HiSoft Pascal

ENTERPRISE
PROGRAMS

©Hisoft 1985

INDEX

SECTION 0 PRELIMINARIES

0.0 Introduction 1
0.1 Loading HiSoft Pascal into your Enterprise 1
0.2 Scope of this manual 1
0.3 Compiling and running 2

SECTION 1 SYNTAX AND SEMANTICS

1.1 Identifier 4
1.2 Unsigned Integer 4
1.3 Unsigned number 4
1.4 Unsigned constant 5
1.5 Constant 6
1.6 Simple type 6
1.7 Type 7
1.8 Field list 9
1.9 Variable 9
1.10 Factor 10
1.11 Term 10
1.12 Simple expression 10
1.13 Expression 11
1.14 Parameter list 11
1.15 Statement 11
1.16 Block 14
1.17 Program 15
1.18 Strong typeing 16

SECTION 2 PREDEFINED IDENTIFIERS

2.1 Constants 17
2.2 Types 17
2.3 Variables 17
2.4 Procedures and functions 17

SECTION 3 COMMENTS AND COMPILER OPTIONS

3.1 Comments 25
3.2 Compiler options 25

SECTION 4 THE INTEGRAL EDITOR 27

SECTION 5 TURTLE GRAPHICS 30

APPENDIX 1 ERRORS 34

APPENDIX 2 RESERVED WORDS AND PREDEFINED IDENTIFIERS 36

APPENDIX 3 DATA REPRESENTATION AND STORAGE 36

APPENDIX 4 EXAMPLE HISOFT PASCAL PROGRAMS 40

BIBLIOGRAPHY

SECTION 0 PRELIMINARIES

0.0 INTRODUCTION

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

The FILE structure is not supported; although variables may be saved to and loaded from tape or disc.
A RECORD type may not have a VARIANT part.
PROCEDUREs and FUNCTIONs are not valid as parameters.

Some extra functions and procedures are included to reflect the microcomputer environment (and EXOS in particular); among these are INLINE, POKE, PEEK, ADDR and EXOS.

The compiler occupies approximately 12K of storage, the editor takes up 2K and the run times roughly 5K. The package is supplied on cassette tape.

0.1 LOADING HISOFT PASCAL INTO YOUR ENTERPRISE

HiSoft Pascal is supplied as an Application Program running at address £100 (throughout this manual £ is used to denote hexadecimal) on the Enterprise computers; to load it into your computer you should place the cassette into your tape deck with the label showing HiSoft Pascal Compiler facing upwards, ensure that the tape is rewound to the beginning and then press:

FUNCTION KEY 1 PLAY	from within BASIC and on your tape recorder
or	
FUNCTION KEY 1 ENTER PLAY	from within the word processor and on your tape recorder

HiSoft Pascal will then load and autostart, producing a menu of the editor commands.

0.2 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 describes the integral editor used to create, amend and save your Pascal programs.

Section 5 describes the Turtle Graphics package supplied with the Pascal on the reverse side of the tape.

The above Sections should be read carefully by all users.

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

Appendix 2 lists the predefined identifiers and reserved words.

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

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

0.3 COMPILING AND RUNNING

When writing a Pascal program for use with HiSoft Pascal you can type in your source text using the editor contained within the Pascal package or any editor that produces a pure ASCII text stream (each line ending with a CR and LF character pair) on tape or disc.

For example type the following immediately after you have loaded the Pascal (i.e. from within the integral editor), ending each line with the ENTER key:

```
I10,10      ENTER

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

  STOP

C     ENTER
```

The source code will be compiled and assuming no errors are detected the object code will be created in memory. This can be run by simply typing:

```
R     ENTER
```

If there are any errors you should hit 'E' to exit the compiler. Then use the editor to edit the program and correct the typing mistake.

When this program is run it will prompt you to enter numbers and give the factorial of each number input until you type STOP which will return you to the editor.

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 HOLD ; then use STOP to return to the editor or HOLD again to continue the compilation.

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 this line, 'P' to return to the editor and edit the previous line (useful for missing semi-colons), STOP to abort, or any other key to continue the compilation. Normally it is best to use 'P' or 'E' on reaching the first error.

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 the message 'No Table Space' will be displayed and control returned to the editor. You should then use the editor's 'A' command to increase the table size. See Section 4 for details.

If a compilation contains any errors then the number of errors detected will be given and any object code produced will be deleted.

During a run of the object code various runtime error messages may be generated (see Appendix 1). You may suspend a run while it is outputting to the screen or printer by using HOLD ; subsequently use STOP to abort the run and return to the editor or HOLD again to resume the run.

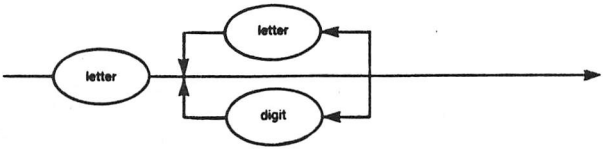
When using HiSoft Pascal the function keys are programmed as follows:

Key Number	Without Shift	With shift
1	HELP [ENTER]	READ(
	PROGRAM	WRITE(
3	CONST [ENTER]	INTEGER
4	TYPE [ENTER]	WRITELN
5	VAR [ENTER]	WHILE
6	BEGIN [ENTER]	UNTIL
7	END	REPEAT [ENTER]
8	PROCEDURE	FUNCTION

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).

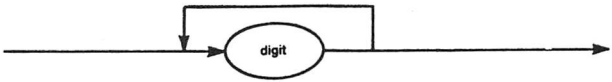
1.1 IDENTIFIER



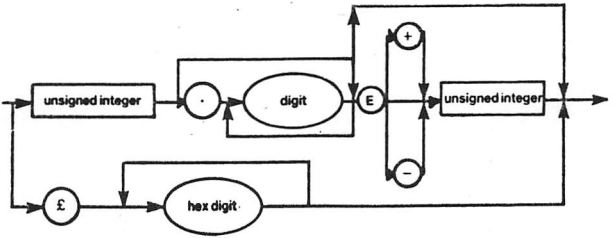
Only the first 10 characters of an identifier are treated as significant.

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 then 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.

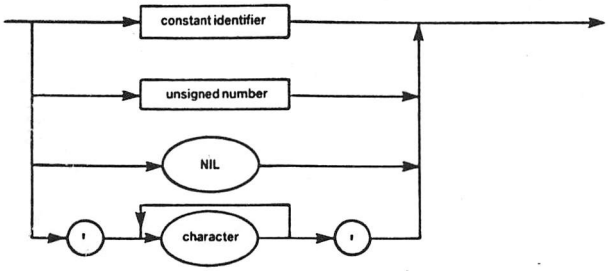
The largest real available is 3.4E38 while the smallest is 5.9E-39.

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.

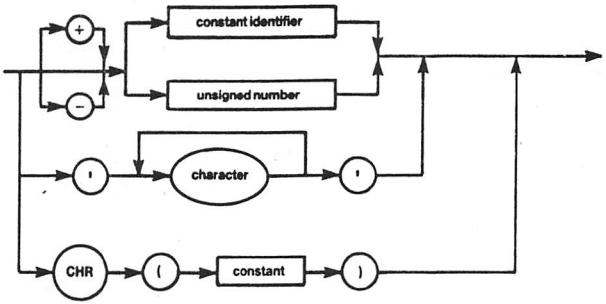
1.4 UNSIGNED CONSTANT



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.

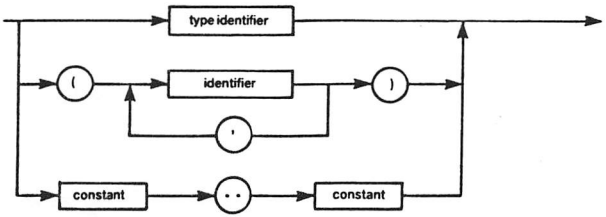
1.5 CONSTANT



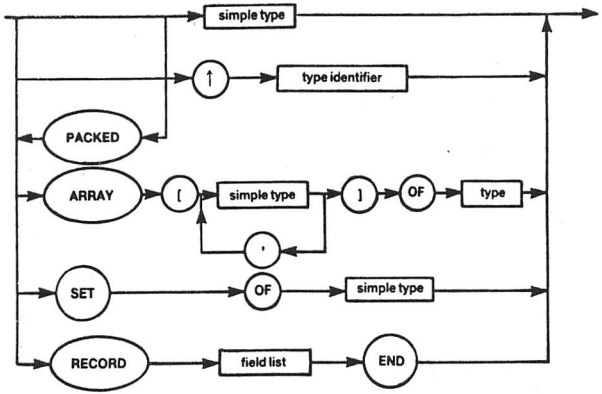
The non-standard CHR construct is provided here so that constants may be used for control characters. In this case the constant in parentheses must be of type integer.

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.



The reserved work 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. The above restriction does not hold for the special case of arrays of a string type, since arrays of this type are always used to represent similar data.

1.7.2 POINTERS

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

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

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

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

        link = ^item;
```

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

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

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

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

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

1.7.3 RECORDs

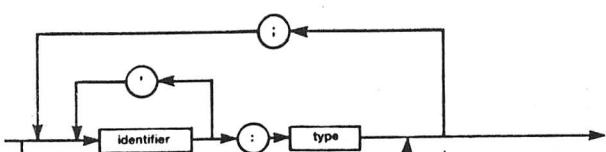
The implementation of RECORDs, structured variables composed of a fixed number of constituents called fields, within HiSoft Pascal is as Standard Pascal except that the variant part of the field list is not supported.

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

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

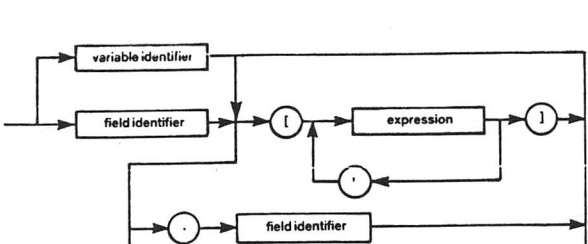
See Appendix 4 for an example of the use of WITH and RECORDs in general.

1.8 FIELD LIST



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

1.9 VARIABLE



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

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

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

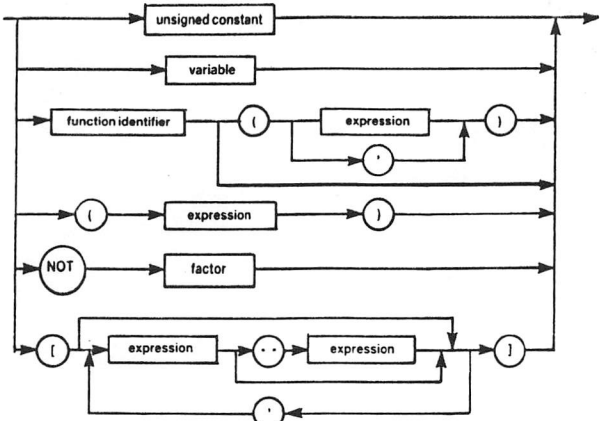
When specifying elements of multi-dimensional arrays the programmer is not forced to use the same form of index specification in the reference as was used in the declaration.

e.g. if variable a is declared as:

ARRAY[1..10] OF ARRAY[1..10] OF INTEGER

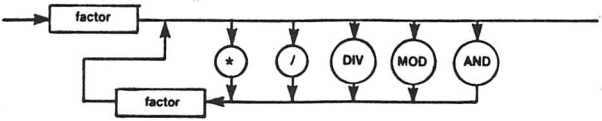
then either a[1][1] or a[1,1] may be used to access element (1,1) of the array.

1.10 FACTOR



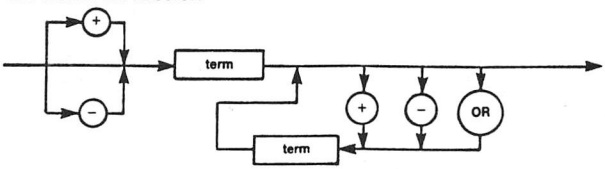
See EXPRESSION in Section 1.13 and FUNCTIONs in Section 3 for more details.

1.11 TERM



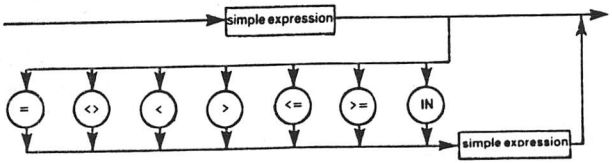
The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

1.12 SIMPLE EXPRESSION



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

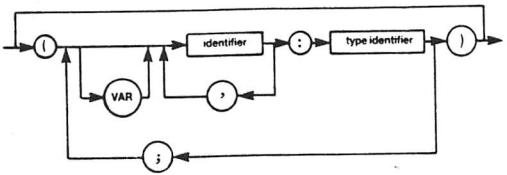
1.13 EXPRESSION



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

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

1.14 PARAMETER LIST



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

Variable parameters as well as value parameters are fully supported.

Procedures and functions are not valid as parameters.

1.15 STATEMENT

Refer to the syntax diagram on page 13.

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

CASE statements:
An entirely null case list is not allowed i.e. CASE OF END; will generate an error (*ERROR* 13).

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

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

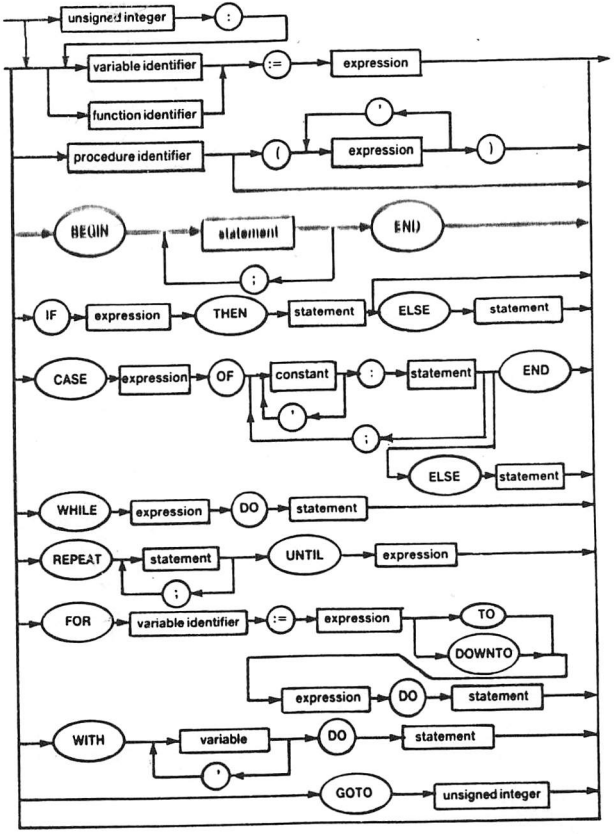
FOR statements:
Variable parameters may not be used as FOR statement variables. This would be inefficient and it is certainly usual practice to use variable parameters sparingly, using local parameters instead, where possible.

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.

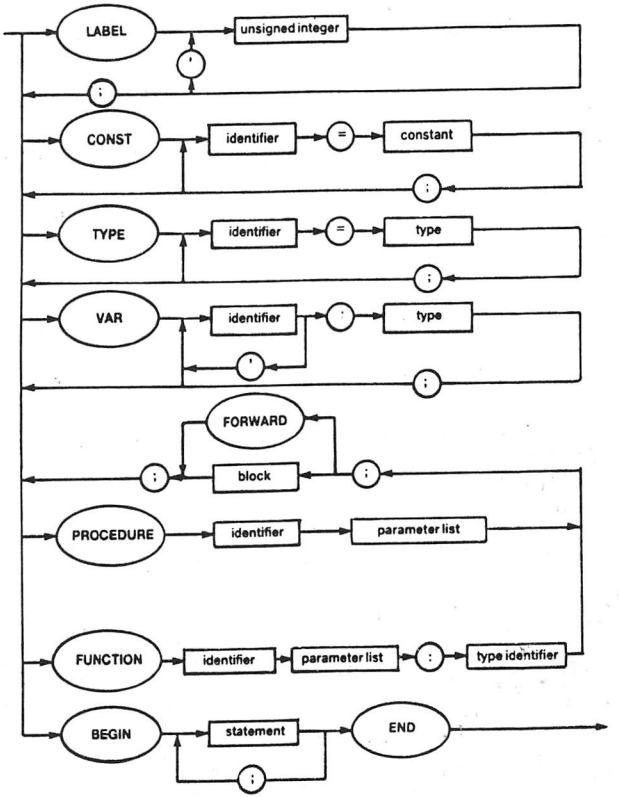
You may not use GOTO to jump out of a FOR statement.

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 — ':'.

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

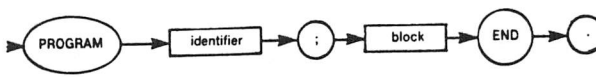

LABEL
unsigned integer
,
;
CONST
identifier
=
constant
;
TYPE
identifier
=
type
;
VAR
identifier
:
type
,
;
FORWARD
;
block
;
PROCEDURE
identifier
parameter list
FUNCTION
identifier
parameter list
:
type identifier
BEGIN
statement
END
;

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 *)
  BEGIN
    ....
    b(q);
    ....
  END;
```

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

1.17 PROGRAM



Note that the 'microcomputer standard' form is used i.e. there are no formal parameters of the program.

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 integers and strings and, sometimes, between integers and reals (perhaps using the '%' sign to denote integers). Then comes Pascal which goes as far as allowing distinct user-enumerated types. At the top of the scale (at present) is a language like ADA in which one can define different, incompatible numeric types.

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

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

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

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

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

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

SECTION 2 PREDEFINED IDENTIFIERS

2.1 CONSTANTS

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

2.2 TYPES

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

2.3 VARIABLES

RA, RB, RC, RD, RE, RH, RL, RBC, RDE, RHL	RA, RB, RC, RD, RE, RH, RL are CHARacter variables which are mapped onto the Z80 registers A, B, C, D, E, H and L respectively. RBC, RDE and RHL are INTEGER variables mapped on to BC, DE and HL.

2.4 PROCEDURES AND FUNCTIONS

2.4.1 Input and Output Procedures

2.4.1.1 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 expression e to the screen or printer as appropriate. Use CHR(16) to toggle the printer on and off when WRITEing.

Generally though:

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

BEGIN WRITE(P1); WRITE(P2);.....; WRITE(Pn);

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

e or e:m or e:m:n or e:m:H

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

We have 5 cases to examine:

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

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

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

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

WRITE (1025:m:H);

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

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

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

If n is not present then the number is output in scientific notation, with a mantissa and an exponent. If the number is negative then a minus sign is output prior to the mantissa, otherwise a space is output. The number is always output to at least one decimal place up

to a maximum of 5 decimal places and the exponent is always signed (either with a plus or minus sign). This means that the minimum width of the scientific representation is 8 characters; if the field width m is less than 8 then the full width of 12 characters will always be output. If m) =8 then one or more decimal places will be output up to a maximum of 5 decimal places (m=12). For m) 12 leading spaces are inserted before the number. Examples:

```
WRITE (—1.23E 10:m);

m=7     gives:    —1.23000E+10
m=8     gives:    —1.2E+10
m=9     gives:    —1.23E+10
m=10    gives:    —1.230E+10
m=11    gives:    —1.2300E+10
m=12    gives:    —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.

2.4.1.2 WRITELN

WRITELN causes subsequent output to occur on a new line.

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

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

2.4.1.3 PAGE

The procedure PAGE causes an ASCII form feed character to be written to the screen or printer. To clear the screen you should use WRITE(CHR(26)).

2.4.1.4 READ

The procedure READ is used to access data from the keyboard.

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.

A character is effectively read from the buffer and assigned 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 the value CHR(13) will be returned by READ. When a read operation is subsequently performed the text window will be positioned at the start of the new line.

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

2] V is of type string.

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

The note in 1] above also applies here.

3] V is 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.

2.4.1.5 READLN

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

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

READLN simply reads in a new buffer from the keyboard; while typing in the buffer you may use the standard Enterprise editing facilities. 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 from 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.

2.4.2 FILE HANDLING FUNCTIONS

2.4.2.1 EOLN

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

2.4.2.2 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. If a key is held down then it will repeat at its normal rate.

2.4.3 TRANSFER FUNCTIONS

2.4.3.1 TRUNC(X)

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

TRUNC(—1.5) returns —1 TRUNC(1.9) returns 1

2.4.3.2 ROUND(X)

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

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

2.4.3.3 ENTIER(X)

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

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

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

2.4.3.4 ORD(X)

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

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

Examples:

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

2.4.3.5 CHR(X)

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 '['

2.4.4 ARITHMETIC FUNCTIONS

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

2.4.4.1 ABS(X)

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

2.4.4.2 SQR(X)

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

2.4.4.3 SQRT(X)

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

2.4.4.4 FRAC(X)

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

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

Examples:

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

2.4.4.5 SIN(X)

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

2.4.4.6 COS(X)

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

2.4.4.7 TAN(X)

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

2.4.4.8 ARCTAN(X)

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

2.4.4.9 EXP(X)

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

2.4.4.10 LN(X)

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

2.4.5 FURTHER PREDEFINED PROCEDURES

2.4.5.1 NEW(p)

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

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

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

2.4.5.2 MARK(v1)

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

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

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

2.4.5.3 RELEASE(v1)

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

See above and Appendix 4 for more details.

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

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

2.4.5.5 USER(V)

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

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

2.4.5.6 HALT

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

2.4.5.7 POKE(X,V)

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

Examples:

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

2.4.5.8 TOUT (NAME,START,SIZE)

TOUT is the procedure which is used to save variables to a device. The first parameter is of type ARRAY[1..n] OF CHAR; and is the EXOS device/filename of the file to be saved. SIZE bytes of memory are dumped starting from address START. Both these parameters are of type INTEGER. For compatibility with other versions of HiSoft Pascal you should use a filename of exactly 8 characters padded to the right with spaces and use the default device.

E.g. to save the variable TEMP to the default device under the name TESTDATA, you should use:

```
TOUT('TESTDATA',ADDR(TEMP),SIZE(TEMP) );
```

The use of actual memory addresses (via ADDR) gives you far more flexibility than the ability to save arrays.

2.4.5.9 TIN(NAME,START)

This procedure is used to load variables etc. that have been saved using the TOUT procedure. NAME is of type ARRAY[1..n] OF CHAR but should be exactly 8 characters in length if you wish compatibility with other versions of HiSoft Pascal, while START is of type INTEGER. The device is searched for the specified filename (if relevant) which is then loaded at memory address START. The length of the file is stored in the file by TOUT.
E.g. to load the variable saved in the above example use:

```
TIN('TESTDATA',ADDR(TEMP) );
```

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

2.4.5.10 OUT(i,c)

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

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

2.4.5.11 RANSEED (X,Y,Z)

This procedure has three INTEGER parameters which are used as seeds for the random numbers produced by RANDOM. See Section 2.4.6.1. These integers must be between and 30000.

In order to use the same random numbers for different runs of a program you should use this procedure.

2.4.6 FURTHER PREDEFINED FUNCTIONS

2.4.6.1 RANDOM

This returns a pseudo-random REAL number between 0.0 and 1.0.

The algorithm used is based on that of B.A. Wichmann and I.D. Hill (NPL Report DITC 6/82). The seeds may be changed by using the procedure RANSEED (See section 2.4.5.11 above).

2.4.6.2 SUCC(X)

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

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

2.4.6.3 PRED(X)

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

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

2.4.6.4 ODD(X)

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

2.4.6.5 ADDR(V)

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

2.4.6.6 PEEK(X,T)

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

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

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

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

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

2.4.6.7 SIZE(V)

The parameter of this function is a variable of any type. The function returns an INTEGER result which is the amount of storage occupied by this variable, in bytes.

2.4.6.8 INP(p)

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

2.4.6.9 EXIS(f)

This powerful function has one INTEGER parameter which is the EXOS function number passed in the A register and returns a character value returned by EXOS in the A register. This is normally a flag to indicate if the operation was successful. CHR(0) is returned in this case. Other arguments are passed to and from EXOS using the predefined variables RB, RC, RD, RE, RH, RL, RDE, RBC and RHL. For examples of the use of the EXOS function please see the source of the Turtle graphics package supplied with HiSoft Pascal.

2.4.6.10 MAKESTR(s)

This function is used to pass strings to EXOS as EXOS requires strings to be stored in a different form to HiSoft Pascal. MAKESTR has one parameter of a string type (ARRAY [1..N] OF CHAR) and returns an integer value of the address of the corresponding EXOS string. This returned value may then be assigned to RDE etc. before calling EXOS. See the Turtle graphics package for an example of its use.

SECTION 3 COMMENTS AND COMPILER OPTIONS

3.1 COMMENTS

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

3.2 COMPILER OPTIONS

Compiler options should be included in your program between comment braces and the first option in the list (which is separated by commas) should be preceded by a dollar symbol:

Examples:

```
(*$O—,A—*)
{$L— }
```

The following options are available:

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

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

DEFAULT: L+

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

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

If O— then the above checks are not made.

DEFAULT: O+

Option 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.

The address of the stack may be set at compilation time using the editor's 'A' command - see Section 4.

If S— then no stack checks are performed.

DEFAULT: S+

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

If A+ and an array index is too high or too low then the message

'Index too high' or 'Index too low'

will be displayed and the program execution halted.

If A— then no such checks are made.

DEFAULT: A+

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

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

DEFAULT: I—

Option P device name:
This option directs the compilation to the device specified after the $P. There must be a space separating the device name and the comment terminator. Thus to start the compiler listing going to the printer you would use:

(*$P PRINTER: *)

All subsequent compiler listing will be printed on your printer until either the end of the source is reached or another $P is encountered.

DEFAULT: P VIDEO:

Option F:
This option letter must be followed by a space and then a valid EXOS device/filename string.

The presence of this option causes inclusion of Pascal source text from the specified file from the end of the current statement - useful if the programmer wishes to build up a 'library' of much-used procedures and function on tape and then include them in particular programs. The device name used can be any valid EXOS input device such as SERIAL: etc. and must be followed by at least one space.

Example: ($F TEST.PAS); will include the text file TEST.PAS from the default device (normally TAPE:).

When using this facility from tape without motor control you should use the L— option so that the compiler and the screen can keep up with the tape.

This option may not be nested.

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

The integral editor supplied with HiSoft Pascal is a combination of the standard HiSoft line editor and the EXOS screen editor; this partnership leads to simple yet powerful creation and editing of your Pascal programs.

While within the editor you may use all the normal screen editing facilities that you would use in BASIC (see pages 36 and 37 of your Enterprise BASIC manual) together with the following Pascal-specific features, each command should be finished with [ENTER].

EXOS commands may be typed by prefixing them with a colon e.g. :HELP as in BASIC.

Below the syntax of each command is given followed by an example and then the description of the command. In what follows

n1,n2,n3	are line numbers.
s,s1,s2	are strings.
i	is a line number increment.

INSERT TEXT **COMMAND In,i**

I100,10

Insert text starting at line 100, incrementing line numbers in 10s; this does a true insert, moving up existing lines as necessary. The default for n and m is 10. Line numbers are generated automatically for you, like BASIC's AUTO. Hit STOP to get out.

EDIT TEXT **COMMAND En**

E100

Edit line 100; this displays the line on the screen - now use the joystick etc. to edit the line and press [ENTER] to save the changes in the file.

DELETE TEXT **COMMAND Dn1,n2**

D10,50

Delete line 10 thru 50 inclusive. The command must have two line numbers to function; these may be the same. To delete just one line you type the line number followed by [ENTER] as in BASIC.

MOVE TEXT **COMMAND Mn1,n2,n3**

M10,30,200

Move the block of text that exists between lines 10 and 30 inclusive to just **before** line 200. Intelligent move up or down the text file. You cannot move a block within itself so that M50,100,65 is illegal and will not be carried out.

RENUMBER TEXT **COMMAND Nn1,i**

N5,10

Renumber the text so that it becomes first line 5, incrementing in steps of 10. If this can't be done, the command aborts. Both numbers must be present for the command to go ahead. If the renumbering would cause a line number greater than 65535 to be created then the numbering is left originally.

LIST TEXT **COMMAND Ln1,n2**

L100,200

List the textfile between lines 100 and 200 inclusive. You may pause this listing by pressing HOLD: then press HOLD again to resume or STOP to abort and return to the editor loop. Without any arguments L lists the whole textfile.

PUT TEXT TO DEVICE **COMMAND Pn1,n2,s**

P1,9999,TEST.PAS

Sends the textfile or part (1..9999) of the textfile to the default or specified external device with the name given; you may interrupt this process by pressing STOP.

GET TEXT FROM EXTERNAL DEVICE **COMMAND G,,s**

G,,TAPE:TEST.PAS

Searches the default or specified external device for the TEST.PAS file; when found it is appended to the current source; press STOP to abort this command. The source that is appended will be renumbered with an increment of one starting from the last line number of the existing text.

VERIFY TEXTFILE **COMMAND V,,s**

V,,TEST.PAS

Verifies a found file on an external device with the textfile in memory; displays 'Verified' or 'Failed!' depending on the outcome of the comparison.

FIND STRING IN TEXT **COMMAND Fn1,n2,s**

F30,50,PROCEDURE

Searches for the string PROCEDURE between lines 30 and 50 inclusive; if the string is found then the line in which it is found is displayed on the screen and edit mode is entered - change at will or use the substitute command by pressing [ENTER] and:

SUBSTITUTE STRING **COMMAND Sn1,n2,s1,s2**

S1,9999,CHAR,INTEGER

Substitutes the string CHAR for the string INTEGER over the specified line range; you are prompted each time to substitute the string by the line being displayed in Edit mode on the screen; hit [ENTER] and then F or S to continue finding and substituting.

CHANGE DELIMITER **COMMAND Q,,s**

Q,,/

This command changes the separator between arguments when editing (it is normally a comma). This lets you search for a string which contains a comma. If you use this command by mistake you can find out what the delimiter is using the Y command.

SHOW DEFAULTS **COMMAND Y**

Y

This command shows the current delimiter and the current defaults for line numbers and strings.

DISPLAY UPPER LINE NUMBER **COMMAND U**

U

Displays the last number in the current textfile on the screen.

COMPILE PROGRAM **COMMAND Cn**

C

Compiles the current Pascal program starting from line number n. If no line number is given then compilation starts from the first line of the text.

RUN OBJECT CODE **COMMAND R**

R

Run the object code that you have created using the Compile command. Note that this cannot be used if you have changed the program or used the 'A'lter command (see below).

TRANSLATE PROGRAM **COMMAND Tn,,s**

T,,TEST.OBJ

This compiles your program as if the code was located directly after the runtime routines and then prompts you to continue. If you answer Y then the runtimes and your code are saved to the device/filename specified by the string s. The code may then be loaded in the standard way using 'START' from BASIC or from the word processor.

This allows you to save compact object code.

ALTER DEFAULTS **COMMAND A**

A

This command lets you view and modify some of the compiler's internal variables. The current value in hexadecimal is displayed after the variables name. If you type [ENTER] then no changes are made, but if you type a hexadecimal value then this is used as the new value of the variable.

The first variable is the Symbol Table Size. Normally it is not necessary to change this, but if the compiler gives the fatal error message 'No Table Space' then this can be used to make the table larger.

Normally the compiler allocates as much RAM as possible for itself and the programs that it generates. This memory is allocated from low addresses to high addresses. The highest value used when a program produced using the Translate command is called the Translate Stack and the highest used when the compiler is in memory is called the Compiler Stack. On an Enterprise 64 the default value for these variables is £C000 and on the Enterprise 128 it is £0000.

On the Enterprise 128 there is normally no need to change these values unless you are writing a program to run on the Enterprise 64 when you should lower the translate stack.

When using the Enterprise 64 to run graphics programs you will need to lower the stack to £8000 to free 16k to the operating system to use as extra video RAM.

Thus when using the Turtle Graphics package on the 64 you should type A[ENTER] [ENTER] [ENTER] 8000 [ENTER] before compiling the program.

CHANGE SCREEN WIDTH **COMMAND W**

W

This command clears the screen and flips from 40 column to 80 column mode or vice versa. If the operating system does not have enough memory to open an 80 column screen then you are left in 40 column mode.

BUFFER LEFT **COMMAND B**

B

This command displays the number of bytes left for your source and object code in hexadecimal. If you start running out of space you should save your text and use the $F compiler option. See Section 3.

LOAD EXOS MODULE **COMMAND X,,s**

X,,MODULE

This command loads an EXOS module from the specified file or device.

SECTION 5 TURTLE GRAPHICS

HiSoft Pascal comes complete with a Logo-style Turtle Graphics package on the 'B' side of the program cassette.

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

G,, [ENTER]

This will load the turtle graphics package and example program.

Before compiling and running the example program on the Enterprise 64 you should use the Alter command to give the operating system some memory to use for the graphics screen. Type A [ENTER] [ENTER] [ENTER] 8000 [ENTER] to do this.

To compile the program type C [ENTER] and then when the compiler asks

Run?

type Y to run the program. You will then be prompted for a number between 1 and 5. Type one in followed by [ENTER] and a space-filling curve will be drawn. You will then be prompted again for another number. After plotting 6 curves the program will return to the editor. If you get bored press RESET once and you will be returned to the editor. Pressing STOP leaves the screen set up for graphics - use the editor's W command to return to a full text screen.

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 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 can 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 O with the procedure TURTLE (see below). The value O 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 Enterprise graphics screen has a size of 1343*971 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 matrix (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 400 and 189 respectively, thus placing the turtle near the middle of his 'pool'.

PENON

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

PROCEDURES

The procedures available are as follows:

WRVAR (I:INTEGER)

updates the value of the EXOS variable number I to the current contents of the RD variable.

TXTOPEN(ROWS:INTEGER)

opens a hardware text mode video screen as channel 120 with the number of lines given in ROWS and an editor channel with a 2K buffer as channel 121.

GRAOPEN

open a high-resolution 2 colour graphics screen as channel 131. This uses the full width of the screen and 20 out of the 28 maximum lines occupying 15288 bytes of memory. This could be modified to use more colours on the Enterprise 128 but this would use the same memory, but give a lower resolution.

DISPLAY (C,NoRows,TopRow:INTEGER)

displays NoRows lines of the video page attached to channel C with TopRow giving the physical row from where the page is displayed.

PRINT (C:INTEGER; A:CHAR) ;

writes the character A to channel C.

PLOT (X,Y:INTEGER);

plots the point X,Y if the pen is down and clears it if the PEN is up.

LINE (DUMMY:BOOLEAN; X,Y:INTEGER) ;

plots a line from (XCOR,YCOR) to (XCOR + X, YCOR+Y) according to the status of PENON. XCOR and YCOR are **not** updated.

PENDOWN (C:INTEGER)

sets the turtle state so that it will leave a trail. It could be extended to leave the trail in the ink colour associated with the parameter C. This procedure assigns TRUE to PENON.

PENUP

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

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 O 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)

an alternative to TURN - RIGHT changes the turtle's heading in the clockwise direction by A degrees.

LEFT (A:REAL)

this is identical to TURN and is provided simply for convenience and compatibility with RIGHT.

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 4.0.

this procedure simply sets the initial state of the turtle and to set up the graphics screen; it is placed in near the middle of the screen, facing EAST (heading of 0), leaving a 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 HiSoft Pascal 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 have HiSoft Pascal loaded before entering the programs.

Examples Programs

In all the example programs given below we assume that you have already loaded HiSoft Pascal and used G,, [ENTER] to load the Turtle Graphics package. Then use N10,10 [ENTER] to renumber the program to make editing easier. Next type D2150,2260 to delete the main part of the demonstration program. Now we are ready to type the example programs:

1. CIRCLES

```
  75  I:INTEGER;
2150  SETXY(600,350);
2160  FOR I:=1 TO 9 DO
2170  BEGIN
2180    ARCR(2.0,360);
2190    RIGHT(40)
2200  END;
2210  READLN;
```

2. SPIRALS

```
D2180,2210
2150  SETXY(600,350);                    (or (9,90,9) or (9,121,9) ... )
2160  SPIRALS(9,95,9);
2170  READLN;
```

3. FLOWER

```
D1830,2090

1830  PROCEDURE PETAL ( S:REAL );
1840  BEGIN
1850    ARCR(S,60);
1860    LEFT(120);
1870    ARCR(S,60);
1880    LEFT(120)
1890  END;
1900  PROCEDURE FLOWER ( S:REAL );
1910  VAR I:INTEGER;
1920  BEGIN
1930    FOR I:=1 TO 6 DO
1940  BEGIN
1950    PETAL(S);
1960    RIGHT(60)
1970  END
1980 END;
```

```
2150  SETXY(600,10);
2160  LEFT(90); FWD(75);
2170  RIGHT(60); PETAL(1.5);
2180  LEFT(60); PETAL (1.5);
2190  SETHD(90); FWD(300);
2200  FLOWER(3.0);
2210  READLN;
```

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

APPENDIX 1 ERRORS

A.1.1 ERROR NUMBERS GENERATED BY THE COMPILER

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

41. Function result must be type identifier.
42. ',' or ']' expected in set.
43. '..' or ',' or ']' expected in set.
44. Type of parameter must be a type identifier.
45. Null set cannot be the first factor in a non-assignment statement.
46. Scalar (including real) expected.
47. Scalar (not including real) expected.
48. Sets incompatible.
49. '<' and '>' cannot be used to compare sets.
50. 'FORWARD', 'LABEL', 'CONST', 'VAR', 'TYPE' or 'BEGIN' expected.
51. Hexadecimal digit expected.
52. Cannot POKE sets.
53. Array too large (> 64K).
54. 'END' or ';' expected in RECORD definition.
55. Field identifier expected.
56. Variable expected after 'WITH'.
57. Variable in WITH must be of RECORD type.
58. Field identifier has not had 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.
71. The parameter of this procedure/function must be a string.

A.1.2 RUNTIME ERROR MESSAGES

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

1. Halt
2. Overflow
3. Out of RAM
4. / by zero — also generated by DIV.
5. Index too low
6. Index too high
7. Maths Call Error
8. Number too large
9. Number expected
10. Line too long
11. Exponent expected

Runtime errors result in the program execution being halted.

APPENDIX 2 RESERVED WORDS AND PREDEFINED IDENTIFIERS

A 2.1 RESERVED WORDS

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

A 2.2 SPECIAL SYMBOLS

The following symbols are used by HiSoft Pascal 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; NEW; MARK; RELEASE; OUT; TIN; TOUT; RANSEED
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; EXOS; MAKESTR;

APPENDIX 3 DATA REPRESENTATION AND STORAGE

A 3.1 Data Representation

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

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

A 3.1.1 Integers

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

Examples:

1	≡	£0001
256	≡	£0100
−256	≡	£FF00

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

A 3.1.2 CHARACTERS, BOOLEANS AND OTHER SCALARS

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

Characters: 8 bit, extended ASCII is used.

'E'	≡	£45
'['	≡	£5B

Booleans:

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

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

A 3.1.3 REALS

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

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

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

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

$-11001_2 *2^{-1}$

$-1.1001_2 *2^3$ when normalis

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

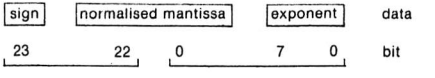
so now we need to do some binary long divisio

```
    0.0001100
101|0.100000000000000
     101
     110
     101
      1000
       101  at this point
            we see that the
            fraction recurs
```

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

$1.1001100_2 *2^{-4}$ answer.

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



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. e.g.

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

A 3.1.5 SETS

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

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

A 3.1.6 POINTERS

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

A 3.2 VARIABLE STORAGE AT RUNTIME

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

a. Global variables - declared in the main program block.
b. Local variables - declared in an inner block.
c. Parameters and return 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.

A 3.3 INTERACTION WITH EXOS

HiSoft Pascal uses the following channel numbers:

105	keyboard
120	video for the editor
121	editor channel itself
122-123	internal for tape,printer etc.

We recommend using channel numbers from 131 upwards for any extra channels you wish to open in your open programs.

APPENDIX 4 SOME EXAMPLE HISOFT PASCAL PROGRAMS

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

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

```
(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
  SizeStr=10;
TYPE
  Entry = RECORD
            Name : ARRAY [1..10] OF CHAR;
            Number : ARRAY [1..10] OF CHAR
          END;
VAR
  Directory : ARRAY [1..SizeStr] OF Entry;
  I : INTEGER;
BEGIN
(Set up the directory..)
```

```
  FOR I:= 1 TO SizeStr 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.
```

```
 10  (Program to shown the use of recursion)
 20
 30  PROGRAM FACTOR;
 40
 50  (This program calculates the factorial of a number input
 60  from the keyboard 1) using recursion and 2) using an
 70  iterative method.)
 80  TYPE
 90      POSINT = 0..MAXINT;
100
110  VAR
120      METHOD : CHAR;
130      NUMBER : POSINT;
140
150  (Recursive algorithm.)
160
170  FUNCTION RFAC(N : POSINT) : INTEGER;
180
190  VAR F : POSINT;
200
210  BEGIN
220      IF N>1 THEN F:= N * RFAC(N—1)      (RFAC invoked N times)
230             ELSE F:= 1;
240      RFAC := F
250  END;
260
270  (Iterative solution)
280
290  FUNCTION IFAC(N : POSINT) : INTEGER;
300
310      VAR I,F: POSINT;
320      BEGIN
330        F := 1;
340        FOR I := 2 TO N DO F := F*I;                    (Simple Loop)
```

```
350          IFAC:=F
360        END;
370
380  BEGIN
390    REPEAT
400      WRITE('Give method (I or R) and number     ');
410      READLN;
420      READ(METHOD,NUMBER);
430      IF METHOD = 'R'
440          THEN WRITELN(NUMBER,'! = ',RFAC(NUMBER))
450          ELSE WRITELN(NUMBER,'! = ',IFAC (NUMBER))
460    UNTIL NUMBER=O
470  END.
```

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

```
10  (Program to show how to 'get your hands dirty'!
20  i.e. how to modify Pascal Variables using machine code.
30  Demonstrates PEEK, POKE, ADDR and INLINE.)
40
50  PROGRAM divmult2;
60
70  VAR r:REAL;
80
90  FUNCTION divby2(x:REAL):REAL; (Function to divide 2 ..
100                                .. quickly)
110 VAR i:INTEGER;
120 BEGIN
130   i:=ADDR(x)+1;              (Point to the exponent of x)
140   POKE(i,PRED(PEEK(i,CHAR) ) ); (Decrement the exponent of x.
150                                  see Appendix 3.1.3.)
160   divby2:=x
170 END;
180
190 FUNCTION multby2(x:REAL):REAL; (Function to multiply by 2..
200                                 .. quickly)
210 BEGIN
220   INLINE(£DD,£34,3)                    (INC (IX+3) — the exponent of x
230                                          — see Appendix 3.2.)
240   multby2:=x
250 END;
260
270 BEGIN
280   REPEAT
290     WRITE('Enter the number r ');
300     READ(r);                          (No need for READLN - see
310                                        Section 2.4.1.4)
320
330     WRITELN('r divided by two is',divby2(r):7:2);
340     WRITELN('r multiplied by two is ',multby2(r):7:2)
350   UNTIL r=0
360 END.
```

BIBLIOGRAPHY

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

PASCAL MANUAL

Please note this manual was issued prior to the availability of PASCAL in cartridge form. If you have purchased the cartridge version, please ignore any instructions refering to tape cassettes.

Enterprise Computers Limited