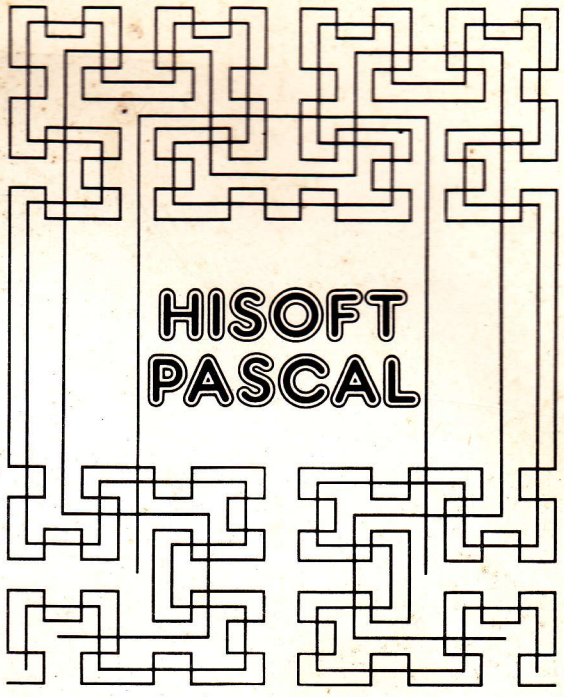


MEMOTECH
HISOFT PASCAL
USER GUIDE

CONTENTS

SECTION 0 PRELIMINARIES.

0.1 Introduction.

Hisoft Pascal for the MTX512 microcomputer 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.
A RECORD type may not have a VARIANT part.
PROCEDUREs and FUNCTIONs are not valid as parameters.

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

The package consists of the compiler itself, a set of runtime routines (which reside in RAM so that they may be saved to tape) and a program editor for creating and editing your programs.

The physical package is a circuit board with Hisoft Pascal mounted in ROM/EPROM; the board has two connectors so that it may be attached to the main MTX512 circuit board at either the left or right hand side of the board. Please ensure that the power to the MTX512 is OFF before connecting the Hisoft Pascal ROM board.

To enter the package, once you have connected the board to your MTX 512, simply type ROM 2 <RET> from within BASIC. If there are any other expansion ROMs in the system you will be prompted with a menu, otherwise you will go straight into Hisoft Pascal. Please read the rest of this section before attempting to use the Pascal package.

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 shows how to use the program editor which is an integral part of Hisoft Pascal and is a combination of a screen editor and the line editor used by MTX BASIC.

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.

Now let's get on with writing and running a few programs:

0.3 Example of Editing, Compiling and Running.

To enter Hisoft Pascal on the MTX512 simply type ROM 2 <RET> from within BASIC - this will either generate a menu, from which you should choose the number associated with Hisoft Pascal, or directly enter the Hisoft Pascal package.

Once entered, Pascal will present a startup message and then ask for Table? - you can answer this question with a positive decimal number if you wish to set the compiler's symbol table size, otherwise simply hit <RET> and a default size of 2K bytes will be assumed. If you try to specify too large a size then you will be prompted with Table? again until you enter a valid size. Normally, just pressing <RET> will suffice.

Now the screen will clear, the bottom line will contain the letters F: and R: and there will be a flashing cursor at the top left of the screen. For editing purposes the actual screen is split into 3 virtual screens, as in BASIC, a listing screen (the top 19 lines), an edit screen (4 lines) and a message screen (the bottom line). Normally the message screen displays the current Find string (after the F:) and the current Replace string (after the R:) - refer to Section 4 for more details. Occasionally the message screen will be used to prompt for various responses.

Let's get started and type in a program:

Firstly, hit the <INS> key to enter Insert mode; the message Insert will be displayed on the message screen and the flashing cursor will be moved to the Edit screen - you can now type in a Pascal program. Try this:

```
PROGRAM HELLO; <RET>
BEGIN <RET>
 WRITELN('HELLO'); <RET>
END. <RET>
```

Now press <RET> by itself - this will take you out of Insert mode and back into Command mode, the whole program is now displayed on the Listing screen. To compile the program simply hit C (for Compile). A compiler listing will be generated (this consists of the memory address at which object code is being placed, followed by the text of the line) followed by the message Run?. Answer Y (or y) to this question and your program will be executed producing the word HELLO on the screen and again you will be asked if you want to Run? your program. You can continue to execute your program in this way until you press a character other than Y (or y) in answer to the Run? question. You will then be returned to the editor.
Now let's change the program:

Press the cursor down key on the MTX keyboard so that the flashing cursor is positioned at the start of the line with BEGIN in it, now press <INS> to enter Insert mode and type:

```
VAR I : INTEGER; <RET>
```

Hit <RET> again to escape from Insert mode, move the cursor down one line and press <INS> again and then enter:

```
FOR I := 1 TO 20 DO <RET>
```

and then hit <RET> again to return to Command mode. Now compile the new program (use C)

and run it (answer Y to Run?) as many times as you like and then return to the editor (answer N to Run?).

Now another program:

Firstly, delete the whole program already there; do this by pressing M when the cursor is at the top of the screen, to set a marker on this line. Now move the cursor to the end of the program by pressing W . Now press O ; you will be prompted with Block? on the message screen, hit Y and the block of text between the marker and the cursor will be deleted. Now press <INS> and type the following:

```
PROGRAM DEMO; <RET>
PROCEDURE SCREEN (SCR,COL : INTEGER); <RET>
BEGIN <RET>
 VS(SCR); <RET>
 PAPER(COL); <RET>
 PAGE; <RET>
 WRITE(CHR(3),CHR(2),CHR(3)); <RET>
END; <RET>
BEGIN <RET>
 CRVS(2,1,1,0,10,8,32); <RET>
 CRVS(3,1,11,8,10,8,32); <RET>
 CRVS(6,1,21,16,10,8,32); <RET>
 SCREEN(2,11); WRITE('HISOFT'); <RET>
 SCREEN(3,3); WRITE('MTX512'); <RET>
 SCREEN(6,9); WRITE('PASCAL'); <RET>
 READLN; <RET>
END. <RET>
```

Hit <RET> by itself to take you out of Insert mode and then hit C to compile the program. Run the program by answering Y to the Run? message.
Finally, to show you how to edit a line, let's modify this program to go on forever (almost!). Get back to Command mode by hitting the <RET> key and then answer N to Run? and then move the cursor down the List screen until it is positioned on the line that begins SCREEN(2.11) .... Now press <INS> and type in:

```
REPEAT <CR>
```

hit <CR> by itself to exit from Insert mode and now hit E to edit the line pointed to by the cursor, move the cursor right (using the cursor right key on the keyboard) until the cursor is positioned over the first 1 in the number 11, press <DEL> twice to delete the 11. Now press <INS> and type:

```
RANDOM(0) MOD 14 + 2 <RET>
```

This will take you out of the Edit mode back into Command mode and replace the old line with the new, edited line that you have just created (note that only 40 characters of the line are displayed - you can always see the whole line by pressing E to edit the line). Now move the cursor down one line and press E to edit the line beginning SCREEN(3,3)..... move the cursor right until it is over the second 3 in this line (the one before the right parenthesis) and press <DEL> to delete it, press <INS> and type:

```
RANDOM(0) MOD 14 + 2 <RET>
```

Now edit the line starting SCREEN(6,9).... by moving the cursor down one line and pressing E, on the Edit screen move the cursor right until it is positioned over the number 9, press <DEL> then <INS> and type:

RANDOM(0) MOD 14 + 2 <RET>

Finally, within Command mode, move the cursor down one line (so it is on the READLN; line), press <INS> and type:

UNTIL FALSE; <RET>

followed by <RET> on its own to get out of Insert mode.

Right, now compile the program (C) and run it (Y to Run?). It will go on indefinitely to break out hit any key to pause the program and then hit <CTRL>C followed by any other key to return to the editor.

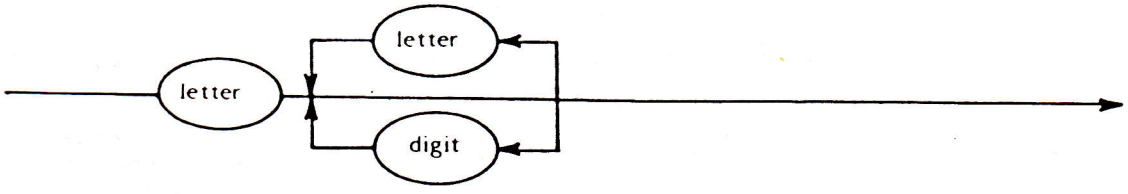
To save this program to tape, go to the beginning of the program (press Q), set a marker there (press M), go to the end of the program (press W), press P to Put to tape and then, in response to the message Name?, type in the name that you want the program to have on the tape, start your recorder in Record mode and then press <RET> to start the dump.

We hope that the above has given you a good idea of how to write, edit, compile, run and save Hisoft Pascal programs on the MTX512. Please refer carefully to the other sections of this manual for more details of using the package.

SECTION 1 SYNTAX AND SEMANTICS.

This section details the syntax and the semantics of Hisoft Pascal MTX512 - unless otherwise stated the implementation is as specified in the Pascal User Manual and Report Second Edition (Jensen/Wirth).

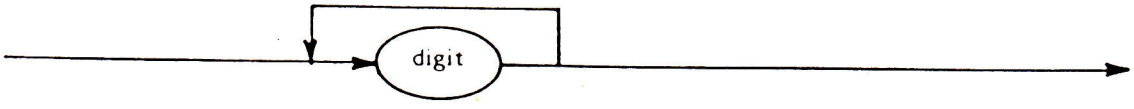
1.1 IDENTIFIER.



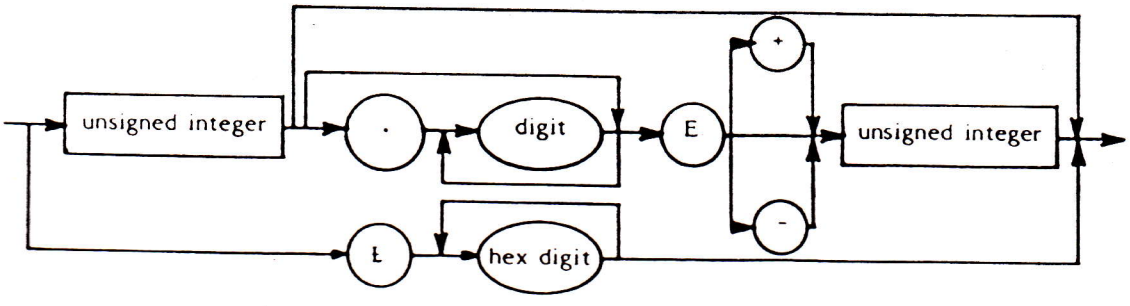
Only the first 10 characters of an identifier are treated as significant.

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

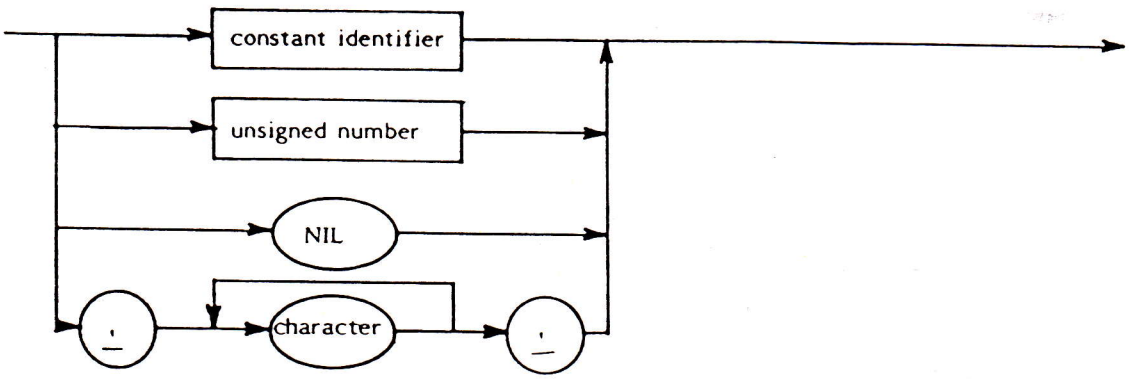
The largest real available is 3.4E38 while the smallest is 5.9E-39.

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.

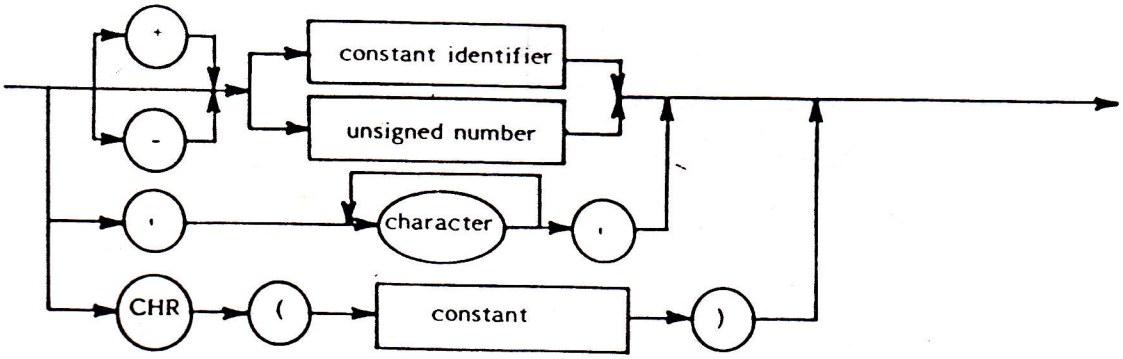
1.4 UNSIGNED CONSTANT.



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.

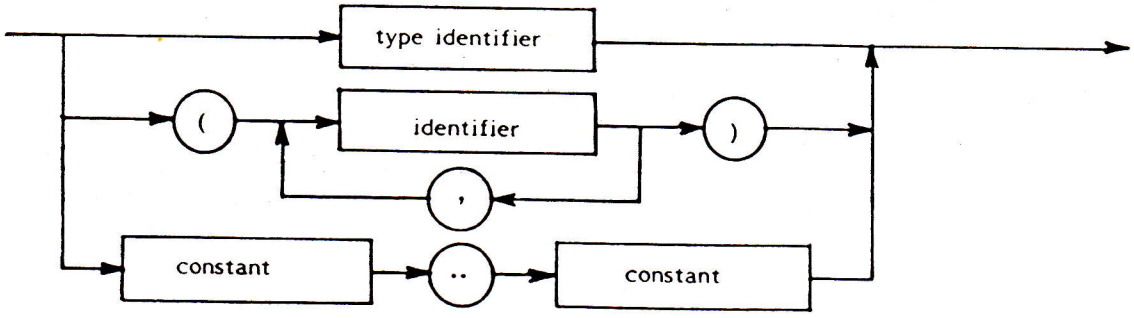
1.5 CONSTANT.



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.

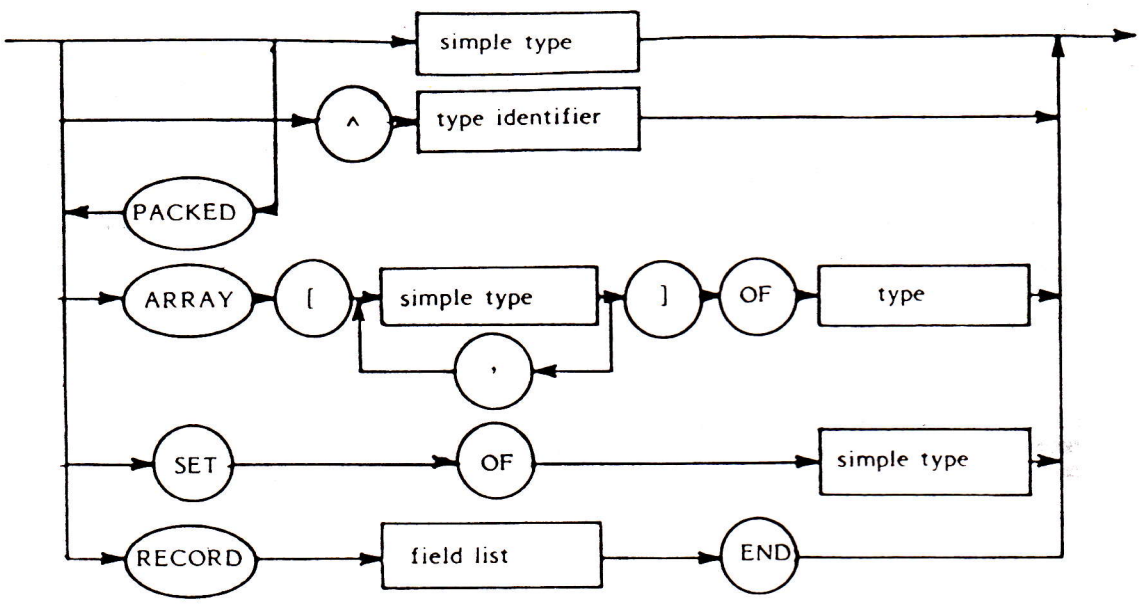
E.g. CONST bs=CHR(10);
cr=CHR(13);

1.6 SIMPLE TYPE.



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

1.7 TYPE.



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

1.7.1 ARRAYs and SETs.

The base type of a set may have up to 256 elements. This enables SETs of CHAR to be declared together with SETs of any user enumerated type. Note, however, that only 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 7.

There are some restrictions on the use of pointers within Hisoft Pascal MTX512. These are as follows:

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

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

    link = ^item;
```

Pointers to pointers are not allowed.

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

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

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

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

1.7.4 RECORDs.

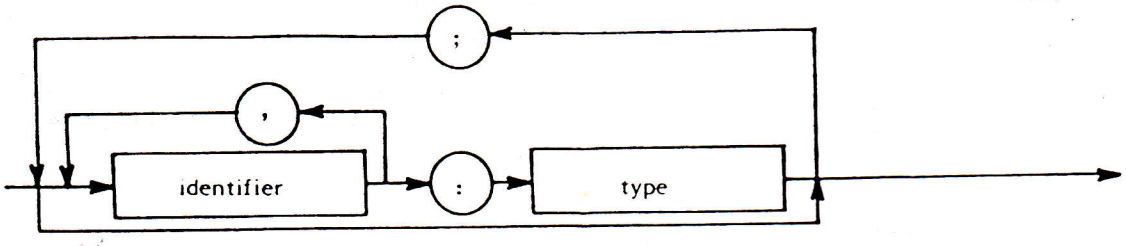
The implementation of RECORDs, structured variables composed of a fixed number of constituents called fields, within Hisoft Pascal MTX512 is as Standard Pascal except that the variant part of the field list is not supported.

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. You should note that WITH statements cannot be called recursively and that WITH does not open a new scope.

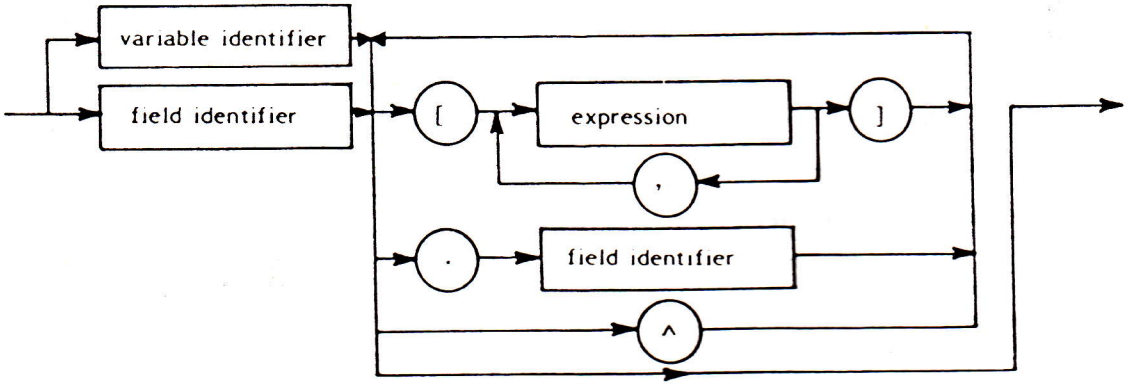
See Appendix 7 for an example of the use of WITH and RECORDs in general.

1.8 FIELD LIST.



Used in conjunction with RECORDs see Section 1.7.4 above and Appendix 7 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.

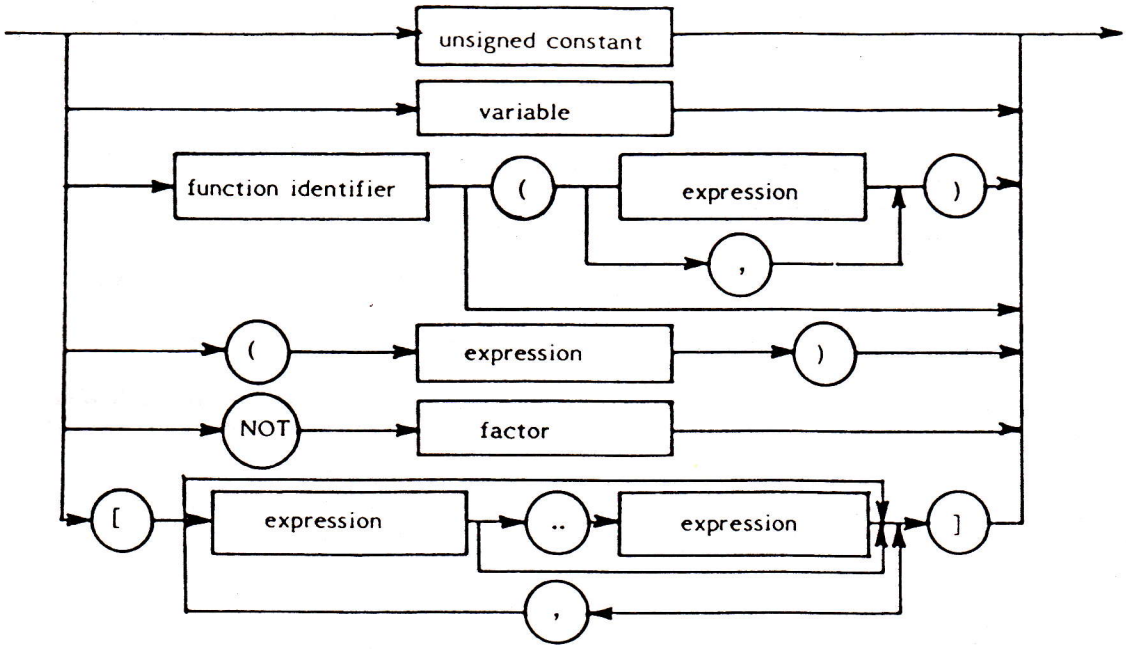
Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type pointer, which contains the address of the dynamic variable.

See Section 1.7.2 and Section 2 for more details of the use of dynamic variables and Appendix 7 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.

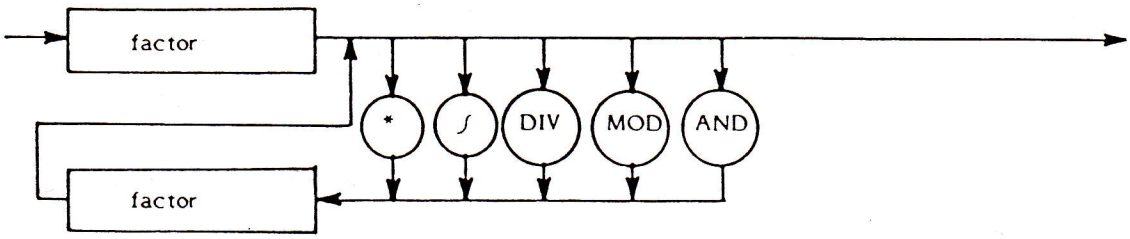
e.g. if variable a is declared as ARRAY[1..10] OF ARRAY[1..10] OF INTEGER then either a[1][1] or a[1,1] may be used to access element (1,1) of the array.

FACTOR.



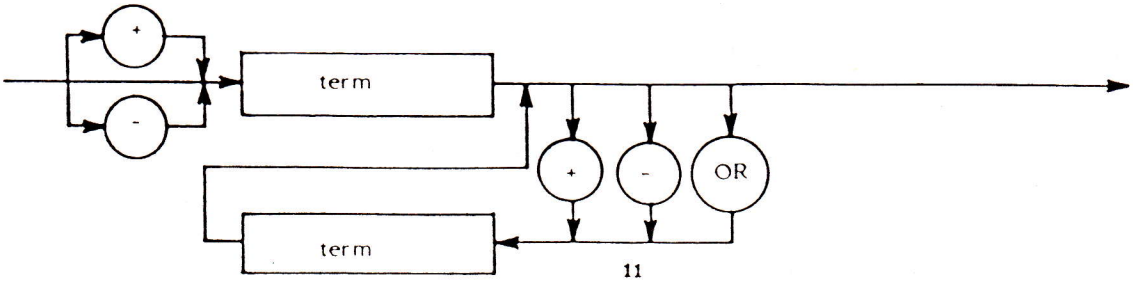
See EXPRESSION in Section 1.13 and FUNCTIONs in Section 3 for more details.

1.11 TERM.



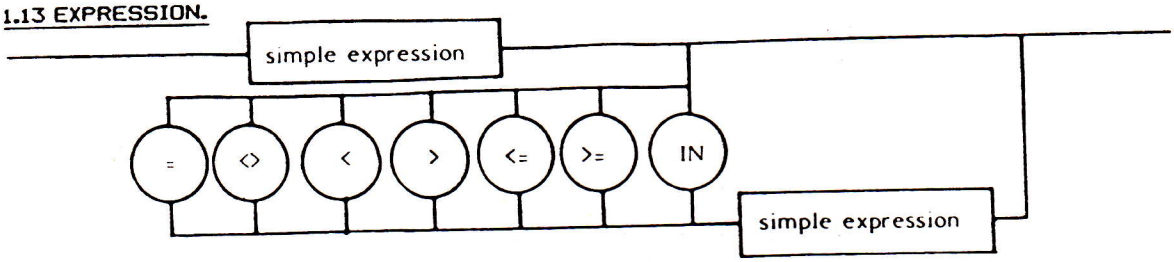
The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

1.12 SIMPLE EXPRESSION.



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

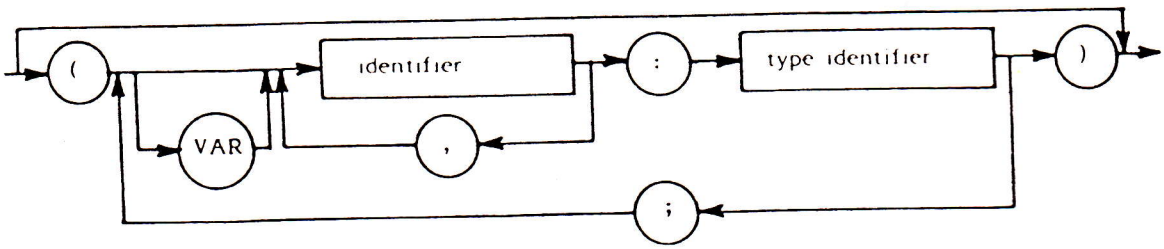
1.13 EXPRESSION.



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 14.

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:

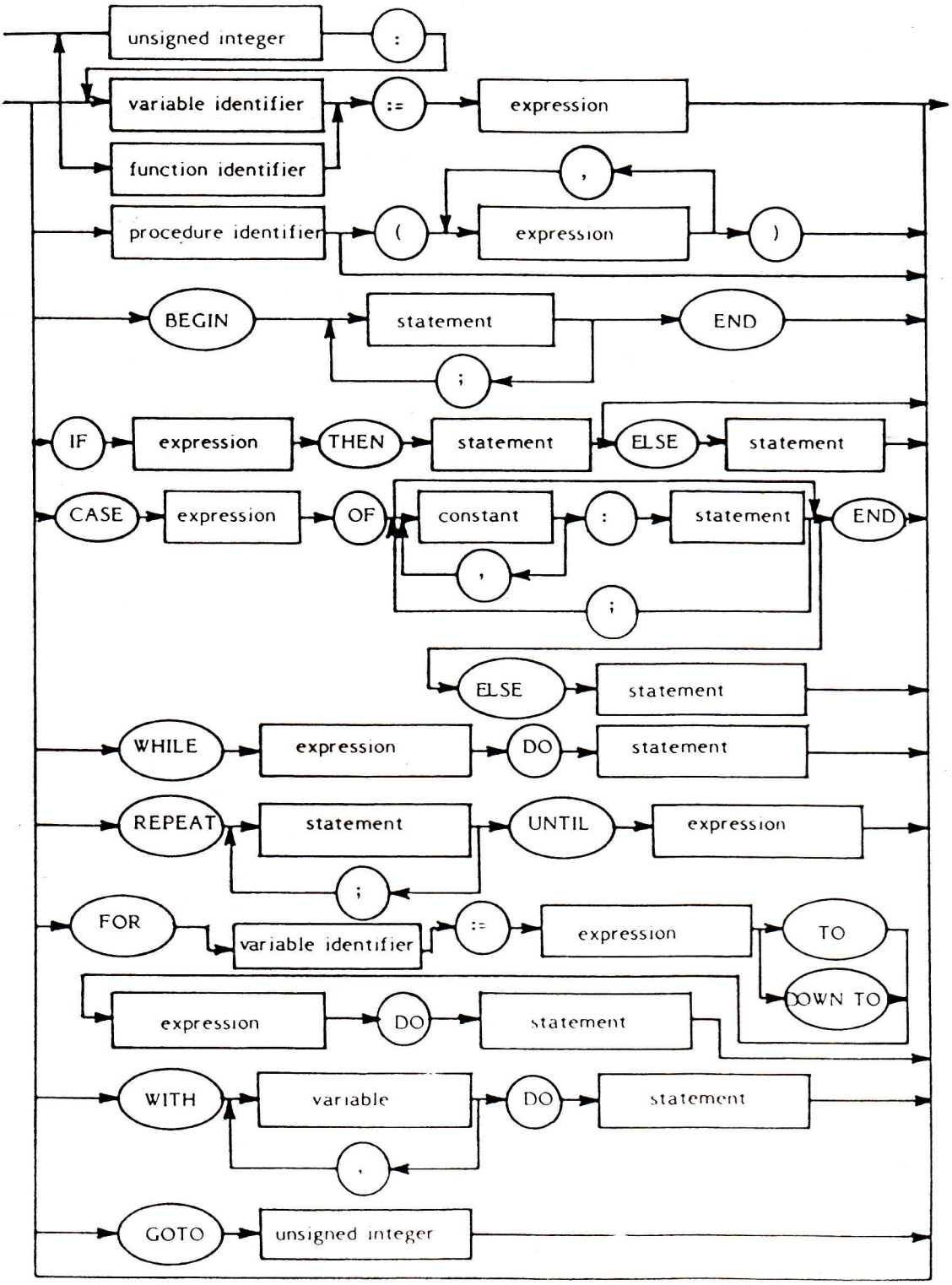
The control variable of a FOR statement may only be an unstructured variable, not a parameter. This is half way between the Jensen/Wirth and draft ISO standard definitions.

GOTO statements:

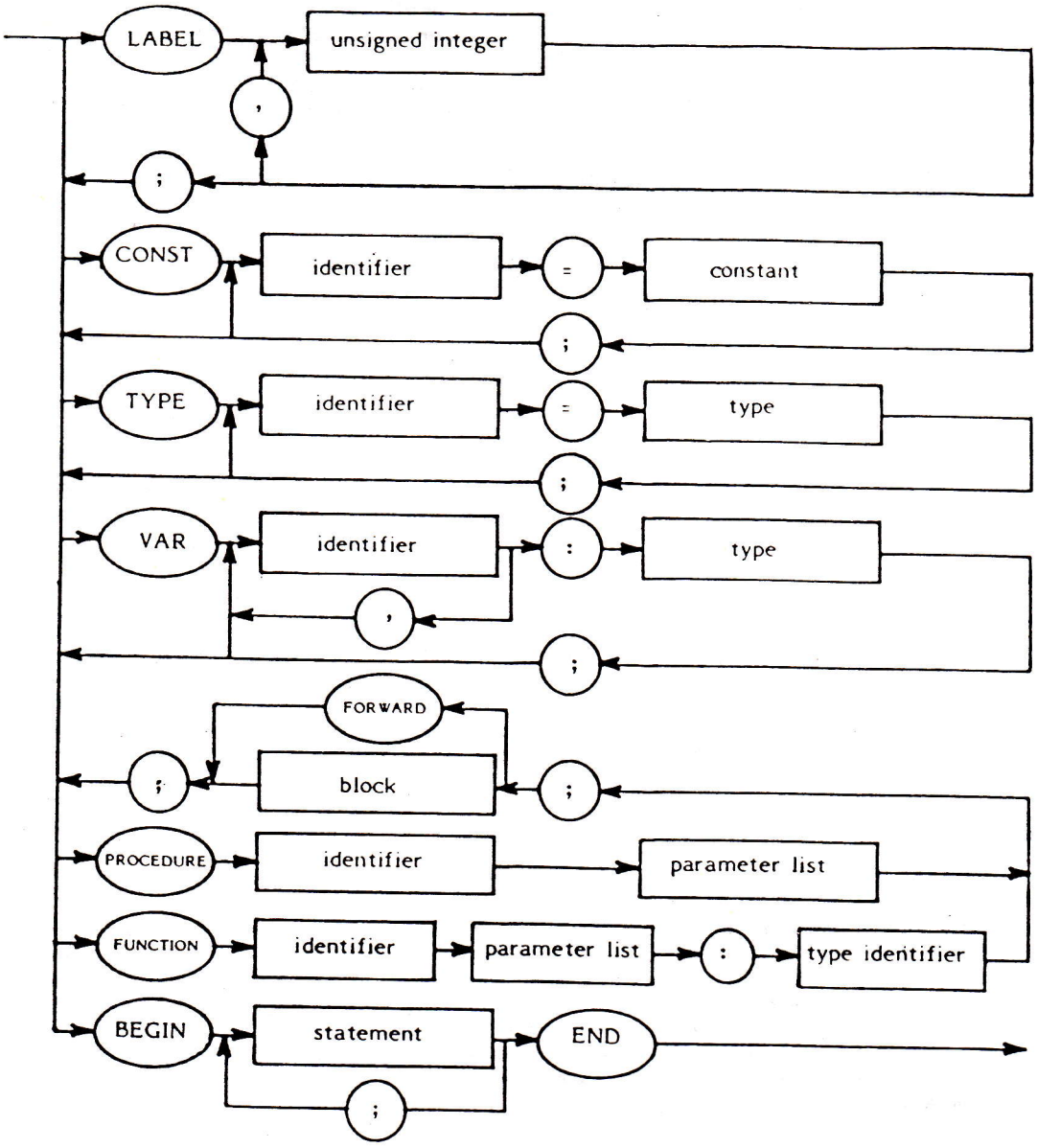
It is only possible to GOTO a label which is present in the same block as the GOTO statement and at the same level. GOTO should not be used to transfer execution out of a FOR..DO loop nor out of a Procedure or Function.

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
DOWN TO
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
;

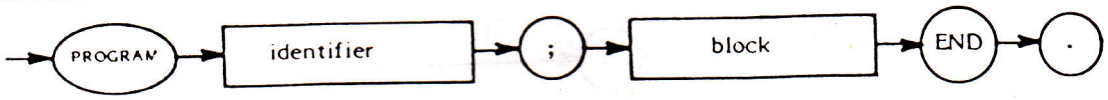
Forward References.

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 there are no formal parameters of the program i.e. you should NOT write PROGRAM A (INPUT,OUTPUT); but simply PROGRAM A; .

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 - 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 PROCEDURES AND FUNCTIONS.

2.3.1 Input and Output Procedures.

2.3.1.1 WRITE

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

Note:

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

Generally though:

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

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

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

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

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

We have 5 cases to examine:

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

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

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

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

WRITE(1025:m:H);

m=1 outputs: 1
m=2 outputs: 01
m=3 outputs: 0401
m=4 outputs: 0401
m=5 outputs: _0401

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

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

If n is not present then the number is output in scientific notation, with a mantissa and an exponent. If the number is negative 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 gives a newline. This is equivalent to WRITE(CHR(13)). Note that a linefeed is included.

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

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

2.3.1.3 PAGE

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

2.3.1.4 READ

The procedure READ is used to access data from the keyboard. It does this through a buffer held within the 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.

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

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

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

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

1) V is of type character.

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

the new line.

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

2] V is of type string.

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

The note concerning in 1] above also applies here.

3] V is of type integer.

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

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

If the first character read, after spaces and end-of-line characters have been skipped, is not a digit or a sign ('+' or '-') then 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 simply reads in a new buffer from the keyboard; while typing in the buffer you may use the various control functions detailed in Section 0.0. Thus EOLN becomes FALSE after the execution of READLN unless the next line is blank.

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

2.3.2 Input Functions.

2.3.2.1 EOLN

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

2.3.2.2 INCH

The function INCH causes the keyboard of the computer to be scanned and, if a key has been pressed, returns the character represented by the key pressed. If no key has been pressed then CHR(0) is returned. The function therefore returns a result of type character.

2.3.3 Transfer Functions.

2.3.3.1 TRUNC(X)

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

TRUNC(-1.5) returns -1 TRUNC(1.9) returns 1

2.3.3.2 ROUND(X)

X must be of type real or integer and the function returns 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. 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 (in hex) at £C000.

2.3.5.8 TOUT (NAME,START,SIZE)

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

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

The use of actual memory addresses gives the user far more flexiblity 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, variables etc. that have been saved using TOUT. NAME is of type ARRAY[1..8] of CHAR and START is of type INTEGER. The tape is searched for a file called NAME which is then loaded at memory address START. The number of bytes to load is taken from the tape (saved on the tape by TOUT).

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

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

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

See 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.

2.3.5.11 CRVS(n,t,x,y,w,h,s)

CRVS is used to set up the definition of a new virtual screen - the new screen may then be selected using the procedure VS (see below). The parameters (all INTEGER) of CRVS are exactly the same as are used within MTX BASIC i.e.

n is the VS identification number (0-7)
t is the screen type (0 for text, 1 for graphics)
x is the x co-ordinate of the top left hand corner of the screen
y is the y co-ordinate of the top left hand corner of the screen
w is the width (in characters) of the screen
h is the height (in characters) of the screen
a should be 40 for a text screen or 32 for a graphics screen

e.g.
CRVS(2,1,11,8,10,8,32);

sets up a graphics screen (80x64 pixels) in the middle of the screen.

2.3.5.12 VS(n)

Select the virtual screen n for subsequent output. Remember that screens 0, 1, 4, 5 and 7 are set up and used by MTX BASIC. n is of type INTEGER.

2.3.5.13 PAPER(n)

This sets the paper colour for the current screen to the colour corresponding to the number (INTEGER) n - see page 184 of the MTX Operator's Manual for the list of colours available. Equivalent to BASIC's PAPER command.

2.3.5.14 INK(n)

INK sets the ink colour for the current screen to the colour corresponding to the number n which is of type INTEGER. Equivalent to BASIC's INK command.

2.3.5.15 PLOT(x,y)

This procedure will only produce valid results when used after a graphics screen has been selected (using VS). It takes two INTEGER parameters x and y and simply plots the point (x,y) on the current graphics screen, in the current ink colour and relative to the origin of the current screen. If an attempt is made to PLOT to a text screen then an MTX BASIC error will occur - type ROM 2 <RET> to return to Pascal.

2.3.5.16 LINE(x1,y1,x2,y2)

LINE takes 4 INTEGER parameters and draws a straight line between points (x1,y1) and (x2,y2) on the current graphics screen and in the current ink colour. If an attempt is made to draw a line on a text screen then an MTX BASIC error will occur - type ROM 2 <RET> to recover Pascal.

2.3.6 Further Predefined Functions.

2.3.6.1 RANDOM(X)

RANDOM generates a pseudo-random number in the range 0 - MAXINT i.e. a positive INTEGER. RANDOM takes one parameter, if this parameter is zero then RANDOM(0) returns the next random number in the sequence otherwise the parameter is taken as the seed for a new random number sequence.

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 MTX512.

2.3.6.8 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.

2.3.6.9 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.

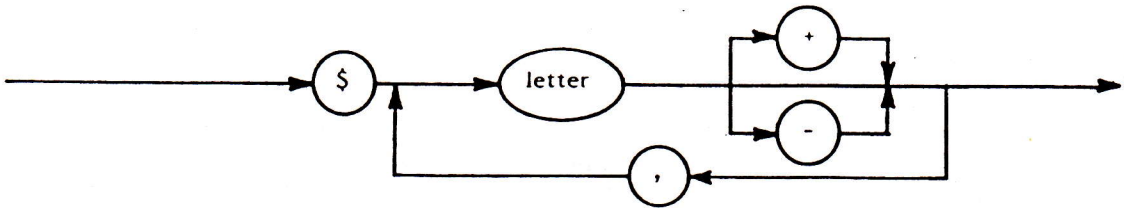
3.1 Comments.

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

3.2 Compiler Options.

Compiler options can only occur within a Pascal program and then only within a comment i.e. between { or (* and } or *).

The syntax for specifying compiler options within a comment is:



The following options are available:

Option L:

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

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

DEFAULT: L+

Option O:

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

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

If O- then the above checks are not made.

DEFAULT: O+

Option C:

During execution of the object code, and assuming you have not used option $C- (see Section 3), you may pause execution by pressing the BRK key; subsequently press <CTRL> X to terminate the run (then press any key) or press another key to continue the execution of the program.

This check is made at the beginning of all loops, procedures and functions. Thus the user may use this facility to detect which loop etc. is not terminating correctly during the debugging process. It should certainly be disabled if you wish the object program to run quickly. It should also be turned off if you are using the INCH function.

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 the user wish to compare such numbers, the use of I+ ensures that the results of the comparison will be correct. The equivalent situation may arise with real arithmetic in which case an overflow error will be issued if the arguments differ by more than approximately 3.4E38 ; this cannot be avoided.

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

DEFAULT: I-

Option P:

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

DEFAULT: The video screen is used.

Option F:

This option letter must be followed by a space and then an eight character filename. If the filename has less than eight characters it should be padded with spaces.

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

The program should be saved using the built-in editor's 'P' command.

The list option L- is forced while including in this manner - otherwise the compiler will not compile fast enough.
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 and using the 'F' option - then only 256 bytes of the source are in RAM at any one time, leaving much more room for the object code.

This option may not be nested.

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 EDITOR

4.1 Introduction to the Editor.

The editor supplied with Hisoft Pascal for the MTX512 is a combination of the line editor used by MTX BASIC and a screen editor specific to Hisoft Pascal.

The interaction of these two editors is as follows: whenever you are inserting or editing a line of text then the MTX BASIC editor has total control, text is typed on the Edit screen (lines 20-23 inclusive) and all the control keys specified on page 7 of the MTX Operator's Manual may be used. Once you have terminated the insertion or edit (you do this by pressing <RET>) the line of text is transferred to the Listing screen (lines 1-19 inclusive) with the first 40 characters of the line displayed. When you are not inserting or editing text you are in the Command mode of the screen editor i.e. text is displayed on the Listing screen and you may enter various one character commands to do things like save text to tape, delete a block of tape, go to the end of the textfile etc. Often, after entering a command, you will be prompted on the Message screen (line 24) with a relevant message before the command is executed: enter Y or y if you wish the command to continue or any other character to abort execution of the command.

While in the Command mode of the screen editor, and not waiting for a command to be executed, the Message screen will contain the current Find and Replace strings - these are the strings of characters used by the F, R and <CTRL>R commands which allow you to search for particular words, inter alia, and optionally replace them with other words. The Find string is displayed after the letters F: while the Replace string is shown after R: .

Initially, both Find and Replace strings are empty, the textfile is empty and the cursor is placed at the top left hand corner of the screen - you are in the Command mode of the screen editor waiting for a command to be entered. There are 24 commands and these are described below.

4.2 Screen Editor Commands.

All screen editor commands are either one letter commands (such as I for Insert) or <CTRL> key commands such as <CTRL>R - <CTRL> key commands require the <CTRL> key to be held down while you press the relevant command letter e.g. hold <CTRL> and R down to reach the <CTRL>R command.

Commands generally take effect immediately (there is no need to press <RET>) although many prompt you on the Message screen first to make sure that you want to continue with the command - usually answer Y or y to continue the command or any other letter to abort it. The commands you may enter are given below.

If at any time there is no room to insert any more text in the textfile then the message Space! will appear on the message screen - hit any key to return to Command mode. You will now have to delete some of your program to make space for any new text.

4.2.1 Cursor Commands.

Command ↑

Cursor up. This simply moves the cursor up one line in the textfile. The cursor cannot be moved before the start of the textfile. When the cursor is at the top of the page and not at the beginning of the file then the cursor up command will bring in the previous line

and the current page will be scrolled down one line.

Command ↓

Cursor down. The cursor is moved down one line in the textfile. It cannot be moved past the end of the file. If the cursor is a the bottom of the page and not at the end of the file then the cursor down command will bring in the next line of text and the current page will be scrolled up one line.

Command Q

Go to the beginning of the textfile.

Command W

Go to the end of the textfile i.e. position the cursor at the start of the last line in the textfile.

Command A

Display the previous page (19 lines) of text to the current page. If the first page of the file is the current page then no action is taken. The cursor position within the page is maintained.

Command S

Display the next page of text. If the last page of text is already displayed then no action is taken. The curosr position within the page is maintained.

Command Z

Take the cursor to the top of the current page.

Command X

Position the cursor at the start of the last line of text on the current page.

Note that the last 6 commands (Q, W, A, S, Z and X) are positioned on a block of keys at the left of the keyboard - this should facilitate their use.

4.2.2 Editing Commands.

Command <INS>

Insert a line before the current cursor position. This takes you into Insert mode, allowing you to continually insert lines of text. While in this mode the word Insert is displayed on the Message screen. To type the line that you wish to insert just type normally on the keyboard, optionally using the various control keys given on page 7 of the MTX Operator's Manual. To finish a line type <RET> and the line will be transferred to the Listing screen - you will remain in Insert mode and can keep typing lines,

terminated by <RET>.

To get out of Insert mode and back to Command mode simply type <RET> as the first character of a line - if you want to insert blank lines into the text then type a space followed by <RET>. In fact any key whose ASCII value is less than 32, when typed as the first character on a line, will take you back to Command mode.

Note that you can type lines of up to 160 characters on the Edit screen, all these characters will be inserted into the textfile but only the first 40 characters of the line will be displayed on the Listing screen.

Command E

Edit the line at the cursor position. The line on which the cursor is positioned is transferred to the Edit screen and you may now use all the commands given on page 7 of the MTX Operator's Manual to edit this line of text. To finish the edit and copy the line back into the textfile simply press <RET>.

Important Note: whenever you are using the Edit screen you should never press <ESC> because this will normally cause an MTX BASIC error which will give BASIC control of the machine again. If you should accidentally press <ESC> while using the Edit screen and find yourself back in BASIC then enter ROM 2 <RET> - this should take you back into Pascal with your text intact.

Command D

Delete the current line. This command first prompts you with Line? on the Message screen. If you wish to delete the line at the cursor then type Y or y , otherwise type any aother character and the command will be aborted.

Command O

Obliterate (delete!) a block of text. You should first set a marker (using the M command) on the first line that you wish to delete and then move the cursor to the start of the line up to which you want to delete. Now press O - the message Block? will appear on the Message screen, hit Y or y to delete the marked block of text or any other character to abort the command and return to Command mode. Text will be deleted from and including the line in which the marker is set up to but not including the line on which the cursor is positioned.

Command V

Set the Values of the Find and Replace strings. This command allows you to define the Find and Replace strings that will be used by subsequent F, R and <CTRL>R commands. Firstly you are prompted on the Message screen for Find? - enter, on the Edit screen, the string of characters, up to 17 characters, for which you want to search and terminate the string with <RET>. Now you will be prompted to enter the Replace? string - again enter up to 17 characters on the Edit screen finishing with <RET>. You may define a null Replace string, simply press <RET> by itself. The strings that you have defined will now appear on the message screen (after F: and R:) in Command mode until you redefine them.

While entering the Find string you may use the ^ character (<SHIFT> ↑) as a wild character e.g. if you enter R^N as the Find string then, whenever you subsequently

search for that string, the second character will be matched against any character in the text so that words like RUN, RAN, R;N etc. will all be matched with the Find string. Note that end-of-line characters will not be matched.

Command F

Find a string. Starting from the current cursor position + 1, search the textfile for the first occurrence of the Find string and then update the Listing screen so that the cursor is positioned at the start of the found occurrence.

Command R

Replace a string. This command first searches, starting from the current cursor position, for the first occurrence of the Find string – if an occurrence is found then it replaces the string with the current Replace string and then searches for the next occurrence of the Find string.

If no occurrence of the Find string is found then no action is taken.

Command <CTRL>R

Global replace. Starting from the current cursor position search the rest of the textfile for all occurrences of the Find string, replacing them with the current Replace string. While the search and replace is in progress the Message screen will contain simply the Replace string as an indication that the command is in progress.

Use this command carefully.

4.2.3 Tape Commands.

Command P

Put to tape. This command saves the text starting from and including the line with a marker set in it up to but not including the line currently containing the cursor. Use M to set the marker. The P command first prompts you to enter the Name? of the file that you are going to save to tape. You should enter a name of up to 8 characters terminated by <RET> . Remember to start your tape recorder in RECORD mode before typing in the name since the dump to tape will start as soon as you have pressed <RET> .

Command G

Get a file from tape. You are first prompted to enter the Name? of the file that you wish loaded. Type up to 8 characters terminated with <RET>, or simply type <RET> by itself if you wish to load the first file on the tape. Now press PLAY on your tape recorder. The tape will be searched for the relevant file and, when found, the file will be loaded into memory starting from the line before the line in which the marker is currently set. Thus you can load up text at any position within the textfile by specifying a marker (use M) at the desired position. Remember that the marker is set by default to the beginning of the file.

You can abort a tape load by pressing <CTRL> X and holding it down while the tape load is in progress.

Data is stored on tape in blocks and each block has a checksum associated with it – if a

checksum error should be encountered on loading back the data then CS ERR will be displayed on the screen; hit any key to return to Command mode. You will now have to rewind the tape and try to load the file again from the beginning.

Command <CTRL>G
Verify a tape file. You are first prompted (as in the G command) to enter the file-name of up to 8 characters, enter <RET> by itself if you want to verify the first file on the tape. Now press PLAY on your tape recorder and the command will search for the requested file. When found, the file on the tape will be compared with textfile in memory, starting from the line with the marker in it. If the comparison is good then, at the end of the file, you will simply be returned to Command mode; if the comparison fails the message ERR will appear on the message screen, hit any key to return to Command mode. You may now want to dump the textfile again, preferably onto a new tape.
You may abort a verify command by pressing <CTRL>X and holding it down while the tape is being searched.

As in the G command a checksum error may occur.

4.2.4 Compiling and Running.

Command C

Compile. The text will be compiled starting from the line with the marker in it; remember that the marker is, by default, normally at the beginning of the file.

Once it has been invoked, and assuming the $L- option (see Section 3) has not been specified, then the compiler will generate a compiler listing on the screen consisting of the approximate memory address at which object code is being generated followed by the text of the line. This listing may be directed to your printer by use of the compiler option $P , see Section 3.

You may pause the listing at the end of a line by hitting the BRK key; then hit <CTRL> X to halt the compilation (then hit any key) or any other key to continue the compilation.

If an error is detected while compilation is in progress, then the offending line will be displayed and underneath it the word *ERR* will be displayed followed by an up-arrow (^) symbol, which points after the symbol that generated the error, and an error number (see Appendix 1 for a list of error numbers). The listing will now pause; hit E to return into Edit mode and edit the line in which the error was detected or P to edit the previous line (often useful when the error is *ERR* 2 - missing semi-colon).

If the program terminates incorrectly (i.e. without END.) then the message Text? will be shown on the Message screen and control will return to the screen editor.

If the compiler runs out of symbol table space while compiling then the word Space! will appear on the screen - hit any key to return to the editor. If you run out of symbol table space then you can cut down on the number of global variables used, decrease the length of identifiers or, most easily, save your program to tape, hit RESET, reload the Pascal package (ROM 2 <RET>) and specify a larger Table? size - the default is 2048 bytes.

If the compilation terminates correctly then the End: address of the object code will be displayed and you will be prompted to Run? the code - answer Y or y to run or any other character to return to the editor.

During execution of the object code, and assuming you have not used option $C- (see Section 3), you may pause execution by pressing the BRK key; subsequently press <CTRL> X to terminate the run (then press any key) or press another key to continue the execution of the program.

During the running of a program various runtime error messages may be displayed (e.g. OV at PC=xxxx), refer to Appendix 1 for an explanation of these messages. Runtime errors cause control to be returned to the editor or BASIC. It is also possible that you will run out of runtime stack, if this happens the message Space! at PC=xxxx will be displayed, hit any key to return to the editor (or BASIC).

Command <CTRL>O

Object code to tape. When you have finished debugging your Pascal program and would like to save the object code to tape, then you should use the <CTRL>O command. You should only do this when the program is fully working, at previous stages it makes more sense to save the textfile of the program.

<CTRL>O compiles the program and then returns to MTX BASIC. You can now save the object to tape from within BASIC by the following method:

First type A.10 <RET>
Now type <CLS> <RET>
and now SAVE "NAME"

where NAME is the filename that you wish the code to have on tape. The object code will now be saved as a BASIC program and may be loaded into an MTX512 computer subsequently by typing LOAD "" from within BASIC, whether or not Hisoft Pascal ROMs are present within the system. Then simply RUN the program from within BASIC. Any runtime errors that occur within the Pascal program will now cause a return to BASIC - you must then RUN the program again.
Hisoft has no objection to your selling programs developed using Hisoft Pascal for the MTX512 and indeed would encourage you to do so but we would ask you to acknowledge our copyright on the runtime routines that you will be selling along with your compiled code. Thus we would be grateful if you could include the following within the startup message of your program and also within the documentation for the product:

Produced using Hisoft Pascal
Pascal runtime routines Copyright Hisoft 1983,4

You should note that the <CTRL>O command destroys a part of the compiler and you must hit RESET and then ROM 2 <RET> to do any more work with the compiler. Thus <CTRL>O destroys your textfile and should only be used at the very end of the development process.

4.2.5 Other Commands.

Command B

Back to BASIC. This command prompts you to Exit? . Type Y or y to return to BASIC or any other character to return to the editor. Once you are back in MTX BASIC you may re-enter the Pascal, assuming you have not corrupted the runtimes (by entering a BASIC program for example!), by typing ROM 2 <RET>, normally this will recover the compiler and your textfile.

Command M

Set the Marker. Set the marker to the start of the line currently addressed by the cursor. This marker is used by various other commands to define one end of a block of text, the other end of the block is defined by the cursor position.

On entry to the compiler the marker is set to the beginning of the file and any commands that cause text to be inserted or deleted from the textfile (including R) will reset the marker to the beginning of the textfile.

Command L

Print text. The block of text defined by the current Marker position and the current cursor position is written out to the printer. This command first prompts with Print? on the message screen. Answer Y or y to print out the block of text, but make sure your printer is on-line first, or tpye any other character to abort the command and return to the screen editor Command mode.

APPENDIX 1 ERRORS.

A.1.1 Error numbers generated by the compiler.

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

51. Hexadecimal digit expected.
52. Cannot POKE sets.
53. Array too large (> 64K).
54. 'END' or ';' expected in RECORD definition.
55. Field identifier expected.
56. Variable expected after 'WITH'.
57. Variable in WITH must be of RECORD type.
58. Field identifier has not had asociated WITH statement.
59. Unsigned integer expected after 'LABEL'.
60. Unsigned integer expected after 'GOTO'.
61. This label is at the wrong level.
62. Undeclared label.
63. 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.	OV	Overflow
3.	Space!	No Stack or Heap space
4.	/0	Division by zero
5.	IL	Array index too low
6.	IH	Array index too high
7.	MC	Maths call error
8.	NL	Number too large
9.	NE	Number expected
10.	EE	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;
 PAPER; INK; CRVS; VS; PLOT; LINE;

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 MTX512.

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 representd by 0.

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

A 3.1.3 Reals.

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

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

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

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

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

$= -1.1001 *2^{-3}$ when normalised.

$0.1 \equiv 1.0*10^{-1}$ 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 division..

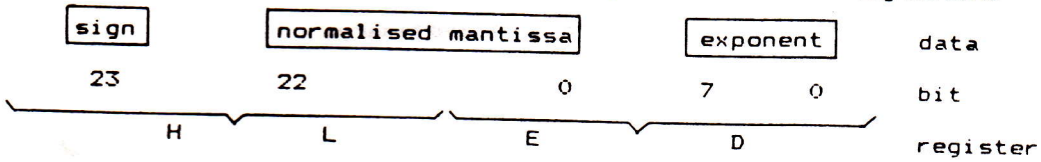
$$
\begin{array}{r}
0.0001100 \\
101 \overline{)\,0.1000000000000000} \\
\underline{101} \\
110 \\
\underline{101} \\
1000 \\
\underline{101} \\
\end{array}
$$

at this point we see that the fraction recurs

$= \frac{0.1_2}{101_2} = 0.\dot{0}0011\dot{0}0_2$

$1.1001\dot{1}0\dot{0} * 2^{-4}$ 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 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 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 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 <CTRL>C to exit}
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  and  2) using an iterative method.}

TYPE
  POSINT = 0..MAXINT;

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 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.
```

SOME RECOMMENDED READING

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

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

J. Tiberghien — THE PASCAL HANDBOOK.
SYBEX 1981.

W. Findlay
D.A. Watt — PASCAL. AN INTRODUCTION TO SYSTEMATIC PROGRAMMING.
Pitman Publishing. 1978, 1982.

J. Welsh
J. Elder — INTRODUCTION TO PASCAL.

MEMOTECH LTD STATION LANE WITNEY OXON OX8 6BX TEL· 0993-2977 TLX· 83372 MEMTEC G
MEMOTECH CORP 99 CABOT STREET NEEDHAM MA 02194 TEL· 617-449-6614 TLX· 7103212035 MEMOTECH NEDM
HISOFT 180 HIGH STREET NORTH DUNSTABLE BEDFORDSHIRE LU6 1AT TEL· 0582 696 421

MEMOTECH ROM BOARD LINKS

If several ROM boards are installed inside the MTX, one of them must provide a menu to allow the user to choose which board he wants to use.

In order to do this, there are six links on each ROM board, some of which can be changed manually to be either high or low. These links must be set up in the following manner.

1. The links on one of the boards must all be set to low. This board will provide a menu if several boards are present.

2. All boards fitted must have different link set-ups.

3. If the PASCAL board is fitted with other boards, the links on one of the other boards must all be set to low, ie the PASCAL board cannot provide the menu.

4. If only one board is installed, all links must be low and the MTX will automatically select this board.

INSTALLING THE PASCAL ROM BOARD

The PASCAL PCB, which fits inside the MTX Computer, should be installed as follows:

Remove the MTX side plates with a 2 mm Allen Key, and raise the top of the MTX -- it is hinged along its front edge. Insert the PCB into the MTX base so that the two edges of the PCB slide into the channels in the aluminium. The new PCB lies flat and in the same plane as the main computer PCB, and the plastic edge socket on the PASCAL PCB clips over the tin PCB tracks on the existing board. As you slide the new board in, gently move any wires to one side so they do not get pinched or bent. If the "existing board" is the main computer board, it is the Hi-Fi and Monitor wires that must be safeguarded -- if your MTX has been fitted with an RS232, you must remove the RS232 board, install the ROM board, then replace the RS232.

Once the PASCAL PCB has been firmly plugged into the existing board, the metal side plates can be screwed back on.