
HISOFT
microdrive
versions
PASCAL
DEVPAC

HiSoft is pleased to announce ZX Microdrive-compatible versions of both Pascal and Devpac - version numbers 1.6M and 3.2M respectively.

These versions are available from the 7 March 1984 and give the user the ability to store and retrieve text-files to and from the Microdrives, 'include' text from a Microdrive cartridge whilst compiling or assemble and save the resultant object code onto a Microdrive cartridge. These releases incorporate enhancements and bug-fixes to Versions 1.5M and 3.0M of Pascal and Devpac respectively.

Any existing customer who purchased Devpac 3M or Pascal 1.5M may exchange their tape(e) for these new versions for a handling charge of #1.50 (#2.50 outside the UK). To upgrade to these versions from previous versions of Pascal and Devpac please return your original tape with #3.45 (#4 outside the UK).

We are working on even more powerful versions of these programs which will include many extra Microdrive-related features such as sequential FILE handling (for Pascal) and easy assembly from Microdrive to Microdrive in one step, We shall be announcing these versions when they become available.

The remainder of this document details the Microdrive extensions to HiSoft Pascal Version 1.6M and HiSoft Devpac Version 3.2M that are currently (from 7 March 1984) available.

HiSoft DEVPAC VERSION 3.2M

Loading and Making Back-up Copy.

The loading procedure for both GENS3M2 and MDWS3M2 is identical to that given in the Devpac 3 Programmer's manual except that you should allow a little more memory space for both packages. Once you have loaded either GENS3M2 or MONS3M2 (without executing them) you can make a back-up copy to either cassette tape or Microdrive cartridge as follows:

GENS3M2

SAVE "GENS3M2" CODE XXXXX,10034 to cassette

SAVE *"M";1;"GENS3M2" CODE XXXXX,10034 to Microdrive

MONS3M2

SAVE "MONS3M2" CODE XXXXX,6068 to cassette

SAVE *"M";1;"MONS3M2" XXXXX,6068 to Microdrive

where: XXXXX is the address at which you loaded the program.

Added Features

There are no added Microdrive-related-features within MONS3M2, the code has simply been altered so that it is possible to single-step code that uses the Interface 1 ROM, Please note that it in not possible to single-step the Interface 1 ROM itself.

One feature that has been added is related to the action taken when a breakpoint is encountered. Previously, when a breakpoint was found, a break was made immediately to the Front Pane! display; in Version 3.2M, when a breakpoint is found, a short tone is sounded through the SPECTRUM'S loudspeaker and execution pauses, waiting for you to hit any key - when you hit a key the Front Panel display is entered.

Also it is now not necessary to have the same value in the Memory Pointer and the Program Counter when single-stepping.

The extra features available in GENS3M2 (compared with GENS3) are as follows:

The Editor 'P' and 'B' commands.

These commands remain identical for 'P'utting or 'G'etting text to and from cassette tape but have been modified so that, if the first 2 characters of the filename consist of a digit (1-8 inclusive) followed by a colon ':', then the text will be stored to or retrieved from the relevant Microdrive. Thus, to store lines 10 to 190 of a text-file to Microdrive 1 you simply use:

P1,190,1:TEST

and the text-file will be stored on Microdrive 1 with the name 'TEST'.

Note that, when 'P'utting, if the file already exists then you will be asked 'File Exists Delete (Y/N)?' - answer Y to delete the file and continue or any other key to return to the editor without deleting the file. If, when 'G'etting, the file does not exist then the message 'Absent' is displayed.

You must specify a filename if you wish to 'G'et a file from a Microdrive cartridge – you cannot load the first file since there isn't one!

If you do not have Microdrives attached to your system then you must ensure that the second character of any filename you use is not a colon.

The '*F' Assembler Command.

'*F' is used to 'include' text from a storage device while assembling. In GENS3M5 this command has been extended to allow you to include text from a Microdrive cartridge - again the method is simply to include a Microdrive number (1-8 inclusive) before the filename. On cassette you must use the 'T' editor command to dump out text that you wish to 'include' at a later stage and the use of '*F' automatically turns off all other assembler commands. On Microdrive these two restrictions can be relaxed - you can 'include' a text-file that was stored on cartridge using the 'F' command and all assembler commands are available while including. Thus, to include the text-file 'TEST' (see the example above) you might have something like this:

```
        LD   A,B
        CALL AOUT

*H  Include the  file  'TEST'  from Microdrive

        *F  1:TEST
```

*H Next Routine

Note that you must specify a filename when using ' *F' with Microdrives and, if the Included code contains an error, you must continue with the assembly — do not try to get back to the editor (via 'E') while including text from a Microdrive.

The Editor 'O' command.

The 'O' command has been added to allow you to dump out object code direct!y to cassette tape or microdrive cartridge. The format of this command is simply O,, filename where the filename can be up to 8 characters in length. If the filename begins with a number followed by a colon ':' then the object code will be saved to the Microdrive specified by the number preceding the colon; otherwise the object code is saved to cassette tape.

You should note that only the last block of code produced by GENS3M2 can be saved in this way i.e. if you have more than one ORG directive in your program then only the code produced after the last ORG is saved.

Code must have been produced in the memory of the SPECTRUM before it can be saved using 'O'.

The Editor 'V' command.

This command has been modified to display the current delimiter (a comma initially) as well as the default line numbers and strings. It is useful to know what the delimiter is if you have inadvertently changed it using the 'S' command.

Extended CATalogue.

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

Type of files:
D - for a data or print-type file
P - for a program file
B - for a CODE-file
S - for a string array
N - for a numeric array

Also, for a CODE file, its length and start address are displayed in decimal while, for a program file, it's length and auto-execution address are displayed, in decimal.

To use the program either type in the hex codes directly (using MONS3M2 or POKE within BASIC) or use GENS3M2 (or another assembler, heaven forbid!) to assemble the source of the program. Note that the program is current!y positioned at location 60000 - you may of course change this if you re-assemble.

Once the code is in the computer you may execute from within BASIC or patch it into the editor in GENS3M2.

To use it from BASIC, first POKE the Microdrive number you with to catalog into location 60345 (assuming the code starts at 60000) and then use RANDOMIZE USR 60000.

To use the code from within GENS3M2 you must patch the dummy 'Z' command to jump to the CATalogue code. To do this patch, load up GENS3M2 and then POKE the address of the extended CATalogue routine + 2 into locations Start of GENS3M2 + 7790 and Start of GENS3M2 + 7791 (low order byte first). For example, say you have loaded GENS3M2 at 26000 and the extended CATalgue routine is at 60000. Then to effect the patch simply do P0KE 33790,98 and POKE 33791,234 and then enter GENS3H2 in the normal way.

You can now use 2n from within the editor to CATalogue drive number n.

Below it the lifting of the extended CATalogue programs:

```
                        1 #H Microdrive Extended CATalogue        28 February 1984
                        2
                        3 ;Produced with Hisoft's GEN assembler.
                        4
                        5 ;Copyright Hisoft 1984
                        6 ;With many thanks to Andrew Pennell for his
                        7 ;Microdrive 'bible' and the Stream 14 routine.
                        8
                        9 ;Equates
                       10
000D                   11 CR      EQU  13
                       12
0002                   13 PRINT   EQU  2
0043                   14 RECFLG  EQU  67
                       15
15D4                   16 WAIT_K  EQU  #15D4
1655                   17 MAKE_S  EQU  #1655
                       18
5C16                   19 STRMS   EQU  #5C16
5C3A                   20 ERR_NR  EQU  #5C3A
5C4F                   21 CHANS   EQU  #5C4F
5C53                   22 PROG    EQU  #5C53
                       23
5CD6                   24 D_STR1  EQU  #5CD6
5CD8                   25 S_STR1  EQU  #5CD8
5CDA                   26 N_STR1  EQU  #5CDA
5CDC                   27 T_STR1  EQU  #5CDC
5CE6                   28 HD_00   EQU  #5CE6
5CE7                   29 HD_0B   EQU  #5CE7
5CE9                   30 HD_0D   EQU  #5CE9
5CEB                   31 HD_0F   EQU  #5CEB
5CED                   32 HD_11   EQU  #5CED
                       33
0022                   34 OPEN_M  EQU  #22
0023                   35 CLOSE_  EQU  #23
0031                   36 NEWVAR  EQU  #31
0032                   37 SHADOW  EQU  #32
                       38
                       39 ;For GENS3M2 only.
                       40
1A0C                   41 NUMlof  EQU  7297
                       42
                       43
EA60                   44         ORG  60000
                       45
                       46 ;BASIC entry point
                       47
EA60 1817              48         JR   BasEnt
                       49
                       50 ;GENS3M2 entry point
                       51
EA62 E1                52         POP  HL
EA63 54                53         LD   D,H
EA64 5D                54         LD   E,L
EA65 E5                55         PUSH HL
EA66 21811C            56         LD   HL,NUMlof
EA69 19                57         ADD  HL,DE
EA6A 7E                58         LD   A,(HL)
EA6B 23                59         INC  HL
EA6C B6                60         OR   (HL)
EA6D 2002              61         JR   NZ,NumSet
EA6F 3E01              62         LD   A,1
EA71 E60F              63 NumSet  AND  %00001111
EA73 2600              64         LD   H,0
EA75 6F                65         LD   L,A
EA76 2289EB            66         LD   (DRIVE),HL
                       67
                       68 ;Initialise
                       69
EA79 210AEC            70 BasEnt  LD   HL,SPACE
EA7C 2208EC            71         LD   (POINTER),HL
EA7F FD213A5C          72         LD   IY,#5C3A
                       73
EA83 21C5EB            74         LD   HL,SIGNON
EA86 0634              75         LD   B,SIGNEND-SIGNON
EA88 CD9AEB            76         CALL WRstring
                       77
                       78 ;Set up new channel and attach to Stream 14
                       79
EA8B 2A535C            80         LD   HL,(PROG)
EA8E 2B                81         DEC  HL
EA8F E5                82         PUSH HL
EA90 010B00            83         LD   BC,11
EA93 CD5516            84         CALL MAKE_SPACE
EA96 2191EB            85         LD   HL,CH14out
EA99 D1                86         POP  DE
EA9A D5                87         PUSH DE
EA9B EB                88         EX   DE,HL
EA9C 73                89         LD   (HL),E
EA9D 23                90         INC  HL
EA9E 72                91         LD   (HL),D
EA9F 23                92         INC  HL
EAA0 EB                93         EX   DE,HL
EAA1 21FDEB            94         LD   HL,CI4INFO
EAA4 010900            95         LD   BC,11-2
EAA7 EDB0              96         LDIR
EAA9 E1                97         POP  HL
EAAA 23                98         INC  HL
EAAB ED4B4F5C          99         LD   BC,(CHANS)
EAAF B7               100         OR   A
EAB0 ED42             101         SBC  HL,BC
EAB2 22325C           102         LD   (STRMS+2B),HL
                      103
                      104 ;Now read in bare catalogue
                      105
EAB5 D9               106         EXX
EAB6 E5               107         PUSH HL
EAB7 D9               108         EXX
                      109
EAB8 CF               110         RST  8
EAB9 31               111         DEFB NEWVARS
                      112
```

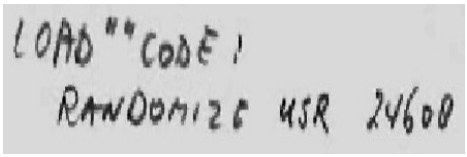
```
EABA 3E0E      113         LD   A,14
EABC 32DB5C    114         LD   (S_STR1),A
EABF 2AB9EB    115         LD   HL,(DRIVE)
EAC2 22D65C    116         LD   (D_STR1),HL
EAC5 2A06EC    117         LD   HL,(CAT)
EAC8 22ED5C    118         LD   (HD_11),HL
EACB FB        119         EI
EACC CF        120         RST  8
EACD 32        121         DEFB SHADOW
               122
               123 ;Now process bare catalogue
               124
EACE 210AEC    125         LD   HL,SPACE
EAD1 060B      126         LD   B,11
EAD3 CD9AEB    127         CALL WRstring
EAD6 060F      128 CatL1   LD   B,15
EAD8 C5        129 CatLoo  PUSH BC
EAD9 23        130         INC  HL
EADA 7E        131         LD   A,(HL)
EADB FE0D      132         CP   CR          ;finished?
EADD 2879      133         JR   Z,CatEnd
EADF 2B        134         DEC  HL
EAE0 060B      135         LD   B,11
EAE2 E5        136         PUSH HL
EAE3 CD9AEB    137         CALL WRstring
EAE6 E3        138         EX   (SP),HL
EAE7 23        139         INC  HL
EAE8 EB        140         EX   DE,HL
EAE9 21DC5C    141         LD   HL,T_STR1
EAEC 73        142         LD   (HL),E
EAED 23        143         INC  HL
EAEE 72        144         LD   (HL),D
EAEF 2AB9EB    145         LD   HL,(DRIVE)
EAF2 22D65C    146         LD   (D_STR1),HL
EAF5 210A00    147         LD   HL,10
EAF8 22DA5C    148         LD   (N_STR1),HL
EAFB FB        149         EI
EAFC CF        150         RST  8
EAFD 22        151         DEFB OPEN_M
EAFE CDB5EB    152         CALL Space
EB01 DDCB4356  153         BIT  PRINT,(IX+RECFLG)
EB05 2007      154         JR   NZ,NotPrint
EB07 3E44      155         LD   A,"D"
EB09 CDACEB    156         CALL CONOUT
EB0C 1841      157         JR   CatBack
EB0E DDE5      158 NotPri  PUSH IX
EB10 D1        159         POP  DE
EB11 215200    160         LD   HL,82
EB14 19        161         ADD  HL,DE
EB15 EB        162         EX   DE,HL
EB16 1A        163         LD   A,(DE)
EB17 13        164         INC  DE
EB18 21F9EB    165         LD   HL,TYPETAB
EB1B 4F        166         LD   C,A
EB1C 0600      167         LD   B,0
EB1E 09        168         ADD  HL,BC
EB1F 7E        169         LD   A,(HL)
EB20 CDACEB    170         CALL CONOUT
EB23 CDB5EB    171         CALL Space
EB26 EB        172         EX   DE,HL
EB27 79        173         LD   A,C
EB28 B7        174         OR   A
EB29 2013      175         JR   NZ,NotProg
EB2B 23        176         INC  HL
EB2C 23        177         INC  HL
EB2D 23        178         INC  HL
EB2E 23        179         INC  HL
EB2F 5E        180         LD   E,(HL)
EB30 23        181         INC  HL
EB31 56        182         LD   D,(HL)
EB32 23        183         INC  HL
EB33 CD66EB    184         CALL DEOUTS
EB36 5E        185         LD   E,(HL)
EB37 23        186         INC  HL
EB38 56        187         LD   D,(HL)
EB39 CD66EB    188         CALL DEOUTS
EB3C 1811      189         JR   CatBack
EB3E FE03      190 NotPro  CP   3
EB40 200D      191         JR   NZ,CatBack
EB42 5E        192         LD   E,(HL)
EB43 23        193         INC  HL
EB44 56        194         LD   D,(HL)
EB45 23        195         INC  HL
EB46 CD66EB    196         CALL DEOUTS
EB49 5E        197         LD   E,(HL)
EB4A 23        198         INC  HL
EB4B 56        199         LD   D,(HL)
EB4C CD66EB    200         CALL DEOUTS
EB4F CF        201 CatBac  RST  8
EB50 23        202         DEFB CLOSE_M
EB51 E1        203         POP  HL
EB52 C1        204         POP  BC
EB53 1083      205         DJNZ CatLoop
EB55 C3D6EA    206         JP   CatL1
               207
EB58 C1        208 CatEnd  POP  BC
EB59 2B        209         DEC  HL
EB5A 0605      210         LD   B,5
EB5C CD9AEB    211         CALL WRstring
EB5F D9        212         EXX
EB60 E1        213         POP  HL
EB61 D9        214         EXX
EB62 010000    215         LD   BC,0
EB65 C9        216         RET
               217
               218 ;Output DE in decimal
               219
EB66 E5        220 DEOUTS  PUSH HL
EB67 DDE5      221         PUSH IX
EB69 EB        222         EX   DE,HL
EB6A 0605      223         LD   B,5
EB6C DD21B8EB  224         LD   IX,TENTAB
EB70 DD5E00    225 DEloop  LD   E,(IX)
EB73 DD5601    226         LD   D,(IX+1)
EB76 3EFF      227         LD   A,-1
EB78 3C        228 TenLoo  INC  A
```

```
EB79 B7          229         OR   A
EB7A ED52        230         SBC  HL,DE
EB7C 30FA        231         JR   NC,TenLoop
EB7E 19          232         ADD  HL,DE
EB7F F630        233         OR   #30
EB81 CDACEB      234         CALL CONOUT
EB84 DD23        235         INC  IX
EB86 DD23        236         INC  IX
EB88 10E6        237         DJNZ DEloop
EB8A CDB5EB      238         CALL Space
EB8D DDE1        239         POP  IX
EB8F E1          240         POP  HL
EB90 C9          241         RET
                 242
                 243 ;Output for Stream 14
                 244
EB91 2A0BEC      245 CH14ou  LD   HL,(POINTER)
EB94 77          246         LD   (HL),A
EB95 23          247         INC  HL
EB96 220BEC      248         LD   (POINTER),HL
EB99 C9          249         RET
                 250
                 251 ;Write a string of length B from (HL)
                 252
EB9A 7E          253 WRstri  LD   A,(HL)
EB9B CDACEB      254         CALL CONOUT
EB9E 23          255         INC  HL
EB9F 10F9        256         DJNZ WRstring
EBA1 C9          257         RET
                 258
                 259 ;Open a stream
                 260
EBA2 E5          261 ChOpen  PUSH HL
EBA3 D5          262         PUSH DE
EBA4 C5          263         PUSH BC
EBA5 CD0116      264         CALL #1601
EBA8 C1          265         POP  BC
EBA9 D1          266         POP  DE
EBAA E1          267         POP  HL
EBAB C9          268         RET
                 269
                 270 ;Output to stream 2
                 271
EBAC F5          272 CONOUT  PUSH AF
EBAD 3E02        273         LD   A,2
EBAF CDA2EB      274         CALL ChOpen
EBB2 F1          275         POP  AF
EBB3 D7          276         RST  #10
EBB4 C9          277         RET
                 278
                 279 ;Output a space to stream 2
                 280
EBB5 3E20        281 Space   LD   A," "
EBB7 18F3        282         JR   CONOUT
                 283
                 284
                 285 ;Storage
                 286
```

```
EBB9 0100        287 DRIVE   DEFW 1
                 288
EBBB 1027E803    289 TENTAB  DEFW 10000,1000,100,10,1
     64000A00
     0100
                 290
EBC5 0D          291 SIGNON  DEFB CR
EBC6 4869736F    292         DEFM "Hisoft Extended CAT Listing"
     66742045
     7874656E
     64656420
     43415420
     4C697374
     696E67
EBE1 0D          293         DEFB CR
EBE2 436F7079    294         DEFM "Copyright Hisoft 1984"
     72696768
     74204869
     736F6674
     20313938
     34
EBF7 0D0D        295         DEFB CR,CR
EBF9             296 SIGNEN  EQU  $
                 297
EBF9 504E5342    298 TYPETA  DEFM "PNSB"
                 299
EBFD C415        300 C14INF  DEFW #15C4
EBFF 5A          301         DEFB "Z"
EC00 28002800    302         DEFW #28,#28,11
     0B00
                 303
                 304 ;***this value may change with new Interface 1 ROM***
                 305
EC06 581C.       306 CAT     DEFW #1C58
                 307
                 308 ;*************************************************
                 309
                 310
EC08             311 POINTE  DEFS 2
EC0A             312 SPACE   DEFS 512
                 313

Pass 2 errors: 00
```

HiSoft PASCAL VERSION 1.6M



Loading and Making a Back-up Copy.

To load Pascal 1.6M into the Spectrum simply use 'LOAD "" (LOAD is on the 'J' key) and start your tape recorder. Once loaded the program will run automatically and you will be prompted with 'Top of RAM?' - refer to Section 0 of the Programmer's Manual for details of how to answer this and the two following questions. Normally, pressing ENTER in response to the prompts will suffice.

Once you are in the editor mode (there will be a '>' prompt on the left hand side of the screen} you can make a back-up copy of the Pascal by using the 'B'ye command to return to BASIC and then either!

SAVE "HP4TM16" CODE 24598,19736 for cassette tape

or

SAVE *"M";"1";"HP4TM16" CODE 24598,19736 for Microdrive

This will save a configured version of the Pascal which you can load back into the machine simply using LOAD "" CODE or LOAD *"M";"1";"HP4TM16" CODE. Once you have re-loaded it in this way then you must enter HP4TM16 via either RANDOMIZE USR 24598 (for a cold start, destroying any text) or RANDOMIZE USR 24603 (for a warm start which preserves any existing text. You may re-enter the Pascal from BASIC using these 'addresses at any time.

You should note that HiSoft Pascal Version 1.6M does not have a BASIC loader and thus any reference to this loader in your Implementation Note should be disregarded. Specifically, to re-enter HiSoft Pascal at any time, you do not GOTO 9 or GOTO 12 as in the Implementation Mode} instead you use the cold and warm start addresses given above. In addition you should ignore the instructions on, making a back-up copy given in the Implementation Note, use those above instead.

Also, because there is no BASIC interface program, the cassette commands 'P' and 'B' take effect immediately and display 'Busy..' as stated in the HiSoft Pascal Programmer's Manual.

We now look at the extra features that have been implemented in HiSoft Pascal Version 1.6M.

Firstly note that, on any return to BASIC from within HP4TM16, the display will first pause, waiting for you to hit any key. This has been added so that runtime error messages etc.. may be inspected when running a Translated program.

The Editor 'V' Command.

This command has been modified to display the current delimiter (initially a comma ",") as well as the default line numbers and strings.

The Editor F* and *G' commands.

These commands remain identical - For 'P'utting or 'G'etting text to and from cassette

tape but have been modified so that, if the first 2 characters of the filename consist of a digit (1-0 inclusive) followed by a colon :, then the text will be stored to or retrieved from the relevant Microdrive. Thus, to store lines 10 to 190 of a text file to Microdrive 1 you simply use:

P10,190,1:TEST

and the text-file will be stored on Microdrive1 with the name 'TEST'.

Note that, when 'P'utting, if the file already exists then it is automatically deleted whereas if, when 'G'etting, the file does not exist then the message 'Absent' is displayed. You must specify a filename if you wish to 'G'et a file from a Microdrive cartridge – you cannot land the first file since there isn't one!

If you do not have Microdrives attached to your system then you must ensure that the second character of any filename you use is not a colon.

The 'F' Compiler Option.

While compiling, you may 'include' a text-file from either cassette tape or from a Microdrive by using the '$F' compiler option.

The format of this option when used to compile a file from cassette tape is explained in Section 3 of the HiSoft Pascal Programmer s Manual. When used with Microdrives, the only difference is that the first two characters of the filename should be a Microdrive number (1-8 inclusive) followed by a colon ':' .

Thus to include the file 'TEST', stored to Microdrive in the example above, you could use the following;

```
10 PROGRAM A;
20 BEGIN
30
40  ($F 1:  TEST    )
50
60 END.
```

Note that, to include a file from cassette tape, you must have used the 'W' editor command to dump out the text in a blocked format whereas, with Microdrives, you include a text-file saved using the normal 'P'ut command.

Do not try to return to the editor (via 'E' or 'P') if there is a compilation error in a file that you are including - you must continue the compilation.

TIN and TOUT.

Data structures may be saved to cassette or Microdrive using the TIN procedure from within a Pascal program and retrieved using TOUT - see Sections 2.3.5.8 and 2.3.5.9 of the HiSoft Pascal Programmer's Manual for details. To direct data to or from Microdrives simply start the required filename with a Microdrive number (1-8 inclusive) and a colon ':' e.g.

```
10  PROGRAM A;
20  VAR  A,B : ARRAY[1..20]  OF CHAR;
30  BEGIN
40  (some code to set up the array A)
```

```
50
60   T0UT ('1:ARRA   ', ADDR(A), SIZE (A));
70
8O   (some more program)
90
100  TIN ('1:ARRA  ',ADDR(B));
110 (etc,  etc. }
```

Remember that the number of characters within the ' ' in TIN or TOUT must be exactly 8.

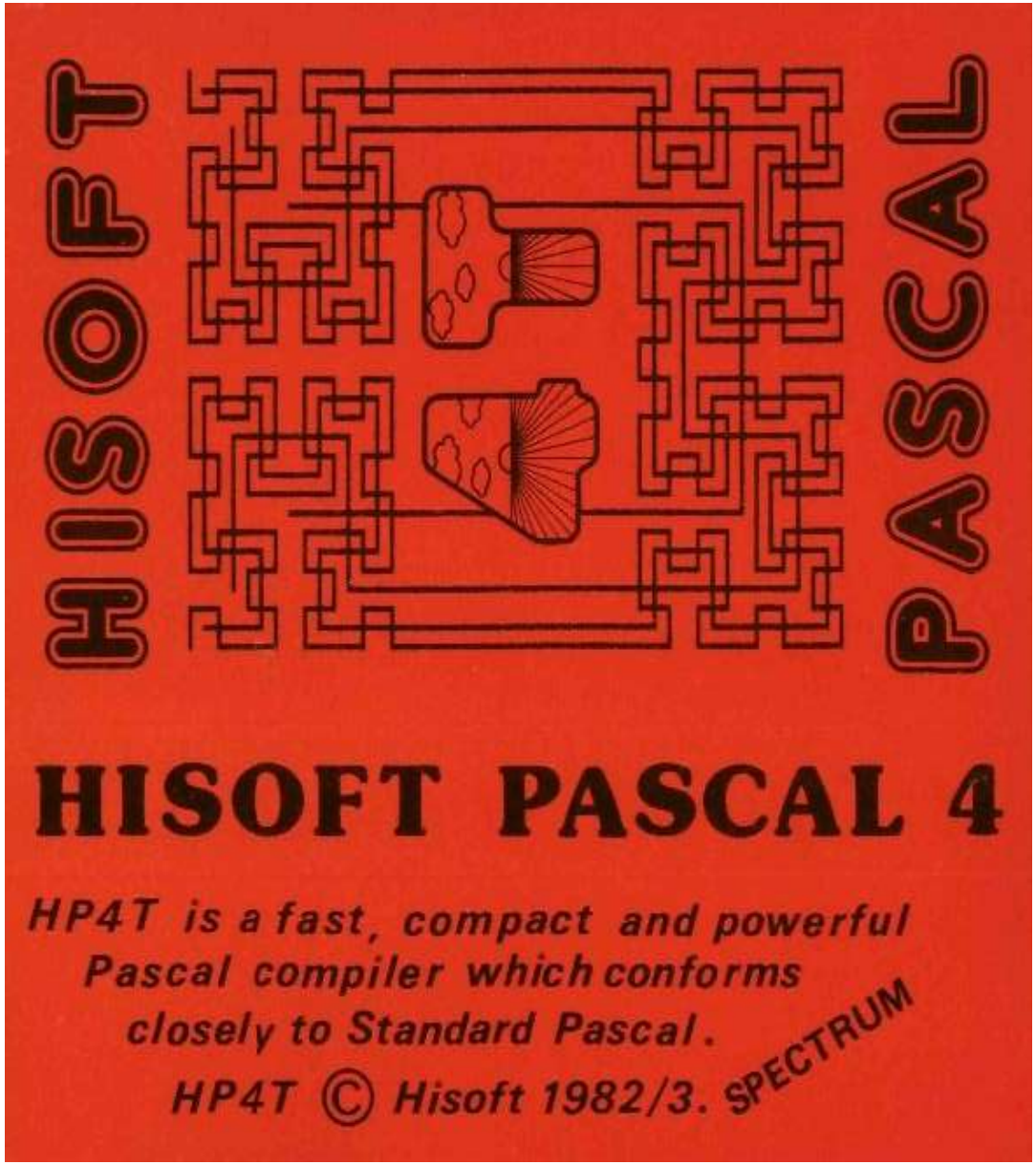
'T'ranslation of Object Code.

The editor *T' command can be used to 'translate' the Pascal object code so that it may be stored on cassette tape or Microdrive. Remember that, when translating, the compiler is deleted and the run-times together with the object code are saved to the storage device Thus you must reload the compiler after a translation if you wish to compile some more text - however this is not the usual practice since you will only save complete and tested object code.

To translate the object code to cassette use the command as specified in Section 4.2.5 of the HiSoft Pascal Programmer's Manual. To reload the object code from tape simply use LOAD "" CODE and then, to execute it, use RANDOMIZE USR 24608.

To translate the object code to Microdrive use the same 'T' command as in Section 4.2.5 but put a Microdrive number and a colon at the front of the filename e.g. T1,9999,1:OBJ . The run-times and object will then be stored on the relevant Microdrive as a CODE file. To reload this file from within BASIC, simply use LOAD *"M";1;"OBJ" CODE and then execute it by RAMDOMIZE USR 24608.

Versions 1.6M and 3.2M
August 1984


HISOFT
PASCAL
HISOFT PASCAL 4
HP4T is a fast, compact and powerful
Pascal compiler which conforms
closely to Standard Pascal.
HP4T © Hisoft 1982/3.
SPECTRUM

HiSoft Pascal

The Pascal Compiler for your ZX Spectrum

More High Quality Software for the ZX Spectrum

HiSoft

HiSoft
High Quality Software

HiSoft

HiSoft Pascal **for the ZX Spectrum**

HiSoft Pascal is a full-featured, fast, easy-to-use and standard implementation of the Pascal programming language.

Key features:

- All the powerful control structures of Pascal: CASE, REPEAT...UNTIL, IF...THEN...ELSE, WHILE...DO etc.
- Full range of data structures including records, sets, arrays, pointers, integer and floating-point arithmetic.
- Procedures and functions may be fully recursive with value *and variable parameters.*
- Compiles at very high speed and generates phenomenally fast code which can be executed independently.
- Full support of the Spectrum including Logo-style Turtle Graphics for easy production of graphic displays.
- Uses only 20K of memory allowing large programs to be created, especially if the source is compiled from tape/disk.
- Integrated editor with optional 51-column mode.
- Easy interface with the Spectrum's ROM through the use of PEEK, POKE, USER and INLINE.

HiSoft Pascal has been developed over many years and is a highly tuned and stable product. It is equally ideal for the student who wishes to supplement his/her coursework at home, the professional programmer who can produce commercial programs with the package and, in fact anyone who wants to learn or use this very popular, structured programming language.

HiSoft Pascal requires:

- Any ZX Spectrum computer. Special +3 version is available
- A tape recorder, microdrive or Disciple/Opus disk system

This package contains:

- 80 page manual detailing every aspect of the HiSoft Pascal implementation including many example programs
- ☑ 1 cassette (also suitable for microdrives)
- ☐ 1 Disciple-format diskette
- ☐ 1 Opus Discovery-format diskette

HiSoft
The Old School
Greenfield, Bedford
MK45 5DE UK
Tel: (0525) 718181
Fax: (0525) 713716

HiSoft
High Quality Software

HP4T

HiSoft PASCAL 4 V1.5 (1983)
21 January 1985

PRODUCT APPLICATION NOTE PS 1.1

ZX SPECTRUM SOUND AND GRAFICS WITH HiSoft PASCAL 4T

This note gives details on controlling the sound and graphics capabilities of the ZX SPECTRUM using Pascal procedures from within HiSoft Pascal 4T.

1. Sound.

The following two procedures (defined in the order given below) are required to produce sound with HP4T.

```
(*This procedure uses machine code to pick up its parameters and then passes them to the BEEP routine
within the SPECTRUM ROM.*)

PROCEDURE BEEPER (A, B : INTEGER);

 BEGIN
  INLINE(#DD, #6E, 2, #DD, #66, 3, (*LD L,(IX+2) : LD H,(IX+3)*)
         #DD, #5E, 4, #DD, #56, 5, (*LD E,(IX+4) : LD D,(IX+5)*)
         #CD, #B5, 3, #F3)         (*CALL #3B5   : DI *)
 END;

(*This procedure traps a frequency of zero which it converts into a period of silence. For non-zero
frequencies the frequency and length of the note are approximately converted to the values required by
the SPECTRUM ROM routine and this is then called via BEEPER.*)

PROCEDURE BEEP (Freq : INTEGER; Length : REAL);

VAR  I : INTEGER;

 BEGIN
  IF Freq=0 THEN FOR I:=1 TO ENTIER(12000*Length) DO

  ELSE BEEPER(ENTIER(Freq*Length),ENTIER(437500/Freq-30.125))

  FOR I:= 1 TO 100 DO         (*short delay between notes*)
 END;
```

Example of the use of BEEP:

```
BEEP ( 262, 0.5 );      (*sounds middle C for 0.5 seconds*)
BEEP ( 0, 1 );          (*followed by a one second silence.*)
```

2. Graphics.

Three graphics procedures are given: the first plots a given (X,Y) co-ordinate whilst the second and third are used to draw lines from the current plotting position to a new position which is defined relative to the current plot position and which then becomes the current plot position.

Both PLOT and LINE take a BOOLEAN variable, ON which, if TRUE, will cause any point to be plotted regardless of the state of the pixel in that plot position or, if FALSE, will cause any pixel already present at the plot position to be flipped i.e. if on it becomes
****

There are occasions where it is useful to output directly through the SPECTRUM ROM RST #10 routine rather than use WRITE(LN). For example, when using the PRINT AT control code
- this code should be followed by two 8 bit values giving the (X,Y) co-ordinate to which the print position is to be changed. However, if this is done using a Pascal WRITE statement then
certain values of X and Y (e.g. 8 which is interpreted by HP4T as BACKSPACE) will not be passed to the ROM and thus the print position will not be correctly modified.

You can overcome this problem by using the following procedure:
****

```
(*SPOUT outputs the character passed as a parameter directly

WYGRUVa the 'ENTER' key.
CC      via CAPS SHIFT and 1.          EDIT
CH      DELETE i.e. CAPS SHIFT and 0.   DELETE
CI      via CAPS SHIFT and 8.          ==>
CP      via CHR(16) in a WRITE or WRITELN statement.
CX      via CAPS SHIFT and 5.          <==
CS      via CAPS SHIFT and SPACE.
```

The ZX SPECTRUM keyword entry scheme is not supported (we see this as a positive advantage), instead all text must be inserted using the normal alphanumeric keys. Using SYMBOL SHIFT and key (except I) will always reach the ASCII symbol associated with that key and not the keyword e.g. SYMBOL SHIFT T gives '>' and SYMBOL SHIFT G gives '}'. You
must not use the single symbols <=, <> and >=; instead these should be entered as a combination of the symbols <, > and =.

The editor comes up in upper case mode, this may be toggled in the normal way using CAPS SHIFT and 2.

You have control over the temporary attributes of the various character position on the screen through the use of the standard control codes (e.g. WRITE(CHR(17),CHR(4)) will make the 'paper' green) but you cannot change the permanent attributes. If, while using these control codes, an invalid sequence is detected then the message 'System Call Error' will be displayed and the execution aborted. When you dump out text or object code to tape the message 'Start tape, then press any key' will be displayed twice - you must respond to it each time.

There is no need to save the loader since an automatic loader is always dumped with the object program - if you have used the Translate command to save the code and run-times on Tape then to load and run the program simply enter 'LOAD""' from within BASIC. After the execution of the object code has finished you can run it again, assuming it has not corrupted anything, by entering 'GOTO 9' to perform a warm start i.e. preserving the Pascal program or 'GOTO 12' to do a cold start, re-initialising the Pascal and clearing any existing Pascal text.

The ZX Printer is supported via the use of the compiler 'P' option (see Section 3.2 of the Programmer's Manual) and via CHR(16) in a WRITE or WRITELN statement. Note that, as a result, you cannot use CHR(16) within a WRITE(LN) statement to specify the INK colour - instead you can use CHR(15) to set the INK.

To make a working copy of HP4S proceed as follows:

2.1. Load up the BASIC interface. HiSoft Pascal 4 on the SPECTRUM is in two parts, an interface to BASIC and the compiler, run-times etc. proper. The BASIC interface is recorded first on the master tape, followed by the code of the compiler, run-times and editor. Normally the BASIC interface program automatically loads the rest of the code and then auto-starts the compiler. We must prevent this. So, load the master HP4S cassette into your tape recorder, type LOAD"" (simply J"" using keyword entry) and PLAY the tape; when the SPECTRUM finds the BASIC interface program header it will display 'Program: HP4S' and then load the interface program itself. After the interface program has been loaded the screen will go blank - the computer is now automatically searching for the compiler code. Press BREAK (simply the SPACE bar will do) to abort the search once the interface program has been loaded. Stop the tape and do not rewind it.

2.2. Load the code of the compiler, run-times and editor by typing LOAD "HP4S" CODE and restart the tape. Remember you use keyword entry for LOAD and CODE.

2.3. Now load a separate, blank tape into your tape recorder, you are going to record the new HP4S master on this. Type SAVE "HP4S" LINE 1 to save the BASIC interface program so that it automatically executes. Set your tape recorder into recording mode and proceed as normal to record on tape.

2.4. When the dump of the BASIC interface program has finished leave the tape recorder in recording mode and type SAVE "HP4S" CODE 24598,21105. This will save the compiler, run-times and editor on the tape.

2.5. You have now created a new master tape which you can use in exactly the same way as the tape supplied by HiSoft. You are advised to make one copy and then store the HiSoft master tape in a safe place, away from magnetic fields and damp.

Note that you are authorized by HiSoft to make only one working copy.

Please do not hesitate to contact us if you experience any difficulty with HiSoft Pascal 4 - we can only solve the problems if we know what they are!

HiSoft PASCAL 4 - VERSION 1.4

The release number of HiSoft Pascal 4D and HiSoft Pascal 4T is now 1.4, effective from October 1982.

The differences between the previous version of HiSoft Pascal and version 1.4 are given below:

1. A bug in the evaluation of expressions (which led to expressions like 1+SQR(2) being incorrectly evaluated) has been corrected.

2 A bug in the evaluation of expressions (which led to expressions like I+(I+1) being incorrectly evaluated) has been corrected.

3. A bug that caused the incorrect evaluation of the result of an integer-to-real comparison has been corrected.

4. **HiSoft Pascal 4T only:** the documented bug in the editor 'S' sub-command (see page 38 of the Programmer's Manual) has been corrected - 'S' can now be used at any time.

5. **HiSoft Pascal 4T only:** using the 'F' command from within the editor to find a character string now positions the editing cursor at the beginning of any found occurrence of the string of pages 37 and 38 of the Programmer's Manual.

6. **HiSoft Pascal 4T only:** a new editor command has been incorporated. The 'X' command displays, in hexadecimal, the current end address of the compiler. This information allows the user to make a working copy of the compiler and thus minimizes the danger of corrupting the master tape. ** Note that you are authorised by HiSoft to make only one working copy of HP4T.** To make a working copy, first find the start address of the package from your Implementation Note and then find the end address using the 'X' command. Now use the relevant operating system command to dump this block of memory to tape. Note that you will not use the HP4T loader to load the compiler etc. When saved in this way - instead you should use the relevant operating system load-from-tape command and then execute a cold start into the editor as given in your Implementation

Note.

Note: ZX SPECTRUM users cannot use the above method of making a copy and should consult their Implementation Note for details of making a working copy of HP4T.

As usual, HiSoft welcomes any queries concerning this note.

HiSoft PASCAL 4T - VERSION 1.5

As from 1 April 1983, the version number of HiSoft Pascal 4T will be 1.5. Owners of previous versions of HP4T may upgrade to version 1.5 by returning their old tape together with a handling charge of pound 3 + VAT. The differences between version 1.4 and version 1.5 are as follows:

1. Functions may now return a POINTER result.

2. A bug in the pre-defined procedure NEW which occasionally resulted in the incorrect allocation of memory for a dynamic variable has been fixed.

3. A new editor command has been incorporated. The 'V' command takes no arguments and simply displays the current default values of the line range, the Find string and the Substitute string. The current default line range is shown first followed by the two strings (which may be empty) on separate lines. It should be remembered that certain editor commands (like 'D' and 'N') do not use these default values but must have values specified on the command line - see the HP4T Programmers Manual.

4. **ZX SPECTRUM only** The 'Include' option ('$F' see Section 3.2 of the HP4T Programmer's Manual) is now available on the ZX SPECTRUM version of HP4T. If you wish to use this option then the source text that is to be subsequently 'included' **must** be dumped out using a new editor command, the 'W' command (SPECTRUM only). 'W' works exactly like 'P' except that it does not dump out in HP4T standard tape format and it begins to save the text to tape immediately the command line is terminated - thus you must have the tape recorder in RECORD mode before typing ENTER at the end of the command line. When
dumping text out using 'W' or reading text in using the 'F' compiler option, then you may use the BREAK key at any time; the use of this key will return you to the editor. Note that, if you wish to read in text from the editor via the editor's 'G' command, then the text must have been
dumped out using 'P' and **not** 'W'.

Example of use:

To write out a section of a program use:

```
W50,120,PLOT          ;write out the PLOT procedure.
```

To 'include' the section in another program:

```
100  END;
110
120  (*$F PLOT    'include' the PLOT procedure here.*)
130
140  PROCEDURE MORE;    (*the rest of program.*)
150
```

HiSoft hope that the above improvements and correction will enhance the use of HP4T and, as usual, we welcome any correspondence regarding this document.

C O N T E N T S

0 PRELIMINARIES.

0.0 Getting Started

HiSoft Pascal 4T (HP4T) is a fast, easy-to-use and powerful version of the Pascal language as specified in the Pascal User Manual and Report (Jensen/With Second Edition). Omissions from the specification are as follows:

FILEs are not implemented although variables may be stored on tape. A RECORD type may not have a VARIANT part. PROCEDUREs and FUNCTIONs are not valid as parameters.

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

The compiler occupies approximately 12K of storage while the run-times take up roughly 4K. Both are supplied on cassette tape in the tape format used by the run-times. All interfacing between HP4T and the host machine takes place through vectors conveniently placed at the start of the run-times (see HP4T Alteration Guide) - this makes it easy for the user to write his own customised I/O routines if necessary.

HiSoft Pascal 4T uses various control codes, mostly within the editor. Of course, different systems can have very different keyboard designs and thus will have different ways of reaching control codes. In this manual the control characters used will be refereed to as RETURN, CC, CH, CI, CP, CS and CX. The attached Implementation Note will tell you the corresponding keys for your system.

Whenever HP4T is waiting for a line of input, the control characters can be used as follows:

RETURN is used to terminate the line.
CC returns to the editor. CAPS SHIFT 1
CH deletes the last character typed. CAPS SHIFT 0
CI move to the next TAB position. CAPS SHIFT 8
CP directs output to the printer (if available)
or if output was going to the printer then
it returns to the screen. WRITE(CHR(16))
CX deletes the whole line typed so far. CAPS SHIFT 5

A simple loader is also supplied in the package so that the user can load, from tape, data which has been recorded in HP4T format.

Thus, to load the compiler and run-times from the master tape supplied by HiSoft, the user must first load the loader – when applicable this is supplied in a form suitable for loading by the user's operating system. If the user is unable to access the computer's operating system then the boot loader must be entered into the computer's memory directly either through the use of an assembler or a high level language such as BASIC - details of how to do this and a skeletal loader appear in the HP4T Alteration Guide.

Once the loader has been executed it will proceed to search for a file recorded in HP4T tape format. When a file of this format has been found the loader will load the file into memory. If at any stage an error is detected while reading the tape a message will be displayed - you must then rewind the tape to the beginning of the file and attempt to load it again. If you experience repeated errors then adjust the volume level on your tape recorder - if this is not successful then please return the tape to HiSoft and we will send you a replacement tape.

Thus the loader will automatically load the compiler and run-times into memory for you.

When the compiler has been successfully loaded it will execute automatically and produce the message:

```
Top of RAM?
```

You should respond to this by either entering a positive decimal number up to 65536 (followed by RETURN) or by hitting RETURN (See Implementation Note).

If you enter a number then this is taken to represent the highest RAM location + 1 otherwise the first non-RAM location is automatically computed. The compiler's stack is set to this value and thus you can reserve high memory locations (perhaps for extensions to the compiler) by deliberately giving a value less than the true top of RAM. In the ZX Spectrum version the 'true' top of RAM is taken to be start of the user-defined graphics area (UDG in the Sinclair manual).

You will then be prompted with:

```
Top of RAM for 'T'
```

Here you can enter a decimal number or default to the 'Top of RAM' value previously specified. What you enter will be taken as the stack when the resultant object code is executed after using the editor 'T' command (See Section 4 for details). You will need to define a runtime stack different from the top of RAM if, for example, you have written extensions to the run-times and wish to place them safely in high memory locations.

Finally you will be asked:

```
Table size?
```

What you enter here specifies the amount of memory to be allocated to the compiler's symbol table.

Again, either enter a positive decimal number followed by RETURN or simply RETURN by itself in which case a default value of (available RAM divided by 16) will be taken as the symbol table size. In nearly all cases the default value provides more than enough space for symbols. The symbol table may not extend above machine address #8000*(32768 decimal). If you specify so large value that this happens then you will be prompted again for 'Top of RAM' etc.

You may, optionally, include an 'E' before the number after this prompt - if you do so then the internal line editor will not be retained for use by the compiler. So do this if you wish to use your own editor with the compiler (see HP4T Alteration Guide for details on how to do this).

At this point the compiler and integral editor (if retained) will be relocated at the end of the symbol table and execution transferred to the supported editor.

**Note: throughout this Manual, the hash sign "#" (decimal 35, hexadecimal 23, shift '3') is replaced by the number sign on all systems which do not use U.K. ASCII. Numbers that are preceded by this symbol are in hexadecimal.

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 4.

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 4, from CONSTants to FUNCTIONs.

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

Section 4 shows how to use the line editor which is an integral part of HP4T; if you do not wish to use this editor but want to interface your own editor, then you should consult the HP4T Alteration Guide.

The above Sections should be read carefully by all users.

Appendix 1 details the error messages generated both by the compiler and the run-times.

Appendix 2 lists the predefined identifiers and reserved words.

Appendix 3 gives details on the internal representation of data within HiSoft Pascal 4 - 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 4 programs.

0.2 Compiling and Running.

For details of how to create, amend, compile and run an HP4T program using the integral line editor see Section 4 of this manual. For information on what to do if you are using your own editor see the HP4T Alteration Guide.

Once it has been invoked 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' if supported (see Section 3).

You may pause the listing at any stage by pressing CS; subsequently use CC to return to the editor or any 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 EDITOR to edit the line displayed, 'P' to return to the editor and edit the previous line (if it exists) 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 the editor.

If the compiler runs out of table space then the message 'No Table Space' will be displayed and control returned to the editor. Normally the programmer will then save the program on tape, re-load the compiler and specify a larger 'Table size' (see Section 0.0).

If the compilation terminates correctly but contained errors then the number of errors detected will be displayed and the object code deleted. If the compilation is successful then the message 'Run?' will be displayed; if you desire an immediate run of the program then respond with 'Y', otherwise control will be returned to the editor.

During a run of the object code various runtime error messages may be generated (see Appendix 1). You may suspend a run by using CS; subsequently use CC to abort the run 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 the 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 which distinguishes between numbers 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 4 uses name equivalence for RECORDs and ARRAYs. The consequences of this are clarified in Section 1 - let 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 4 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;
```

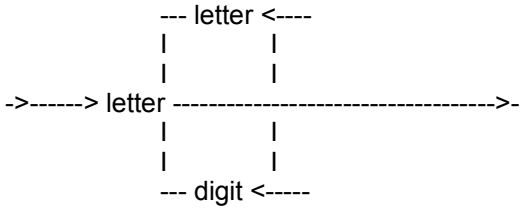
and now the user can freely assign A to B and vice versa since only one 'TYPE record' has been created.

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.

1 SYNTAX AND SEMANTICS.

This section details the syntax and the semantics of HiSoft Pascal 4 - unless otherwise stated the implementation is as specified in the Pascal User Manual and Report Second Edition (Jensen/Wirth).

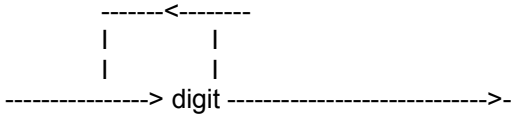
1.1 IDENTIFIER.



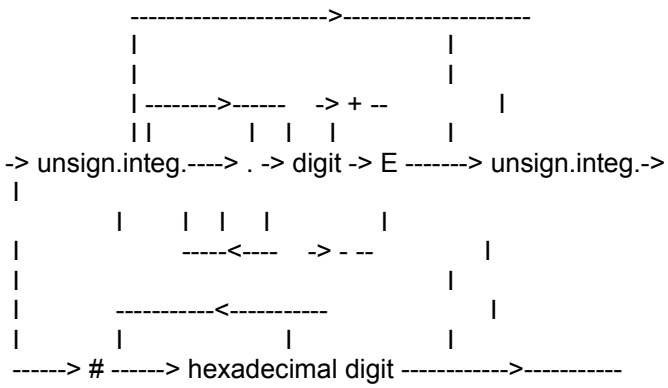
Only the first 10 characters of an identifier are treated as significant.

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 pre-definded 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 PASCAL 4. Larger whole numbers are treated as reals.

The mantissa of reals is 23 bits in length. The accuracy attained 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 argument 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.

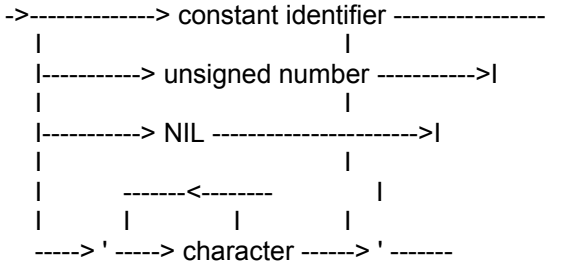
The largest real available is 3.4E38 while the smallest is 5.9E-39.

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.

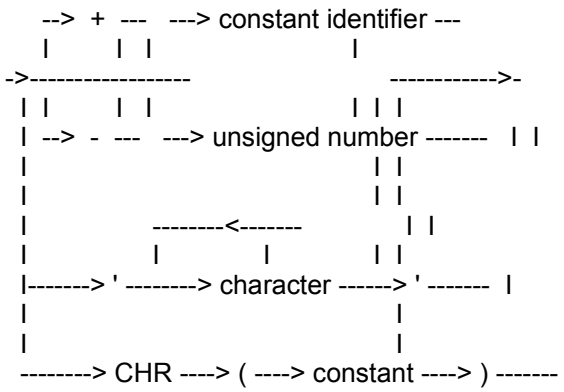
1.4 UNSIGNED CONSTANT.



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.

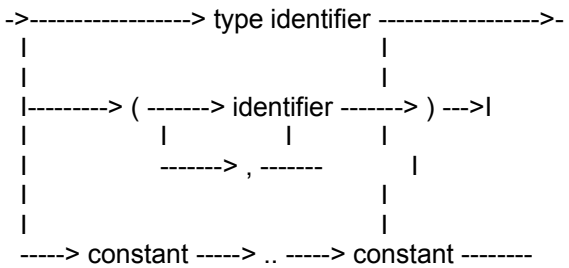
1.5 CONSTANT.



The non-standard CHR construct is provided here so that constants may be used for control characters. In this case the constant in parentheses must be of type integer.

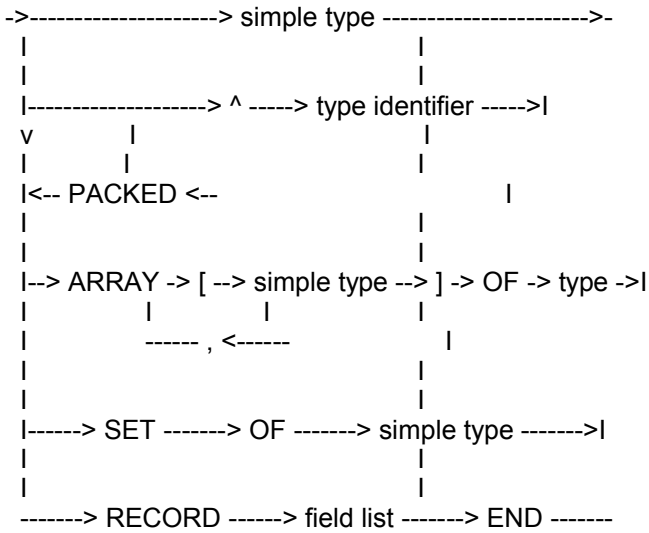
E.g. CONST bs=CHR(10);
cr=CHR(13);

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 is more 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 sub-ranges 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 4 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 4. 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 are 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.

1.7.4 RECORDs.

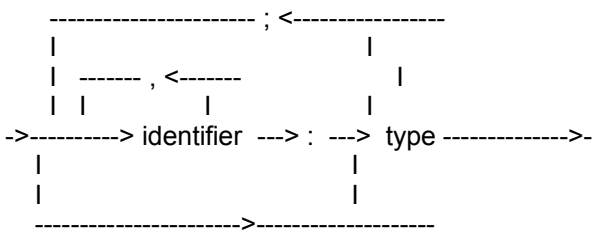
The implementation of RECORDs, structured variables composed of a fixed number of constituents called fields, within HiSoft Pascal 4 is as Standard Pascal except that the variant part of the field list is not supported.

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.

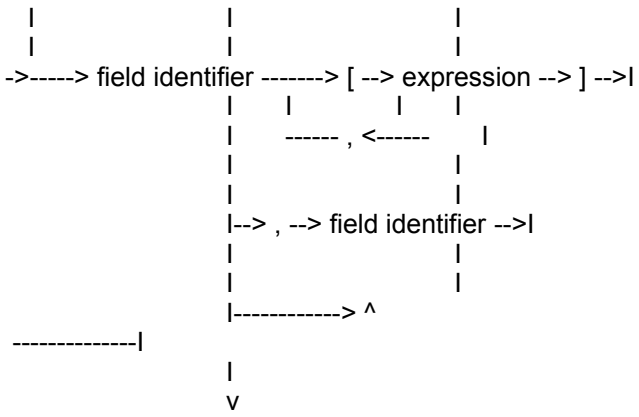
See Appendix 4 for an example of the use of WITH and RECORDs in general.

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 4; 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.

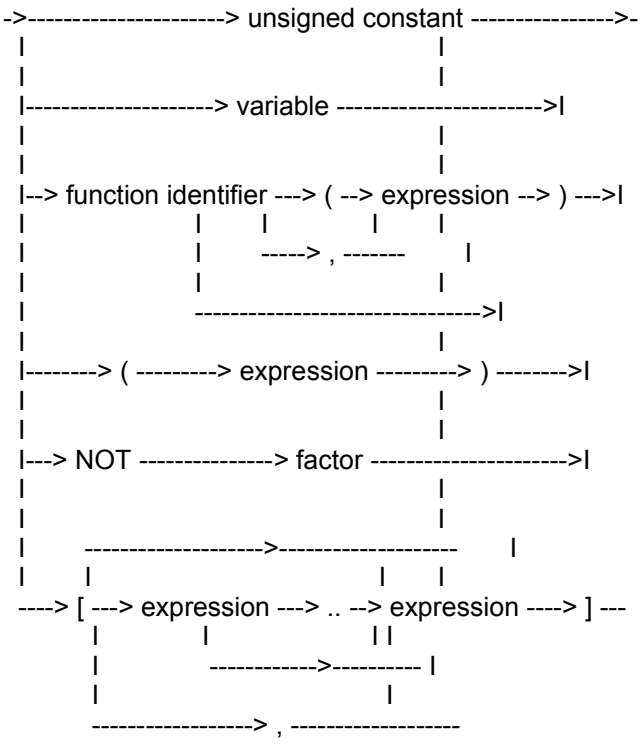
Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type pointer, which contains the address of the dynamic variable.

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 - this is a change from HiSoft Pascal 3.

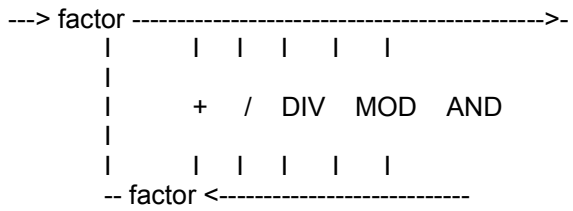
e.g. if a variable a is declared as ARRAY[1..10] OF ARRAY[1..10] OF INTEGER then either a[1][1] or a [1,1] may be used to access element (1,1) of the array.

1.10 FACTOR.



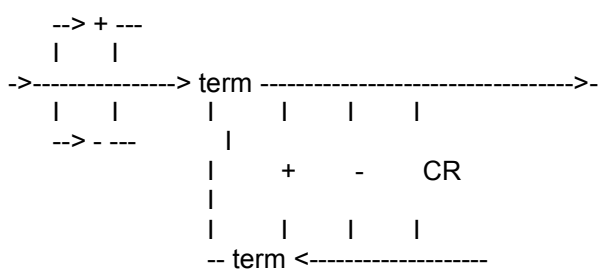
See EXPRESSION in Section 1.13 and FUNCTION in Section 3 for more details.

1.11 TERM.



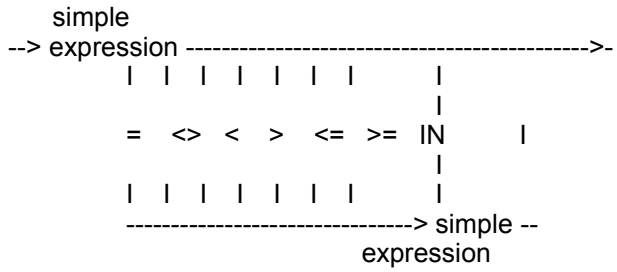
The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

1.12 SIMPLE EXPRESSION.



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

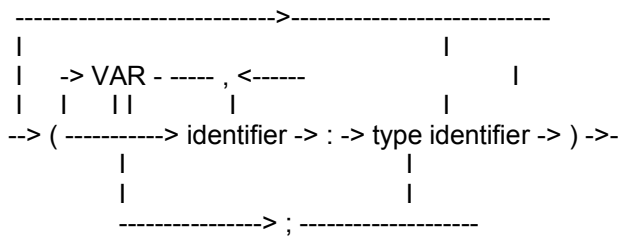
1.13 EXPRESSION.



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.

1.15 STATEMENT.

Refer to the syntax diagram on page below.

Assignment statements:

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).

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 the control is passed to the statement following the END.

FOR statements:

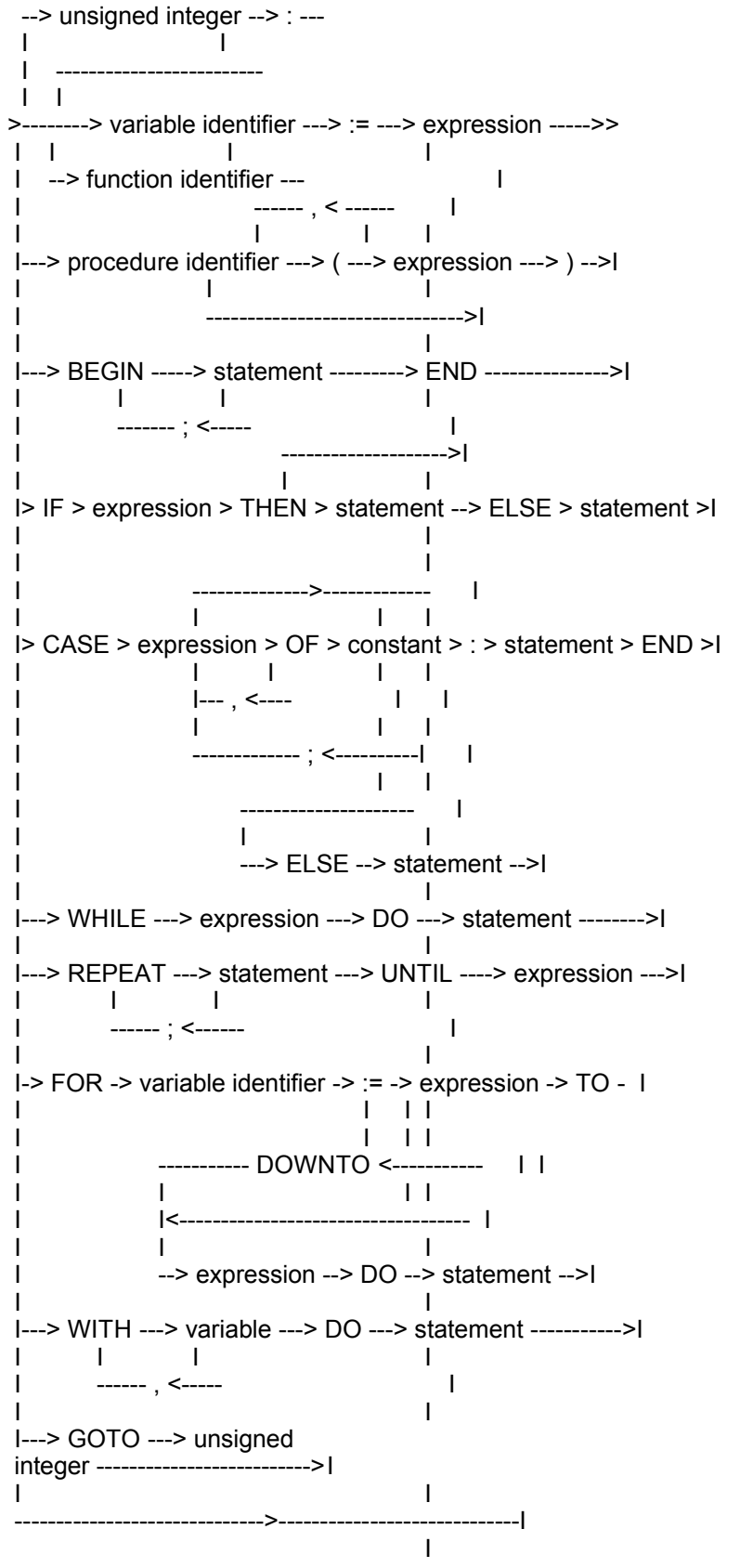
The control variable of a FOR statement may only be an unstructured variable, not a parameter. This is half way between Jensen/Wirth and draft ISO standard definitions.

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.

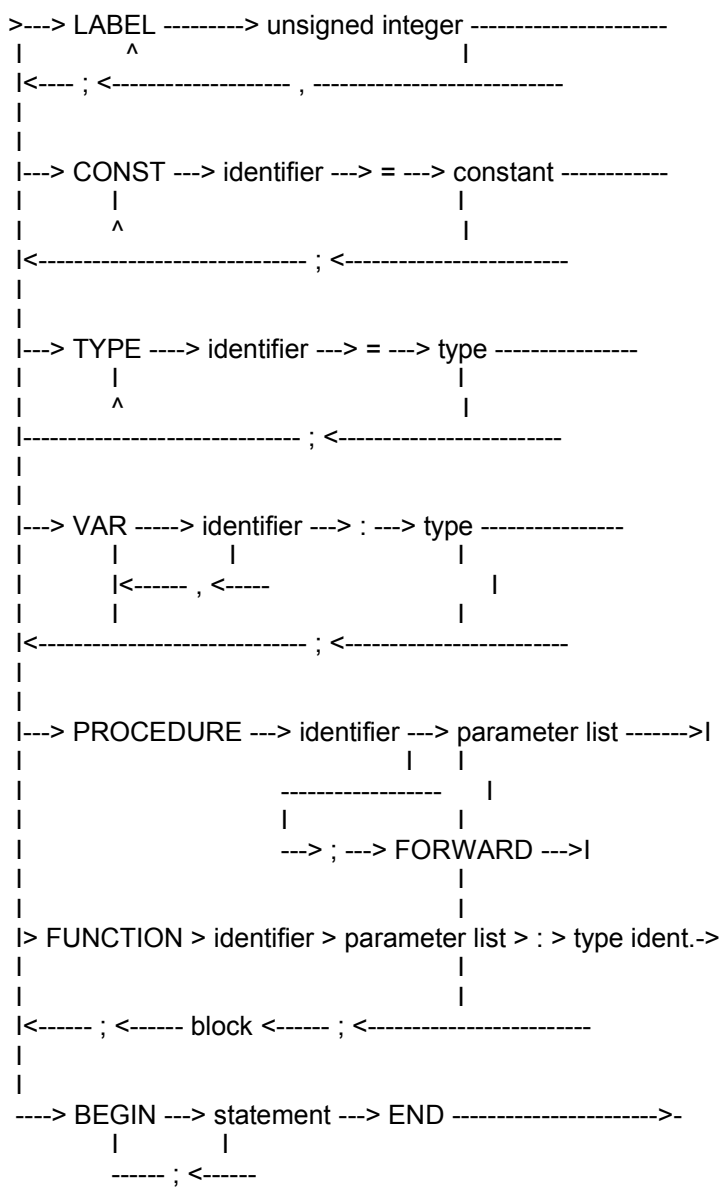
Labels must be declared (using the Reserved Word LABEL) in the block in which they are used; a label consists of at least one and up to four digits. When a label is used to mark a statement it must appear at the beginning of the statement and be followed by a colon - ':'.

STATEMENT.


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

v

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 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 are 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);                    (*procedure a referenced.*)
  ....
  END;
PROCEDURE a;           (*actual declaration of procedure a.*)
  BEGIN
  ....
  b(g);
  ....
  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.

--> PROGRAM --> identifier --> ; --> block --> END -->-

Since files are not implemented there are no formal parameters of the program.

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.

2.3 PROCEDURES AND FUNCTIONS.

2.3.1 Input and Output Procedures.

2.3.1.1 WRITE

The procedure WRITE is used to output data to the screen or printer. When the expression to be written is simply of type character then WRITE(e) passes the 8 bit value represented by the value of the expression e to the screen or printer as appropriate.

Note:

CHR(8) (CTRL H) gives a destructive backspace on the screen. CHR(12) (CTRL L) clears the screen or gives a new page on the printer. CHR(13) (CTRL M) performs a carriage return
and line feed. CHR(16) (CTRL P) will normally direct output to the printer if the screen is in use or vice versa.

Generally though:

```
WRITE(P1,P2,.....Pn); is equivalent to:
BEGIN WRITE(P1); WRITE(P2); .....; WRITE(Pn) END;
```

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 the 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: -1.23000E+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.

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 for 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.

2.3.1.2 WRITELN

WRITELN output gives a newline. This is equivalent to WRITE(CHR(13)). Note that a linefeed is included.

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

BEGIN WRITE(P1,P2,.....P3); WRITELN END;

2.3.1.3 PAGE

The procedure PAGE is equivalent to WRITE(CHR(12)); and causes the video screen to be cleared or the printer to advance to the top of a new page.

2.3.1.4 READ

The procedure READ is used to access data from the keyboard. It does this through a buffer held within the run-times - this buffer is initially empty (except for an end-of-line marker). We can consider that any accesses to this buffer take place through a text window over the buffer through which we can see one character at a time. If this text window is positioned over an end-of-line marker then before the read operation is terminated a new line of text will be read into the buffer from the keyboard. While reading in this line all various control codes detailed in Section 0.0 will be recognised. Now:

READ(V1,V2,.....Vn); is equivalent to:
BEGIN READ(V1); READ (V2); ....; READ(Vn) END;
where V1, V2 etc. may be of type character, string, integer or real.

The statement READ(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(V) simply reads a character from the input buffer and assigns it to V. If the text window on the buffer is positioned on a line marker (a CHR(13) character) then the function EOLN will return the value TRUE and a new line of text is read in from the keyboard. When a read operation is subsequently performed then the text window will be positioned at the start of the new line.

Important note: Note that EOLN is TRUE at the start of the program. This means that if the first READ 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 until the number of characters defined by the string has been read or EOLN = TRUE. If the string is not filled by the read (i.e. If end-of line 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 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 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 a 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; see 1] and 3] above.

2.3.1.5 READLN

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

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

READLN simply read in a new buffer from the keyboard; while typing in the buffer you may use the various control functions detailed in Section 0.0. Thus EOLN becomes FALSE after the execution of READLN unless the next line is blank.

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

2.3.2 Input Functions.

2.3.2.1 EOLN

The function EOLN is a Boolean function which returns the value TRUE if the next char to be read would be an end-of-line character (chr(13)). Otherwise the function returns the value FALSE.

2.3.2.2 INCH

The function INCH causes the keyboard 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.

2.3.3 Transfer Functions.

2.3.3.1 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

2.3.3.2 ROUND(X)

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

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

2.3.3.3 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.

2.3.3.4 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

2.3.3.5 CHR(X)

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

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

2.3.4 Arithmetic Functions.

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

2.3.4.1 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.

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

2.3.4.3 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.

2.3.4.4 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

2.3.4.5 SIN(X)

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

2.3.4.6 COS(X)

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

2.3.4.7 TAN(X)

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

2.3.4.8 ARCTAN(X)

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

2.3.4.9 EXP(X)

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

2.3.4.10 LN(X)

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

2.3.5 Further Predefined Procedures.

2.3.5.1 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.

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).

2.3.5.2 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.

2.3.5.3 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.

2.3.5.4 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.

2.3.5.5 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 4 holds integers in two's complement form (see Appendix 3) then in order to refer to addresses greater than #7FFF (32767) negative values of V must be used. For example #C000 is -16384 and so USER(-16384); would invoke as 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.

2.3.5.6 HALT

This procedure causes program execution to stop with the message 'Halt at the 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 maked. This will normally be used during de-bugging.

2.3.5.7 POKE(X,V)

POKE stores the expression V in the computer's memory address X. X is of type integer and V can be of any type except SET. See Section 2.3.5.5 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 0E 80 70 (in
hexadecimal) at location #C000.
```

2.3.5.8 TOUT (NAME,START,SIZE)

TOUT is the procedure which is used to save variables on tape. The first parameter is of type ARRAY [1..8] OF CHAR and is the name of the file to be saved. SIZE bytes of memory are dumped starting at the address START. Both these parameters are of type INTEGER.

E.g. to save the variable V to tape under the name 'VAR ' use:

```
TOUT('VAR  ',ADDR(V),SIZE(V))
```

The use of actual memory addresses gives the user far more flexibility than just the ability to save arrays. For example if a system has a memory mapped screen, entire screen-fulls may be saved directly. See Appendix 4 for an example of the use of TOUT.

2.3.5.9 TIN(NAME,START)

This procedure is used to load, from tape, variable etc. that have been saved using TOUT. NAME is of type ARRAY[1..8] of CHAR and START is of type INTEGER. The tape is searched for a file called NAME which is then loaded at memory address START. The number of bytes to load is taken from the tape (saved on the tape by TOUT).

E.g. to load the variable saved in the example in Section 2.3.5.8 above use:

TIN('VAR '),ADDR(V))

Because source files are recorded by the editor using the same format as that used by TIN and TOUT, TIN may be used to load text files into ARRAYs of CHAR for processing (see the HP4T Alteration Guide).

See Appendix 4 for an example of the use of TIN.

2.3.5.10 OUT(P,C)

This procedure is used to directly access the Z80's output ports without using the procedure INLINE. The value of the integer parameter P is loaded in to the BC register, then the character parameter C is loaded in to the A register and the assembly instruction OUT (C),A is executed.

E.g. OUT(1,'A') outputs the character 'A' to the Z80 port 1.

2.3.6 Further Predefined Functions.

2.3.6.1 RANDOM

This returns a pseudo-random number between 0 and 255 inclusive. Although this routine is very fast it gives poor results when used repeatedly within loops that do not contain I/O operations.

If the user requires better results than this function yields then he/she should write a routine (either in Pascal or machine code) tailored to the particular application.

2.3.6.2 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'

2.3.6.3 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

2.3.6.4 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.

2.3.6.5 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 4 see Appendix 3. For an example of the use of ADDR see Appendix 4.

2.3.6.6 PEEK(X,T)

The first parameter of this function is of type integer and is used to specify a memory address (see Section 2.3.5.5). 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.

In all PEEK and POKE (the opposite of PEEK) operations data is moved in HiSoft Pascal 4'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 4.

2.3.6.7 SIZE(V)

The parameter of this function is a variable. The integer result is the amount of storage

taken up by that variable, in bytes.

2.3.6.8 INP(P)

INP is used to access the Z80's ports directly without using the procedure INLINE. The value of the integer parameter P is loaded into the BC register and the character result of the function is obtained by executing the assembly language instruction IN A,(C).

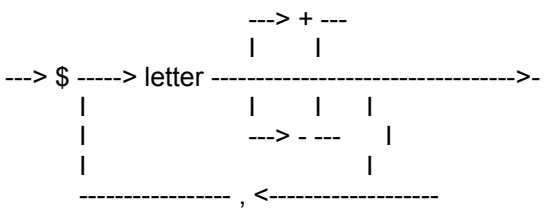
3 COMMENTS AND COMPILER OPTIONS.

3.1 Comments.

A comment may occur between any two reserved words, numbers,identifiers or special symbols - see Appendix 2. A comment starts with a '{' character or the '(*' character pair. Unless the next character is a '$' all characters are ignored until the next '}' character or '*)' character pair. If a '$' was found then the compiler looks for a series of compiler options (see below) after which characters are skipped until a '}' or '*)' is found.

3.2 Compiler Options.

The syntax for specifying compiler options is:



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

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 CC is pressed then execution will return to with a HALT message - see Section 2.3.5.6.

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. It should certainly be disabled if you wish the object program to run quickly.

If C- then the above check is not made.

DEFAULT: C+

Option S:

Controls whether or not stack checks 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; if a 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 utilizing recursion then it is possible for the function to be halted unnecessarily.

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 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 the P option is used the device to which the compilation listing is sent is changed i.e. if the video screen was being used the printer is used and vice versa. Note that this option is not followed by a '+' or '-'.
DEFAULT: The video screen is used.

Option F:

This option letter must be followed by a space and then an eight character file name. If the file name has less than eight characters it should be padded with spaces. The presence of this option causes inclusion of Pascal source text from the specified file from the end of the current line - useful if the programmer wishes to build up a 'library' of much-used procedures and functions on tape and then include them in particular programs. The program should be saved using the built-in editor's 'P' command. On most systems the list option L- should be used - otherwise the compiler will not compile fast enough.

Example: (*$L-,F MATRIX include the text from a tape file MATRIX*);

When writing very large programs there may not be enough room in the computer's memory for the source and object code to be present at the same time. It is however possible to compile such programs by saving them to tape and using the 'F' option - then only 128 bytes of the source are in RAM at any one time, leaving much more room for the object code. This option may not be nested and is not implemented in the ZX Spectrum version. (*In the HP4S ZX Spectrum version this option is implemented. The Pascal source text, which is to be included, should be saved using the built-in editor's command, instead of 'P'.*)

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.

4 THE INTEGRAL EDITOR.

4.1 Introduction to the Editor.

The editor supplied with all versions of HiSoft Pascal 4T is a simple, line-based editor designed to work with all Z80 operating systems while maintaining ease of use and the ability to edit programs quickly and efficiently. Text is held in memory in a compacted form; the number of leading spaces in a line is held as one character at the beginning of the line and all HP4T Reserved Words are tokenised into one character. This leads to a typical reduction in text size of 25%.

NOTE: throughout this section the DELETE key is referred to instead of the control code CH. It appears more natural to do this.

The editor is entered automatically when HP4T is loaded from tape and displays the message:

followed by the editor prompt '>'.

In response to the prompt you may enter a command line of the following format:

C N1, N2, S1, S2

followed by a RETURN where:

C is the command to be executed (see Section 4.2 below).
N1 is a number in the range 1 - 32767 inclusive.
N2 is a number in the range 1 - 32767 inclusive.
S1 is a string of characters with a maximum length of 20.
S2 is a string of characters with a maximum length of 20.

The comma is used to separate the various arguments (although this can be changed - see the 'S' command) and spaces are ignored, except within the strings. None of the arguments are mandatory although some of the commands (e.g. the 'D'elete command) will not proceed without N1 and N2 being specified. The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line. The values of N1 and N2 are initially set to 10 and the strings are initially empty. If you enter an illegal command line such as F-1,100,HELLO then the line will be ignored and the message 'Pardon?' displayed - you should then retype the line correctly e.g. F1,100,HELLO. This error message will also be displayed if the length of S2 exceeds 20; if the length of S1 is greater than 20 then any excess characters are ignored. Commands may be entered in upper or lower case. While entering a command line, all the relevant control functions described in Section 0.0 may be used e.g. CX to delete to the beginning of the line.

This lists the current text to the display device from line number n to line number m inclusive. The default value for n is always 1 and the default value for m is always 32767 i.e. default values are not taken from previously entered arguments. To list the entire text file simply use 'L' without any arguments. Screen lines are formatted with a left hand margin so that the line number is clearly displayed. The number of screen lines listed on the display device may be controlled through use of the 'K' command - after listing a certain number of lines the list will pause (if not yet at line number m), hit control function CC to return to the main editor loop or any other key to continue the listing.

Command: K n

'K' sets the number of screen lines to be listed to the display device before the display is paused as described in 'L' above. The value (n MOD 256) is computed and stored. For example use K5 if you wish a subsequent 'L'ist to produce five screen lines at a time.

4.2.3 Text Editing.

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

Command: D <n,m>

All lines from n to m inclusive are deleted from the text file. If m<n or less than two arguments are specified then no action will be taken; this is to help prevent careless mistakes. A single line may be deleted by making m=n ; this can also be accomplished by simply typing the line number followed by RETURN.

Command: M n,m

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

Command: N <n,m>

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

Command: F n,m,f,s

The text existing within the line range n<x<m is searched for an occurrence of the string f - the 'find' string. If such an occurrence is found then the relevant text line is displayed and the Edit mode is entered - see below. You may then use commands within the Edit mode to search for subsequent occurrences of the string f within the defined line range or to substitute the string s (the 'substitute' string) for the current occurrence of f and then search for the next occurrence of f; see below for more details. Note that the line range and the two strings may have been set up previously by any other command so that it may only be necessary to enter 'F' to initiate the search - see the example in Section 4.3 for clarification.

Command: E n

Edit the line with line number n. If n does not exist then no action is taken; otherwise the line is copied into a buffer and displayed on the screen (with the line number), the line number is displayed again underneath the line and the Edit mode is entered. All subsequent editing takes place within the buffer and not in the text itself; thus the original line can be recovered at any time. In this mode a pointer is imagined moving through the line (starting at the first character) and various sub-commands are supported which allow you to edit the line. The sub-commands are:

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

DELETE (or BACKSPACE) - decrement the text pointer by one to point at the previous character in the line. You cannot step backwards beyond the first character in the line.

CI (control function) - step the text pointer forwards to the next tab position but not beyond the end of the line.

RETURN - end the edit of this line keeping all the changes made.

Q - quit the edit of this line i.e. leave the edit ignoring all the changes made and leaving the line as it was before the edit was initiated.

R - reload the edit buffer from the text i.e. forget all changes made on this line and restore the line as it was originally.

L - list the rest of the line being edited i.e. the remainder of the line beyond the current pointer position. You remain in the Edit mode with the pointer re-positioned at the start of the line.

K - kill (delete) the character at the current pointer position.

Z - delete all characters from (and including) the current pointer position to the end of the line.

F - find the next occurrence of the 'find' string previously defined within a command line (see the 'F' command above). This sub-command will automatically exit the edit on the current line (keeping the changes) if it does not find another occurrence of the 'find' string in the current line. If an occurrence of the 'find' string is detected in a subsequent line (within the previously specified line range) then the Edit mode will be entered for the line in which the string is found. Note that the text pointer is always positioned at the start of the line after a successful search.

S - substitute the previously defined 'substitute' string for the currently found occurrence of the 'find' string and then perform the sub-command 'F' i.e. Search for the next occurrence of the 'find' string. This, together with the above 'F' sub-command, is used to step through the text file optionally replacing occurrences of the 'find' string with the 'substitute' string - see Section 4.3 for an example.
**#4 Important#5 Note ** In the current version of HP4T there is a known bug in the operation of the sub-command 'S' - this sub-command should only be used immediately after an 'F' command, an 'F' sub-command or an 'S' sub-command. In practice this should pose no problems.

I - insert characters at the current pointer position. You will remain in this sub-mode until you press RETURN - this will return you to the main Edit mode with the pointer positioned after the last character that you inserted. Using DELETE (or BACKSPACE) within this sub-mode will cause the character to the left of the pointer to be deleted from the buffer while the use of CI (control function) will advance the pointer to the next tab position, inserting spaces.

X - this advances the pointer to the end of the line and automatically enters the insert sub-mode detailed above.

C - change sub-mode. This allows you to overwrite the character at the current pointer position and then advances the pointer by one. You remain in the change sub-mode until you press RETURN whence you are taken back to the Edit mode with the pointer positioned after the last character you changed. DELETE (or BACKSPACE) within this sub-mode simply decrements the pointer by one i.e. moves it left while CI has no effect.

4.2.4 Tape Commands.

Text may be saved to tape or loaded from tape using the commands 'P' and 'G':

Command: P n,m,s

The line range defined by n<x<m is saved to tape in HP4T format under the file name specified by the string s. Remember that these arguments may have been set by a previous command. Before entering this command make sure that your tape recorder is switched on and in RECORD mode. While the text is being saved the message 'Busy..' is displayed.

Command: G,,s

The tape is searched for a file in HP4T tape format and with a file name of s. While the search is taking place the message 'Busy..' will be displayed. If a valid HP4T tape file is found but has the wrong file name then the message 'Found' followed by the file name that was found on the tape is displayed and the search continued. Once the correct file name is found then 'Found' will appear on the list device and the file will be loaded into memory. If an error is detected during the load then 'Checksum error' is shown and the load aborted. If this happens you must rewind the tape, press PLAY and type 'G' again.

If the string s is empty then the first HP4T file on the tape will be loaded, regardless of its file name.

While searching of the tape is going on you may abort the load by holding CC down; this will interrupt the load and return to the main editor loop.

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

Command: W n,m,s (*in the HP4S Spectrum version only*)

The line range n<x<m is saved to tape under the file name specified by the string s, in a format which can be loaded with the compiler option F (inclusion of Pascal source text). Before entering this command make sure that your tape recorder is switched on and is in RECORD mode.

4.2.5 Compiling and Running from the Editor.

Command: C n

This causes the text starting at line number n to be compiled. If you do not specify a line number then the text will be compiled from the first existing line. For further details see Section 0.2.

Command: R

The previously compiled object code will be executed, but only if the source has not been expanded in the meantime – see Section 0.2 for more detail.

Command: T n

This is the 'T'ranslate command. The current source is compiled from line n (or from the start if n is omitted) and, if the compilation is successful, you will be prompted with 'Ok?': if you answer 'Y' to this prompt then the object code produced by the compilation will be moved to the end of the run-times (destroying the compiler) and then the run-times and the object code will be dumped out to tape with a file name equal to that previously defined for the 'find' string. You may then, at a later stage, load this file into memory, using the HP4T loader, whereupon it will automatically execute the object program. Note that the object code is located at and moved to the end of the run-times so that, after a 'T'ranslate you will need to reload the compiler - however this should present no problems since you are only likely to 'T'ranslate a program when it is fully working.

If you decide not to continue with the dump to tape then simply type any character other than 'Y' to the 'Ok?' prompt; control is returned to the editor which will still function perfectly since the object code was not moved.

4.2.6 Other Commands.

Command: B

This simply returns control to the operating system. For details of how to re-enter the compiler refer to the HP4T Alteration Guide and your Implementation Note.

Command: O n,m

Remember that text is held in memory in a tokenised form with leading spaces shortened into a one character count and all HP4T Reserved Words reduced to a one character token. However if you have somehow got some text in memory, perhaps from another editor, which is not tokenised then you can use the 'O' command to tokenise it for you. Text is read into a buffer in an expanded form and then put back into the file in a tokenised form - this may of course take a little time to perform. A line range must be specified, or the previously entered values will be assumed.

Command: S,,d

This command allows you to change the delimiter which is taken as separating the arguments in the command line. On entry to the editor the comma ',' is taken as the delimiter; this may be changed by the use of the 'S' command to the first character of the specified string d.

Remember that once you have defined a new delimiter it must be used (even within the 'S' command) until another one is specified.

Note that the separator may not be a space.

4.3 An Example of the use of the Editor.

Let us assume that you have typed in the following program
(using I10,10):

```
 10 PROGRAM BUBBLESORT
 20 CONST
 30   Size = 2000;
 40 VAR
 50   Numbers : ARRAY [1..Size] OF INTEGER;
 60   I, Temp : INTEGER;
 70 BEGIN
 80   FOR I:=1 TO Size DO Number[I] := RANDOM;
 90   REPEAT
100     FOR I:=1 TO Size DO
110     Noswaps := TRUE;
120     IF Number[I] > Number[I+1] THEN
130       BEGIN
140       Temp := Number[I];
150       Number[I] := Number[I+1];
160       Number[I+1] := Temp;
170       Noswaps := FALSE
180     END
190   UNTIL Noswaps
200 END.
```

This program has a number of errors which are as follows:

Line 10 Missing semi-colon.
Line 30 Not really an error but say we want a size of 100.
Line 100 Size should be Size-1.
Line 110 This should be at line 95 instead.
Line 190 Noswapss should be Noswaps.

Also the variable Numbers has been declared but all references are to Number. Finally the BOLEAN variable Noswaps has not been declared.

To put all this right we can proceed as follows:

F60,200,Number,Numbers and then use sub-command 'S' repeatedly.

E10 then the sequence X ; RETURN RETURN

E30 then _____ K C 1 RETURN RETURN

F100,100,Size,Size-1 followed by the sub-command 'S'.

M110,95 .
110 followed by RETURN.

E190 then X DELETE RETURN RETURN

65 Noswaps : BOOLEAN; .

N10,10 to renumber in steps of 10.

You are strongly recommended to work through the above example actually using the editor - you may find it a little cumbersome at first if you have been used to screen editors but it should not take long to adapt.

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 expression of this type.
28. Should be either type INTEGER or 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. Lower-bound greater than upper-bound.
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 associated WITH statement.
59. Unsigned integer expected after 'LABEL'.
60. " " " " 'GOTO'.
61. This label is at the wrong level.
62. Undeclared label.
63. The parameter of SIZE should be a variable.
64. Can only use equality tests for pointers.
65.
66.
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.

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. Often the source of the error will be obvious; if not, consult the compilation listing to see where in the program the error occurred, using XXXX to cross reference. Occasionally this does not give the correct result.

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

Runtime errors result in the program execution being halted.

Appendix 2 RESERVED WORDS AND PREDEFINED IDENTIFIERS.

A.2.1 Reserved Words.

AND ARRAY BEGIN CASE CONST DIV DO DOWNTO
ELSE END 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 4 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.*)

PROCEDURE WRITE; WRITELN; READ; READLN; PAGE; HALT; USER;
          POKE; INLINE; OUT; NEW; MARK; RELEASE; TIN; TOUT;

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

Appendix 3 DATA REPRESENTATION AND STORAGE.

A.3.1 Data Representation.

The following discussion details how data is represented internally by HiSoft Pascal 4.

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 represented 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

$$2 = 2 * 10^{0} \quad \text{or} \quad 1.0 * 2^{1}$$

$$1 = 1 * 10^{0} \quad \text{or} \quad 1.0 * 2^{0}$$

$$-12.5 = -1.25 * 10^{1} \quad \text{or} \quad -25 * 2^{-1}$$

$$= -11001 * 2^{-1}$$

$$= -1.1001 * 2^{3} \text{ when normalized.}$$

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

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

```
        0.0001100
      ----------------------
101  I  0.10000000000000000
          101
          ---
           110
           101
           ---
            1000
             101    at this point we see
             ---  that the fraction recurs


     0.1
       2
=  -------  =  0.0001100
     101              2
       2

                    -4
          1.1001100 * 2     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          normalized 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.

normalized mantissa: the mantissa normalized to the form
1.XXXXXX with the top bit (bit 22)
always 1 except when representing
zero (HL=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 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*s.

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 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 and returned values

Values parameters are treated like local variables and, like these variables, the earlier 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 HP4T PROGRAMS.

The following programs should be studied carefully if you are in any doubt as to how to program in HiSoft Pascal 4T.

```
(*Program to illustrate the use of TIN and TOUT.
 The program constructs a very simple telephone  directory on tape and then reads it back. You  should
write any searching required.*)

PROGRAM TAPE;

CONST
  Size = 10;          (*Note that 'Size' is in upper
                    and lower case - not 'SIZE'.*)
TYPE
  Entry = RECORD
           Name : ARRAY [1..10] OF CHAR;
           Number : ARRAY [1..10] OF CHAR
         END;

VAR
  Directory : ARRAY [1..Size] OF Entry;
  I : INTEGER;

BEGIN                 (*Set up the directory..*)
  FOR I:= 1 TO Size DO
  BEGIN
    WITH Directory[I] DO
    BEGIN
      WRITE('Name please');
      READLN;
      READ(Name);
      WRITELN;
      WRITE('Number please');
      READLN;
      READ(Number);
      WRITELN
    END
  END;

(*To dump the directory to tape use..*)

  TOUT('Director',ADDR(Directory),SIZE(Directory))

(*Now to read the array back to the following..*)

  TIN('Director',ADDR(Directory))

(*And now you can process the directory as you wish.....*)

END.
```

```
 10  (*Program to list lines of a file in reverse order.
 20   Shows use of pointers, records, MARK and RELEASE.*)
 30
 40  PROGRAM ReverseLine;
 50
 60  TYPE elem=RECORD          (*Create linked-list structure*)
 70           next: ^elem;
 80           ch: CHAR
 90          END;
100      link=^elem;
110
120  VAR prev,cur,heap: link;   (*all pointers to 'elem'*)
130
140  BEGIN
150   REPEAT                   (*do this many times*)
160     MARK(heap);             (*assign top of heap to 'heap'*)
170     prev:=NIL;             (*points to no variable yet.*)
180     WHILE NOT EOLN DO
190       BEGIN
200         NEW(cur);          (*create a new dynamic record*)
210         READ(cur^.ch);      (*and assign its field to one
220                         character from file.*)
230         cur^.next:=prev;   (*this field's pointer addresses*)
240         prev:=cur          (*previous record.*)
250       END;
260
270  (*Write out the line backwards by scanning the records
280   set up backwards.*)
290
300     cur:=prev;
310     WHILE cur <> NIL DO     (*NIL is first*)
320       BEGIN
330         WRITE(cur^.ch);  (*WRITE this field i.e. character*)
340         cur:=cur^.next      (*Address previous field.*)
350       END;
360     WRITELN;
370     RELEASE(heap);       (*Release dynamic variable space.*)
380     READLN                (*Wait for another line*)
390   UNTIL FALSE               (*Use CC to exit*)
400  END.
```

```
 10  (*Program to show the use of recursion*)
 20
 30  PROGRAM FACTOR;
 40
 50  (*This program calculates the factorial of a number input
 60   from the keyboard 1) using recursion and 2) using an
 70   iterative method.*)
 80  TYPE
 90    POSINT = 0..MAXINT;
100
110  VAR
120    METHOD : CHAR;
130    NUMBER : POSINT;
140
150  (*Recursive algorithm.*)
160
170  FUNCTION RFAC(N : POSINT) : INTEGER;
180
190   VAR F : POSINT;
200
210   BEGIN
220     IF N>1 THEN F:= N * RFAC(N-1)   (*RFAC invoked N times*)
230           ELSE F:= 1;
240     RFAC := F
250   END;
260
270  (*Iterative solution*)
280
290  FUNCTION IFAC(N : POSINT) : INTEGER;
300
310    VAR I,F: POSINT;
320    BEGIN
330     F := 1;
340     FOR I := 2 TO N DO F := F*I;    (*Simple Loop*)
350     IFAC := F
360    END;
370
380  BEGIN
390    REPEAT
400      WRITE('Give method (I or R) and number    ');
410      READLN;
420      READ(METHOD,NUMBER);
430      IF METHOD = 'R'
440           THEN WRITELN(NUMBER,'! = ',RFAC(NUMBER))
450           ELSE WRITELN(NUMBER,'! = ',IFAC(NUMBER))
460    UNTIL NUMBER=0
470  END.
```

```
 10  (*Program to show how to 'get your hands dirty'!
 20   i.e. how to modify Pascal variables using machine code.
 30   Demonstrates PEEK, POKE, ADDR and INLINE.*)
 40
 50  PROGRAM divmult2;
 60
 70  VAR r:REAL;
 80
 90  FUNCTION divby2(x:REAL):REAL;      (*Function to divide
100                                     by 2.... quickly*)
110  VAR i:INTEGER;
120  BEGIN
130
 i:= ADDR(x)+1;              (*Point to the exponent of x*)
140   POKE(i,PRED(PEEK(i,CHAR))); (*Decrement the exponent of x.
150                            see Appendix 3.1.3.*)
160   divby2:=x
170  END;
180
190  FUNCTION multby2(x:REAL):REAL;   (*Function to multiply by
200                                    by 2.... quickly*)
210  BEGIN
220   INLINE(#DD,#34,3);        (*INC (IX+3) - the exponent
230                           of x - see Appendix 3.2.*)
240   multby2:=x
250  END;
260
270  BEGIN
280   REPEAT
290    WRITE('Enter the number r ');
300    READ(r);                 (*No need for READLN - see
310                           Section 2.3.1.4*)
320
330    WRITELN('r divided by two is',divby2(r):7:2);
340    WRITELN('r multiplied by two is',multby2(r):7:2)
350   UNTIL r=0
360  END.
```

B I B L I O G R A P H Y.

K.Jensen and N.Wirth — PASCAL USER MANUAL AND REPORT. Springer-Verlag 1975.

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.

The first and third books above are useful for reference purposes whereas the second and fourth books are introductions to the language and aimed towards beginners.