HiSoft Pascal

INTEGRATED COMPILER & EDITOR

48K Spectrum, Spectrum 128
& Spectrum Plus 2

HiSoft

High Quality Software

Corrections to the HiSoft Pascal for the ZX Spectrum Manual (4th Edition)

We would like to apologise for the following omissions from your **HiSoft Pascal** manual:

Page 3 Making a backup

After you are asked `Would you like 51 Columns?` you will be asked three further questions; just hit `Return` for each of these.

Page 5

In fact you won't be asked if you would like 51 columns when entering the example programs.

Page 9

When you load the compiler for the first time you will be asked three questions, normally just press `Return` for each. The following describes when you might wish to enter something different:

`Top of RAM ?`

The default for this is as described on Page 9 of the manual. You will need to use a different value if you wish to use a machine code routine at the top of memory whilst the compiler is loaded.

`Top of RAM for 'T'ranslate ?`

This is the top of memory when a stand-alone program produced by the 'Translate command is running. This defaults to the value used for the Top of RAM. You would change this if you are using a machine code routine only when running a translated program.

`Symbol Table Size?`

This is the amount of memory (in bytes) used by the compiler's symbol table. It defaults to one sixteenth of the available memory (about 1300 bytes). If you run out of symbol table space when compiling you will need to increase this.

Page 76

The turtle graphics example progams should all start their main sections at line 1390 not 1360.

We hope that these corrections will enable you to use HiSoft Pascal to the full.

HiSoft Pascal

Fast, Interactive Compiler

System Requirements:
ZX Spectrum 48K & 128K, ZX Spectrum Plus, ZX Spectrum Plus 2.
Opus and Disciple versions supplied on disk.

HiSoft ZX Pascal Manual
1st Edition July 1983
2nd Edition August 1984
3rd Edition October 1985
4th Edition December 1987

Set using an Apple Macintosh™, Microsoft Word™ and Apple Laserwriter™.

Table of Contents

Section 0
Introducing HiSoft Pascal

0.0 Getting Started

Congratulations! You are now the owner of an almost complete implementation of the Standard Pascal programming language for the ZX Spectrum. **HiSoft Pascal** is a very powerful tool that enables you to build highly structured and easy-to-understand programs that run *very quickly*. However, as with any computer language, it is going to take you some time to get used to using **HiSoft Pascal**. We strongly recommend that you adopt the following procedure when using this package for the first time:

- read the rest of this Section very carefully, trying out the example programs until you understand how they are created, compiled and executed.

- now read the editor section of this manual (**Section 4**) and try out the example at the end of the Section.

- go through the above steps until you feel comfortable using the editor and compiling/running a Pascal program.

- if you come to a complete halt in understanding what is happening, leave the computer and do something else for a while and then return and start from scratch - it is easy to get confused when operating in a new environment.

- if you are convinced that your problems are not of your own making then please do not hesitate to contact HiSoft during our technical support hour (3-4pm weekdays); our programming staff will be available to you.

Now read on.....

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

FILEs are not implemented although variables may be stored on tape, microdrive and disk (Opus Discovery and Disciple disk systems only). A RECORD type may not have a VARiant part. PROCEDUREs and FUNCTIONs are not valid as parameters.

A Spectrum Plus 3 version of **HiSoft Pascal** exists which removes most of the above restrictions.

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 runtimes take up roughly 4K and the editor uses 2k. Thus, typically, the total size of the package is roughly 19K, leaving the rest of your Spectrum's memory for the Pascal source and object programs.

0.1 Loading HiSoft Pascal

Loading from Cassette

Load the supplied cassette into your cassette recorder with **HiSoft Pascal** uppermost and type:

```
LOAD "" <ENTER>
```

on the Spectrum, and

<PLAY>

on your tape recorder.

Loading from Disk

Do not try to load **HiSoft Pascal** from the master disk that we supply, first make a backup (see below) onto another disk.

Once you have done this you can load the compiler from disk, using the name under which you saved it (see **Making a Backup** below), as a code file. You can load it at any convenient address but we would recommend 24700 as the lowest practical address.

0.3 Making a Backup

Making a Backup from Cassette

Load up **HiSoft Pascal** from cassette using the default address (i.e. simply type `LOAD ""`). Once it is loaded, you will be prompted with the question:

```
Would you like 51 columns?
```

Answer with a `Y` and, once in editor mode (with a > prompt) type:

`B <ENTER>` to get back to BASIC.

You can now make a backup to cassette or microdrive with the following commands:

`SAVE "HPCODE" CODE 24700,19200 <ENTER>` to cassette, or

`SAVE *"M";1;"HPCODE" CODE 24700,19200 <ENTER>` to microdrive 1

To load the compiler back into memory at any subsequent time use:

`LOAD "" CODE <ENTER>` from cassette, or
`LOAD *"M";1;"HPCODE" CODE <ENTER>` from microdrive.

Once loaded, you can enter the compiler at its start address. This is normally 24700, so use:

`RANDOMIZE USR 24700 <ENTER>`

If you have come out into BASIC and want to re-enter Pascal preserving any program you have entered, then execute Pascal at the start address + 3 i.e. normally 24703.

Making a Backup from Disk

If you have purchased **HiSoft Pascal** on Opus Discovery or Disciple disk, firstly format a new disk on which to copy your backup. Then insert your **HiSoft Pascal** master disk in drive A of your disk system and type:

```
LOAD "HPCODE" CODE <ENTER>
```

When the code of the compiler has loaded remove the master disk, insert your freshly-formatted blank disk and type:

```
SAVE *"M";1;"HPCODE" CODE 24700,19200 <ENTER>
```

To load the compiler back into memory at any subsequent time use:

```
LOAD *"M";1;"HPCODE" CODE <ENTER>
```

You can, of course, use the shorter forms of the LOAD and SAVE commands for the disk systems if you wish.

Once loaded, you can enter the compiler at its start address. This is normally 24700, so use:

```
RANDOMIZE USR 24700 <ENTER>
```

If you have come out into BASIC and want to re-enter Pascal preserving any program you have entered, then execute Pascal at the start address + 3 i.e. normally 24703.

0.4 Backing-up the Turtle Graphics

To make a copy of the turtle graphics routines you should enter the editor, use the G command to get the text of the graphics routines and then use the P command to save out the text onto cassette, microdrive or disk. See **Section 4** for details of the P and G editor commands.

0.5 Please ...

Please note that we allow you to make a backup of **HiSoft Pascal** for your own use so that you can protect your valuable investment and program with confidence. Please do not copy **HiSoft Pascal** to give away or sell to your friends; we supply very reasonable-priced software with a full after-sales service but if enough people copy our software illegally we shall not be able to continue this - ***please buy, don't steal!***

0.6 Your First HiSoft Pascal Program

To whet your appetite, and to give you a feel for the package, we now give an example of using the editor and compiler to create, compile and run a small Pascal program.

Firstly, let's load the Pascal into the computer: using your backup tape or disk type:

```
CLEAR 24699 <ENTER>
LOAD "" CODE <ENTER>          and start your tape recorder, or
LOAD *"M";1;"HPCODE" CODE <ENTER>
                              with your backup disk inserted.
```

Once the Pascal has been loaded into the computer type:

```
RANDOMIZE USR 24700 <ENTER>
```

You will be asked:

```
Would you like 51 columns?
```

Type Y or y to use 51-column editing, N or n to edit using the standard 32 columns. Once you have answered this question, the **HiSoft Pascal** editor will appear, letting you know it's there with a message and the > prompt. Let's type the following:

```
I10,10 <ENTER>
```

you should now be prompted with the number 10 ; this is a line number and what you subsequently type will be entered into the Pascal textfile at line 10, typing <ENTER> will terminate the line and then you will be prompted with line number 20 and so on. You can continue to enter text like this (with the line numbers being generated automatically for you) until you press the key combination <CAPS SHIFT> and 1 or <EDIT>.

So, type the following program, remembering that you do not need to type in the line numbers:

```
10 PROGRAM HELLO;
20 BEGIN
30  WRITELN('HELLO WORLD!');
40 END.
50 <EDIT>
```

Right, to compile this program type:

```
C <ENTER>
```

You should see a listing of you program appear with some extra numbers at the front - this is a compiler listing. If the program compiles correctly, the message Run? will appear; answer Y to this question and the program will run and print out:

```
HELLO WORLD!
```

and return to the editor.

You can now run the program again by:

```
R <ENTER>
```

If your program did not compile correctly and produced an *ERROR* message then press E followed by <ENTER> to get back to the editor, then press:

```
L <ENTER>
```

to list your program and compare it with the one given. If you see a mistake in any line then simply type that line number, then a space and then the correct text followed by <ENTER>.

Now, let's try another program:

```
I10,10 <ENTER>
10 PROGRAM CHARTOASC;
20 VAR CH : CHAR;
30 BEGIN
40 REPEAT
50 WRITE('ENTER A CHARACTER ');
60 READLN;
70 READ(CH);
80 WRITELN(CH,' IS ',ORD(CH),' IN ASCII.');
90 UNTIL CH=' ';
100 END.
110 <EDIT>
```

Now compile (C) and run this program; it will prompt you to enter a character, followed by <ENTER> and then print out the ASCII equivalent of that character. This will repeat until you enter a space as the character.

For those who know Pascal well, we would encourage you to study the use of READLN and READ in the above example of reading characters - also study the relevant sub-sections (**2.3.1.4** and **2.3.1.5**).

We hope that the above examples have given you some idea of how to use **HiSoft Pascal**; now read the rest of this section and Section 4. *Good Luck!*

0.7 Scope of this manual

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

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

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

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

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

Section 4 shows how to use the line editor which is an integral part of **HiSoft Pascal**.

The above Sections should be read carefully by all users.

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

Appendix 2 lists the predefined identifiers and reserved words.

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

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

Appendix 5 explains the use of the Turtle Graphics package supplied with **HiSoft Pascal** and shows you how to call the Spectrum ROM.

0.8 Compiling and Running

For details of how to create, amend, compile and run a program using the integral line editor see **Section 4** of this manual.

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 (see **Section 3**).

You may pause the listing at any stage by pressing <CAPS SHIFT> and <SPACE> subsequently use <CAPS SHIFT> 1 to return to the editor or any other key to restart the listing.

If an error is detected during the compilation then the message *ERROR* will be displayed, followed by an up-arrow (^), which points after the symbol which generated the error, and an error number (see **Appendix 1**). The listing will pause; hit E to return to the 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. *This is very unlikely to happen.*

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 <CAPS SHIFT> and <SPACE>; subsequently use <CAPS SHIFT> 1 to abort the run or any other key to resume the run.

0.9 Technical Notes

Use of the Spectrum Keyboard

When using **HiSoft Pascal**, the keyword entry scheme of the 48K Spectrum is not used. For example, to get SIN simply type the individual letters, S then I then N. <CAPS SHIFT> and <SYMBOL SHIFT> work in the normal way so that to get ' (single quote) you would use <SYMBOL SHIFT> and 7. Extended mode is not used. Those characters that are normally reached using Extended mode are accessed by typing <SYMBOL SHIFT> and the relevant key e.g. <SYMBOL SHIFT> Y gives [(square bracket). Note that the Pascal not equals operator <> is type as two characters, < then >. Finally, the ^ character that is used in pointers is <SYMBOL SHIFT> H.

Plus 2 and Plus 3 owners should note that to get the (,), { or } characters, you should hold down <SYMBOL SHIFT> and press Y, U, F or G respectively.

Memory Considerations

The compiler is, by default, loaded at address 24700 but you can load it at any practical address of your choice, say x, using:

```
CLEAR x-1 <ENTER>
LOAD "" CODE <ENTER>
RANDOMIZE USR x <ENTER>
```

The cold start address will then be x and the warm start address (the entry point to preserve any Pascal text) will be X+3.

The lower you load the Pascal, the more space you will have for your Pascal program but the less you will have for BASIC. The lowest possible value when using microdrives without a BASIC loader is about 24598. On a cassette-only machine you should be able to load the Pascal even lower.

51 or 32 Columns

Normally you should answer Y or y to the question:

```
Would you like 51 columns?
```

that appears on a cold start entry. If you don't, you will have an editor that uses 32 columns and 500 fewer bytes of RAM. This version may be used in the same way as the 51 character version and you should use it if you find 51 characters across the page difficult to read or you need that extra bit of memory or want to use the user defined graphics.

In the 51 character version, the default value for the top of RAM is 65525 whilst it is UDG in the 32 column version. On the Opus 51 column package the default top of RAM is 65522 because higher values cause problems for the Opus catalogue code.

Note that the 51 character code supports only a subset of the Spectrum control codes and does not support the keyword characters of the UDG. This keeps it fast and compact.

Using your Printer

HiSoft Pascal will print to any printer connected to Spectrum stream 3.

To get a compilation listing use the P option within your program (see **Section 3** for details).

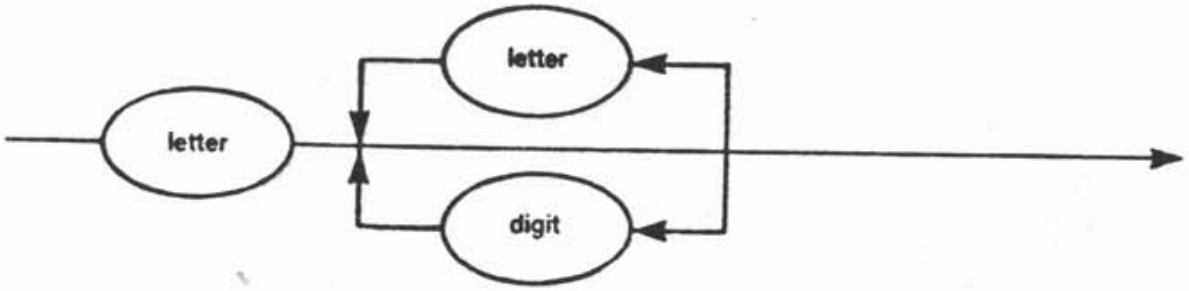
For a listing without compilation use <CAPS SHIFT> 3 followed by L <ENTER> and then <CAPS SHIFT> 3 again when the listing has finished. What <CAPS SHIFT> 3 does is toggle all the output between the printer and the screen.

To print from within your program use WRITE(CHR(16)); to toggle the printer on and off as described in **Section 2.**

Section 1 Syntax and Semantics

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

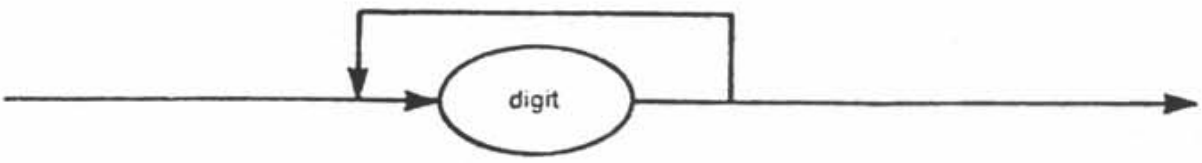
1.1 IDENTIFIER



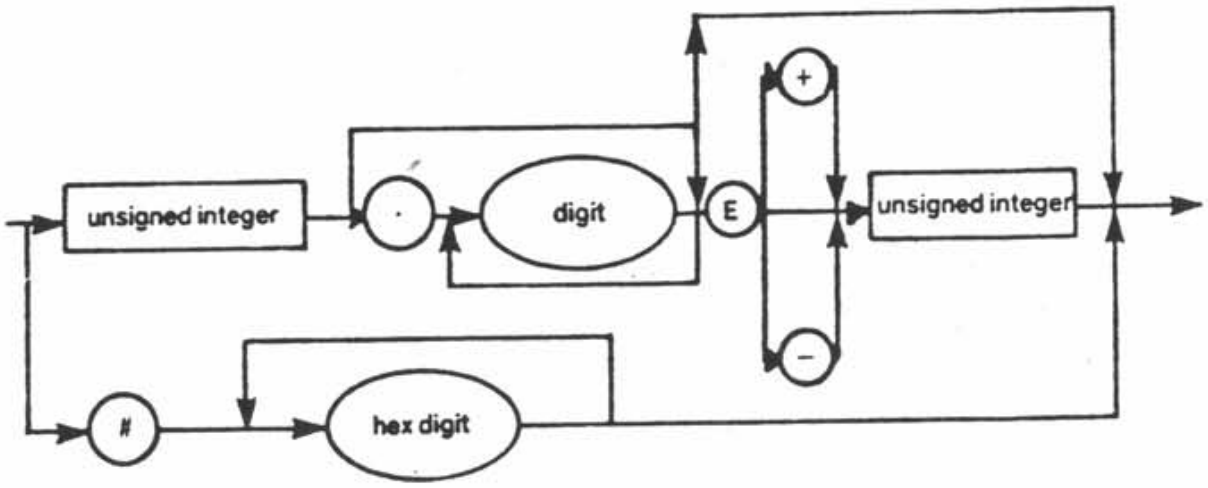
Only the first 10 characters of an identifier are treated as significant.

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 predefined identifiers may only be entered in upper case.

1.2 UNSIGNED INTEGER



1.3 UNSIGNED NUMBER



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

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

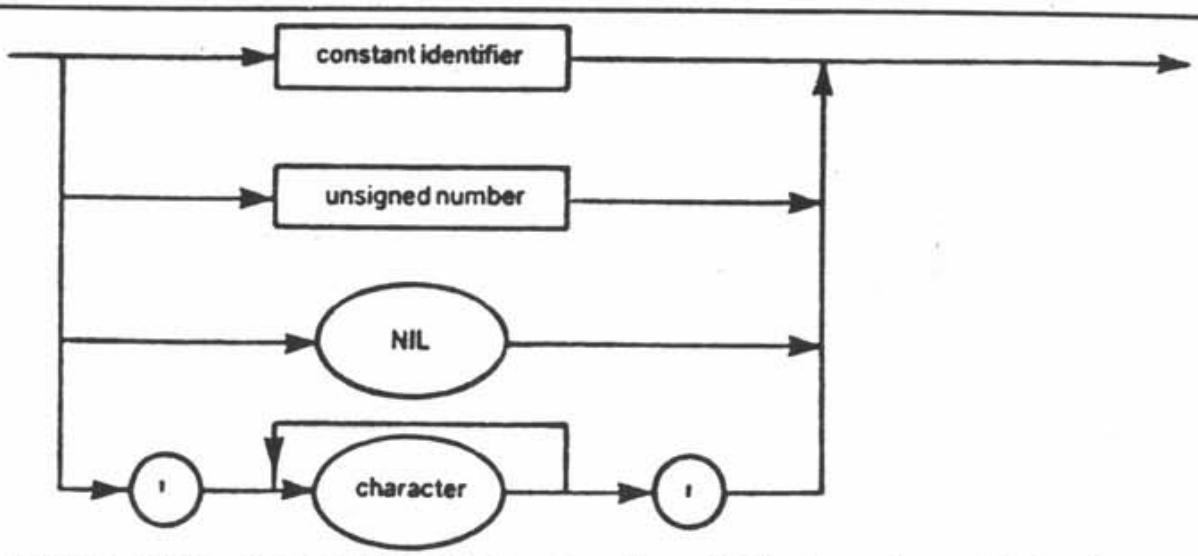
The largest real available is 3.4E38 while the smallest is 5.9E-39.

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.

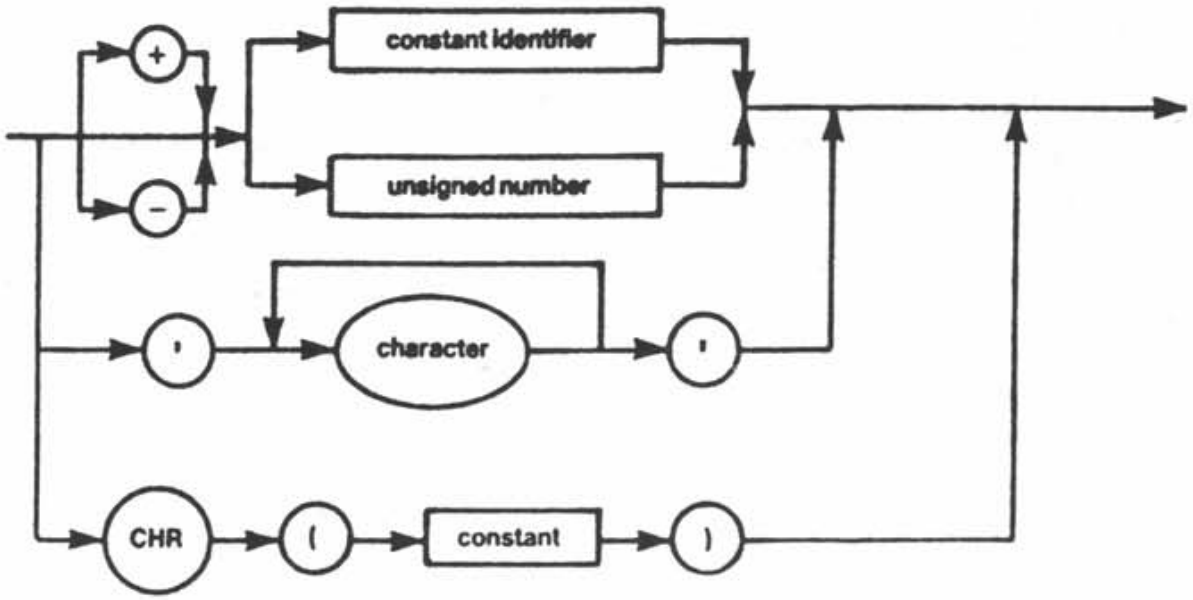
1.4 UNSIGNED CONSTANT



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.

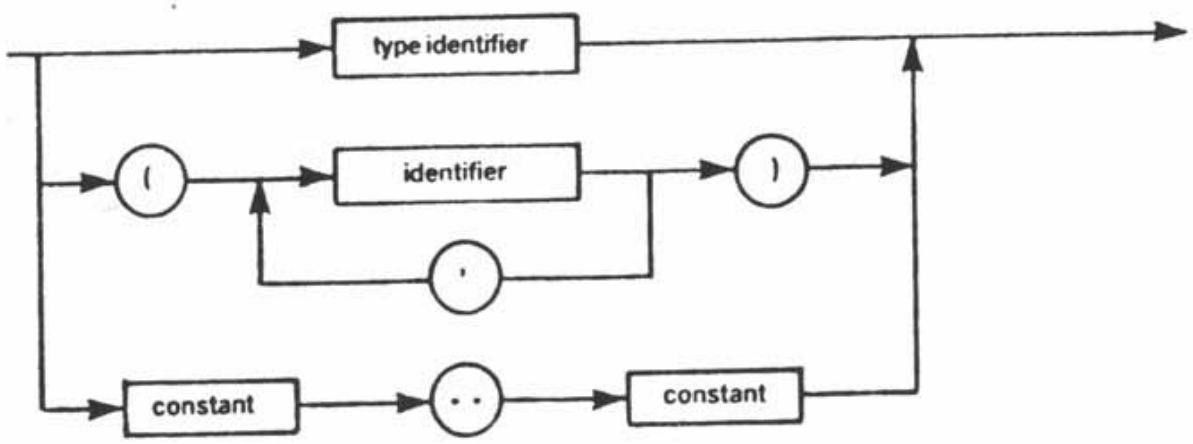
The characters available are the full expanded set of ASCII values with 256 elements. To maintain compatibility with Standard Pascal the null character is *not* represented as " ; instead CHR(0) should be used.

1.5 CONSTANT



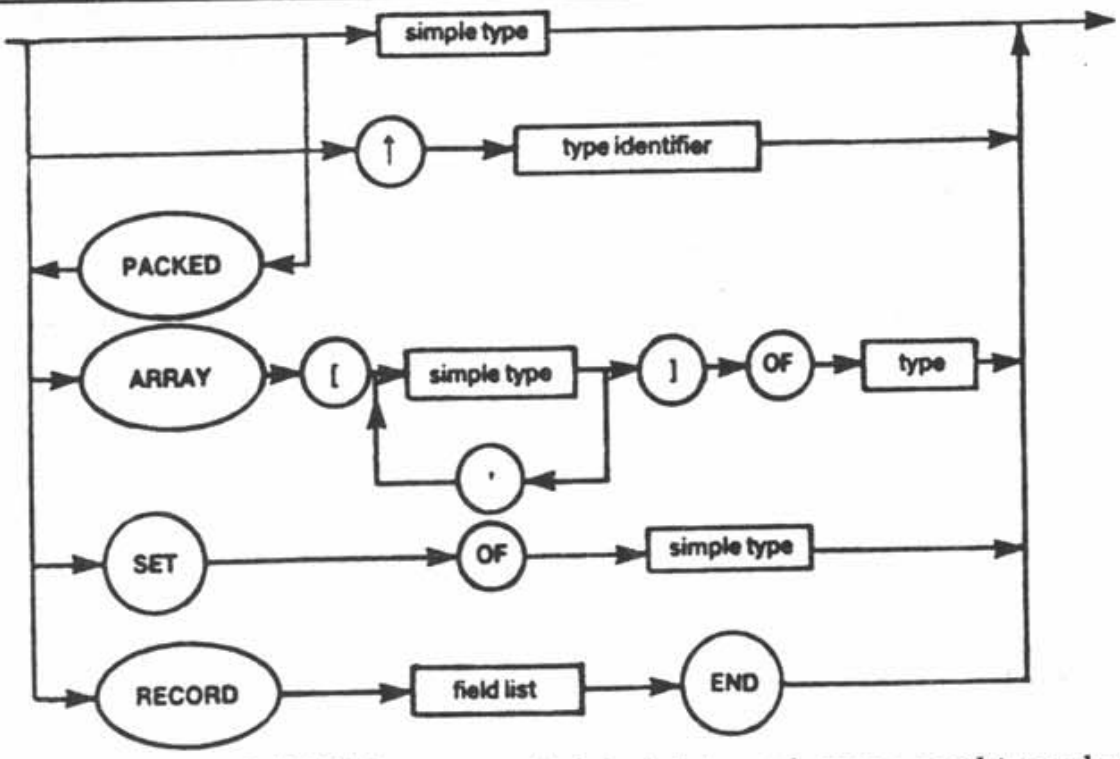
The non-standard CHR construct is provided here so that constants may be used for control characters. In this case the constant in parentheses must be of type integer e.g. `CONST bs=CHR(8); cr=CHR(13);`

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 subranges of integers can be used as base types. All subsets of integers are treated as sets of 0..255.

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

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

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

So a variable of type tablea may not be assigned to a variable of type tableb. This enables mistakes to be detected such as assigning two tables representing different data. This 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. See **Section 1.18** for a further discussion of this strong TYPEing.

1.7.2 Pointers

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

There are some restrictions on the use of pointers within **HiSoft Pascal**. These are as follows:

Pointers to types that have not been declared are not allowed. This does not prevent the construction of linked list structures since type definitions may contain pointers to themselves e.g.

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

Pointers to pointers 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.

1.7.3 RECORDs

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

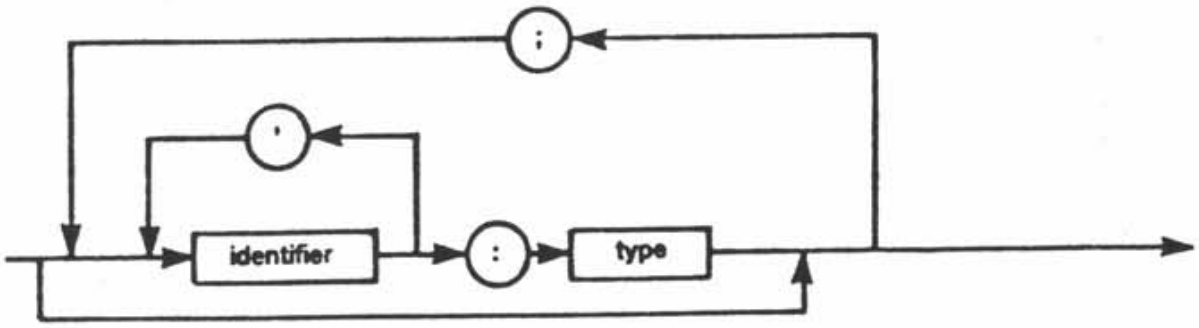
Two record types are only treated as equivalent if their declaration stems from the same occurrence of the reserved word RECORD see **Section 1.7.1** above.

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

RECORD declarations and WITH statements do not open a new scope. This means that you should not use the same field identifiers in two different record declarations or use the same name as a variable and a field identifier.

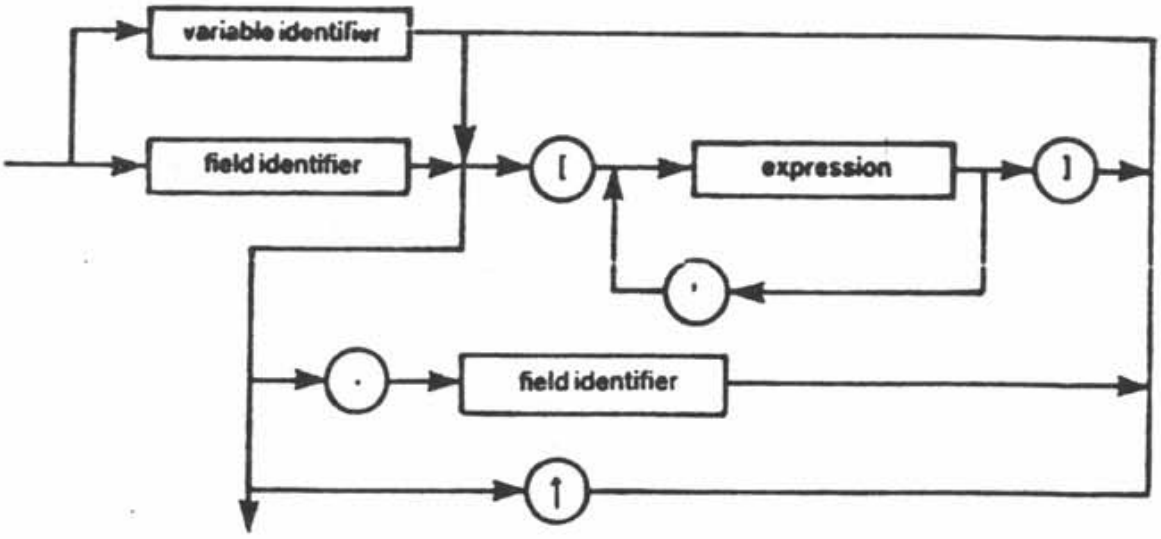
See **Appendix 4** for an example of the use of WITH and RECORDs in general.

1.8 FIELD LIST



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

1.9 VARIABLE



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

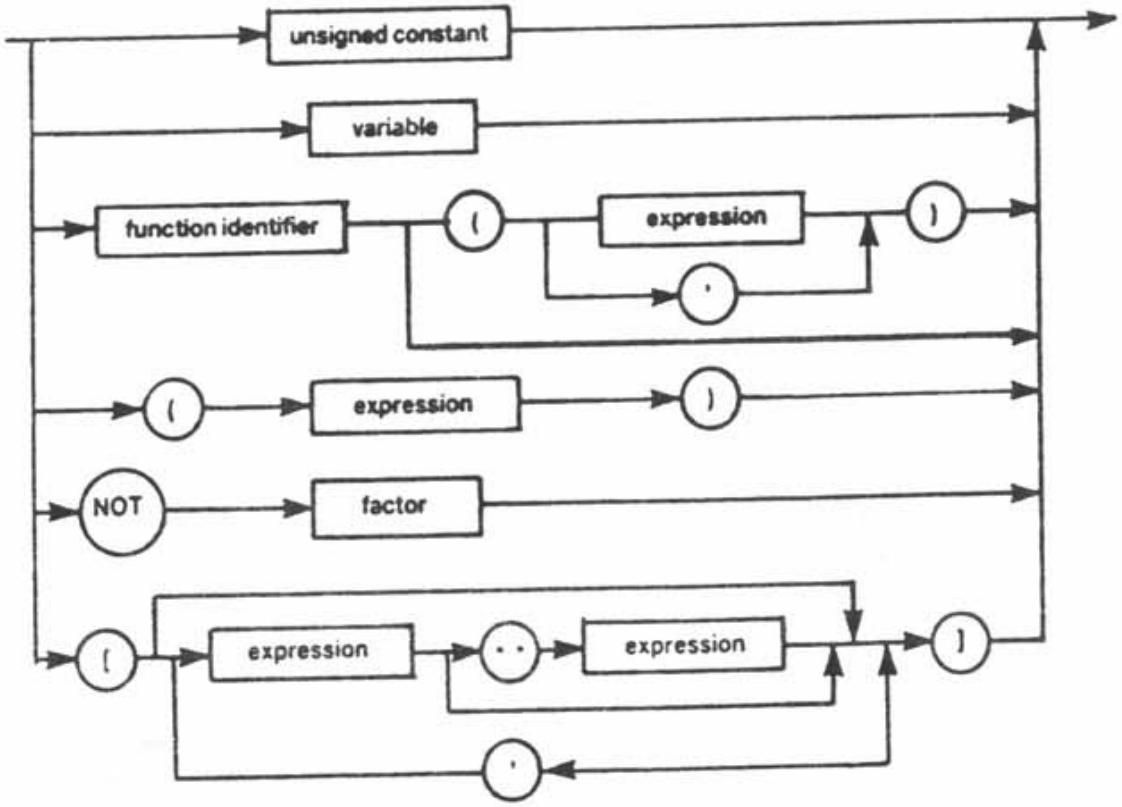
Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type pointer, which contains the address of the dynamic variable.

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

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

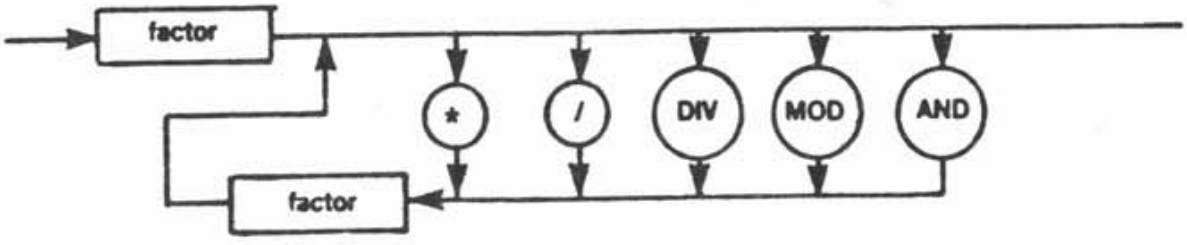
e.g. if variable a is declared as ARRAY(1..10) OF ARRAY(1..10) OF INTEGER then either a(1)(1) or a(1,1) may be used to access element (1,1) of the array.

1.10 FACTOR



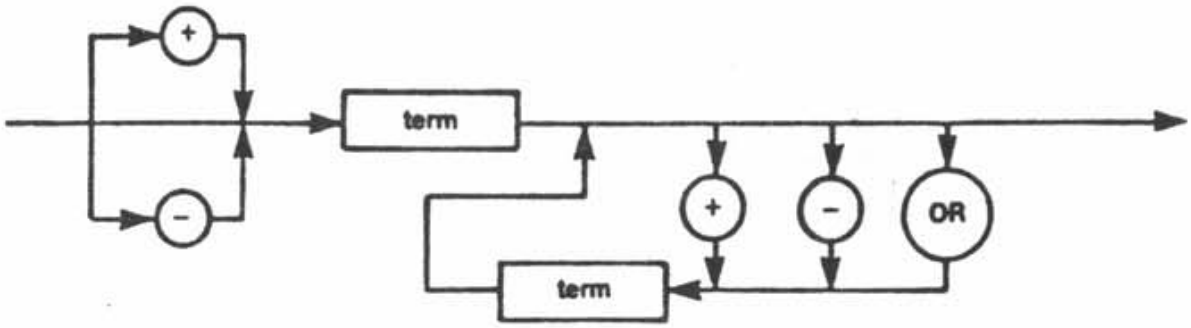
See EXPRESSION in **Section 1.13** and FUNCTIONs in **Section 3** for more details.

1.11 TERM



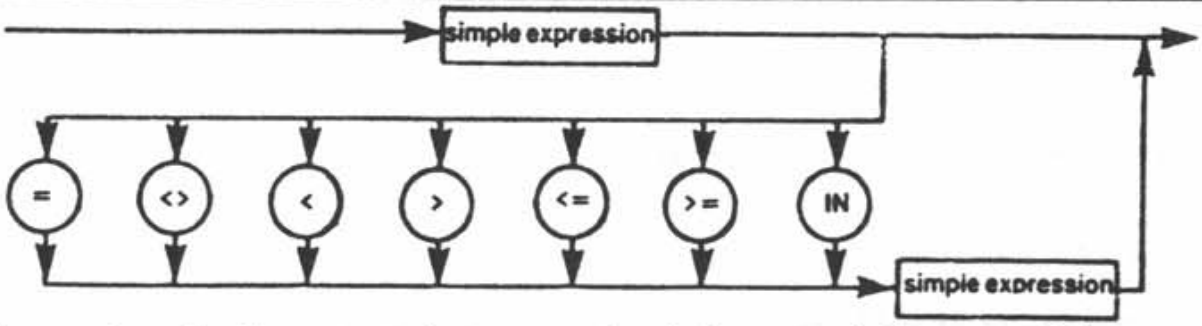
The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

1.12 SIMPLE EXPRESSION



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

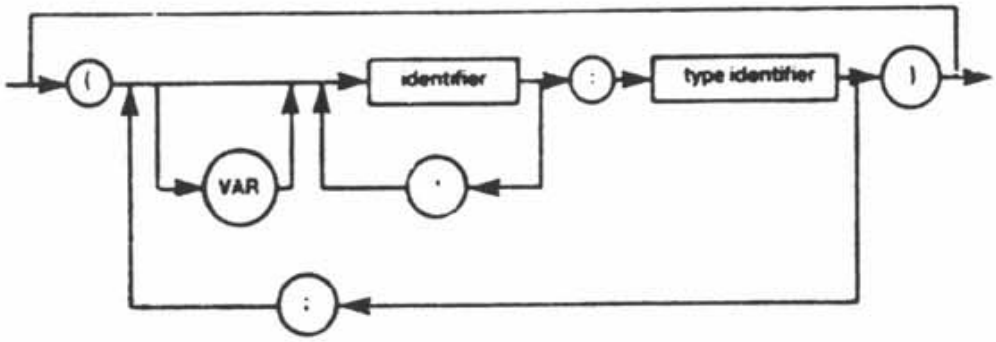
1.13 EXPRESSION



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

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

1.14 PARAMETER LIST



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

Variable parameters as well as value parameters are fully supported.

Procedures and functions are not valid as parameters.

1.15 STATEMENT

Refer to the syntax diagram on page 22.

Assignment statements

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

When assigning to subrange variables the value is not checked for being within the subrange for efficiency reasons.

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).

If the END terminator is used and the selector is not found then 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 the 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.

You may not use GOTO to jump out of a FOR statement.

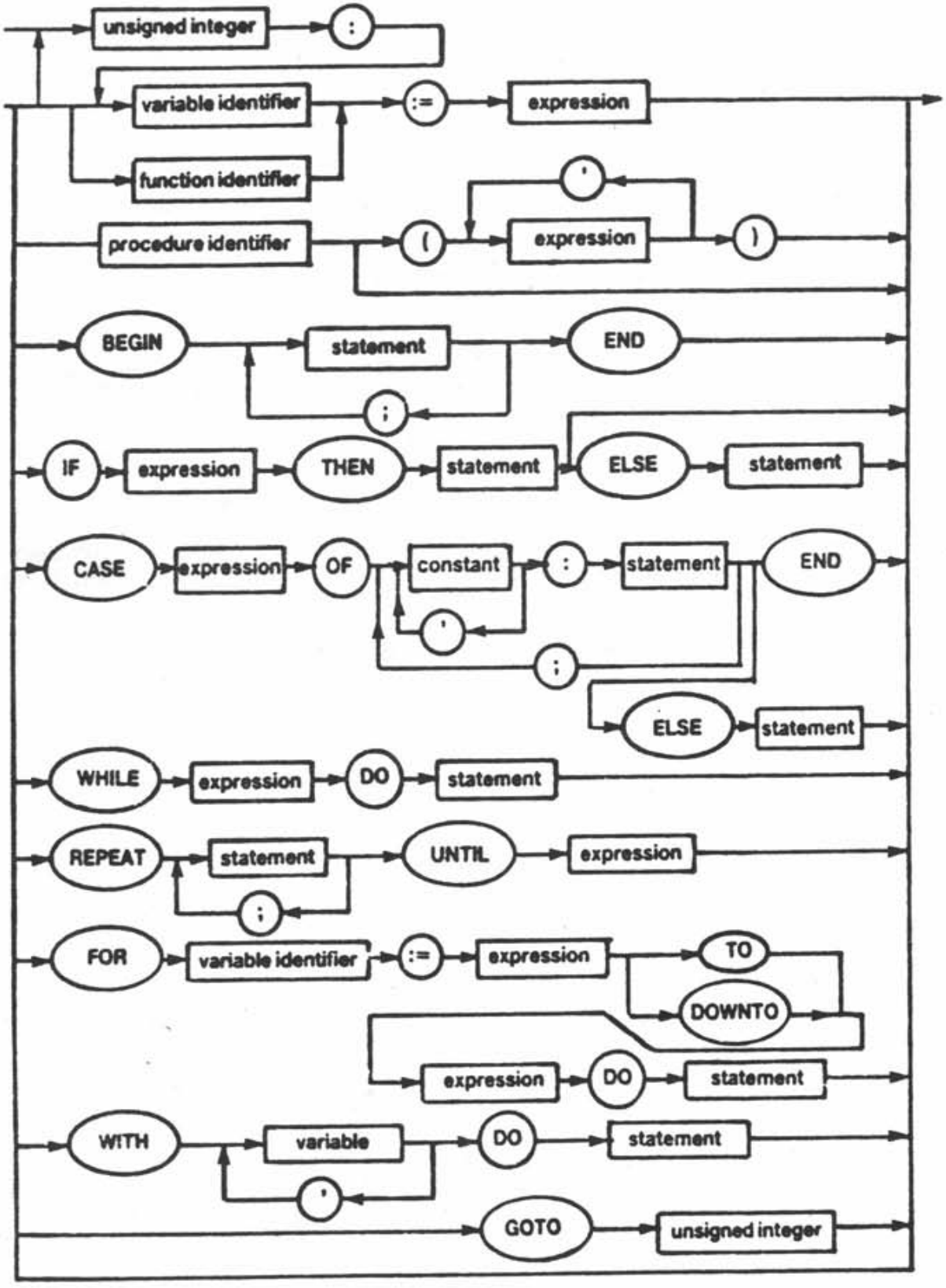
Labels must be declared (using the Reserved Word LABEL) in the block in which they are used; a label consists of at least one and up to four digits. When a label is used to mark a statement it must appear at the beginning of the statement and be followed by a colon.

WITH statements

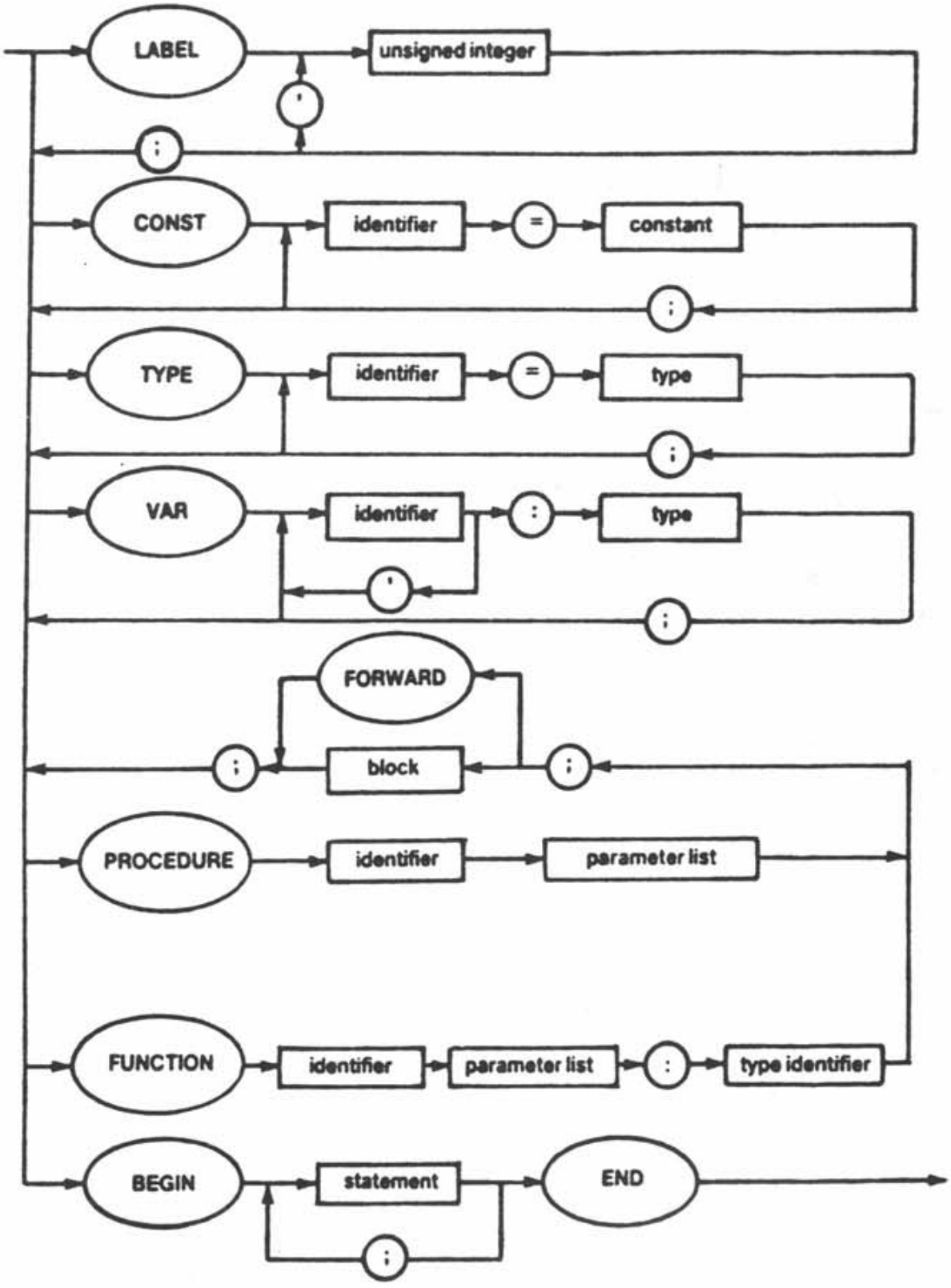
WITH statements may not be used recursively; use the full specification of a field identifier when using it as a parameter to a procedure which is called recursively.

The syntax diagram for Statement is on the next page.

STATEMENT


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

1.16 BLOCK


LABEL
unsigned integer
,
;
CONST
identifier
=
constant
;
TYPE
identifier
=
type
;
VAR
identifier
:
type
,
;
FORWARD
;
block
;
PROCEDURE
identifier
parameter list
FUNCTION
identifier
parameter list
:
type identifier
BEGIN
statement
END
;

Forward References

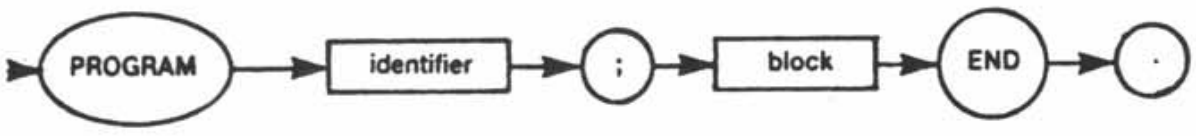
As in the Pascal User Manual and Report (Section 11.C.1) procedures and functions may be referenced before they declared through use of the Reserved Word FORWARD e.g.

```
PROCEDURE a(y:t) ; FORWARD; {procedure a declared to be}
PROCEDURE b(x:t);           {forward of this statement}
 BEGIN
 ....
 a(p);                      {procedure a referenced.}
 ....
 END;

PROCEDURE a;          {actual declaration of procedure a.}
 BEGIN
 ....
 b(q);
 ....
 END;
```

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

1.17 PROGRAM



Since Files are not implemented in HiSoft Spectrum Pascal, there are no formal parameters of the program.

1.18 Strong TYPEing

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

At one end of the scale there is machine code where no checks whatever are made on the *type* of variable being referenced. Next we have a language like the Byte 'Tiny Pascal' in which character, integer and Boolean data may be freely mixed without generating errors. Further up the scale comes BASIC which distinguishes between 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** uses name equivalence for RECORDs and ARRAYs. The consequences of this are clarified in **Section 1.7** *et al* - let it suffice to give an example here; say two variables are defined as follows:

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

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

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

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

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

Section 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	(FALSE,TRUE). 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 umber of characters to be output. If m is not sufficient for e to be written or m is not present then e is written out in full, with a trailing space, and m is ignored. Note that, if m is specified to be the length of e without the trailing space then no trailing space will be output.

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

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

WRITE(1025:m:H);

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

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

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

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

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

m=7	gives:	-1.23000E+10
m=8	gives:	-1.2E+10
m=9	gives:	-1.23E+10
m=10	gives:	-1.230E+10
m=11	gives:	-1.2300E+10
m=12	gives:	-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 if m is sufficiently large.

5] *e is of type Boolean.*

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

2.3.1.2 WRITELN

`WRITELN` outputs 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 runtimes - 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 various editing keys will be recognised (e.g. <DELETE>, <CAPS SHIFT>-8 etc.) 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 you 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 the runtime error Number expected will be reported and the program aborted.

4] *V is of type real.*

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

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

Reals, like integers, may be read immediately; 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 by itself simply reads in a new buffer from the keyboard; while typing in the buffer you may use <CAPS SHIFT>-0 to delete the previous character. 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**. *Note that you should always disable keyboard checks when using INCH i.e. always specify compiler option $C-.*

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 the nearest integer to X (according to standard rounding rules). Examples:

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

2.3.3.3 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   [
```

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 and 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** 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 a a call to the memory address #C000. However, when using a constant to refer to a memory address, it is more convenient to use hexadecimal.

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

2.3.5.6 HALT

This procedure causes program execution to stop with the message

Halt at PC=XXXX

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

2.3.5.7 POKE(X,V)

POKE stores the expression V in the computer's memory starting from the memory address X. X is of type integer and V can be of any type except 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 0B 80 70 (hex) at location #C000.
```

2.3.5.8 TOUT (NAME,START,SIZE)

TOUT is the procedure which is used to save variables on tape, microdrive or disk (if your disk system is supported by **HiSoft Pascal**). The first parameter is an ARRAY OF CHAR that can have a *maximum* of 12 characters in it 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));
```

To save the variables onto files on a microdrive cartridge or a supported disk drive, simply precede the filename with a drive number (1 to 8 for microdrives) and a colon (e.g. `1:TEST`).

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 screenfuls 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, microdrive or disk variables etc. that have been saved using TOUT. NAME is of type ARRAY OF CHAR and can have a *maximum* of 12 characters while START is of type INTEGER. The tape, microdrive or disk 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 front of the file (saved by TOUT). E.g. to load the variable saved in the example in **Section 2.3.5.8** above use:

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

To load the variables from files on a microdrive cartridge or a supported disk drive, simply precede the filename with a drive number (1 to 8 for microdrives) and a colon (e.g. `1:TEST`).

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, 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 you require better results than this function yields then you 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.

2.3.6.6 ADDR(V)

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

2.3.6.7 PEEK(X,T)

The first parameter of this function is of type integer and is used to specify a memory address (see **Section 2.3.5.5**). 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**'s own internal representation detailed in **Appendix 3**. For example: if the memory from #5000 onwards contains the values: 50 61 73 63 61 6C (in hexadecimal) then:

```
WRITE(PEEK(#5000,ARRAY[1..6] OF CHAR))    gives    Pascal
WRITE(PEEK(#5000,CHAR))                   gives    P
WRITE(PEEK(#5000,INTEGER))                gives    24912
WRITE(PEEK(#5000,REAL))                   gives    2.46227E+29
```

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

2.3.6.7 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 be BC register and the character result of the function is obtained by executing the assembly language instruction IN A,(C).

Section 3
Comments and Compiler Options

3.1 Comments

A comment within a Pascal program 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. For example:

```
i:=i+1;  {bump the loop count}
{$l+  turn the listing on from now}
```

3.2 Compiler Options

Compiler options are included in the program between comment brackets and are the first option in the list is prefaced by a dollar symbol $.

Example:

(*$C-,A-*) to turn keyboard and array checks OFF.

The following options are available:

Option L

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

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

DEFAULT: L+

Option O

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

If O+ then checks are made on integer addition and subtraction.

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 <CAPS SHIFT> and <SPACE> are pressed together, execution will pause; hit <EDIT> to return to the editor or BASIC with a HALT message (see **Section 2.3.5.6**) or any other key to continue execution.

This check is made at the beginning of all loops, procedures and functions. Thus you 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 utilising 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 occurs when performing a >, <, >=, or <= operation if the arguments differ by more than MAXINT (32767). If this occurs then the result of the comparison will be incorrect. This will not normally present any difficulties; however, should you 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 up to 12 characters of filename. If you are using microdrive or disk then the filename must start with a drive number (1, 2, etc.) followed by a colon.

The presence of this option causes inclusion of Pascal source text from the specified file from the end of the current line - useful if you wish to build up a 'library' of much-used procedures and functions on tape/microdrive/disk and then include them in particular programs.

The program should be saved using the built-in editor's W command if cassette is being used or the P command if microdrive/disk is used. The list option L- (turn listing off) should be used when including from tape, otherwise the compiler cannot compile quickly enough in the inter-block gaps.

Example:

```
{$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/microdrive/disk and using the F option - then only a small portion of the source is in RAM at any one time, leaving much more room for the object code.

This option may not be nested.

General

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.

Section 4
The Integral Editor

4.1 Introduction to the Editor

The editor supplied with the ZX Spectrum version of **HiSoft Pascal** is a simple, line-based editor designed for 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 **HiSoft Pascal** 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 mean the key that deletes the character behind the cursor; this is <CAPS SHIFT> 0 or <DELETE> on Spectrum keyboards.

The editor is entered automatically when **HiSoft Pascal** is executed and displays the message:

```
Copyright HiSoft 198x
All rights reserved
```

followed by the editor prompt >.

In response to the prompt you may enter characters that make up a command line of the following format:

C N1, N2, S1, S2

followed by <ENTER> 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 Delete command) will not proceed without N1 and N2 being specified.

The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line. The values of N1 and N2 are initially set to 10 and the strings are initially empty.

If you enter an illegal command line such as F-1,100,HELLO then the line will be ignored and the polite 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, various keys may be used to edit the line e.g. <DELETE> to delete the last character, <CAPS SHIFT> 5 to delete to the beginning of line etc.

The following sub-section details the various commands available within the editor - note that wherever an argument is enclosed by the symbols < > then that argument *must* be present for the command to proceed.

4.2 The Editor Commands

4.2.1 Text Insertion

Text may be inserted into the textfile either by typing a line number, a space and then the required text or by use of the I command. Note that if you type a line number followed by <ENTER> (i.e. without any text) then that line will be deleted from the text if it exists.

Whenever text is being entered then the control functions <CAPS SHIFT> 5 (delete to the beginning of the line), <CAPS SHIFT> 8 (go to the next tab position), <EDIT> (return to the command loop) and <CAPS SHIFT> 3 (toggle the printer) may be employed. The <DELETE> key will produce a destructive backspace (but not beyond the beginning of the text line). Text is entered into an internal buffer within **HiSoft Pascal** and if this buffer should become full then you will be prevented from entering any more text - you must then use <DELETE> or <CAPS SHIFT> 5 to free space in the buffer.

Command: I n,m

Use of this command gains entry to the automatic insert mode: you are prompted with line numbers starting at n and incrementing in steps of m. You enter the required text after the displayed line number, using the various control codes if desired and terminating the text line with <ENTER>. To exit from this mode use <EDIT>.

If you enter a line with a line number that already exists in the text then the existing line will be deleted and replaced with the new line, after you have pressed <ENTER>. If the automatic incrementing of the line number produces a line number greater than 32767 then the Insert mode will exit automatically.

If, when typing in text, you get to the end of a screen line without having entered 128 characters (the buffer size) then the screen will be scrolled up and you may continue typing on the next line - an automatic indentation will be given to the text so that the line numbers are effectively separated from the text.

4.2.2 Text Listing

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

Command: L n,m

This lists the current text to the display device from line number n to line number m inclusive. The default value for n is *always* 1 and the default value for m is *always* 32767 i.e. default values are not taken from previously entered arguments.

To list the entire textfile simply use L <ENTER> 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 <EDIT> to return to the main editor loop or any other key to continue the listing.

Command: K n

Kn sets the number of screen lines to be listed to the display device before the display is paused as described in List above. The value (n MOD 256) is computed and stored. For example use K5 if you wish a subsequent list 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 textfile. If m<n or less than two arguments are specified then no action will be taken; this is to help prevent careless mistakes. A single line may be deleted by making m=n ; this can also be accomplished by simply typing the line number followed by <ENTER>.

Command: M n,m

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

Command: N <n,m>

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

Command: F 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 <ENTER> 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>	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.
<CAPS SHIFT> 8	step the text pointer forwards to the next tab position but not beyond the end of the line.
<ENTER>	end the edit of this line keeping all the changes made.
Q	quit the edit of this line i.e. leave the edit ignoring all the changes made and leaving the line as it was before the edit was initiated.
R	reload the edit buffer from the text i.e. forget all changes made on this line and restore the line as it was originally.
L	list the rest of the line being edited i.e. the remainder of the line beyond the current pointer position. You remain in the Edit mode with the pointer re-positioned at the start of the line.
K	kill (delete) the character at the current pointer position.
Z	delete all the characters from (and including) the current pointer position to the end of the line.

F — find the next occurrence of the *find* string previously defined within a command line (see the F command above). This sub-command will automatically exit the edit on the current line (keeping the changes) if it does not find another occurrence of the *find* string in the current line. If an occurrence of the *find* string is detected in a subsequent line (within the previously specified line range) then the Edit mode will be entered for the line in which the string is found. Note that the text pointer is always positioned at the start of the found string 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 textfile optionally replacing occurrences of the *find* string with the *substitute* string - see **Section 4.3** for an example.

I — insert characters at the current pointer position. You will remain in this sub-mode until you press <ENTER> - this will return you to the main Edit mode with the pointer positioned after the last character that you inserted. Using <DELETE> within this sub-mode will cause the character to the left of the pointer to be deleted from the buffer while the use of <CAPS SHIFT> 8 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 <ENTER> whence you are taken back to the Edit mode with the pointer positioned after the last character you changed. <DELETE> within this sub-mode simply decrements the pointer by one i.e. moves it left while <CAPS SHIFT> 8 has no effect.

4.2.4 Tape/Microdrive/Disk Commands

Text may be saved to or loaded from tape or microdrive or disk using the commands P, W and G:

Command: P n,m,s

The line range defined by n < x < m is saved to tape in **HiSoft Pascal** format under the filename specified by the string s. Remember that these arguments may have been set by a previous command.

If the first two characters of the filename are a number followed by a colon then the text is saved to the relevant microdrive or disk drive instead of to tape. Examples:

P10,90,TEST <ENTER>	save lines 10 through 90 to tape under the name TEST
P10,1000,1:PRIMES <ENTER>	save lines 10 through 1000 to microdrive/disk under the name PRIMES

When saving to microdrive or disk, any file with the same name as the file you are saving will be deleted first.

Before entering this command make sure that your tape recorder is switched on and in RECORD mode.or that there is a disk or microdrive cartridge in the given drive.

Command: Wn,m,s

This command behaves like the P command except that it saves your program to tape in a blocked format so that it can be *included* using the compiler option F at a later stage. Use this command if you want the text to be included from tape. Use the P command if you want your text to be included from microdrive or disk.

Command: G,,s

The tape/microdrive/disk is searched for a file with a filename of s, which, if found, will be loaded into the editor so that it can be changed/compiled.

Cassette tape:

While the search is taking place the message Searching.. will be displayed. If a valid **HiSoft Pascal** tape file is found but has the wrong filename then the message Found followed by the filename that was found on the tape is displayed and the search continued. Once the correct filename is found then Using will appear and the file will be loaded into memory. If an error is detected during the load then an error message is displayed 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 **HiSoft Pascal** file on the tape will be loaded, regardless of its filename.

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

Microdrive/Disk:

If the specified file cannot be found, it is treated as error and the message Absent is displayed. Use the X command to check that the required file is on the microdrive/disk.

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

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 0.8**.

Command: R

The previously compiled object code will be executed, but only if the source has not been expanded in the meantime - see **Section 0.8** for more detail.

Command: T n,,s

This is the Translate 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 or y to this prompt then the object code produced by the compilation will be moved to the end of the runtimes (destroying the compiler) and then the runtimes and the object code will be dumped out to tape/microdrive/disk with a filename s. As usual, if the first two characters of the filename are a number followed by a colon, the code will be saved onto the relevant microdrive cartridge or disk.

You may then, at a later stage, load this code into memory, and execute the object program. Load it from BASIC by typing:

```
LOAD "" CODE <ENTER>                    from cassette or
LOAD *"M";1;"name" CODE <ENTER>         from microdrive or disk
```

Once loaded, you can execute the program by:

`RANDOMIZE USR 24709 <ENTER>` or the *start of the compiler* + *9* if you had loaded the compiler at an address other than the default one of 24700 when you saved the code.

Note that the object code is located at and moved to the end of the runtimes so that, after a Translate you will need to reload the compiler; however this should present no problems since you are not likely to Translate a program until it is fully working.

If you decide not to continue with the save then simply type any character other than Y or 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 Spectrum BASIC. To re-enter the compiler type:

`RANDOMIZE USR 24700 <ENTER>` for a cold start or

`RANDOMIZE USR 24703 <ENTER>` for a warm start, preserving any text

These addresses assume that you loaded Pascal at 24700; if not, use addresses based on the address at which you loaded the package.

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 **HiSoft Pascal** 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. Also, please do not confuse this command with the S sub-command which is used only in the Edit mode (after using the E command).

Command: V

The V command takes no arguments and displays the current default values of the line range, the two strings (*find* and *substitute*) and the current delimiter. The line range is shown first followed by the two strings, which may be empty, and lastly the delimiter. Remember that certain editor command (e.g. D and N) do not use these defaults but must have values specified on the command line.

Command: Xn

Gives a catalogue of the microdrive or disk drive n e.g.

```
x2 <ENTER>
```

catalogues drive 2

If a parameter is not given then drive 1 is assumed.

Do not use this command without a microdrive or disk drive connected.

4.3 An Example of the use of the Editor

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

```
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 Noswapss;
200 FOR I:=1 TO Size DO WRITE(Number[I]:4);
210 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 BOOLEAN variable Noswaps has not been declared.

To put all this right we can proceed as follows:

```
F60,200,Number,Numbers <ENTER>
```
and then use the sub-command S repeatedly.

```
E10 <ENTER>
```
then the sequence `X ; <ENTER> <ENTER>`

```
E30 <ENTER>
```
then (9 spaces) `K C 1 <ENTER> <ENTER>`

```
F100,100,Size,Size-1 <ENTER>
```
followed by the sub-command S.

```
M110,95 <ENTER>
```

```
E190 <ENTER>
```
then `X <DELETE> <ENTER> <ENTER>`

```
65 Noswaps : BOOLEAN; <ENTER>
```

```
N10,10 <ENTER>
```
to renumber in steps of 10.

You are strongly recommended to work through the above example actually using the editor.

Appendix 1
Errors

A.1.1 Error numbers generated by the Compiler

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

38. ']' or ',' expected in ARRAY declaration.
39. Lowerbound greater than upperbound.
40. Set too large (more than 256 possible elements).
41. Function result must be type identifier.
42. ',' or ']' expected in set.
43. '..' or ',' or ']' expected in set.
44. Type of parameter must be a type identifier.
45. Null set cannot be the first factor in a non-assignment statement.
46. Scalar (including real) expected.
47. Scalar (not including real) expected.
48. Sets incompatible.
49. '<' and '>' cannot be used to compare sets.
50. 'FORWARD', 'LABEL', 'CONST', 'VAR', 'TYPE' or 'BEGIN' expected.
51. Hexadecimal digit expected.
52. Cannot POKE sets.
53. Array too large (> 64K).
54. 'END' or ';' expected in RECORD definition.
55. Field identifier expected.
56. Variable expected after 'WITH'.
57. Variable in WITH must be of RECORD type.
58. Field identifier has not had associated WITH statement.
59. Unsigned integer expected after 'LABEL'.
60. Unsigned integer expected after 'GOTO'.
61. This label is at the wrong level.
62. Undeclared label.
63. The parameter of SIZE should be a variable.
64. Can only use equality tests for pointers.
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	FORWARD	FUNCTION
GOTO	IF	IN	LABEL	MOD	NIL
NOT	OF	OR	PACKED	PROCEDURE	PROGRAM
RECORD	REPEAT	SET	THEN	TO	TYPE
UNTIL	VAR	WHILE	WITH		

A 2.2 Special Symbols

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

```
+   -   *   /
=   <>  <   <=  >=  >
(   )   [   ]
{   }   (*  *)
^   :=  .   ,   ;   :
'   ..
```

A 2.3 Predefined Identifiers

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

For further information see **Section 2.**

```
CONST       MAXINT = 32767;

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

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.**

The information on the amount of storage required in each case should be of use to most programmers (the SIZE function may be used see **Section 2.3.6.7**); other details may be needed by those attempting to merge Pascal and machine code programs.

A 3.1.1 Integers

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

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

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

A 3.1.2 Characters, Booleans and other Scalars

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

Characters:

8 bit, extended ASCII is used.

```
'E'      ≡      #45
'['      ≡      #5B
```

Booleans:

`ORD(TRUE) = 1` so TRUE is represented by 1.

`ORD(FALSE) = 0` so FALSE is 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:

$2 \equiv 2*10^0$ or $1.0_2 * 2^1$

$1 \equiv 1*10^0$ or $1.0_2 * 2^0$

$-12.5 \equiv -1.25*10^1$ or $-25*2^{-1}$

-11001_2*2^{-1}

$-1.1001_2 *2^3$ when normalised.

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

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

```
          0.0001100
       ------------------
101 ) 0.1000000000000000
        101
        ---
        110
        101
        ---
         1000
          101
          ----
```

at this point we see that the fraction recurs ...

$\equiv \quad 0.1_2/101_2 \equiv 0.0001100_2$

$= \underline{1.1001100*2^{-4}}$ 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 format shown on the next page:

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

Thus:

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

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

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

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

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

A 3.1.4 Records and Arrays

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

Arrays: if n=number of elements in the array and s=size of each element then the number of bytes occupied by the array is n*s.

For example:

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 you might need information on how variables are stored at runtime:

Global variables	declared in the main program block.
Local variables	declared in an inner block.
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

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

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

then:

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

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

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

then:

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

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

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

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

Appendix 4
Some Example HiSoft Pascal Programs

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

```
{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
  Max=10;

TYPE
  Entry = RECORD
            Name : ARRAY [1..10] OF CHAR;
            Number : ARRAY [1..10] OF CHAR
          END;

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

BEGIN
{Set up the directory..}
  FOR I:= 1 TO Max 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 do the following..}
  TIN('Director',ADDR(Directory))
{And now you can process the directory as you wish.....}

END.
```

```
{Program to show the use of recursion}

PROGRAM FACTOR;

{This program calculates the factorial of a number input from the
 keyboard  1) using recursion  &  2) using an iterative method.}

 TYPE
   POSINT = 0..MAXINT;100
 VAR
   METHOD : CHAR;
   NUMBER : POSINT;

 {Recursive algorithm.}

 FUNCTION RFAC(N : POSINT) : INTEGER;

  VAR F : POSINT;

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

 {Iterative solution}

 FUNCTION IFAC(N : POSINT) : INTEGER;

   VAR I,F: POSINT;
   BEGIN
    F := 1;
    FOR I := 2 TO N DO F := F*I;                  {Simple Loop}
    IFAC:=F
   END;

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

```
{Program to list lines of a file in reverse order.
 Shows use of pointers, records, MARK and RELEASE.}

PROGRAM ReverseLine;

TYPE elem=RECORD                          {Create linked-list structure}
           next: ^elem;
           ch: CHAR
         END;
     link=^elem;

VAR prev,cur,heap: link;                  {all pointers to 'elem'}

BEGIN
 REPEAT                                   {do this many times}
   MARK(heap);                            {assign top of heap to 'heap'.}
   prev:=NIL;                             {points to no varaible yet.}
   WHILE NOT EOLN DO
      BEGIN
        NEW(cur);                         {create a new dynamic record}
        READ(cur^.ch);                    {and assign its field to one
                                           character from file.}
        cur^.next:=prev;                  {this field's pointer adresses}
        prev:=cur                         {previous record.}
      END;

{Write out the line backwards by scanning the records
 set up backwards.}

   cur:=prev;
   WHILE cur <> NIL DO                    {NIL is first}
     BEGIN
        WRITE(cur^.ch);                   {WRITE this field i.e. character}
        cur:=cur^.next                    {Address previous field.}
     END;
   WRITELN;
   RELEASE(heap);                         {Release dynamic variable space.}
   READLN;                                {Wait for another line}
 UNTIL FALSE                              {Use CC to exit}
END.
```

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

PROGRAM divmult2;

VAR r:REAL;

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

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

BEGIN
 REPEAT
  WRITE('Enter the number r ');
  READ(r);                                 {No need for READLN - see
                                            Section 2.3.1.4}

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

Please note that the above programs are included to show you how to solve particular problems in **HiSoft Pascal**; they are not meant to be examples of the Pascal language in general. You should consult one of the books in the **Bibliography** if you are learning Pascal. Remember, there is no substitution for understanding the program, don't just type it in blindly, with no thought.

Appendix 5
Turtle Graphics and Spectrum Sound & Graphics

It is very easy to use the sound and graphic facilities of the Spectrum with **HiSoft Pascal** through the use of USER and INLINE. In this Appendix, we first detail the supplied Turtle Graphics package and then show you how to interface with the Spectrum ROM so that you can do it yourself if you want to.

A 5.1 Turtle Graphics

HiSoft Pascal comes complete with a Logo-style Turtle Graphics package. This is supplied as a Pascal source program under the name TURTLE. It is to be found on your master **HiSoft Pascal** disk or on the B side of your master cassette.

The package is written in Pascal and can be loaded from within the Pascal editor by using the G command:

```
G,,TURTLE <ENTER>        from cassette or
G,,1:TURTLE              from disk drive 1
```

This will load the turtle graphics program and append it to any existing Pascal program. Note that, in order for it to function correctly, the Turtle Graphics program must be preceded by a normal PROGRAM heading and a VAR declaration; TYPE, CONST and LABEL declarations are optional. There must be no Procedures or Functions declared before the inclusion of the Turtle Graphics package.

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

The facilities available are as follows:

Global Variables

HEADING

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

XCOR, YCOR

These are the current (x,y) REAL co-ordinates of the turtle on the screen. The Spectrum graphics screen has a logical size of 256x176 pixels and the turtle may be positioned on any point within this area; if an attempt is made to move the turtle out of this *pool* (using LINE, see below) then the message Out of Limits will be displayed and the program will be aborted with a HALT message.

Initially XCOR and YCOR are undefined; use of the procedure TURTLE initialises them to 127 and 87 respectively, thus placing the turtle near the middle of his *pool*.

PENSTATUS

An integer variable holding the current status of the *pen* (i.e. the trail left by the turtle). 0 means the pen is down, 1 means the pen is up.

Procedures

The procedures PLOT, LINE and SPOUT included as standard in TURTLE are described later. The only essential difference in their implementation here is that LINE calls the CHECK procedure which ensures that XCOR and YCOR cannot go outside the screen boundaries. The other procedures available are:

INK (C: INTEGER)

This takes an integer between 0 and 8 inclusive and sets the ink colour of the turtle's pen accordingly.

PAPER (C:INTEGER)

Sets the background (paper) colour of the screen to the colour associated with the ink C which is an integer in the range 0 to 8 inclusive.

COPY

Downloads the current screen to the ZX Printer; useful for getting a copy of a completed graphics page.

PENDOWN (C:INTEGER)

Sets the turtle state so that it will leave a trail in the ink colour associated with the parameter C. This procedure assigns 0 to PENSTATUS.

PENUP

Subsequent to a call to this procedure the turtle will not leave a trail. Useful for moving from one graphic section to another. PENUP assigns the value 1 to PENSTATUS.

SETHD (A:REAL)

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

SETXY (X,Y : REAL)

Sets the absolute position of the turtle within the graphics area to the value (X,Y). No check is made within this procedure to ascertain if (X,Y) is out of bounds; procedure LINE does this check.

FWD (L : REAL)

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

BACK (L:REAL)

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

TURN (A : REAL)

Changes the turtle's heading by A degrees without moving it. The heading is increased in the anti-clockwise direction.

VECTOR (A,L : REAL)

Displaces the turtle's position by L units at a heading of A; the turtle's heading remains A after the displacement.

RIGHT (A : REAL)

Changes the turtle's heading by A degrees without moving it. The heading is increased in the clockwise direction.

LEFT (A : REAL)

Changes the turtle's direction by A degrees anti-clockwise. Identical to TURN.

ARCR (R:REAL; A:INTEGER)

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

TURTLE

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

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

Example Turtle Graphics Programs

In all the example programs given below we assume that you have already loaded **HiSoft Pascal** and used G,,TURTLE <ENTER> to load the Turtle Graphics package into the editor (it starts at line 10 and finishes at line 1350). Now proceed with the examples:

CIRCLES

```
   1  PROGRAM CIRCLES;
   2  VAR I:INTEGER;


1390  BEGIN
1400   TURTLE;
1410   FOR I:=1 TO 9 DO
1420    BEGIN
1430     ARCR(0.5,360);
1440     RIGHT(40)
1450    END
1460  END.
```

SPIRALS

```
   1  PROGRAM SPIRALS;
   2  VAR

1390  PROCEDURE SPIRALS ( L,A:REAL );
1400  BEGIN
1410   FWD(L);
1420   RIGHT(A);
1430   SPIRALS(L+1,A)
1440  END;
1450  BEGIN
1460   TURTLE;
1470   SPIRALS(9,95)          {or (9,90)  or  (9,121)  ... }
1480  END.
```

FLOWER

```
   1  PROGRAM FLOWER;
   2  VAR

1390 PROCEDURE PETAL ( S:REAL );
1400  BEGIN
1410   ARCR(S,60);
1420   LEFT(120);
1430   ARCR(S,60);
1440   LEFT(120)
1450  END;
1460  PROCEDURE FLOWER ( S:REAL );
1470  VAR I:INTEGER;
1480  BEGIN
1490   FOR I:=1 TO 6 DO
1500    BEGIN
1510     PETAL(S);
1520     RIGHT(60)
1530    END
1540  END;
1550  BEGIN  TURTLE;
1560   SETXY(127,60);
1570   LEFT(90); FWD(10);
1580   RIGHT(60); PETAL(0.2);
1590   LEFT(60);  PETAL(0.2);
1600   SETHD(90); FWD(40);
1610   FLOWER(0.4)
1620  END.
```

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

A 5.2 Sound & Graphics with the ROM

We can easily define Pascal procedures to call the Spectrum ROM in order to control the sound and graphic capabilities of the Spectrum.

A 5.2.1 Sound

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

```
{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 0 which it converts into a
period of silence. For non-zero frequencies the frequency and length
of the note are converted approximately to the values required by
the Spectrum ROM routine and this is then called via BEEPER}

PROCEDURE BEEP (FREQUENCY : INTEGER; LENGTH : REAL);
VAR I : INTEGER;
BEGIN
 IF FREQUENCY=0 THEN FOR I:=1 TO ENTIER(12000*LENGTH) DO
 ELSE
 BEEPER(ENTIER(FREQUENCY*LENGTH),ENTIER(437500/FREQUENCY-30.125));
 FOR I:1 TO 100 DO                {short delay between notes}
END;
```

Example of the use of BEEP:

```
BEEP(262, 0.5);
BEEP(0, 1);      {sounds middle C for 0.5 secs then a 1 sec silence}
```

A 5.2.2 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 off and *vice versa*. This effect is identical to that caused by Spectrum BASIC's OVER command.

```
{A procedure that mirrors the BASIC PLOT command. Simply plots the
point X,Y ON or OFF depending on whether the first parameter to the
procedure is TRUE or FALSE}

PROCEDURE PLOT (ON : BOOLEAN; X,Y : INTEGER);
BEGIN
 IF ON THEN WRITE(CHR(21), CHR(0))
       ELSE WRITE(CHR(21), CHR(1));

 INLINE(#FD, #21, #3A, #5C,        { LD   IY,#5C3A }
        #DD, #46, 2, #DD, #4E, 4, { LD   B,(IX+2) : LD   C,(IX+4) }
        #CD, #E5, #22);           { CALL #22E5  ;ROM PLOT routine }
END;

{Called by the LINE procedure, LINE1 is used to pass the correct
arguments to the ROM's DRAW routine}

PROCEDURE LINE1(X,Y,SX,SY : INTEGER);
BEGIN
 INLINE(#FD, #21, #3A, #5C,        { LD   IY,#5C3A }
        #DD, #56, 2, #DD, #5E, 4, { LD   D,(IX+2) : LD   E,(IX+4) }
        #DD, #46, 6, #DD, #4E, 8, { LD   B,(IX+6) : LD   C,(IX+8) }
        #CD, #BA, #24)            { CALL #24BA    ;ROM DRAW routine}
END;

{LINE draws a line from the current plot position (x,y) to
(x+X,y+Y). The line may be 'on' or 'off' depending on the value of
the BOOLEAN parameter ON.}
```

```
PROCEDURE LINE(ON : BOOLEAN; X,Y : INTEGER);
VAR
 SGNX, SGNY : INTEGER;
BEGIN
 IF ON THEN WRITE(CHR(21), CHR(0))
       ELSE WRITE(CHR(21), CHR(1));
 IF X<0 THEN SGNX:=-1 ELSE SGNX:=1;   {the DRAW routine that is to}
 IF Y<0 THEN SGNY:=-1 ELSE SGNY:=1;   {called within LINE1 needs}
                                      {the absolute values of X and}
                                      {Y and their signs}
 LINE1(ABS(X), ABS(Y), SGNX, SGNY);   {Plot the line}
END;
```

Example of the use of PLOT and LINE:

```
PLOT(ON, 50, 50);
LINE(ON, 100, -50);     { draws a line from (50,50) to (150,0) }
```

A 5.2.3 Output Directly through the ROM

There are occasions where it is useful to output characters or control codes directly through the Spectrum ROM RST #10 routine rather than use Pascal's 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 moved. However, if this is done using the Pascal WRITE statement, certain values of X and Y will not be passed to the ROM but trapped by Pascal (e.g. 8 which is interpreted by Pascal as *delete last character*) thus causing the PRINT AT command to malfunction. You can overcome this problem by using the following procedure:

```
{SPOUT outputs a character directly through the ROM's RST #10
routine, avoiding any trapping by Pascal of the value output }

PROCEDURE SPOUT (C : CHAR);
BEGIN
 INLINE(#FD, #21, #3A, #5C,          { LD   IY,#5C3A }
        #DD, #7E, 2,                 { LD   A,(IX+2) }
        #D7)                         { RST  #10   ;ROM output }
END;
```

Example of the use of SPOUT:

```
SPOUT( CHR(22) ); SPOUT( CHR(8) ); SPOUT( CHR(13) );
  { sets the print position to line 8, column 13 }
```

Please feel free to use and modify the above routines in any way you see fit.

Bibliography

The following are recommended reading; you may find them in your local library.

Tutorial Style Books

W. Findlay
D.A. Watt
Pascal: An Introduction to Methodical Programming
3rd Edition 1985. Pitman. ISBN 0 273 02188 5

Boris Allan
Introducing Pascal.
1st Edition 1984. Granada. ISBN 0 246 12323 0

R. Forsyth
Pascal at Work and Play.
1st Edition 1982. Chapman & Hall. ISBN 0 412 23380 0

Reference Books

N. Wirth
K. Jensen
Pascal User Manual and Report.
2nd Edition 1975. Springer-Verlag. ISBN 0 387 90144 2

J. Tiberghien
The Pascal Handbook,
1st Edition 1981. Sybex. ISBN 0 89588 053 9

I. Logan
F. O'Hara
Spectrum ROM Disassembly
1st Edition 1983. Melbourne House. ISBN 0 86161 116 0

Other HiSoft Programs for your Spectrum

HiSoft Devpac

The *standard* assembler development system, **HiSoft Devpac** has been used by all the top software houses and is ideal for professional work or simply for learning assembly language.

Devpac is a fast macro assembler/editor coupled with a front panel disassembler/debugger. On a disk system (ZX Plus 3, Opus, Disciple and microdrive) you can assemble disk-to-disk allowing any size program to be assembled. The debugger is small and has a host of features (including single-step, *even in ROM*) that allow you to trace faults in your programs or to hack other programs. *Cassette/microdrive,Opus Discovery, Disciple, Plus 3 versions.*

HiSoft C

Ideal for learning the language, **HiSoft C** is a standard Kernighan & Ritchie compiler for the Spectrum. The only real omission is floating-point arithmetic, **HiSoft C** handles 16-bit integer numbers. Files are fully supported on microdrive and there are extra functions to handle the Spectrum's sound and graphics capabilities. *Cassette/microdrive version.*

HiSoft BASIC

Compiles virtually all of Spectrum BASIC very quickly indeed. Unlike other floating-point BASIC compilers, **HiSoft BASIC** produces code that runs up to 100 times faster than interpreted BASIC. This is because you can specify which variables are floating-point and which are integer, allowing the compiler to optimise your code, just like other language compilers do. Programs up to 40K can be compiled on the 128K Spectrums (30K on 48K machines) at the touch of a key and then saved to tape or disk as stand-alone programs. *Cassette/microdrive, Opus Discovery, Disciple, Plus 3 versions.*

Please write or phone for details of the above products and their prices.