page.

FILL AREA

```
PROCEDURE fillarea(handle,count,pxyarray :INTEGER);
```

This fills a polygon specified in the `pxyarray`. The number of vertices is specified by the parameter `count` and `pxyarray` is the *address* of the array containing the points, in exactly the same the form as `polyline`. The device driver draws a line from the last point to the first point so that the polygon is always closed.

The area is filled using the current **fill area colour, fill style**, and **writing mode**, and the outline of the polygon is drawn using a solid line in the current **writing mode** of the current **fill area colour**.

BAR

```
PROCEDURE bar(handle,x1,y1,x2,y2:INTEGER);
```

This draws a bar given `(x1,y1)` as one corner and `(x2,y2)` as the opposite corner.

The area fill attributes are used as with `fillarea`.

```
e.g. bar(handle,10000,10000,20000,20000);
```
draws a rectangle near the middle of the screen.

SET CHARACTER HEIGHT

```
PROCEDURE setcharheight(handle,yheight:INTEGER;
                        VAR setx,sety,cellx,celly:INTEGER);
```

This sets the size of the text characters printed on a printer. There are 12 different heights.

The `yheight` parameter is the size of the characters given in terms of graphics points.

The `setx` and `sety` parameters are maximum size of the actual characters and the `cellx` and `celly` are the size of the cells in which the characters are printed.

The following table gives the different heights given by the PCW printer using the High resolution driver:

yheight	char size(mm)	cell size(mm)
1 - 383	1 x 1	1 x 1
384 - 574	2 x 3	3 x 3
575 - 766	3 x 4	4 x 4
767 - 958	4 x 6	5 x 6
959 - 1149	5 x 7	6 x 7
1150 - 1341	6 x 8	8 x 8
1342 - 1533	7 x 10	9 x 10
1534 - 1724	8 x 11	10 x 11
1725 - 1916	10 x 13	11 x 13
1917 - 2107	11 x 14	13 x 14
2108 - 2299	12 x 15	14 x 15
2300 - 10000	13 x 17	15 x 17

The following table gives the different heights avaiable using the Low resolution driver:

yheight:	char size(mm)	cell size(mm)
1 - 780	2 x 3	3 x 3
781 - 1170	4 x 6	5 x 6
1171 - 1560	6 x 8	8 x 8
1561 - 1950	8 x 11	10 x 11
1951 - 2340	11 x 14	13 x 14
2341 - 2730	13 x 17	15 x 17
2731 - 3120	15 x 20	18 x 20
3121 - 3510	17 x 22	20 x 22
3511 - 3900	19 x 25	23 x 25
3901 - 4291	21 x 28	25 x 28
4292 - 4681	23 x 31	28 x 31
4682 - 1000	25 x 34	30 x 34

SET ROTATION

```
FUNCTION setrotation(handle,angle:INTEGER):INTEGER;
```

Again this is a function that is only applicable to printers. The `angle` parameter is a measure of the angle at which text will be printed. Normally this is 0 corresponding to the usual East to West. 900 corresponds to upwards; 1800 to upside down and 2700 to down the paper. If a different value is given then the closest match is taken.

The value returned is the set value.

e.g.

```
VAR Upside:ARRAY[1..6] OF CHAR;

Upside:='This is Upside down';

gtext(handle,15000,16000,ADDR(Upside),19);
```

prints

```
This is Upside down
```

COLOUR

```
PROCEDURE colour(handle,index,red,green,blue:INTEGER);
```

This is only useful on CPC 6128 screens; the `index` can be:

0 - 1	in mode 2
0 - 3	in mode 1
0 - 15	in mode 0

red, green and blue are the proportions of the various colours defined.These vary from 0 to 1000. Thus the default for index 0 is 1000,0,0

To change index 1 to be white use:

```
colour(handle,1,1000,1000,1000);
```

and to set ink 8 to be 'dark' blue use:

```
colour(handle,1,0,0,300);
```

The default colours are:

0 black
1 red
2 green
3 blue
4 cyan
5 yellow
6 magenta
7 white

8-15 are also initially white.

SET LINE TYPE

```
FUNCTION setlinetype(handle,style:INTEGER):INTEGER;
```

This function sets the line type that is used. Valid style arguments are:

1. solid
2. dash
3. dot
4. dash,dot
5. long dash

This function returns the style that has been set. If an illegal value is used solid (1) is set.

e.g. After

```
 dummy:=setlinetype(handle,4);
```

then lines will be drawn using dashes and dots.

To make your programs more readable you can define a type:

```
TYPE linetype=(solid,dash,dot,dashdot,longdash);
```

and then use

```
  dummy:=setlinetype(handle,ORD(longdash));
```

to achieve the same effect as the example above.

SET COLOUR

```
FUNCTION setlinecolour(handle,index:INTEGER):INTEGER;
```

This functions sets the colour `index` in which lines will be displayed. Normally the default is index 1 (normally red on the CPC and 'white' on the PCW) (this is set by the open workstation call). It returns the colour that has actually been set, so that if you try to set a colour index that does not exist for a particular screen you will get the highest value possible.

Thus if using either MODE0 or MODE1 on a CPC machine and assuming that colour index 3 has not been redefined using `setcolour` (see above) then

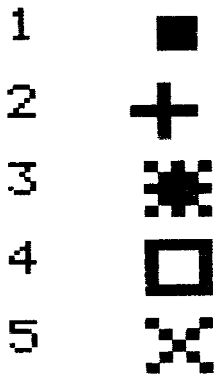
```
 i:=setcolour(handle,3);
```

will cause lines to be drawn in blue.

SET MARKER TYPE

```
FUNCTION setmarktype(handle,symbol:INTEGER):INTEGER;
```

This function sets the current marker type to be one of



As usual this returns the type that is set; this is asterisk (3) if an illegal type is passed.

e.g.

```
  dummy:=setmarktype(handle,5);
```

sets the marker type to be a diagonal cross.

SET MARKER HEIGHT

```
FUNCTION setmarkheight(handle,yheight:INTEGER):INTEGER;
```

This sets the size of markers to be displayed on the printer. There are 12 different sizes available with the two drivers. The PCW High resolution driver gives the following sizes:

yheight	approx height in mm
1 - 383	1
384 - 574	3
575 - 766	4
767 - 958	6
959 - 1149	7
1150 - 1341	8
1342 - 1533	10
1534 - 1724	11
1725 - 1916	13
1917 - 2107	14
2108 - 2299	15
2300 - 10000	17

The PCW Low resolution driver gives the following sizes:

yheight	approx height in mm
1 - 780	3
781 - 1170	6
1171 - 1560	8
1561 - 1950	11
1951 - 2340	14
2341 - 2730	17
2731 - 3120	20
3121 - 3510	22
3511 - 3900	25
3901 - 4291	28
4292 - 4681	31
4682 - 10000	34

SET MARKER COLOUR

```
FUNCTION setmarkcolour(handle,index:INTEGER):INTEGER;
```

This functions sets the colour `index` in which markers will be displayed. It returns the colour that has actually been set, so that if you try to set a colour index that does not exist for a particular screen you will get the highest value possible.

Thus if using either MODE0 or MODE1 on a CPC machine and assuming that colour index 2 has not been redefined using `setcolour` (see above) then

```
 i:=setmarkcolour(handle,2);
```

will cause markers to be drawn in green.

SET TEXT COLOUR

```
FUNCTION settextcolour(handle,index:INTEGER):INTEGER;
```

This function sets the colour index that text is drawn with when using both the `gtext` and `curtext` procedures on the CPC machines.

e.g.

```
 i:=setcolour(handle,5);
```

will cause subsequent text to be output in yellow.

Note that you should always reset this to be 1 before you do a `closework` and then return to CP/M because otherwise you will end up with a blue foreground and background.

SET FILL INTERIOR

```
FUNCTION setfillinterior(handle,style:INTEGER):INTEGER;
```

This function sets the fill interior style to be one of:

0 - hollow
1 - solid
2 - pattern
3 - hatch

For the pattern and hatch options the fill style index (see below) is also used.

So that for example:

```
i:=setfillinterior(handle,0)
```

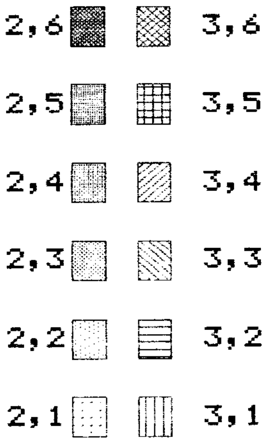
will cause subsequent fills to be done in current background colour.

SET FILL STYLE INDEX

```
FUNCTION setfillstyle(handle,style:INTEGER):INTEGER;
```

This changes how areas are filled if pattern (2) or hatch(3) fill interior styles are in use.

The following table shows the patterns generated for different styles and indices. `2,1` indicates an **fill interior style** of 2 and a **fill style index** of 1.



SET FILL COLOUR

```
FUNCTION setfillcolour(handle,index:INTEGER):INTEGER;
```

This function sets the colour `index` that areas are filled with. It returns the colour that has actually been set, so that if you try to set a colour index that does not exist for a particular screen you will get the highest value possible.

Thus if using either MODE0 or MODE1 on a CPC machine and assuming that colour index 2 has not been redefined using `setcolour` (see above) then:

```
 i:=setcolour(handle,2);
```

will cause areas to be filled in green.

PLACE GRAPHIC CURSOR AT LOCATION

```
PROCEDURE dspgrcursor(handle,x,y:INTEGER);
```

Displays the graphics cursor at the graphics location `(x,y)`. The graphics cursor is of the 'cross hair' type. It is drawn using the XOR mode so that in can be subsequently removed.

e.g.

```
 dspgrcursor(handle,16384,16384);
```

displays the graphics cursor in the middle of the screen.

REMOVE LAST GRAPHIC CURSOR

```
PROCEDURE remgrcursor(handle:INTEGER);
```

This removes the last graphics cursor displayed using `dspgrcursor`. It is removed by writing the graphics cursor in XOR mode. Thus if it has been cleared already it will in fact be redrawn.

SET INPUT MODE

```
PROCEDURE setinputmode(handle,device,mode:INTEGER);
```

This procedure is used to choose between using `getlocator` and `samplelocator`. You must set `device` to be 1 on the Amstrad computers. If `mode` is

1	then `getlocator` is to be used.
2	then `samplelocator` is to be used.

e.g. `setinputmode(handle,1,2)` is used before calling `samplelocator`.

GET LOCATOR

```
FUNCTION getlocator(handle,x,y:INTEGER;
                    VAR xout,yout:INTEGER):CHAR;
```

This routine displays a graphics cursor at graphics position `(x,y)` and then lets the user move it using the cursor keys. Using the cursor keys alone moves the cursor in quite large steps and using the shift keys as well will cause it to move in small steps. When the user presses a non-cursor key this is returned as the value of the function with `(xout,yout)` containing the final co-ordinates of the graphics cursor which is then removed.

e.g.

```
key:=getlocator(handle,16384,16384,xpos,ypos);
WRITELN('You moved the cursor to (',x,',',y,
        ') and pressed ',key);
```

If you have used the samplelocator function (see below) then you must use `setinputmode(handle,1,1);` before calling this function.

SAMPLE LOCATOR

```
FUNCTION samplelocator(handle,x,y:INTEGER;
         VAR xout,yout:INTEGER; VAR c:CHAR):INTEGER;
```

This function is used to find the current position of the graphics cursor, which must be displayed separately at `(x,y)` using `dspgrcursor` (see above) first. The value returned as the result is either:

0	nothing happened.
1	the cursor has been moved; the new cursor position will be in `(xout,yout)`
2	a non-cursor key has been pressed; the character is returned in the parameter `c`.

You *must* call `setinputmode` (see above) before calling this function.

Here is an extended example of the use of `samplelocator` in the form of an entire program:

```
PROGRAM Mouse;

TYPE point= RECORD x,y:INTEGER END;

VAR

{$F GSXVAR }

      st:INTEGER;           { the status returned by samplelocator }
      p: ARRAY[0..1] OF point; {p[0] is the last point p[1] the current }
      term:CHAR;            { any character typed }

{$F GSXLIB }

BEGIN
  openwork(handle,1,1,1,1,1,1,1,1,1,1);

  setinputmode(handle,1,2);          {set sampling mode }

 {$C-  to avoid losing characters }
 WITH p[1] DO BEGIN x:=10000;y:=10000; END; {start at point(10000,10000);}

 p[0]:=p[1];

 dspgrcursor(handle,p[0].x,p[0].y);  {display the cursor initially }

 REPEAT
  st:=samplelocator(handle,p[0].x,p[0].y,p[1].x,p[1].y,term);
  CASE st OF
  0   :;                             {nothing has happened so do nothing }
  1   : BEGIN                        {the cursor has been moved}
         remgrcursor(handle);        {remove the old cursor}
         polyline(handle,2,ADDR(p)); {draw a line between the old and new
                                      points}
         dspgrcursor(handle,p[1].x,p[1].y);    {display the new cursor }
         p[0]:=p[1];                 {make the old point the current one }
        END;
  2   : WRITE(term)                  {echo any characters typed}
  END;
 UNTIL (st=2) AND (term='E');        {finish if E is typed }

 closework(handle)                   {clear the screen etc }
END.
```

THE TERMINAL 'ESCAPE' PROCEDURES

These give text, cursor positioning etc in a machine independent way providing most of the facilities of the CP/M Plus terminal emulator on the Amstrad machines.

INQUIRE CHAR CELLS

```
PROCEDURE inqcharcells(handle:INTEGER;
                       VAR rows,columns:INTEGER);
```

This returns the size of the screen in character positions. This varies depending on whether you are using a PCW or a CPC and whether or nct the status line is enabled.

Thus given the declaration

```
VAR r,c:INTEGER;
```

```
inqcharcells(handle,r,c)
```

will normally return r=31 and c=90 on a PCW with the status line enabled.

EXIT ALPHA MODE

```
PROCEDURE exitalpha(handle:INTEGER);
```

This simply switches the cursor **OFF** on the Amstrad machines.

ENTER ALPHA MODE

```
PROCEDURE enteralpha(handle:INTEGER);
```

This simply switches the cursor **ON** on the Amstrad machines.

ALPHA CURSOR UP

```
PROCEDURE cursorup(handle:INTEGER);
```

This moves the cursor up one line; if it is already on the top line it does not move.

ALPHA CURSOR DOWN

```
PROCEDURE cursordown(handle:INTEGER);
```

This moves the cursor down one line; if it is already on the bottom line it does not move.

ALPHA CURSOR RIGHT

```
PROCEDURE cursorright(handle:INTEGER);
```

This moves the cursor right one character; if it is already on the right most column it does not move.

ALPHA CURSOR LEFT

```
PROCEDURE cursorleft(handle:INTEGER);
```

This moves the cursor left one character; if it is already on the left most column it does not move.

HOME ALPHA CURSOR

```
PROCEDURE cursorhome(handle:INTEGER);
```

This moves the cursor to the top left corner of the screen.

ERASE TO END OF ALPHA SCREEN

```
PROCEDURE eraseEOS(handle:INTEGER);
```

Erases the screen from the current alpha position to the end of page. The cursor does not move.

ERASE TO END OF ALPHA TEXT LINE

```
PROCEDURE eraseEOL(handle:INTEGER);
```

Erases the screen from the current cursor position to the end of the line. The cursor does not move.

DIRECT ALPHA CURSOR ADDRESS

```
PROCEDURE curaddress(handle,row,column:INTEGER);
```

This procedure moves the cursor to alpha position `(row,column)` with `(1,1)` as the top left hand corner. If the position is not on the displayable screen it is moved to the nearest value that can be displayed.

OUTPUT CURSOR ADDRESSABLE ALPHA TEXT

```
PROCEDURE curtext(handle,string,size:INTEGER);
```

This outputs a text string starting at the current alpha cursor position. `string` is the *address* of the string and `size` is its length in characters. If you only wish to output part of the string make `size` less than the true string length.

e.g.
```
VAR str:ARRAY[1..22] OF CHAR;

 curaddress(handle,1,10);
 str:='A HiSoft Powerful Tool';
 curtext(handle,ADDR(str),22);
```

writes `A HiSoft Powerful Tool` on the top line of the screen 10 characters in from the left.

REVERSE VIDEO ON

```
PROCEDURE rvseon(handle:INTEGER);
```

This procedure causes all subsequent alpha text to be displayed in inverse video until `rvseoff` is called.

REVERSE VIDEO OFF

```
PROCEDURE rvseoff(handle:INTEGER);
```

this causes subsequent alpha text to be printed in normal video.

INQUIRE CURRENT ALPHA CURSOR ADDRESS

```
PROCEDURE inqcuraddress(handle:INTEGER;
                      VAR row,column:INTEGER);
```

This procedure returns the current alpha cursor position in the variables `(row,column)` with (1,1) as the top left hand corner.

INDEX

ED80

```
JABBER.WOK        LINE:16   COL:17  ^Q^F   I/AUTO
              He took his vorpal sword in hand;
              Long time the manxome foe he sought
              So rested he by the Tumtum tree,
              And stood awhile in thought.


              And as in uffish thought he stood,
              The Jabberwock, with eyes of flame,
              Came whiffling through the tulgey wood
              And burbled as it came!


              One, two ! One, two! And through and through
              The vorbal blade wqent snicker-snack!
              He left it dead, and with its head
              He went gallumphing back.


              "And hast thou slain the Jabberwock.
              Come to my arms, my beamish boy!
              O frabjous day! Callooh! Callay!"
              He chortled in his joy.

FREE:44836   $JABBERWOCK                        $Jabberwock
```

ED80 is a full screen text editor designed to help you create and amend source programs and other texts quickly and efficiently.

Please take time to read carefully through this manual; there are sections on installing ED80 on your particular computer, the commands available from ED80, some technical details of ED80 and a tutorial; this tutorial is well worth working through even if you have used screen editors before.

The software described by this manual will run on computers with the following specification:

- *a Z80 microprocessor*
- *the CP/M 2.2 disc operating system*
- *a TPA of at least 16K for the ED80 programs*
- *a TPA of at least 36K for the installation programs*

Please check that your system conforms with the above requirements and that you have been supplied with a disc that is correctly formatted for your disc system; the format supplied is shown on the disc label together with your registration number.

Also note that your computer must be running under the CP/M 2.2 operating system before any of the ED80 programs will work.

We strongly encourage you to make one back-up copy of ED80 before using it.

If you cannot get ED80 to run correctly on your computer then please contact your supplier for help or contact HiSoft directly:

HiSoft
180 High Street North
Dunstable LU6 1AT
(0582) 696421

CONTENTS

SECTION 1 ABOUT ED80

ED80 is a full-screen program editor. It is a tool to enable you to type programs into your machine, to inspect and edit them at will and to save them on disc, and has been designed to make the process of program development as easy and convenient as possible. When you use **ED80**, you might immediately notice that it behaves in approximately the same way as a word-processor. Although there are important differences between a program-editor and a word-processor (reflected in the design of **ED80**) the initial set-up of the commands in **ED80** is compatible with the most familiar and widely-used of word-processors, Wordstar. **ED80** has been written, however, so that you may easily tailor the program to your personal preference (See **INSTALLING ED80**) and thus where the manual might read:–

Now move the cursor up one line ([CTRL]–X   )

a space has been left after the Wordstar command ([CTRL]–X) so that if you decide to change it using the **ED80INST** program you can fill in the new command.

The supplied disc contains at least six (and possibly seven) files concerned with **ED80**:–

1) ED80.COM	**(The program editor itself)**
2) ED80.HLP	**(The ED80 help file created by ED80INST)**
3) ED80INST.COM	**(The INSTALL program)**
4) ED80INST.MSG	**(The message file for the INSTALL program)**
5) JABBER.WOK	**(Example text file)**
6) EXTRA.WOK	**(Example text file)**
[7) something.E80	**(An optional installation file)]**

It is recommended that the first-time user work through section 2 of the manual (which is not intended to be a work of reference, but rather, a tutorial guide), after which he/she will be familiar enough with **ED80** to use the program without further reference to documentation.

SECTION 2 USING ED80

2.1 GETTING STARTED

You enter **ED80** simply by typing ED80. You may, however, add a filename preceded by a space eg.. ED80 MYFILE.TXT. This will cause that filename to become the Current File. If the Current File exists then it is loaded from the disc otherwise the user simply starts from scratch with an empty file.

Type:–

```
ED80 JABBER.WOK [ENTER]
```

(from now on the symbol [ENTER] will be used to represent a key that may be labeled **ENTER** or **RETURN** or **CR** on your machine). This will activate **ED80** and load the file **JABBER.WOK** from the disc.

2.2 THE STATUS-LINES

You should now be in **ED80** with some text filling the screen. If there appear to be problems at this stage then **ED80** may require installing on your terminal (See **INSTALLING ED80**). The status-lines are described here for an 80 column screen, but even if your console is different (eg.. 40 columns), the differences in the status-lines will not be very significant.

The upper status-line (the top line of the screen) has six sections and should appear 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 **ED80** and is blank because you haven't issued any yet. To the right of this is the mode (see next section) and finally there is a large space for **ED80**'s messages.

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

```
FREE:XXXXX    $                       $
```

On the left is the amount of free space you have left in memory which will vary depending on the machine and the size of the current text. The value is approximately equal to the number of characters you can type before **ED80** becomes full of text. The two $ signs mark the start of the Find and Substitute strings which are currently undefined.

On a 40 column screen, the space for the commands is on the top status line at the far left and the messages will overwrite the line and column information.

2.3 THE TEXT WINDOW

The screen in **ED80** 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). In **ED80** the scrolling is handled automatically and intelligently to give you the most convenient window onto the text.

2.4 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 (ie.. the one at the cursor) and thus the line remains the same length. **ED80** will not allow you 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.

2.5 A TUTORIAL IN THE USE OF ED80

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 own terminal.

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 **ED80**. As a re-assuring start, verify that the help key ([CTRL]–J) will list the various commands. Remember that [CTRL]–J means control–J ie.. hold down the control key and then press J. After a quick look at the help page, pressing [ENTER] 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 experimenting with 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. Next 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 [ENTER] 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 ie.. 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 **ED80** is in **INSERT** mode (note the word on the upper status-line). You can change (or 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 **ED80** will not allow you to overwrite [ENTER] in **CHANGE** mode.

A word of explanation is in order here about [ENTER]. It is a character as much as is the letter "A" and can be cursored-over, deleted, found and substituted (as [CTRL]–M) as can any other character. In fact the installation program gives you the option of displaying all [ENTER]s so you can see where you are more easily. The most usual way of displaying them is <.

The second verse has been omitted from the poem and this is a good excuse to test the deleting commands in **ED80** 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 [ENTER] twice to give a good separation between the stanzas. You can toggle auto-indent by pressing [CTRL]–O I ie.. [CTRL]–O and then I (or i or [CTRL]–I or [CTRL]–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 the character at the cursor ([CTRL]–G). Press [ENTER] 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 [ENTER]:

```
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 **ED80** has been designed to minimise the effects of the more common errors. Thus whenever you delete a line in **ED80** 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 it 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 ie.. 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 in **ED80** 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 lower status-line. Now type in Jabberwock. (the word Jabberwock followed by a full-stop) (you can use the delete character left ([DEL]) if you make a mistake) and then [ENTER] and just press [ENTER] 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 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 **ED80**. Put the cursor to the start of line 4 ie.. 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 lower 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 [ENTER]

(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 very important to be able to save an edited file to the disc. There are three ways to quit **ED80**. Firstly, one may abandon the file ([CTRL]–K Q). Here nothing is saved and the current text is lost. Then one may want to save the file 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. Finally, the most normal method of quitting **ED80** 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. Issue 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 [ENTER]. 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 3 INSTALLING ED80

The process of installing **ED80** involves three phases. **ED80** is first read in from the disc. Then, sections of the program are modified and finally **ED80** 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 **ED80**.

There are two reasons that you might want to install **ED80**. Primarily, it may be that there are problems with the screen layout and **ED80** 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 ED80 COMMANDS** and **RE-DEFINING USER OPTIONS**. In either case you should first read the next section.

3.1 STARTING UP THE INSTALL PROGRAM

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

```
ED80INST [ENTER]
```

You will now see the **ED80INST** 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 **ED80** 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 ED80 is
read in from a file called ED80.COM
Use another file instead (Y/N)   ?
```

The reply will normally be N, the exception being when you have renamed a version of **ED80**. A reply of Y 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 eg . . . to use **EDIT.COM** as the image, type EDIT [ENTER]). If you typed Y by mistake and really do want to use ED80.COM as the working copy then just type ED80. 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 **ED80** Installation Menu will appear.

There is now a copy of **ED80** in the memory of your machine ready to be altered and the **ED80** Installation Menu on the screen.

```
ED80 INSTALLATION MENU
--------------------
1. Return to CP/M
2. Alter screen codes
3. Save ED80 as <working copy filename> (normally
   ED80.COM)
4. Save ED80 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 **ED80** 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 ED80 COMMANDS**, **REDEFINING USER OPTIONS**, and **USE OF INSTALLATION FILES.**

3.2 TERMINAL INSTALLATION

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

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

and then

in answer to each of these questions you should type in the correct number followed by [ENTER]. Pressing [ENTER] 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 **ED80** 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, **ED80** 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 enter the correct sequence terminated by [ENTER] as above. If your controller does not recognize a sequence to clear the screen (possible but unlikely) then press D.

```
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 **ED80** will perform the function itself by software (although more slowly than the controller would do it).

```
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 **ED80** 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 from **ED80**. 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!

3.3 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 3) ! (which you are. Good)
User patch area starts at #XXXX

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 **ED80** and you will then be returned to the main menu. It is possible to get the message

```
Too many characters in commands
```

when trying to read in the patch-file. This message means that the size of the command definitions and the patch-file combined is too large. You should try to change the command definitions to be shorter and thus quicker (both to type and to execute) and more efficient.

3.4 RE-DEFINING ED80 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.

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 definition should be typed in element by element (up to a maximum of four elements) and terminated by [ENTER] after which the redefined command appears. If you are now sure that the definition is correct then press [ENTER] 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 [ENTER]. 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 [ENTER]
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 as three key presses (ie.. 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 **ED80** 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.

3.5 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 [ENTER] or [ENTER] alone to retain the value in brackets.

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

When the cursor in **ED80** moves off the right-hand edge of the screen, the text window moves to the right (ie.. 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.

3.6 USE OF INSTALLATION FILES

There are many features of **ED80** that are alterable by the user. Every copy of **ED80** 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 [ENTER]. 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 or the size of the patch-file (if you're using one).

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 **ED80** in memory.

The main use of Installation Files is when you are in the long-term process of tailoring your version of **ED80** to suit your own preferences. If you save each successive change you make to the installation of **ED80** 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.

3.7 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 **ED80** (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**) whereas option 4 allows you to change the name by which you will invoke **ED80**. You may have more than one copy of **ED80** 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.

N.B. You may, when saving the copy of **ED80** get the error message

```
Too many characters in commands
```

in which case you should either decrease the size of your command definitions, or the size of the patch-file or both. 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 4 COMMAND REFERENCE GUIDE

This Section is intended as a short reference guide to the commands and features of **ED80**. In all cases, the default command is given in brackets followed by a space so that the user may fill in any new commands. Where possible, the default command is the same as the Wordstar command. A ? means that there is no exact Wordstar equivalent for this command and the sign [CTRL]– means that the control key is held down. [ENTER] indicates that the user should press the requisite key, whereas <CR> indicates a byte of the value £OD (ASCII 13).

4.1 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).

Character left (alt) ([CTRL]–H=backspace   )
As in Wordstar there are two cursor left commands.

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]–O S   : [CTRL]–O D   )
Move the cursor to the last/next tab position. The size of tab stops may be defined by the user (see **INSTALLING ED80**). Wraparound operates.

Start/End of line ([CTRL]–Q S   : [CTRL]–Q D   )
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, **ED80** will behave as though the cursor was at the end of the current line (Ambiguous cursor).

Top/Bottom of screen ?([CTRL]–O E : [CTRL]–O X)
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 25 line screen will move the text window up by 22 lines (25 screen lines-2 status lines-1) and the old top line becomes the new bottom line. Ambiguous cursor operates

Start/End of file ([CTRL]–Q R : [CTRL]–Q C)
Move the cursor to the start/end of the file.

4.2 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]–O T : [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.

4.3 BLOCK COMMANDS

Mark start/end of block ([CTRL]–K B : [CTRL]–K K)
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]–K V)
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]–K C)
Copy the currently marked block from the text to the cursor position.

Delete block ([CTRL]–K Y)
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) the user is asked whether to abandon the block. If the user presses Y then the block will be deleted from the text and not placed in the block buffer. If the user does not 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]–O P)
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]–K R)
The user is asked for a filename. [ENTER] alone aborts the command. A filename followed by [ENTER] will search the disc for the filename given and insert it at the cursor. The response RDR: will read the block from the current logical reader device.

Write block ([CTRL]–K W)
The user is asked for a filename. [ENTER] alone aborts the command. A filename followed by [ENTER] 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).

4.4 QUICK CURSOR MOVEMENT

Goto line ?([CTRL]–O G)
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 destruc-

tive backspace) and after [ENTER] 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]–Q B : [CTRL]–Q K)
Move the cursor to the start/end block marker.

Remember position ([CTRL]–K O)
The current cursor position is stored. The marker is lost if the line in which it lies is subsequently changed.

Return to position ([CTRL]–Q O)
The cursor is positioned at the stored position.

4.5 FIND AND SUBSTITUTE

Find first ([CTRL]–Q F)
The current Find string is displayed. [ENTER] will retain the current string, otherwise the user should type in the required Find string (up to a maximum of 32 characters) and then press [ENTER]. The **DELETE CHAR LEFT** command 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 (eg.. [CTRL]–P then [ENTER] 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 [ENTER] 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]–O L)
The file is searched for the next occurrance 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]–O A)
Starting from the cursor, all occurrances 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.

4.6 LEAVING ED80

Quit and Exit ([CTRL]–K Q)
The user is asked whether to abandon the file. Pressing Y will cause a return to **CP/M** and the current text will be abandoned. Pressing [SPACE] will prompt the user for a filename and the given file will be loaded from the disc thus effectively also abandoning the current text. Any other response will abort the command.

Edit without backup ?([CTRL]–O Q)
The Current Filename is displayed after the prompt Filename:. This may be deleted using the **DELETE CHAR LEFT** command ([DEL]) and altered, or the whole command can be aborted by pressing [CTRL]–U. When the user is satisfied with the filename, [ENTER] or [SPACE] will cause the current text to be saved on the disc under the filename given. If [ENTER] is used as the terminator then after saving the file the user is returned to **CP/M**. After [SPACE], however, the user is prompted for a filename and then will be able to edit another file without leaving **ED80**. Thus the normal response after this command will be [ENTER] or [SPACE] alone which will save the file with its original name. A file already on the disc with the same name will be lost.

Exit with a backup ([CTRL]–K X)
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
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 the user 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).

4.7 TOGGLES

Toggle 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]–O I)
Auto indent will only operate in **INSERT** mode. The message INSERT will become I/AUTO. When indent is on and [ENTER] 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]–O F)
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.

4.8 MISCELLANEOUS

Deliver tab ([CTRL]–I)
Will return a tab character ([CTRL]–I or ASCII 9). The size of tabs and thus the position of tab stops may be defined by the user. 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]–O R)
If the user is in the process of editing a line then this command will restore the line to what it was when the user first positioned the cursor on it. If the user is 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. The user 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]–K F)
The prompt Filename: is given. See **RULES FOR FILENAMES.** A reply of [ENTER] or [SPACE] alone will abort the command as will pressing [CTRL]–U. After the filename is terminated by [ENTER] 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 the user to the current text.

Erase file ([CTRL]–K J)
The prompt Filename: is given. See **RULES FOR FILENAMES.** A reply of [ENTER] or [SPACE] alone will abort the command, as will pressing [CTRL]–U. After the filename is terminated by [ENTER] 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 (eg..

[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 **ED80** and how to access them.

SECTION 5 PROMPTS AND MESSAGES IN ED80

There are four prompts produced by **ED80**. They appear on the lower status line. Two require a single key response and the other two require a string of characters terminated by [ENTER].

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 the user responds 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 the user has issued any command the execution of which would overwrite the block in the block buffer. If the user responds Y then the block in the block buffer will be lost and the command executed while any other response will abort the command.

Abandon text: Sure?

This prompt requires a single character response. It is produced after the **QUIT AND EXIT** command. If the user responds Y then the current text will be lost and the user returned to **CP/M**. If the user responds [SPACE] then he will be further prompted to produce a filename (see later) and the named file will be loaded and the old current text abandoned. Any other response will abort the command.

Filename:

This prompt requires a string of characters terminated by [ENTER] or [SPACE]. 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 the two **EXIT** commands are issued, the Current Filename is given for convenience although it may be deleted if desired and another name substituted. If the name returned is null ie.. [ENTER] or [SPACE] alone, or [CTRL]–U is pressed at any stage 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 [ENTER]. 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. [ENTER] alone aborts the command.

RULES FOR FILENAMES

A filename consists of three fields. The drivename, the filename and the filetype. (eg.. 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 (ie.. **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 directory command 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
*.BAK	Addresses all files of type .BAK on the current drive
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 only be used for the **DISC DIRECTORY** command. Addresses all files on drive B

There are thirteen messages produced by **ED80** and they appear on the lower 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 (ie., 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.

SECTION 6 TECHNICAL DETAILS

6.1 INTERNAL FILE FORMAT

Text in **ED80** 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 **ED80** compatible with normal **CP/M** text files.

The maximum line-length in **ED80** 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 remains 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).

6.2 NON-PRINTING CHARACTERS

Characters of ASCII value less than 32 decimal (Control characters) are treated as far as the user is concerned as any other character. They may be entered into the file by first 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] [ENTER] 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 ie.. 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 ?).

6.3 DATA AND EXTERNAL DEVICES

Whenever the user gives a command that would normally access the disc (ie.. **READ BLOCK, WRITE BLOCK, EXIT WITHOUT BACKUP, EXIT WITH A BACKUP)** there are three responses to the prompt File-name: 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 **ED80** 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 **ED80** by use of PUN: and RDR:.

SECTION 7 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 #XXXX
;so assuming the #XXXX was #2438 the statement should be
;        ORG   #2438
;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 that 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 #F060      ;This is an example
          LD   HL,#FFC8   ;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   (#FFD0),A  ;An example routine
          LD   A,C        ;to position the cursor
          LD   (#FFD2),A  ;Note that the
          PUSH IX         ;routine saves the IX
          CALL #F076      ;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  $
```

The questions we are asked most often

Here are the answers to the questions we get asked most frequently - if you get stuck read this:

Q. Why do I get error 3 (undeclared identifier) when I use one of `SIN`, `COS`, `TAN`, `ARCTAN`, `EXP`, `LN`, `RANDOM`, `RANSEED` or `FRAC` ?
A. You haven't used the command line option `T`. See Page P-3.

Q. Why do I get `BDOS ERROR ON A: R/O` ?
A. You're using CP/M 2 and have changed discs without typing [CTRL] C at the A> prompt.

Q. My program looks O.K. but the compiler doesn't generate a .COM file and finishes with the error '`No more text`'.
A. The last line of your program needs to finish with '`END.`' and a newline. Like many CP/M programs (e.g. Digitial Research's SUBMIT, RMAC) Pascal80 must have a CR/LF before the end of file. You should edit your program; go to the end of text (with [CTRL] Q [CTRL] C) and type ENTER.

Q. When I load ED80 I get rubbish on the screen as well as the text. How do I make it look alright?
A. ED80 isn't installed for your screen. Run the ED80 install program as described in the ED80 manual Page E-9 and change the screen codes. If you don't know the screen codes, look in the documentation that came with your computer. If this doesn't help contact your dealer or the computer's manufacturer.

Q. I seem to have installed ED80 OK but if the cursor is on the bottom line and I do cursor down, then the display gets messed up.
A. It sounds as if you have the number of lines on the screen installed incorrectly; check this carefully.