Supermax System V

Pascal User's Guide

Dansk Data Elektronik A/S
15. Sep 1987

Version 1.1

Copyright 1987
Dansk Data Elektronik A/S

Table of Contents

Page

1. Introduction 1.1

2. Language description 2.1
2.0.1. A Note on Implementation 2.1
2.1. Letters and Symbols (J&W Report section 3) 2.1
2.1.1. Letters 2.1
2.1.2. Symbols 2.1
2.2. Identifiers (J&W Report section 4) 2.2
2.3. Types (J&W Report section 6) 2.2
2.3.1. Integers 2.2
2.3.2. Long integers 2.2
2.3.3. Reals 2.3
2.3.4. Characters 2.3
2.3.5. Packed types 2.3
2.3.5.1. Packed arrays 2.3
2.3.5.2. Packed records 2.5
2.3.5.3. Using Packed Variables as Parameters 2.7
2.3.6. Record Types 2.7
2.3.7. String Types 2.8
2.3.7.1 STRING and LONGSTRING 2.8
2.3.7.2 CSTRING 2.12
2.3.8. Set Types 2.13
2.3.9. Files 2.14
2.3.9.1. Sequential Files 2.15
2.3.9.2. Direct Access Files 2.15
2.3.9.3. Predeclared Files 2.16
2.3.10. Pointers 2.16
2.4. Operators (J&W Report section 8.1) 2.16
2.4.1. DIV and MOD 2.16
2.4.2. = and <> 2.17
2.5. Statements (J&W Report section 9) 2.17
2.5.1. Goto Statements (J&W Report section 9.1.3) 2.17
2.5.2. CASE Statements (J&W Report section 9.2.2.2) 2.18
2.5.3. LOOP Statement 2.20
2.5.4. Declarations 2.21

2.6. Procedures and Functions (J&W Report section 10-11) 2.21
2.6.1. Parameter Restrictions 2.21
2.6.2. FORWARD Declarations 2.21
2.6.3. EXTERNAL Declarations 2.23
2.6.4. Segmentation 2.24
2.6.5. Separate Compilation 2.27
2.6.5.1 Modular Compilation 2.28
2.6.5.2 External Pascal Routines 2.32
2.7. Text Libraries 2.34
2.7.1. Advantages 2.34
2.7.2. Format of Text Files 2.35
2.7.3. Naming Text Libraries 2.36
2.7.4. Using Text Libraries 2.37

3. Standard Procedures and Functions 3.1
3.1. ABS 3.1
3.2. ALPHACMP 3.1
3.3. ARCTAN 3.2
3.4. CHAIN 3.3
3.5. CHR 3.4
3.6. CLEARSCREEN 3.4
3.7. CLOSE 3.5
3.8. CONCAT 3.5
3.9. COPY 3.6
3.10. COPYL 3.7
3.11. COS 3.7
3.12. CTOLONG and LONGTOC 3.8
3.13. DELAY 3.8
3.14. EDIT 3.9
3.15. EOF 3.10
3.16. EOLN 3.11
3.17. EXIT 3.11
3.18. EXP 3.13
3.19. FILLCHAR 3.14
3.20. GET 3.14
3.21. GETENVR 3.15
3.22 GETOPT 3.15
3.23. GOTOXY 3.16
3.24. IORESULT 3.17
3.25. ISLETTER 3.18

3.26. LENGTH 3.18
3.27. LOADTEXT 3.19
3.28. LN 3.19
3.29. LOCK AND UNLOCK 3.19
3.30. LONG 3.20
3.31. MARK and RELEASE 3.20
3.32. MAXINT 3.22
3.33. MEMAVAIL 3.22
3.34. METAMORPH 3.23
3.35. MOVELEFT 3.23
3.36. MOVERIGHT 3.24
3.37. NEW 3.26
3.38. NEWEDIT 3.26
3.39. ODD 3.27
3.40. ORD 3.28
3.41. PAGE 3.29
3.42. POS 3.29
3.43. PRED 3.29
3.44. PUT 3.30
3.45. PWROFTEN 3.31
3.46. READ 3.31
3.47. READLN 3.32
3.48. RELEASE 3.33
3.49. RESET 3.33
3.50. REWRITE 3.35
3.51. ROUND and ROUNDL 3.37
3.52. SCAN 3.37
3.53. SEEK 3.39
3.54. SETIORESULT 3.39
3.55. SETLENGTH 3.39
3.56. SHORT 3.40
3.57. SHORTSTRING 3.40
3.58. SIN 3.40
3.59. SIZEOF 3.40
3.60. SQR 3.41
3.61. SQRT 3.41
3.62. STACKAVAIL (only interpreted pascal) 3.42
3.63. STDTEXT 3.42
3.64. STRCAT 3.42
3.65. SUCC 3.43

3.66. TAN 3.44
3.67. TIME 3.44
3.68. TRUNC and TRUNCL 3.44
3.69. UNLOCK 3.45
3.70. WRITE 3.45
3.71. WRITELN 3.46

4. Pascal Preprocessor 4.1
4.1. Textual Replacement 4.1
4.2. Inclusion of Files 4.3
4.3. Conditional Compilation 4.3
4.4. Using the Preprocessor 4.4

5. Passing Parameters to the Pascal Program 5.1

6. Aborting the Program 6.1

Appendix A. DO's and DONT's A.1

1. Introduction

This manual describes Supermax Pascal Assembler Compiler and Supermax Interpreted Pascal. The Supermax Pascal compilers accept source programs written in the standard Pascal high level computer language (with minor restrictions and extensions). The manual refers to versions of the Pascal systems dated August 1987 or later.

The Supermax Compilers generate code to be run under Supermax Operating System V.

In the Supermax Pascal Assembler system the Pascal source code is translated to an assembler program and subsequently a load module can be produced. The Supermax Pascal Assembler system is described in the manual Supermax Running Pascal Assembler.

In the Supermax Interpreted Pascal system the Pascal source code is translated into code for a hypothetical computer known as the "P-machine". This code may subsequently be executed by using an interpreter program, which simulates the P-machine on an Supermax computer. The Supermax Interpreted Pascal system is described in the manual Supermax Running Interpreted Pascal.

This manual is a reference manual describing the differences between Supermax Pascal and standard Pascal as defined in

Kathleen Jensen and Niklaus Wirth
PASCAL: User Manual and Report
Third Edition
Springer Verlag New York Inc. 1985

Throughout this manual many references will be made to the above Pascal standard reference guide (referred to as "J&W").

Please note that this edition of J&W describes the ISO standard for Pascal. There are still some differences between Supermax Pascal and the ISO standard. Appendix E in J&W describes the differences between the two versions of Pascal described in second and third edition.

The reader of this manual is expected to be familiar with the Pascal language. This manual is not a tutorial book on Supermax Pascal.

The user must also be familiar with the Supermax operating system as described in the manual Supermax System Operation Guide.

Note:
Throughout this manual the term subroutine is used as a generic term for procedures and functions. The term word is used to designate a 2-byte (16 bit) entity.

The Supermax Pascal system is a modified version of the socalled UCSD Pascal compiler and interpreter version I.4e constructed by a team at the University of California, San Diego, (UCSD), directed by professor Kenneth Bowles. However, any malfunction of the system is the sole responsibility of Dansk Data Elektronik A/S.

Portions of this manual have been adapted from the UCSD release I.4 Pascal manual edited by K. A. Shillington.

Dansk Data Elektronik A/S reserves the right to change the specifications in this manual without warning. Dansk Data Elektronik A/S is not responsible for the effects of typographical errors or other inaccuracies in this manual, and cannot be held liable for the effects of the implementation and use of the structures described herein.

2. Language description.

This chapter describes the language differences between standard Pascal as described in J&W and Supermax Pascal.

Standard procedures and functions added to the system are not described in this chapter. Chapter 3 contains an alphabetical list of all the available procedures and functions.

2.0.1. A Note on Implementation.

The UCSD Pascal system was originally developed for a 16-bit computer. Doing some programming archaeology on the present Pascal system reveals traces of this 16-bit history in the implementation.

One such trace is the fact that a variable takes up an even number of bytes. Thus even a variable of type BOOLEAN, which ideally can be stored in just one bit, takes up one word (16 bits). Boolean elements of packed arrays or records, however, take up only one bit.

2.1. Letters and Symbols (J&W Report section 3).

2.1.1. Letters.

The definition of <letter> has been extended to include the character _ (underscore). National characters are not included.

Upper and lower case characters are considered equivalent. For example, the identifiers XYZ and xYz are equivalent, and the keyword BEGIN may also be written Begin or begin.

2.1.2. Symbols.

Brackets and braces can be used as follows:

- In subscripts (of arrays and strings) the brackets [and] can be replaced by (. and .) or by (and). The programmer may use whichever of the three options she prefers.

- In sets the brackets [and] can be replaced by (. and .).

- Comments can be delimited by (* and *) or by braces.

The definition of <special symbol> has been extended to include the keywords FORWARD, EXTERNAL, OTHERWISE, UNIV, LOOP, EXITIF, and ENDLOOP. Furthermore the keyword SEGMENT is a special symbol in interpreted Pascal and the keywords SUBPROGRAM and GLOBAL are special symbols in the pascal assembler system.

2.2. Identifiers (J&W Report section 4).

In identifiers only the first 14 characters are significant. Thus the identifiers ABCDEFGHIJKLMNO and ABCDEFGHIJKLMN denote the same object.

2.3. Types (J&W Report section 6).

2.3.1. Integers.

Integers are represented internally in one word 2's complement. Thus integer values range from -32768 to 32767.

Warning:
No error is indicated by the system if an arithmetic operation on an integer causes an overflow.

2.3.2 Long integers.

Long integers are represented internally in two words, two's complement. Thus long integer values range from -2147483648 to 2147483647.

Long constants greater than 32767 must be specified followed by l or L, e.g. 45321L or 45321l.

The long integer type is denoted LONGINT, thus a variable of this type, f.ex. L, is declared:

```
VAR L: LONGINT;
```

2.3.3. Reals.

Real numbers are represented internally as floating point numbers in four words in IEEE format.

2.3.4. Characters.

The type CHAR is implemented as the 8-bit ASCII character set. This character set is the ISO-8859/1 which contains national characters for the West European countries. A table on the character set is given in Supermax Virtual Terminal Guide page 3.2. A variable of type CHAR is represented internally in one word with the least significant byte containing the character.

2.3.5. Packed types.

2.3.5.1. Packed arrays.

The Supermax Pascal compilers perform packing of arrays and records if the ARRAY or RECORD declaration is preceded by the word PACKED. For example, consider the following declarations:

```
A: ARRAY (0..9) OF CHAR;
B: PACKED ARRAY (0..9) OF CHAR;
```

The array A will occupy ten 16-bit words of memory, with each element of the array occupying 1 word. The PACKED ARRAY B on the other hand will occupy a total of only 5 words since each 16 bit word contains two 8-bit characters. In this manner each element of the PACKED ARRAY B is 8 bits long.

Packed arrays need not be restricted to arrays of type CHAR. The following are examples of other legal constructs:

```
C: PACKED ARRAY [0..1] OF 0..3;
D: PACKED ARRAY [1..9] OF SET OF 0..15;
E: PACKED ARRAY [0..239,0..319] OF BOOLEAN;
```

Each element of the PACKED ARRAY C is only 2 bits long, since only 2 bits are needed to represent the values in the range 0..3. Therefore C occupies only one 16 bit word of memory, and 12 of the bits in that word are unused. The PACKED ARRAY D is a 9 word array, since each

element of D is a SET which can be represented in a minimum of 16 bits. Each element of a PACKED ARRAY OF BOOLEAN, as in the case of E in the above example, occupies only one bit.

The following two declarations are not equivalent due to the recursive nature of the compiler:

```
F: PACKED ARRAY [0..9] OF ARRAY [0..3] OF CHAR;
G: PACKED ARRAY [0..9,0..3] OF CHAR;
```

The second occurrence of the reserved word ARRAY in the declaration of F causes the packing option in the compiler to be turned off. The net result is that F becomes an unpacked array of 40 words. On the other hand, the PACKED ARRAY G is an array occupying 20 total words. If F had been declared as

```
       F: PACKED ARRAY [0..9] OF PACKED ARRAY [0..3] OF CHAR;
or as  F: ARRAY [0..9] OF PACKED ARRAY [0..3] OF CHAR;
```

then F and G would have had identical configurations.

In short, the reserved word PACKED only has true significance before the last appearance of the reserved word ARRAY in a declaration of a PACKED ARRAY. When in doubt a good rule of thumb when declaring a multidimensional PACKED ARRAY is to place the reserved word PACKED before every appearance of the reserved word ARRAY to ensure that the resultant array will in fact be packed.

The resultant array will only be packed if the final type of the array is a user-defined enumeration type, boolean, CHAR, subrange, or a set which can be represented in 8 bits or less. The following declaration will not result in any packing because the final type of the array cannot be represented in a field of 8 bits:

```
H: PACKED ARRAY [0..3] OF 0..1000;
```

H will be an array which occupies 4 16-bit words.

Packing never occurs across word boundaries. This means that if the type of the element to be packed requires a number of bits which does not divide evenly into 16, there will be some unused bits at the high order end of each of the words which comprise the array.

Note that a string constant may be assigned to a PACKED ARRAY OF CHAR but not to an unpacked ARRAY OF CHAR. Likewise, comparisons between an ARRAY OF CHAR and a string constant are illegal. Because of their different sizes, packed arrays cannot be compared to ordinary unpacked arrays.

A PACKED ARRAY OF CHAR may be output with a single write statement:

```
PROGRAM VERYSLICK;
VAR T: PACKED ARRAY [0..10] OF CHAR;
BEGIN
  T := 'HELLO THERE';
  WRITELN( T )
END.
```

Initialization of a PACKED ARRAY OF CHAR can be accomplished very efficiently by using the SIZEOF and FILLCHAR procedures defined in chapter 3.

Supermax Pascal does not support the standard procedures PACK and UNPACK defined in J&W page 192.

2.3.5.2. Packed records.

The following RECORD declaration declares a RECORD with 4 fields. The entire record occupies one 16 bit word as a result of declaring it to be a PACKED RECORD.

```
VAR R: PACKED RECORD
            I, J, K: 0..31;
            B: BOOLEAN
         END;
```

The variables I, J, K each take up 5 bits in the word. The boolean variable B is allocated in the 16th bit of the same word.

In much the same manner that packed arrays can be multidimensional, packed records may contain fields which themselves are packed records or packed arrays. Again, slight differences in the way in which decla-

rations are made will affect the degree of packing achieved. For example, note that the following two declarations are not equivalent:

```
VAR A: PACKED RECORD
         C: INTEGER;
         F: PACKED RECORD
              R: CHAR;
              K: BOOLEAN
            END;
         H: PACKED ARRAY [0..3] OF CHAR
       END;

VAR B: PACKED RECORD
         C: INTEGER;
         F: RECORD
              R: CHAR;
              K: BOOLEAN
            END;
         H: PACKED ARRAY [0..3] OF CHAR
       END;
```

As with the reserved word ARRAY, the reserved word PACKED must appear with every occurrence of the reserved word RECORD in order for the packed record to retain its packed qualities throughout all fields of the record. In the above example, only the record A is as completely packed as possible. In B, the F field is not packed and therefore occupies two 16 bit words. In contrast A.F has all of its fields packed into one word. However, it is important to note that a packed or unpacked array or record which is a field of a packed record will always start at the beginning of the next word boundary. This means that in the case of A in the above example, even though the F field does not completely fill one word, the H field starts at the beginning of the next word boundary.

A case variant may be used as the last field of a packed record, and the amount of space allocated to it will be the size of the largest variant among the various cases. The actual nature of the packing is beyond the scope of this manual.

```
VAR K: PACKED RECORD
         B: BOOLEAN;
         CASE F: BOOLEAN OF
              TRUE:  (Z: INTEGER);
              FALSE: (M: PACKED ARRAY [0..3] OF CHAR)
         END
       END;
```

In the above example the B and F fields are stored in two bits of the first 16 bit word of the record. The remaining 14 bits are not used. The size of the case variant field is always the size of the largest variant, so in the above example, the case variant field will occupy two words. Thus the entire packed record will occupy 3 words.

2.3.5.3. Using Packed Variables as Parameters.

No element of a packed array or field of a packed record may be passed as a variable (call-by-reference) parameter to a subroutine. Packed variables may, however, be passed as call-by-value parameters (as stated in J&W).

2.3.6. Record Types.

Contrary to the syntax diagrams for <field list> on page 226 of J&W a semicolon is not allowed before the END of a record type declaration if the record type contains a variant part. A semicolon is, however, allowed if the declaration does not contain a variant part.

If a record declaration contains a variant part the run-time system does not prevent the user from addressing undefined fields. For example, consider the following declaration:

```
VAR REC: RECORD
           A: BOOLEAN;
           CASE B: INTEGER OF
             1: (C: REAL);
             2: (D: INTEGER)
         END;
```

Addressing REC.D, even if REC.B contains the value 1, does not cause an error, therefore the user must be careful to check the value of the field controlling the variant part.

2.3.7. String types.

Supermax Pascal has three predeclared string types: STRING, LONGSTRING and CSTRING.

Variables of type STRING or LONGSTRING are essentially packed arrays of characters that have a dynamic length attribute, the value of which is returned by the function LENGTH (described in chapter 3).

The type CSTRING has been implemented to ease the passing of string parameters to external subroutines programmed in the language C. Thus variables of type CSTRING cannot be used as liberally as the other string types. Variables of type CSTRING can be assigned values, be read or written and can be used as parameters to external subroutines. (also see LONGTOC and CTOLONG described in chapter 3).

2.3.7.1 STRING and LONGSTRING.

The difference between the two types of strings lies in the number of characters they may contain. For STRING the dynamic length is stored (unsigned) in one byte thus allowing the string to contain a maximum of 255 characters whereas the dynamic length of a LONGSTRING is stored (signed) in two bytes thus allowing the string to contain 32767 characters.

It is not possible to access the dynamic length-part of a string, but the length can be read by the function LENGTH and changed by the procedure SETLENGTH. Special care should be taken when the dynamic length is changed by SETLENGTH; the user must assure that the new dynamic length is welldefined for the string in consideration.

The default maximum length of a variable of type STRING is 80 characters, and of type LONGSTRING 256 characters.

This default maximum length can be overridden in the declaration of a string variable by appending the desired length of the string variable

within [ ] (or (. .) or ( )) after the reserved type identifier STRING or LONGSTRING. Examples of declarations of string variables appear below:

```
TITLE: STRING;      (* defaults to a maximum length of 80
                       characters *)
NAME: STRING[20];   (* allows the string to contain a
                        maximum of 20 characters *)

LTITLE: LONGSTRING;(* defaults to a maximum length of 256
                       characters *)
LNAME: LONGSTRING[20];(* allows the string to contain a
                        maximum of 20 characters *)
```

The maximum string length given in a declaration must not exceed 255 characters for variables of type STRING and 32767 for variables of type LONGSTRING.

Assigning a value to a string variable includes assigning a dynamic length to the variable. Thus the assignment

```
TITLE := 'ABCD';
```

means: Store the characters 'A' through 'D' in the first four constituent characters of TITLE and assign a dynamic length of four to TITLE. The dynamic length must not exceed the maximum length given in the declaration of the string variable.

The individual characters within a string are indexed from 1 to the length of the string. For example:

```
TITLE(1) := 'A';  (* Sets the first character of the string
                     TITLE to 'A' *)

TITLE( LENGTH(TITLE) ) := 'Z';(* Sets the last character of
                                 the string TITLE to 'Z' *)
```

A string variable may not be indexed beyond its current dynamic length. The following sequence will result in an invalid index run time error:

```
TITLE := '1234';
TITLE(5) := '5';    (* Index error here *)
```

However the length can be set by SETLENGTH, and the following sequence will execute without error:

```
TITLE := 'ABCD';       (* Dynamic length := 4 *)
SETLENGTH(TITLE,5);    (* Dynamic length := 5 *)
TITLE(5) := 'E';       (* No index error here *)
```

The programmer should assume nothing about the dynamic length of a string before it has been initialized, not even that it is less than the indicated maximum length.

String variables (i.e. of type STRING or LONGSTRING) may be compared to any other variable of type STRING or LONGSTRING or a string constant no matter what its current dynamic length may be. Unlike comparisons involving variables of other types, string variables may be compared to items of a different length. The resulting comparison is lexicographical. Lower case characters are greater than upper case characters. The following program is a demonstration of legal comparisons involving variables of type string:

```
PROGRAM COMPARESTRINGS;
VAR S: STRING;
    T: LONGSTRING[40];

BEGIN
  S := 'SOMETHING';
  T := 'SOMETHING BIGGER';

  IF S=T THEN
    WRITELN('Strings don''t work too well')
  ELSE IF S>T THEN
    WRITELN(S,' is greater than ',T)
  ELSE IF S<T THEN
    WRITELN(S,' is less than ',T);
```

```
    IF S='SOMETHING' THEN WRITELN(S,' equals ',S);

    IF S>'SAMETHING' THEN
      WRITELN(S,' is greater than SAMETHING');

    IF S='SOMETHING ' THEN
      WRITELN('Blanks don''t count')
    ELSE
      WRITELN('Blanks make a difference');

    S := 'XXX';
    T := 'ABCDEF';

    IF S>T THEN
      WRITELN(S,' is greater than ',T)
    ELSE
      WRITELN(S,' is less than ',T);
    IF 'LETTER'<'letter' THEN
      WRITELN('LETTER is less than letter')
    ELSE
      WRITELN('LETTER >= letter')
END.
```

The above program will produce the following output:

```
SOMETHING is less than SOMETHING BIGGER
SOMETHING equals SOMETHING
SOMETHING is greater than SAMETHING
Blanks make a difference
XXX is greater than ABCDEF
LETTER is less than letter
```

As the comparison of strings is made using the ordinal value of the characters in the strings concerned, this comparison is not necessarily alphabetical. The order in which the national characters occur in for instance the Danish alphabet is not the same as the order in which they occur in the 8-bit ASCII alphabet. Alphabetical comparisons on strings can be performed using the standard procedure ALPHACMP - see chapter 3.

When a string variable is a parameter to the standard procedures READ and READLN, all characters up to the end of line character in the

source file will be assigned to the string variable. Care must be taken when reading string variables. The single statement

```
READLN(S1,S2)
```

is equivalent to the two statement sequence

```
READ(S1); READLN(S2)
```

In both cases the string variable S2 will be assigned the empty string. For further comments on this, see chapter 3.

As a string is essentially a packed array of characters, the elements of a string may not be used as variable (call-by-reference) parameters in procedure or function calls.

2.3.7.2 CSTRING.

A variable of type CSTRING is essentially a null-terminated packed array of characters, similar to strings in the programming language C. A CSTRING can contain up to 32767 characters. The default maximum length of a variable of type CSTRING is 256 characters. This default maximum length can be overridden in the declaration of a string variable by appending the desired length of the string variable within brackets after the reserved type identifier CSTRING.

CSTRINGS are usually used to pass strings to and from external routines written in C. Of course variables of type LONGSTRING and STRING can also be passed to C-routines, but then the programmer must take care and handle the dynamic length fields correctly. This is not needed when using CSTRINGs, since a declaration:

```
VAR CSTR: CSTRING[80];
```

allocates exactly the same memory as the C-declaration:

```
unsigned char CSTR[80];
```

Variables of type CSTRING can be assigned values directly in an assignment statement or by converting from a variable of type LONGSTRING

using the standard routine LONGTOC, and the resulting CSTRING from an external C-routine can be converted to a LONGSTRING by the standard routine CTOLONG (see section 3). Furthermore the standard procedures READ/READLN and WRITE/WRITELN can be used on variables of type CSTRING.

2.3.8. Set Types.

Supermax Pascal supports all of the constructs defined for sets on pages 77-78 of J&W. However the value of the elements in a set must lie in the range 0..4079. Thus the declaration

```
VAR S: SET OF 0..10;
```

is legal, whereas the declaration

```
VAR S: SET OF -1..10;
```

is illegal.

The maximum number of bits in a set thus becomes 4080, which is equivalent to 255 words.

Comparisons and operations on sets are allowed only between sets which are either of the same base type or subranges of the same underlying type. For example, in the sample program below, the base type of the set S is the subrange type 0..49, while the base type of the set R is the subrange type 0..100. However, the underlying type of both sets is the type INTEGER, which by the above definition of compatibility implies that the comparisons and operations on the sets S and R in the program on the following page are legal:

```
PROGRAM SETCOMPARE;
VAR S: SET OF 0..49;
    R: SET OF 0..100;

BEGIN
  S := [ 0,5,10,15,20,25,30,35,40,45 ];
  R := [ 10,20,30,40,50,60,70,80,90 ];

  IF S = R THEN
    WRITELN(' ...oops... ')
  ELSE
    WRITELN('sets work');
END.
```

However, in the following example the construct I = J is not legal since the two sets are of different underlying types.

```
PROGRAM ILLEGAL;
TYPE NUMBERS=(ZERO,ONE,TWO);
VAR I: SET OF NUMBERS;
    J: SET OF 0..2;

BEGIN
  I := [ ZERO ];
  J := [ 1,2 ];
  IF I=J THEN ;            <<<< ERROR
END.
```

2.3.9. Files.

Supermax Pascal supports two kinds of files: sequential files and direct access files.

The difference between the two types of files lies in the record structure within the files. A sequential file contains variable length records while direct access files contain fixed length records. A file may not contain records of both types.

In a variable length record file each record carries information about its own length. Variable length record files must be read or written sequentially, that is access to record no. n is possible only after

reading the preceding n-1 records. The main advantage of this file type is a better utilization of disk space because no record occupies unnecessary space.

In a fixed length record file the record length is defined when the file is opened, and all records in the file have the same length. The main advantage of this record structure is that it permits direct and fast access to any record in the file using the position file subroutine, SEEK (see chapter 3):

No assignment is allowed on file variables.

A file variable used as a formal parameter of a procedure or function must be a variable (call-by-reference) parameter.

2.3.9.1. Sequential Files.

Sequential files with variable length records are declared in the Pascal program as 'FILE OF CHAR' or simply 'TEXT'. Sequential files have a record length that does not exceed 255 bytes.

Sequential files are normally read and written using the procedures READ, READLN, WRITE, and WRITELN. These procedures are described in chapter 3 of this manual.

In a sequential file, a record corresponds to a line on a terminal or a printer.

2.3.9.2. Direct Access Files.

Files with direct access have fixed length records. Direct access files are declared in the Pascal program as 'FILE OF XXX', where XXX is some type other than CHAR (typically a RECORD).

Direct access files are read and written using the procedures GET and PUT. The procedure SEEK is used to position a file to a specific record. The procedures LOCK and UNLOCK are used to lock/unlock bytes (records) in files. These procedures are described in chapter 3 of this manual.

2.3.9.3 Predeclared Files.

The sequential files INPUT, OUTPUT, and ERROR are predeclared. These correspond directly to the standard UNIX iounits.

2.3.10 Pointers.

The standard procedure DISPOSE defined on page 191 of J&W is not implemented in Supermax Pascal. However, the function of DISPOSE can be approximated by a combined use of the Supermax Pascal procedures MARK and RELEASE, which are described in chapter 3 of this manual.

The programmer should note that there is no protection in the system that prevents a program from addressing a memory area through an uninitialized pointer or a pointer whose value is NIL.

2.4. Operators (J&W Report section 8.1).

2.4.1. DIV and MOD.

J&W leave undefined the result of the operators DIV and MOD if either of the operands are negative. In Supermax Pascal the following rules apply:

```
  11  DIV   3  =  3          11  MOD   3  =  2
  10  DIV   3  =  3          10  MOD   3  =  1
   9  DIV   3  =  3           9  MOD   3  =  0

(-11) DIV   3  = -4        (-11) MOD   3  =  1
(-10) DIV   3  = -4        (-10) MOD   3  =  2
 (-9) DIV   3  = -3         (-9) MOD   3  =  0

  11  DIV (-3) = -4          11  MOD (-3) = -1
  10  DIV (-3) = -4          10  MOD (-3) = -2
   9  DIV (-3) = -3           9  MOD (-3) =  0

(-11) DIV (-3) =  3        (-11) MOD (-3) = -2
(-10) DIV (-3) =  3        (-10) MOD (-3) = -1
 (-9) DIV (-3) =  3         (-9) MOD (-3) = -0
```

2.4.2. = and <>.

The operators = and <> are allowed for comparison of any array or record structure.

2.5. Statements (J&W Report section 9).

2.5.1. Goto Statements. (J&W Report section 9.1.3).

Supermax Pascal has a more limited form of the GOTO statement than is defined as the standard in J&W. Supermax Pascal's GOTO statement may not transfer control to a label which is not within the same block as the GOTO statement itself.

As UCSD Pascal was originally created for a university environment, where good programming style is (or should be) taught, and as GOTO statements are generally considered bad programming style and often are a symptom of poor program structure, GOTO statements are normally not allowed in UCSD Pascal (and hence Supermax Pascal) programs. If the programmer insists on using GOTO statements, she must explicitly tell the compiler to allow them. This is done by including the compiler directive (*$G+*) described in Supermax Running Pascal Assembler and Supermax Running Interpreted Pascal before the first occurrence of a GOTO.

For example:

```
PROGRAM TEST;
(*$G+*)
LABEL 4711;
BEGIN
  ...
  GOTO 4711;
  ....
  4711: ...
END.
```

2.5.2. CASE Statements (J&W Report section 9.2.2.2).

The syntax for a case statement has been altered. The final END of a case statement may be replaced by the key word OTHERWISE followed by a statement (possibly a compound statement, which is several statements enclosed between a BEGIN and an END).

J&W state that the result of the case statement is undefined if there is no label equal to the value of the case statement selector. This is not the case in Supermax Pascal, where one of the following happens:

- If the case statement ends with an END, nothing happens.

- If the case statement ends with an OTHERWISE followed by a statement, then that statement is executed.

Example:

```
        FOR I:=1 TO 4 DO
          CASE I OF
            1: WRITELN('ONE');
            3: WRITELN('THREE');
          END;
```

will output
ONE
THREE

whereas

```
        FOR I:=1 TO 4 DO
          CASE I OF
            1: WRITELN('ONE');
            3: WRITELN('THREE');
          OTHERWISE
           WRITELN('WHAT?');
```

will output
ONE
WHAT?
THREE
WHAT?

Furthermore it is possible to give a subrange of the selector type as a label: (C is of type char)

```
CASE C OF
  'a','e','i','o','u','y': writeln('a vowel');
  'b'..'d',
  'f'..'h',
  'j'..'n',
  'p'..'t',
  'v'..'x',
  'z'         : writeln('a consonant');
OTHERWISE
 WRITELN('not a letter');
```

Note:

Interpreted Pascal:

When translating a case statement the Pascal compiler lays out a jump table containing one entry (one word) for each value between the lowest and the highest label constants. This means that a case statement such as

```
CASE I OF
  1:   WRITELN('ONE');
  2:   WRITELN('TWO');
  500: WRITELN('ERROR');
END;
```

creates a jump table with 500 entries, which take up 1000 bytes plus the code required by the three WRITELN calls. This should therefore be avoided.

When using a statement selector of type LONGINT, the label values may not exceed 32767.

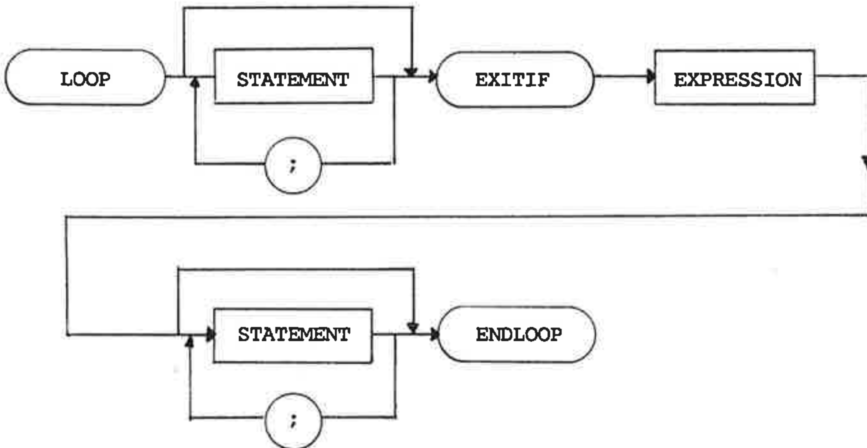
Pascal assembler

The Pascal assembler compiler does not use a jump table when a case statement is used; instead the statement is interpreted as a nested if-then-else construct. Execution speed may therefore be improved by moving the mostly used caselabels to the beginning of the case statement.

2.5.3. LOOP-statement.

Standard Pascal supports three kinds of repetitive statements: FOR, REPEAT, and WHILE statements which differ in the way the repetition stops. The statements to be repeated in a FOR statement are executed an exact number of times computed when entering the statement. In the other two kinds of statements the involved statements are executed as long as a given condition is TRUE - this condition is tested before the statements to be executed in a WHILE statement and after in a REPEAT statement.

Supermax Pascal supports a fourth kind of repetitive statement: the socalled LOOP-statement. The syntax is given below.



Since STATEMENT can be an empty statement it is possible to exit from a LOOP statement in the beginning or the end of the loop or between statements.

2.5.4. Declarations.

To ease the use of including source code from several files while compiling the order in which declarations must occur has been relaxed. Thus declarations of LABEL, CONST, TYPE, VAR, PROCEDURE and FUNCTION can be mixed but each type of declaration must be preceeded by the proper reserved word.

2.6. Procedures and Functions (J&W Report sections 10-11).

2.6.1. Parameter Restrictions.

Supermax Pascal does not yet allow subroutines as formal parameters in the parameter list of a subroutine.

2.6.2. FORWARD Declarations.

Supermax Pascal requires that a subroutine be declared before it is used. This may cause problems in two cases:

1) We have two subroutines, A and B. If A refers to B and B refers to A, then, by the above rule, either must be declared before the other.

2) We have a segment subroutine (see section 2.6.4) A which calls a non-segment subroutine B. As the code of segment subroutines must be given before the code of non-segment subroutines, the code of A must be given before the code of B. However, as A calls B, B must be declared before A.

Note that segmentation is only implemented in Interpreted Pascal.

These problems are solved by using the so-called FORWARD declarations.

A subroutine declaration may be given without the corresponding subroutine block. This is done by replacing the block of the subroutine by the key word FORWARD. Later in the program the declaration of the subroutine may be repeated with the corresponding block. In this repeated declaration, the parameter list of the subroutine and the type of the value returned, if the subroutine is a function, is omitted.

The solution to problem 1 above may then be this:

```
PROCEDURE A(I: INTEGER; R: REAL);
FORWARD;

PROCEDURE B(J: INTEGER);
VAR X,Y: REAL;
BEGIN
  ....
  A( 3, 5.5 );    (* Reference to procedure A is OK
                     because A was declared above *)
  ....
END;

PROCEDURE A; (* Repeated declaration with no parameters *)
VAR S: STRING;
BEGIN
  ....
  B( 4 );         (* Reference to procedure B is OK
                     because B was declared above *)
  ....
END;
```

Often it improves program readability if the second declaration of A includes the parameters of A in a comment:

```
PROCEDURE A(* I: INTEGER; R: REAL *);
```

This makes it clear that the I and R used in the block of procedure A are parameters, not global variables.

The solution to problem 2 above may be this:

```
FUNCTION B(S: STRING): REAL;
FORWARD;

SEGMENT PROCEDURE A(J: INTEGER);
VAR X,Y: REAL;
BEGIN     (* The code of A comes before the code of B *)
  ....
  X:=B('ALFA');  (* Reference to procedure B is OK
                    because B was declared above *)
  ....
END;

FUNCTION B;     (* No parameters or return value type *)
   BEGIN
  ....
END;
```

2.6.3. EXTERNAL Declarations.

A subroutine block may be replaced by the key word EXTERNAL. This tells the compiler that the body of this particular subroutine will not be given in the pascal program itself. An external subroutine is compiled seperately and can be used (that is called) from different Pascal programs. In interpreted Pascal only subroutines written in C can be used as EXTERNAL, whereas EXTERNAL subroutines used in the Pascal Assembler system can be written in either Pascal or C.

Example:

```
PROGRAM TEST;
VAR I: INTEGER;

PROCEDURE ALPHA(LEN: INTEGER);
EXTERNAL;

BEGIN
  READ(I);
  ALPHA(I);
END.
```

Names of external subroutines written in C may not exceed 14 characters.

If a parameter is to be used as a variable declared file of xxx or text or (unpacked) record further typecheck can be avoided by declaring the variable using the form UNIV.

Example:
```
TYPE REC = RECORD X : XX END;
     F= FILE OF REC;
PRODECURE YYY(UNIV FF : F); EXTERNAL;
```

With the above declaration of YYY, the procedure can be called using any type of direct access file.

A further discussion on external subroutines is given in Supermax Pascal C Interface concerning C-routines and in section 2.6.5.2 of this manual concerning Pascal-routines.

2.6.4. Segmentation.

Segmentation is only possible in interpreted Pascal (the Pascal Com/Int System).

Declarations of segment subroutines are identical to declarations of normal subroutines except they are preceded by the reserved word 'SEGMENT', for example:

```
SEGMENT PROCEDURE INITIALIZE;
BEGIN
  (* Pascal code *)
END;
```

Note: segment subroutines are only implemented in interpreted Pascal.

Program behavior differs, however, in that code for a segment subroutine is in memory only while there is an active invocation of that subroutine. The code for a segment subroutine is loaded when the sub-

routine is called, and the memory area it occupies is released when a return is made from the subroutine.

The user may put large pieces of one-time code, for example, initialization code, into a segment subroutine. After performing the initialization, the memory area of the now useless code is released thus increasing the available memory space.

The disk which holds the code file for the program must be ready in the computer whenever a new segment subroutine is to be called. A maximum of 15 segment subroutines are ordinarily available to the user. Code for segment subroutines must be given before code for non-segment subroutines.

The following program is an example of the use of segment subroutines.

```
PROGRAM SEGMENTDEMO;

SEGMENT PROCEDURE ONE;

  PROCEDURE TWO;
  BEGIN
    WRITELN('TWO')
  END;

BEGIN
  WRITELN('ONE');
  TWO
END;

SEGMENT PROCEDURE THREE;
BEGIN
  WRITELN('THREE')
END;

BEGIN
  ONE;
  THREE;
  WRITELN('I''M DONE')
END.
```

```
ONE
TWO
THREE
I'M DONE
```

Procedure TWO above is an integrated part of segment procedure ONE and is loaded when ONE is called. Procedure TWO might have been declared a segment procedure as well, but in this example nothing would be gained by that.

The fact that the code of procedure TWO is given before the code of segment procedures ONE and THREE does not violate the rule that the code of segment subroutines be given before the code of non-segment subroutines, for TWO is merely a part of segment procedure ONE.

The declaration of a non-segment subroutine may precede the declaration of a segment subroutine as long as the code for the segment subroutine is given before the code for the non-segment subroutine, as in the following example, where BETA is a local procedure within ALPHA:

```
PROGRAM SEGMENTDEMO;

  PROCEDURE ALPHA;

    SEGMENT PROCEDURE BETA;
    BEGIN
      ...
    END;

  BEGIN
    ...
  END;

BEGIN
  ...
END.
```

Note: care should be taken when combining FORWARD and SEGMENT. For example the declaration shown below is erroneous:

```
PROGRAM SEGMENTERROR;
  ...
  PROCEDURE A; FORWARD;
  ...
  SEGMENT PROCEDURE A;
  BEGIN ... END;
  ...
BEGIN ... END.
```

as A first is declared (and forwarded) in one segment and then is declared in another.

2.6.5 Separate compilation.

Seperate compilation is only possible when using the Pascal-Assembler System.

Supermax Pascal supports two different ways of compiling programs separately: modular compilation and external Pascal routines.

By modular compilation of Pascal programs is meant the possibility of splitting a large program into smaller subprograms that can be compiled separately producing several relocatable modules that can be linked together thus creating the final machine code of the large program.

Thus a single error in the program need not require recompilation of the whole program, but just the subprogram containing the error.

A modular compiled module is thus part of a singular Pascal program and cannot be linked as part of any other program. In modular compilation global structures from the unsplit program can be made known in all modules and type checking will always be performed.

On the other hand it is possible to write external subroutines in Pascal, that can be linked to several programs. Here it is not possible to perform type checking on parameters passed to the routines, and global variables can only be referenced if they are passed as parameters.

2.6.5.1 Modular compilation.

A Pascal program which is to be modularly compiled must consist of a main module and one or more submodules. The main module is characterized by the first symbol in the program text being the special symbol PROGRAM. Likewise a submodule is characterized by the special symbol SUBPROGRAM.

The main block of the program must be located in the main module. This module can also contain routines declared on level 0 (i.e. their body is on level 1) - the main body of the program is at level 0.

A submodule consists of one or more subroutines declared on level 0. Global structures, that is: types, constants and/or variables declared on level 0 in the program are made available to a submodule by including a file containing the original declarations; the inclusion is made using the compiler directive F. The only other declarations allowed on level 0 are declarations of subroutines.

A submodule is ended by the END; belonging to the last procedure in the module.

Furthermore a subroutine declared on level 0 in the mainmodule or in a submodule can be called in any other module provided the routine is declared to be global. This is done analogous to FORWARD declarations:

In every module where one wishes to call the subroutine the routine is declared with the body of the routine substituted by the special symbol GLOBAL. In the module where the subroutine is declared with the body of the routine the parameters are left out (like for FORWARD).

The GLOBAL declarations of subroutines can be placed in the F-inclusion file. This file can be used in all modules; the compiler recognizes the main module and generates code for variables declared in the F-file when compiling the main module whereas the declarations are used for syntaxchecking in the submodules. Likewise the GLOBAL declarations of subroutines are used for syntaxchecking. The inclusion of a file using F must be located before any other declarations.

Example 1:

Often a large Pascal program consists of several files that are included in a main file. One of these files can contain all declarations and FORWARD declarations of several subroutines. Another file may contain an initialization subroutine while the rest of the program is stored in the rest of the files. Splitting a program like this into a main module, a F-file and one or more submodules, compilation time can be saved. One way of doing this is to gather all global declarations in a F-file together with all FORWARD declarations changed to GLOBAL declarations. It is obvious to have a submodule containing the initialization subroutine and perhaps more subroutines can be gathered in other submodules.

Example 2:

This example shows how a program can be split so that it can be modularly compiled.

Consider the following program:

```
PROGRAM EXAMPLE;
CONST
  PI = 3.14;
TYPE
  VECTOR = RECORD
             X,Y: REAL;
           END;

  CIRCLE = RECORD
             CENT: VECTOR; (*CENTER*)
             PERIF: VECTOR; (*PERIPHERAL POINT*)
           END;
```

```
VAR
  C: CIRCLE;
  RES: REAL;

PROCEDURE SUB(A,B: VECTOR; VAR C: VECTOR);
BEGIN
  C.X := A.X - B.X;
  C.Y := A.Y - B.Y;
END;

FUNCTION VLENGTH(A: VECTOR): REAL;
BEGIN
  VLENGTH := SQRT(SQR(A.X) + SQR(A.Y));
END;

FUNCTION AREA(A: CIRCLE): REAL;
VAR
    V: VECTOR;
    RADIUS: REAL;
BEGIN
  SUB(A.CENT,A.PERIF,V);
  RADIUS := VLENGTH(V);
  AREA := PI * SQR(RADIUS);
END;

BEGIN (*MAIN PROGRAM*)
  WRITE('READ CENTER (X Y): ');
  READ(C.CENT.X, C.CENT.Y);
  WRITELN;
  WRITE('READ PERIFERAL POINT (X Y): ');
  READ(C.PERIF.X, C.PERIF.Y);
  WRITELN;
  RES := AREA(C);
  WRITELN('AREA OF CIRCLE: ',RES);
END.
```

The program reads the coordinates of two points one being the center of a circle and the other a peripheral point. After some calculations the area of the circle is printed.

In the following the program will be split into a F-file: decl.p, a main module: main.p and a submodule: module.p.

F-file: decl.p

```
CONST
  PI = 3.14;
TYPE
  VECTOR = RECORD
             X,Y: REAL;
           END;

  CIRCLE = RECORD
             CENT: VECTOR; (*CENTER*)
             PERIF: VECTOR; (*PERIPHERAL POINT*)
           END;
VAR
  C: CIRCLE;
  RES: REAL;

FUNCTION AREA(A: CIRCLE): REAL; GLOBAL;
```

Main module: main.p

```
PROGRAM EXAMPLE;
(*$Fdecl.p*)
BEGIN (*MAIN PROGRAM*)
  WRITE('READ CENTER (X Y): ');
  READ(C.CENT.X, C.CENT.Y);
  WRITELN;
  WRITE('READ PERIFERAL POINT (X Y): ');
  READ(C.PERIF.X, C.PERIF.Y);
  WRITELN;
  RES := AREA(C);
  WRITELN('AREA OF CIRCLE: ',RES);
END.
```

Submodule: module.p

```
SUBPROGRAM ROUTINES;
(*$Fdecl.p*)
PROCEDURE SUB(A,B: VECTOR; VAR C: VECTOR);
BEGIN
  C.X := A.X - B.X;
  C.Y := A.Y - B.Y;
END;

FUNCTION VLENGTH(A: VECTOR): REAL;
BEGIN
  VLENGTH := SQRT(SQR(A.X) + SQR(A.Y));
END;

FUNCTION AREA;
VAR
    V: VECTOR;
    RADIUS: REAL;
BEGIN
  SUB(A.CENT,A.PERIF,V);
  RADIUS := VLENGTH(V);
  AREA := PI * SQR(RADIUS);
END;
```

In the F-file only the function AREA is declared GLOBAL, since the other subroutines only are called in the module containing their body.

The names of variables, constants and subroutines declared on level 0 are generated in the assembler-code in capital letters. Subroutines declared on level 1 and more are not generated using their own name but by the name of the subroutine on level 0 in which they are contained followed by a number indicating what number subroutine (in the text of the module) it is.

2.6.5.2. External Pascal Routines.

As mentioned before it is possible to write external subroutines in Pascal. Such a module is recognized by the first symbol in the program text being the special symbol EXTERNAL followed by a name.

External modules may contain one or more subroutines and local declarations, for example variables which are only used in the module. This is analogous to static declarations in C. It is only possible to use global variables from another Pascal program by passing these as parameters to the external subroutine.

In the program in which the subroutine is to be called the subroutine is declared with it's body replaced by the special symbol EXTERNAL. Since this is the same way as external subroutines written in C are declared, it is not possible to recognize by the usage of an external subroutine whether the routine originally was written in C or in Pascal.

An external module is not part of a specific program (such as submodules). If the same external module is linked to different programs, it is the user's responsibility, that the types used in the programs and in the external modules are the same.

The names of subroutines declared in external modules are generated in the assembler code using small letters and only the first fourteen characters of the name are used.

Example:

Consider example 2 in section 2.6.5.1. Presume that a programming package concerning calculations on vectors is to be made. Then the subroutine VLENGTH which calculates the length of a vector may be needed in several programs, and it would be convenient to have the source code of this routine as a separate file instead incorporating it into the source text of all the programs in which it is to be used.

This is solved by declaring VLENGTH as an EXTERNAL in the programs in which it is to be called, and placing the function itself in an external module. The function does it's calculations on a variable of type VECTOR, it is important, that this type is declared in the same way in the program and in the external module.

```
EXTERNAL VECT;
TYPE
  VECTOR = RECORD
             X,Y: REAL;
           END;

FUNCTION VLENGTH(A:VECTOR):REAL;
BEGIN
  VLENGTH := SQRT(SQR(A.X) + SQR(A.Y));
END;
```

In the programs using VLENGTH the subroutine is declared:

```
FUNCTION VLENGTH(A:VECTOR):REAL; EXTERNAL;
```

2.7 Text Libraries.

Supermax Pascal has been enhanced to support so-called Native Language Support of programs. By this is meant the possibility of making program's use of string constants language independent.

When a program is written it is assumed that the string constants needed are located in a special format text file, that can be loaded into main memory by the program, and the individual strings can be indexed from here.

This chapter describes the format of the text file, the advantages of using text libraries and which routines are used when programming.

Note: each Pascal program can load just one text library.

2.7.1 Advantages.

Application programs often contain string constants holding information to be written to the user of the application. Up till now these have often been written directly in the program, and when the application should be able to run in different countries writing it's messages in the relevant native language, this has been accomplished

by including the strings in all wanted languages and by using conditional compilation a version in each language has been created.

Using this method a simple change in a string constant necessitates recompilation of the program, and furthermore running the same application in two different languages on one machine requires two versions of the program code to be loaded into main memory.

When using text libraries only one instance of the program need be in main memory and via user environments the text file containing the string constants to be used can be loaded. A change in a string constant only requires changing the string in the text file.

Altogether the use of text libraries saves space in main memory, saves recompilation of programs and produces less redundant code.

2.7.2 Format of text files.

Example:

```
#0
Welcome to the program
#1
File is not found
#2
File is in use
#3
This is long line, "\" before newline indicates \
that the line continues
```

A line starting with a # and followed by a number indicates the index of the string constant on the following line. A long string constant can be continued on the following line by ending the line with a \.

2.7.3 Naming Text Libraries.

The name of a text library (text file) is often identical to the name of the application in which it is used. The user can name a text library as she wishes using a usual UNIX name, however some rules apply to what directory to place the file in.

All text libraries are searched for using a path given in the environment NLSPATH. The value of this environment is:

```
NLSPATH=/<path-prefix>/%L/%N.cat:/<path-prefix>/%N/%L
```

where <path-prefix> is the first part of the path searched, and it can differ for different users or different products. %L is substituted by the value of an environment LANGUAGE and %N is substituted by the name of the file to be loaded.

The default value of NLSPATH is:

```
NLSPATH=/nlslib/%L/%N.cat:/nlslib/%N/%L
```

The default value of the environment LANGUAGE is uk indicating english to be used.

example

LANGUAGE=dk

The text library is loaded using the standard procedure LOADTEXT which has a text name as parameter.

When LOADTEXT('editor') is written in the pascal program and the program is run, the system will use the NLSPATH environment substituting %N with editor and %L with the value of LANGUAGE.

If /nlslib/dk/editor.cat is found this text will be loaded. Otherwise /nlslib/editor/dk is loaded if it exists. If neither exist IORESULT is set to 205 (iounit does not exist) telling that it isn't possible to load the wanted library.

2.7.4 Using Text Libraries.

By changing the value of LANGUAGE different users can utilize the same programcode but have messages (string constants) in different languages.

Once LOADTEXT has been called successfully the individual text constants can be acquired by using the standard function STDTEXT, which has an integer parameter (index of the string constant wanted) and returns the string as a LONGSTRING.

Example:

```
VAR MESSAGE:LONGSTRING;
.
.
LOADTEXT('editor');
IF IORESULT <> 0 THEN BEGIN
  WRITELN('TEXT LIBRARY NOT LOADED');
  EXIT(PROGRAM);
END;

WRITELN(STDTEXT(0)); (* print message 0 to user *)
.
.
MESSAGE := STDTEXT(23);
.
.
```

3. Standard Procedures and Functions.

This chapter contains a complete alphabetical list of all standard subroutines (procedures and functions) available in Supermax Pascal.

The first line(s) of each description contains a Pascal declaration which describes the parameters of the subroutine and the return value type if the subroutine is a function. Often this declaration has to be given in pseudo-Pascal, because many of the subroutines have facilities which cannot be expressed in normal Pascal. For example, WRITE has a variable number of parameters, COPY returns a value of type STRING, ABS may be either an integer function with an integer parameter or a real function with a real parameter.

3.1 ABS

```
FUNCTION ABS(A: INTEGER): INTEGER;
or
FUNCTION ABS(A: REAL): REAL;
```

ABS returns the absolute value of its argument.

Example:
ABS(3)=3   ABS(-3)=3

3.2 ALPHACMP

```
FUNCTION ALPHACMP(STR1, STR2: xxx; <rel.op.>): BOOLEAN;
```

where xxx can be of type STRING or LONGSTRING, and <rel.op.> is one of the following relational operators: <, <=, >, >=, <>, =.

ALPHACMP performs an alphabetical comparison of the two strings using the given operator and returns TRUE if

STR1 <rel.op.> STR2

and FALSE otherwise.

The alphabetic comparison is made using a character table describing the alphabet to be used, for example a table on the Danish alphabet. The name of this table must be given in the environment ALPHABET, f.ex. ALPHABET=dk for the Danish alphabet. The tables themselves are stored in the directory /usr/lib/alphabet. A table describing the Danish alphabet (/usr/lib/alphabet/dk) is supplied with the Pascal System.

If the ALPHABET environment is not set, ALPHACMP will default perform the usual Pascal comparison on strings - as if the infix operator had been used. This is similar to English alpabetization.

Example:
Consider the following call of ALPHACMP:

```
ALPHACMP('Ølgod','Ågård',<)
```

if ALPHABET=dk the value TRUE is returned, as Ølgod is alphabetically before Ågård in Danish. If the environment ALPHABET has not been set the usual Pascal comparison is used and the call is the same as

```
'Ølgod' < 'Ågård'
```

which is FALSE as the ordinal value of Ø is 216 and the ordinal value of Å is 197.

3.3 ARCTAN

```
FUNCTION ARCTAN(A: REAL): REAL;
```

ARCTAN returns a value in the range $-\pi/2..\pi/2$. This value is the radian value of the inverse tangent of the argument of the function.

Example:
ARCTAN(1)*4=3.141592635389.

3.4 CHAIN

```
PROCEDURE CHAIN(PROGRAMNAME, PARMSTRING: xxxx;
                VAR PID: ^INTEGER);
```

Both xxxx parameters can be STRING or LONGSTRING. This procedure is used to start the execution of another program. When CHAIN is called, the specified program is started. The process which executes the specified program can be a produced og gemmated process, or it can be an offspring process (spawned). The standard iounits for the started program can be specified in the CHAIN command.

The return value of the CHAIN call is the process number for the new process given in the first two bytes of PID. (PID can be used as the parameter of the external subroutine PWAIT).

The format of the parameter PROGRAMNAME is:

unitname ! ^unitname !@unitname

where unitname is the iounit name of a file containing the program. If nothing precedes the unitname the new process will be spawned. If ^ precedes the unitname the process will be gemmated. It will be produced if @ precedes the unitname.

The format of the parameter PARMSTRING is (the brackets '()' indicate that the parameters are optional):

(program parameter)(standard unit list)

where program parameter is passed to the started program as the parameter string. Standard unit list indicates what iounits the started program should use for standard iounits. When no standard unit list is given the started program inherits the standard iounits from the program calling CHAIN. Iounits are passed using the format:

```
input=<filename>
output=<filename>
error=<filename>
```

A call of CHAIN affects IORESULT.

The priority of the new process is 10.

Example:
The following code is equivalent to the program start command

```
$ alpha pppp < infil > udfil
```

```
PROGRAM START;
VAR PID:   ^INTEGER;
    IOVAL: INTEGER;
BEGIN
  CHAIN('ALPHA','PPPP input=infil output=udfil',PID);
  IF IORESULT<>0 THEN
    BEGIN
      IOVAL:=IORESULT;
      WRITELN('ERROR ',IOVAL)
    END
END.
```

3.5 CHR

```
FUNCTION CHR(A: INTEGER): CHAR;
or
FUNCTION CHR(A: LONGINT): CHAR;
```

This function returns the character that has the ASCII value A. The inverse of this function is the function ORD.

Examples:
`CHR(97)='a'   CHR( ORD('X') +2 )='Z'`

3.6 CLEARSCREEN

```
PROCEDURE CLEARSCREEN;
```

This procedure clears the terminal screen and places the cursor in the upper left corner.

3.7 CLOSE

```
PROCEDURE CLOSE(VAR F: FILE OF xxx);
or
PROCEDURE CLOSE(VAR F: TEXT);
```

This procedure closes the file given as its parameter. If the file is sequential and was opened by a REWRITE call (see section 3.50) an end-of-file mark is written onto the file at the present file position even if no write operation has been performed on the file.

A possible error may be detected by the IORESULT function; however, the system checks this itself, unless the programmer has specified the (*$C-*) compiler directive (see Supermax Running Interpreted Pascal or Supermax Running Pascal Assembler).

Often the programmer need not explicitly close a file, as the system automatically closes all files declared within a subroutine or program, when an exit is made from that subroutine or program.

After calling the CLOSE procedure, the file variable may be used in a new opening of a file.

CLOSE should not be applied to the predeclared files INPUT, OUTPUT, and ERROR.

3.8 CONCAT

```
FUNCTION CONCAT(A, B, C, ... : xxx): STRING;
```

Where xxx can be STRING or LONGSTRING. This function takes any number of strings as parameters and returns a STRING which is the concatenation of the parameters. (See also STRCAT, section 3.64, concerning concatenation of strings giving a LONGSTRING as result).

Example:

```
TEXT1 := 'WE HOLD';
TEXT2 := 'THESE TRUTHS TO BE ';
TEXT2 := CONCAT(TEXT1,' ',TEXT2,'SELF EVIDENT');
WRITELN( TEXT2 );
```

will print
WE HOLD THESE TRUTHS TO BE SELF EVIDENT

Note that a variable of type CHAR may not be used as a parameter in the CONCAT call, but a string of length 1 may.

Note: Due to an implementation error, expressions involving two calls of CONCAT do not work properly. Thus, for instance, the expression CONCAT(...)=CONCAT(...) may erroneously yield the value TRUE even if the two strings are not identical.

3.9 COPY

```
FUNCTION COPY(S: xxx; INDEX, SIZE: INTEGER): STRING;
```

Where xxx can be STRING or LONGSTRING. This function extracts a substring from the string S. The length of the substring is the value of the parameter SIZE, and the substring starts at index INDEX in the source string.

SIZE must be at least 1.

An index error will be reported as a run time error when the COPY call is executed, even if index checking has been disabled through a (*$R-*) compiler directive (see Supermax Running Pascal Assembler or Supermax Running Interpreted Pascal).

Example:
```
COPY( 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 4, 3 )='DEF'
```

Note: Due to an implementation error, expressions involving two calls of COPY do not work properly. Thus, for instance, the expression COPY(...)=COPY(...) may erroneously yield the value TRUE even if the two strings are not identical.

3.10 COPYL

```
PROCEDURE COPYL(VAR L: LONGSTRING; S: STRING;
                INDEX, SIZE: INTEGER);
or
PROCEDURE COPYL(VAR L: LONGSTRING; S: LONGSTRING;
                INDEX, SIZE: INTEGER);
```

This procedure extracts a substring from the string S. The length of the substring is the value of the parameter SIZE, and the substring starts at index INDEX in the source string. The substring is returned in the long string specified by L.

SIZE must be at least 1.

An index error will be reported as a run time error when the COPY call is executed, even if index checking has been disabled through a (*$R-*) compiler directive (see Supermax Running Pascal Assembler or Supermax Running Interpreted Pascal).

A run-time error will occur if the length of the substring is greater than the declared length of the resulting string.

Example:

```
VAR L: LONGSTRING;
.
.
COPYL(L, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 4, 3 );
WRITELN(L);
```

will print:

```
DEF
```

3.11 COS

```
FUNCTION COS(A: REAL): REAL;
```

The function returns the cosine value of its parameter. The parameter is given in radians.

3.12 CTOLONG and LONGTOC.

PROCEDURE CTOLONG(C: CSTRING; VAR L: LONGSTRING);
and
PROCEDURE LONGTOC(L: LONGSTRING; VAR C: CSTRING);

These procedures perform conversion between strings of type LONGSTRING and strings of type CSTRING.

C-strings are intended only for use as parameters to external subroutines written in C.

The following program code examplifies the use of the procedures. Assume that xxx(s) is an external procedure written in C, where s is declared : unsigned char *s; and the routine modifies the string in some way, that is we want to examine the resulting string.

```
program test;
var str: longstring;
    cstr: cstring;
procedure xxx(var s:cstring);external;
begin
  read(str);
  longtoc(str,cstr);
  xxx(cstr);
  ctolong(cstr,str);
  writeln(str);
end.
```

Running the program the modified string will be printed.

3.13 DELAY

PROCEDURE DELAY(T: INTEGER);

This procedure delays the program for the time period given in the parameter T. This parameter is treated as an unsigned value, specifying the delay time in centiseconds (sic!), that is, 10-millisecond periods. The accuracy of the delay time, however, is coarser than 10 milliseconds, normally 40 milliseconds.

A delayed program uses no CPU time.

Example:
To delay a program for approximately 2 seconds the following call may be used:

```
DELAY(200)
```

3.14 EDIT

```
PROCEDURE EDIT(VAR S: xxx);
```

Where xxx can be STRING or LONGSTRING. The parameter for this procedure may be given either simply as a string variable, og as a string variable followed by specification of an edit length and/or the initial cursor offset. The syntax for the EDIT parameter is:

```
      str ! str:e1 ! str:e1:e2 ! str::e2
```

where str is assumed to be a variable of some string type and e1 and e2 are integer expressions. e1 is the edit length. e2 is the cursor offset relative to the first modifiable character.

The EDIT procedure is a combined output and input procedure. The contents of the string variable given as a parameter are output to the terminal, whereupon the user may edit the string. When the editing is finished, the user strikes the RETURN or the ESC key, whereupon program execution continues with the modified value of the string variable.

If no edit length is given in the EDIT call, the length of the field which the user is allowed to edit is the current dynamic length of the string variable. The dynamic length of the string is not modified by the EDIT procedure.

If an edit length smaller than the current dynamic length of the string variable is given, the dynamic length of the string variable is set to the edit length. The length of the field which the user is allowed to edit will be this new dynamic length, that is, the specified edit length.

If an edit length greater than the current dynamic length of the string variable is given, the dynamic length of the string variable is set to the edit length by adding blank characters to the end of the string. The length of the field which the user is allowed to edit will be this new dynamic length, that is, the specified edit length.

If an edit length greater than the maximum length of the string variable is given, a run time error occurs.

If the edit length is zero, or if the edit length is not specified and the length of the string is zero, no input/output operation takes place.

The value of IORESULT is not affected by EDIT.

3.15 EOF

FUNCTION EOF(F: FILE OF xxx): BOOLEAN;
or
FUNCTION EOF(F: TEXT): BOOLEAN;
or
FUNCTION EOF: BOOLEAN;

Disk files:

When specified with a disk file parameter this function returns TRUE if end-of-file has been reached on that file, otherwise FALSE is returned.

Sequential files have an end-of-file mark written in the file, and EOF becomes true when that mark is read. For direct access files EOF becomes true if an attempt is made to GET a record with a number greater than the uptil now greatest number of a written record.

After a RESET or a REWRITE call for a file F, EOF(F) is false. EOF returns true for a closed file. When EOF(F) is true, EOLN(F) is also true and FÜ is undefined. If EOF(F) becomes true during a GET(F) or a READ(F,...) call, the data thereby obtained may not be valid.

The INPUT file:

EOF(INPUT) returns true if the user terminated the last input operation on the terminal by striking the CTRL-D. EOF(INPUT) returns false if the user terminated the last input operation on the terminal by striking the RETURN key.

EOF without a parameter is equivalent to EOF(INPUT).

The OUTPUT file:

EOF(OUTPUT) returns true if a function key has been pressed. A subsequent call of EOF(OUTPUT) will return FALSE.

3.16 EOLN

```
FUNCTION EOLN(F: FILE OF CHAR): BOOLEAN;
or
FUNCTION EOLN(F: TEXT): BOOLEAN;
```

EOLN indicates the end of a line (record) in a sequential file.

After a RESET or a REWRITE call for a file F, EOLN(F) is true. EOLN returns true for a closed file.

3.17 EXIT

```
PROCEDURE EXIT(P: PROGRAM);
or
PROCEDURE EXIT(P: PROCEDURE);
or
PROCEDURE EXIT(P: FUNCTION);
```

This procedure causes the program to return from the the subroutine or program whose identifier is specified as the parameter.

If the subroutine passed as a parameter to EXIT is recursively called, the most recent invocation of that subroutine will be exited.

EXIT(PROGRAM) is equivalent to EXIT(xxx) where xxx is the program name.

Example:

```
PROGRAM EXITTEST;
VAR I: INTEGER;

  PROCEDURE BETA; FORWARD;

  PROCEDURE ALPHA;
  BEGIN
    WRITELN('ALPHA 1');
    CASE I OF
      1: EXIT(ALPHA);
      2: EXIT(BETA); (* Legal because of the above FORWARD
                                              declaration *)
      3: EXIT(EXITTEST);
      4: EXIT(PROGRAM);
    END;
    WRITELN('ALPHA 2')
  END;

  PROCEDURE BETA;
  BEGIN
    WRITELN('BETA 1');
    ALPHA;
    WRITELN('BETA 2')
  END;

BEGIN
  WRITE('WRITE A NUMBER: ');
  READ(I);
  BETA;
  WRITELN('NORMAL PROGRAM TERMINATION')
END.
```

This program may the be executed in the following ways (user input shown underlined):

```
WRITE A NUMBER: 0
BETA 1
ALPHA 1
ALPHA 2
BETA 2
NORMAL PROGRAM TERMINATION
```

or

```
WRITE A NUMBER: 1
BETA 1
ALPHA 1
BETA 2
NORMAL PROGRAM TERMINATION
```

or

```
WRITE A NUMBER: 2
BETA 1
ALPHA 1
NORMAL PROGRAM TERMINATION
```

or

```
WRITE A NUMBER: 3
BETA 1
ALPHA 1
```

or

```
WRITE A NUMBER: 4
BETA 1
ALPHA 1
```

3.18 EXP

```
FUNCTION EXP(A: REAL): REAL;
```

This function returns the number e (the base of the natural logarithm) raised to the power A, where A is the parameter specified in the EXP call.

Example:
EXP(0)=1 EXP(1)=2.718281828458

3.19 FILLCHAR

```
PROCEDURE FILLCHAR(VAR DESTINATION: PACKED ARRAY (x..y) OF CHAR;
                       LENGTH: INTEGER;
                       CHARACTER: CHAR);
```

This procedure takes a (subscripted) packed array of characters given as the first parameter and fills it with the a number of identical characters. The number of characters inserted is given in the parameter LENGTH, and the character inserted is given in the parameter CHARACTER.

Example:

Assuming that S is of type STRING executing the following statements

```
S:='          ';
FILLCHAR( S(1), LENGTH(S)-1, 'X' );
```

will give S the value 'XXXXXXXXX '.

It is important here that S(1) and not S is specified as the first parameter, for this parameter is not of type STRING but some packed array of characters. If S were given as the first parameter, element number zero (the dynamic length of S) would also receive the value 'X', which, of course, would give a completely wrong dynamic length.

3.20 GET

```
PROCEDURE GET(VAR F: FILE OF xxx);
or
PROCEDURE GET(VAR F: FILE OF xxx; LOCK: BOOLEAN);
or
PROCEDURE GET(VAR F: TEXT);
```

This procedure assigns the value of the next component (the next record in a direct access file) in the file F to the variable FÜ and advances the file position to the next component (record).

If EOF(F) becomes true during the GET call, the value of FÜ is undefined. The effect of GET(F) is defined only if EOF(F) is false before GET is called.

In contrast to what is stated in J&W page 190, a GET(F) is required after a RESET call to assign the first file component to FÜ.

If LOCK is TRUE the record accessed will be locked for other users. This is only relevant for files opened for Unix or selective update. (See REWRITE section 3.50)

3.21 GETENVR

FUNCTION GETENVR(ENVR: xxx; VAR RES: xxx): BOOLEAN;

The type xxx can be STRING or LONGSTRING. The parameter ENVR must specify an environment. If the specified environment exists GETENVR returns TRUE and the value of the environment is placed in RES. Otherwise, if the environment is not found, FALSE is returned and RES has length 0.

Example:
GETENVR('unit',PROGRAMNAME);

3.22 GETOPT

FUNCTION GETOPT(OPT:xxx; VAR RES: xxx): BOOLEAN;

Where xxx can be of type STRING or LONGSTRING. OPT is a string containing the option to be looked for. An option in UNIX is a letter prefixed by '-'.

GETOPT returns TRUE if the option is found in the parameterlist of the program. If OPT is an option followed by ':' GETOPT will check if the option is followed by one or more characters (f.ex. a filename). A single space character between the option and characters is allowed. The characterstring from the first character following the option (but skipping the space character if any) and up to the next space is returned in RES.

example:
consider a program, prog, which is called as follows:

$ prog -i name1 -o name2

where name1 following the i-option could specify a source file and name2 following the o-option could specify a destination.

In the Pascal program prog.p the parameters are fetched by:

```
GETOPT('i:',SRCNAME)  and  GETOPT('o:',DSTNAME)
```

where SRCNAME and DSTNAME are of type STRING or LONGSTRING.

Note that ':' in OPT does not mean that the option must be followed by a string (parameter); if no parameter is given, the empty string will be returned in RES.

This implies that the following construction often is necessary:

```
IF GETOPT('i:',NAME) AND (NAME <> '') THEN
```

3.23 GOTOXY

PROCEDURE GOTOXY(COLUMN, LINE: INTEGER);

This procedure moves the cursor on the terminal to the position specified by the parameters.

The reaction of the terminal to cursor coordinates outside the terminal screen depends on the terminal type.

Examples:

GOTOXY(1,1) will place the cursor in the upper left corner.

GOTOXY(80,24) will place the cursor in the lower right corner on most terminals, namely those having 24 lines of 80 characters.

3.24 IORESULT

FUNCTION IORESULT: INTEGER;

After any I/O operation, except EDIT, IORESULT returns the resulting file system error code. The meaning of these error codes is explained in Supermax System Operation Guide, appendix A. The meaning of the value of IORESULT is special for the routine NEWEDIT (section 3.37).

In some cases IORESULT can return a negative value:

1) -2 is returned if the I/O operation failed because of end-of-file.

2) -1 is returned if an attempt to GET a record from a file is made <u>after</u> end-of-file has become TRUE.

The user should inspect the value of IORESULT after every I/O operation. However, in most cases, the Pascal system performs this check automatically and aborts program execution with an error message if IORESULT>0.

In the following cases no automatic checking of IORESULT is performed and thus <u>must</u> be done by the programmer:

1) After a RESET or REWRITE call.

2) After a CHAIN call.

3) After a call of LOCK with the WAIT parameter set to FALSE.

4) If the compiler directive (*$C-*) has been specified. (See Supermax Running Pascal Assembler or Supermax Running Interpreted Pascal).

Program behavior is unpredictable if the value of IORESULT is not inspected by the program in these cases.

Note:
The following is an instance of a typical programming error:

```
RESET(F,FILENAME);
IF IORESULT<>0 THEN
  WRITELN('FILE SYSTEM ERROR ',IORESULT);
```

In case of an error, the following text will be output:

```
FILE SYSTEM ERROR 0
```

This may look strange, because it is clearly tested that IORESULT<>0. However, IORESULT always returns information about the latest I/O operation, and at the time when IORESULT is written, the latest I/O operation is the writing of the string 'FILE SYSTEM ERROR '. IORESULT is therefore zero, because the writing of the string caused no error.

The following statements will work correctly (IOVAL is assumed to be of type INTEGER):

```
RESET(F,FILENAME);
IF IORESULT<>0 THEN
  BEGIN
    IOVAL:=IORESULT;
    WRITELN('FILE SYSTEM ERROR ',IOVAL)
  END;
```

3.25 ISLETTER

```
FUNCTION ISLETTER(C: CHAR): BOOLEAN;
```

The function checks whether or not the parameter C is a letter (counting all the special national letters as letters, too) and returns TRUE is so and otherwise FALSE.

3.26 LENGTH

```
FUNCTION LENGTH(STR: xxx): INTEGER;
```

Where xxx is of type STRING or LONGSTRING. This function returns the current dynamic length of the string given as its parameter.

Example:

```
PROGRAM LENGTHTEST;
VAR S: STRING(10);
BEGIN
  S:='ABC';
  WRITELN(LENGTH(S))
END.
```

will output

3

3.27 LOADTEXT

PROCEDURE LOADTEXT(NAME: xxx);

Where xxx is of type STRING or LONGSTRING. LOADTEXT uses the NAME parameter and the value of the environments LANGUAGE and NLSPATH to determine which file is to be used as a text library. This file is loaded, and the lines in it can be indexed using the subroutine STDTEXT (see section 3.63).

3.28 LN

FUNCTION LN(A: REAL): REAL;

This function returns the natural logarithm of the number given as its parameter.

3.29 LOCK and UNLOCK

PROCEDURE LOCK(F: FILE OF XXX; RECNO: INTEGER; WAIT: BOOLEAN);

PROCEDURE UNLOCK(F: FILE OF XXX; RECNO: INTEGER);

LOCK locks the record in F specified by RECNO; if the record is already locked WAIT specifies whether or not the process should suspend itself until it is possible to access the record and lock it.

UNLOCK unlocks the record in F specified by RECNO.

No I/O checking will be done if WAIT is FALSE and an attempt to lock an already locked record is made, but IORESULT is affected and the user should check this value.

3.30 LONG

FUNCTION LONG(I: INTEGER): LONGINT;

This function returns the longint value of the integer parameter.

3.31 MARK and RELEASE

PROCEDURE MARK(VAR HEAP: ^INTEGER);

PROCEDURE RELEASE(HEAP: ^INTEGER);

The standard procedure DISPOSE defined on page 191 of J&W is not implemented in Supermax Pascal. However, the function of DISPOSE can be approximated by a combined use of the Supermax Pascal procedures MARK and RELEASE. The process of recovering memory space as described below is only an approximation to the function of DISPOSE in that one cannot explicitly ask that the storage occupied by one particular variable be released by the system.

Supermax Pascal allocates storage for variables created by use of the standard procedure NEW in a stack-like structure called the 'heap'. The following program is a simple demonstration of how MARK and RELEASE can be used to cause changes in the size of the heap:

```
PROGRAM SMALLHEAP
TYPE PERSON = RECORD
                NAME: STRING;
                ID: INTEGER
              END;
VAR P: ^PERSON;
    HEAP: ^INTEGER;

BEGIN
  MARK( HEAP );
  NEW( P );          (* ALLOCATE RECORD *)
  P^.NAME := 'FINKELSTEIN, SAM';
  P^.ID   := 999;
  RELEASE( HEAP ) (* RELEASE SPACE OCCUPIED BY RECORD *)
END.
```

The above program first calls MARK to place the address of the current top-of-heap into the variable HEAP given as its parameter. This parameter supplied to MARK must be a pointer variable, but need not be declared to be a pointer to an INTEGER as is traditional.

Next, the program calls the standard procedure NEW and this results in a new variable P^ which is located in the heap as shown in the diagram below:

```
NEW TOP OF HEAP  -->   :---------------:
                       :               :
                       :       P^      :
                       :               :
                       :---------------:  <-- OLD TOP OF HEAP
                       : contents of   :
                       : heap at start :
                       : of program    :
                       :               :
```

Once the program no longer needs the variable P^ and wishes to release this memory space to the system for other uses, it calls RELEASE which resets the top-of-heap to the address contained in the parameter.

If the above program had done a series of calls to the standard procedure NEW between the calls to MARK and RELEASE, the effect would have been that the storage occupied by several variables would have been released at once. Also note that due to the stack nature of the heap it it not possible to release the memory space used by a single item in the middle of the heap.

It should be noted that careless use of the procedures MARK and RELEASE can lead to pointers which point to areas of memory which are no longer a part of the defined heap space.

3.32 MAXINT

CONST MAXINT=32767;

Although this is not a subroutine it has been included in this chapter because it is logically associated with the other pre-declared identifiers.

MAXINT is the highest value that an integer variable can contain. This value is represented by 15 binary 1's.

3.33 MEMAVAIL

FUNCTION MEMAVAIL: INTEGER; (Interpreted Pascal)

FUNCTION MEMAVAIL: LONGINT; (Pascal Assembler)

This function returns the number of 16-bit words available in the heap. (See the manuals Supermax Running Pascal Assembler and Supermax Running Interpreted Pascal)

A run time error occurs when trying to get more heap space than available.

3.34 METAMORPH.

PROCEDURE METAMORPH(PROGNAME, PARM: xxxx);

The parameters can be of type STRING and LONGSTRING. The program in which the procedure is called is substituted by the Pascal program specified by PROGNAME and PROGNAME is called with the parameters specified by PARM. The differences between the CHAIN call and the METAMORPH call are that by using METAMORPH a new process is not created and if an error occurs during the metamorphosis, control will not be returned to the calling program, and that only Pascal programs can be started by the call.

METAMORPH is thus useful when one wants to change to another program during the same process.

Note: files opened in the program calling METAMORPH (either by REWRITE or RESET) are not closed when PROGNAME is started. The user should assure that she has closed the files explicitly.

In the Pascal Assembler system METAMORPH is not restricted to starting Pascal programs.

3.35 MOVELEFT

```
PROCEDURE MOVELEFT(     SOURCE:       PACKED ARRAY (...) OF CHAR;
                    VAR DESTINATION: PACKED ARRAY (...) OF CHAR;
                        LENGTH:       INTEGER);
```

This procedure copies the number of bytes specified in the parameter LENGTH from the character array SOURCE to the character array DESTINATION. These arrays may be indexed to specify the first (leftmost) element taking part in the move operation.

The move is destructive if the character arrays overlap.

MOVELEFT moves the leftmost character (lowest index) first.

Examples:

```
PROGRAM MOVETEST;
VAR S: STRING;
BEGIN
  S:='ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  MOVELEFT( S(2), S(1), LENGTH(S)-1 );
  WRITELN(S);
END.
```

will output

```
BCDEFGHIJKLMNOPQRSTUVWXYZZ
```

whereas

```
PROGRAM MOVETEST;
VAR S: STRING;
BEGIN
  S:='ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  MOVELEFT( S(1), S(2), LENGTH(S)-1 );
  WRITELN(S);
END.
```

will output

```
AAAAAAAAAAAAAAAAAAAAAAAAAA
```

because S(1) is moved into S(2), whereupon S(2) is moved into S(3), etc.

Note that S(1) is specified instead of S. This is done because element number zero of S, which contains the dynamic length of S, must not take part in the move operation.

3.36 MOVERIGHT

```
PROCEDURE MOVERIGHT(     SOURCE:      PACKED ARRAY (...) OF CHAR;
                     VAR DESTINATION: PACKED ARRAY (...) OF CHAR;
                         LENGTH:      INTEGER);
```

This procedure copies the number of bytes specified in the parameter LENGTH from the character array SOURCE to the character array

DESTINATION. These arrays may be indexed to specify the first (left-most) element taking part in the move operation.

The move is destructive if the character arrays overlap.

MOVERIGHT moves the rightmost character (highest index) first.

Examples:

```
PROGRAM MOVETEST;
VAR S: STRING;
BEGIN
  S:='ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  MOVERIGHT( S(1), S(2), LENGTH(S)-1 );
  WRITELN(S);
END.
```

will output
AABCDEFGHIJKLMNOPQRSTUVWXY
whereas

```
PROGRAM MOVETEST;
VAR S: STRING;
BEGIN
  S:='ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  MOVERIGHT( S(2), S(1), LENGTH(S)-1 );
  WRITELN(S);
END.
```

will output
ZZZZZZZZZZZZZZZZZZZZZZZZZZ

because S(26) is moved into S(25), whereupon S(25) is moved into S(24), etc.

Note that S(1) is specified instead of S. This is done because element number zero of S, which contains the dynamic length of S, must not take part in the move operation.

3.37 NEW

```
PROCEDURE NEW(VAR P: ^xxxx);
or
PROCEDURE NEW(VAR P: ^xxxx; T1, T2, ...);
```

This procedure allocates a new variable of the type xxxx on the system heap and assigns the address of this variable to the pointer variable given as parameter to the NEW procedure. If the type of the allocated variable is a record with variants the second form of the NEW procedure may be used with T1, T2, etc. specifying the values of the fields controlling the variant parts of the record.

The two forms of the NEW procedure work identically; the second form is allowed to provide compatibility with standard Pascal.

A run time error occurs when trying to get more heap space than available. (See Supermax Running Pascal Assembler and Supermax Running Interpreted Pascal)

3.38 NEWEDIT

```
PROCEDURE NEWEDIT(VAR STR: STRING; VAR CURPOS: INTEGER;
                  LINLGD: INTEGER;
                  NOWRIT, ORUN, TYPAHD: BOOLEAN);
or
PROCEDURE NEWEDIT(VAR STR: LONGSTRING; VAR CURPOS: INTEGER;
                  LINLGD: INTEGER;
                  NOWRIT, ORUN, TYPAHD: BOOLEAN);
```

This procedure is the Pascal version of the system call medit (see SMOS User's Manual part 2).

It is possible to edit a typed text consisting of several lines using one call of NEWEDIT per line. The characters are collected in a buffer as not to loose characters between the calls of NEWEDIT. The buffer contains non echoed characters; the buffered characters are echoed when they are passed to a call of NEWEDIT.

The parameters are:

STR: the string that is to be edited

OFFSET: the position of the cursor at the beginning and end of the NEWEDIT call (relative to the first character) LINLGD: the edit field length

NOWRIT: FALSE: the contents of the string STR are written before the editing takes place
TRUE: the contents of the string STR are not written before the editing takes place

ORUN: TRUE: the editing is automatically stopped if overflow occurs
FALSE: overflow can not occur. The SIOC ignores the key giving overflow but reports the error by the bell.

TYPAHD: TRUE: the characters collected in the input buffer after last clearing/NEWEDIT call are used as the first characters i the next NEWEDIT call.
FALSE: the buffer is cleared before the editing takes place

The buffer can be cleared by a call of NEWEDIT with TYPAHD FALSE. The editing terminates when a function key that is not supported by the SIOC is pressed or if overflow occurs and ORUN is TRUE.

IORESULT is affected by NEWEDIT. The value of IORESULT can be:

0 : the call terminated without error.
>0: the value of the functionkey terminating the editing
or
the value of the character/functionkey typed when the cursor is placed to the right of the current line (overflow)
or
the value of the functionkey pressed when the cursor is at the beginning of the current line

3.39 ODD

```
FUNCTION ODD(I: INTEGER): BOOLEAN;
or
FUNCTION ODD(I: LONGINT): BOOLEAN;
```

This function returns TRUE if the value of the parameter is odd, FALSE if the value of the parameter is even.

3.40 ORD

FUNCTION ORD(C: xxxx): INTEGER;

xxxx is INTEGER, CHAR, BOOLEAN, or a user-defined scalar type.

If xxxx is INTEGER ORD simply returns the value of the parameter.

If xxxx is CHAR, ORD returns the integer ASCII value of the character given as parameter.

If xxxx is BOOLEAN, ORD returns zero for FALSE and one for TRUE.

If xxxx is a user-defined scalar type, ORD returns the position of the value of the parameter in the value enumeration, the numbering starting with zero.

Examples:
ORD('A')=65 ORD(17)=17

Given the declaration

```
VAR GREEK: (ALPHA, BETA, GAMMA, DELTA, EPSILON);
```

the statements

```
GREEK:=ALPHA;
WRITELN( ORD(GREEK) );
GREEK:=EPSILON;
WRITELN( ORD(GREEK) );
```

will output

```
0
4
```

3.41 PAGE

```
PROCEDURE PAGE(VAR F: FILE OF CHAR);
or
PROCEDURE PAGE(VAR F: TEXT);
```

This procedure outputs the string '\05ES>' to the specified file. When this string is sent to a printer, the paper will be advanced to the next page. Thus if the list device is a printer, PAGE(LIST) will cause the printer to advance the paper to the next page. If the list device is a terminal, PAGE(LIST) has no effect.

3.42 POS

```
FUNCTION POS(PATTERN, SOURCE: xxxx): INTEGER;
```

The type xxxx can be STRING or LONGSTRING. This function returns the position of the first occurrence in the parameter SOURCE of the pattern in the parameter PATTERN. The value returned is the index of the first character in the matched pattern.

Zero will be returned, if the pattern is not found.

Example:

```
S:='THIS IS VERY, VERY SMART';
WRITELN( POS( 'VERY', S ) );
WRITELN( POS( 'very', S ) );
```

will print

```
9
0
```

3.43 PRED

```
FUNCTION PRED(C: xxxx): xxxx;
```

xxxx is INTEGER, LONGINT, CHAR, BOOLEAN, or any user-defined scalar type.

PRED returns the value which immediately precedes the parameter in the enumeration of values of type xxxx.

Examples:

```
PRED('B')='A'        PRED(TRUE)=FALSE        PRED(17)=16
```

Given the declaration

```
TYPE T=(UN, DEUX, TROIS, QUATRE, CINQ);
```

we have

```
PRED(TROIS)=DEUX
```

3.44 PUT

```
PROCEDURE PUT(VAR F: FILE OF xxx);
or
PROCEDURE PUT(VAR F: FILE OF xxx; UNLOCK: BOOLEAN);
or
PROCEDURE PUT(VAR F: TEXT);
```

This procedure outputs the contents of the buffer variable F^ to the file F. The file position is moved to the next component of the file (the next record of a direct access file). The file is extended automatically, if required.

EOF(F) need not be true before PUT is called, and the value of F^ is not changed by the PUT operation.

If UNLOCK is TRUE the record will be unlocked, otherwise if the record is locked and UNLOCK is FALSE, the record will remain locked for other users. This is relevant only for files opened UNIX update mode or selective update mode.

3.45 PWROFTEN

FUNCTION PWROFTEN(EXPONENT: INTEGER): REAL;

This function returns the value of 10 raised to the power of the parameter. The parameter must lie in the range 0..126.

Example:

PWROFTEN(32)=1E32

3.46 READ

PROCEDURE READ(VAR F: FILE OF CHAR; VAR A, B, C, ...);
or
PROCEDURE READ(VAR F: TEXT; VAR A, B, C, ...);
or
PROCEDURE READ(VAR A, B, C, ...);

This procedure will read data in ASCII format from the input buffer associated with the file F and assign the data to the parameters. New records are read from the file into the input buffer if required.

If a file is not specified, INPUT is assumed.

READ may have any number of parameters. The parameters may have the type INTEGER, LONGINT, REAL, CHAR, STRING, LONGSTRING or CSTRING.

If a parameter is of type STRING, LONGSTRING or CSTRING the rest of the input buffer is assigned to this parameter and its length is adjusted accordingly.

Example:

READ(S), where S is a string variable, will most often result in the empty string (a string of length zero) being assigned to S and no input operation takes place. This is due to the special handling of string input: The remainder of the input buffer is assigned to the string variable; if the buffer is empty, then this empty string is assigned to S.

The following code should normally be used for string input:

```
READLN;     (* Force reading of data into the input buffer *)
READ(S);    (* Assign the contents of the input buffer to S *)
```

For further examples, see the READLN procedure (section 3.47).

3.47 READLN

```
PROCEDURE READLN(VAR F: FILE OF CHAR; A, B, C, ...);
or
PROCEDURE READLN(VAR F: TEXT; A, B, C, ...);
or
PROCEDURE READLN(A, B, C, ...);
```

This procedure works as the READ procedure, except that after all input has been done, a new record (line) is read into the input buffer for use in later READ or READLN calls.

READLN is often used without parameters, or with only the file parameter to force input of a record for later use.

Example:

If I is an integer, the call READLN(I) will have the following effect: Lines are read from the terminal until a non-blank character is encountered This is then interpreted as an integer, which is assigned to I. After this, another line is read from the terminal (!), that is, the user must enter an additional (possibly empty) line followed by a RETURN. The contents of this line will be used in subsequent READ or READLN statements.

The statements

```
READ(I);
...
READ(J);
...
READ(K);
```

will, if given the input line '1 2 3', assign 1 to I, 2 to J, and 3 to K. This may be annoying, because there may be many statements between each READ call and we may want to skip the rest of each previous input line before reading a new integer. This may be done thus:

```
READ(I);
...
READLN;  (* Force input of a new line *)
READ(J);
...
READLN;  (* Force input of a new line *)
READ(K);
```

Often it is desired to have the user enter simply a RETURN as a confirmation of something. A READLN call without parameters will do this job.

3.48 RELEASE

```
PROCEDURE RELEASE(HEAP: ^INTEGER);
```

This procedure is described with the MARK procedure in section 3.31.

3.49 RESET

```
PROCEDURE RESET(VAR F: FILE OF xxx; FILENAME: STRING);
or
PROCEDURE RESET(VAR F: TEXT; FILENAME: STRING);

PROCEDURE RESET(VAR F: FILE OF xxx; FILENAME: LONGSTRING);
or
PROCEDURE RESET(VAR F: TEXT; FILENAME: LONGSTRING);
```

This procedure opens an existing file for reading only. Other users on the computer are allowed to open the same file for reading only, but not for writing. The file is positioned to the first record, but contrary to J&W page 190, the first record is not read.

The first parameter must not be associated with an open file, when the call is made.

The FILENAME parameter is a Supermax file name possibly followed by a colon and one or more indicators of parameters. The format of these indicators is described in section 3.50 (REWRITE). The only relevant indicators for the RESET call are :f followed by a size in bytes indicating the size of the filebuffer used when opening the file and :m followed by an opening mode, 0 or 1. Other indicators used in REWRITE will be ignored by the RESET call.

Mode 0 opens the file in READ mode, that is: open for reading only, and, if the iounit is a file, reserve it non-exclusively, that is, other READ-opens will be allowed, but no other opens of the file are allowed.

Mode 1 opens the file the file in O_RDONLY mode, that is: open with no reservation.

FILENAME can specify a file, box, disc, terminal etc.

Note that no automatic checking of the value of IORESULT is performed by the system after a RESET operation. This must be done by the user.

Example:

RESET(F,'/dev/box/buf') will try to open the box named buf.

RESET(F,'/usr/abc/test.p') will try to open the file test.p in the directory /usr/abc.

RESET(F,'/usr/abc/file1:f4096') will try to open the file named file1 in the directory /usr/abc with a filebuffer of size 4K.

The default buffersize is 2K = 2048 bytes.

3.50 REWRITE

```
PROCEDURE REWRITE(VAR F: FILE OF xxx; FILENAME: STRING);
or
PROCEDURE REWRITE(VAR F: TEXT; FILENAME: STRING);
or
PROCEDURE REWRITE(VAR F: FILE OF xxx; FILENAME: LONGSTRING);
or
PROCEDURE REWRITE(VAR F: TEXT; FILENAME: LONGSTRING);
```

This procedure opens a file for reading and writing. The file is positioned to the first record. EOF(F) is set to false.

The first parameter must not be associated with an open file, when the call is made.

Note that no automatic checking of the value of IORESULT is performed by the system after a REWRITE operation. This must be done by the user.

The FILENAME parameter is a Supermax file name possibly followed by a colon and one or more indicators of parameters. FILENAME can specify a file, box, disk, terminal etc.

The indicators are:

- :s followed by the size of the file in bytes
- :a followed by the access rights in octal notation
- :f followed by the filebuffersize in bytes
- :m followed by 0, 1, or 6 denoting the mode in which the file should be opened. The modes are:

  0: UPDATE
  1: O_RDWR
  6: O_WRONLY

  (the modes are described below)

- :x followed by one or more of the letters a, e, and n.
  (these flags are described below)

If the file exists it will be opened, otherwise it will be created if and only if :s followed by a non-negative size is included in the FILENAME parameter (:s0 can be used for superfile).

The default accessrights are 764, i.e. rwxrw-r--, but this depends on the user's iounit creation mask.

The default filebuffersize is 2K = 2048 bytes.

The default mode is UPDATE.

Modes:

UPDATE: open for reading and writing, and, if the iounit is a file, reserve it exclusively, that is, no other opens of the file are permitted.

O_RDWR: open for reading and writing with no reservation.

O_WRONLY: open for writing with no reservation.

Flags:

a: (O_APPEND) if set, the file-pointer will be set to the end of the file prior to each write.

e: (O_EXCL) if set, the open will fail if the file exists.

n: (O__NDELAY) this flag may affect subsequent reads and writes, for further information please refer to Supermax Operating System, reference manual section 2-3 on open, read, and write).

The order of the :s, :a, :f, :m, and :x parameters is arbitrary and not used indicators must be left out.

Example:

REWRITE(F,'/dev/box/buf') will open the box named buf if it exists. Otherwise an I/O error will occur.

REWRITE(F,'/usr/abc/test.p:m2:a777:s0') will open the file test.p in /usr/abc (if it exists) for selective update otherwise the file will be created with access rights rwxrwxrwx.

3.51 ROUND and ROUNDL

```
FUNCTION ROUND(A: REAL): INTEGER;
or
FUNCTION ROUNDL(A: REAL): LONGINT;
```

This function returns the integer/longint value closest to the parameter value.

A run time error occurs if the parameter lies outside the range of integer/longint values.

Example:

```
ROUND(1.6)=2    ROUND(1.5)=2    ROUND(1.4)=1

ROUNDL(35054.6)=35055   ROUNDL(35054.4)=35054
```

3.52 SCAN

```
FUNCTION SCAN(LENGTH: INTEGER;
              partial expression;
              ARR:    PACKED ARRAY (...) OF CHAR): INTEGER;
```

This function scans the character array given as the parameter ARR until one of the following conditions are met:

1) The 'partial expression' is satisfied by a character in the array.

2) The number of characters specified in the parameter LENGTH have been scanned.

The value returned by SCAN is the number of characters that were scanned before one of the conditions was met: Zero is returned if the first character meets condition 1 above, one is returned if the second character meets condition 1 above, ..., LENGTH-1 is returned if the last character scanned meets condition 1 above, LENGTH is returned if none of the scanned characters meets condition 1 above.

The packed array of characters may be indexed to indicate a starting point for the scanning.

The array is scanned from lower to higher index values if LENGTH is positive. The array is scanned from higher to lower index values if LENGTH is negative, in which case the number of characters scanned is the absolute value of LENGTH, and the value returned by SCAN is also negative.

The 'partial expression' given as the second parameter must consist of a <> or an = followed by a character expression.

Examples:

Assuming the declaration

```
VAR S: STRING;
```

and the assignment

```
S:='0000123456789ABCDEFGHI'
```

we have

```
SCAN( -10,  =' ', S(17) ) =-10
SCAN( 100, <>'0', S(1)  ) =4
SCAN( 100, <>'0', S(2)  ) =3
SCAN( 100,  ='0', S(1)  ) =0
SCAN(  15,  ='9', S(1)  ) =12
SCAN(  -7,  ='C', S(20) ) =-4
```

Note that S(1) is specified instead of S because element number zero, the length indicator, should not take part in the scanning.

3.53 SEEK

PROCEDURE SEEK(F: FILE OF xxx; RECORDNUMBER: yyy);

Where yyy is of type INTEGER or LONGINT. This procedure is used to position the file F, which must be a direct access file, to the record whose number is given in the parameter RECORDNUMBER. The next GET or PUT from/to the file will read/write this record. Records in files are numbered 1, 2,... etc.

Example:

```
SEEK(F,12); (* Position file to record number 12 *)
GET(F);     (* Read record 12,position file to record number 13*)
....        (* Update F^ *)
SEEK(F,12); (* Position file to record number 12 *)
PUT(F);     (* Write record 12,position file to record number 13*)
```

3.54 SETIORESULT

PROCEDURE SETIORESULT(I: INTEGER);

This procedure sets the value of the IORESULT function (see section 3.24) to the value of the parameter.

This procedure provides a convenient way of communicating I/O result codes between user-written I/O procedures and application programs.

3.55 SETLENGTH

PROCEDURE SETLENGTH(VAR STR: xxx; NEWLENGTH: INTEGER);

Where xxx is of type STRING or LONGSTRING. This procedure assigns the value of NEWLENGTH to the byte (bytes) of the specified string(long-string) where then dynamic length of the string(longstring) is saved.

A run-time error will occur if an attempt to set the length of a string(longstring) to more than the declared length is made.

Note: special care should be taken when SETLENGTH is used; the user must be sure that the length specified by NEWLENGTH is a correct dynamic length for the string parameter.

3.56 SHORT

FUNCTION SHORT(L: LONGINT): INTEGER;

This function returns the integer value of the longint parameter. If the longint value exceed the range of the integer a run-time error occur.

3.57 SHORTSTRING

PROCEDURE SHORTSTRING(VAR S: STRING; L: LONGSTRING);

If the length of the long string specified by L is less than og equal to the declared length of the string specified by S, L is converted to a string and placed in S.

A run-time error will occur if the length of L is greater than the declared length of S.

3.58 SIN

FUNCTION SIN(A: REAL): REAL;

This function returns the sine of the value of the parameter. The parameter is given in radians.

3.59 SIZEOF

FUNCTION SIZEOF(xxxx): INTEGER;

xxxx is either a variable or a type identifier. The function returns the number of bytes of storage occupied by the specified variable or by items of the specified type.

Example:

SIZEOF(INTEGER)=2

Note that when SIZEOF is used on a file, the recordsize + 8 is returned.

3.60 SQR

```
FUNCTION SQR(A: INTEGER): INTEGER;
or
FUNCTION SQR(A: LONGINT): LONGINT;
or
FUNCTION SQR(A: REAL): REAL;
```

This function returns the square of the value of the parameter.

Examples:

SQR(3)=9

Note that SQR(1000) returns the value 16960 because an integer parameter is given and an integer overflow occurs, whereas SQR(1000.0) returns the real value 1000000 correctly because a real parameter is given.

3.61 SQRT

```
FUNCTION SQRT(A: REAL): REAL;
```

This function returns the square root of the value of the parameter.

Example:

SQRT(9)=3

3.62 STACKAVAIL

FUNCTION STACKAVAIL: INTEGER; (only Interpreted Pascal)

This function returns the number of words available on the stack. NOTE: by the term STACK in Interpreted Pascal is meant the socalled Pascal Stack (i.e. segment 5).

3.63 STDTEXT

FUNCTION STDTEXT(INDEX: INTEGER): LONGSTRING;

This function returns the textline indexed by INDEX in the text loaded by LOADTEXT (see section 3.27). A runtime error will occur, if STDTEXT is used, when no text has been loaded.

Example:

```
.
.
LOADTEXT('editor');
IF IORESULT <> 0 THEN WRITELN(STDTEXT(10));
.
.
```

will output the textline indexed by 10.

3.64 STRCAT

PROCEDURE STRCAT(VAR L: LONGSTRING; STR: xxx);

Where xxx is of type STRING or LONGSTRING. This procedure takes the LONGSTRING specified by L and the STRING or LONGSTRING specified by STR, concatenates them and returns the resulting LONGSTRING in L. STRCAT is thus a pendant to the function CONCAT.

Example:

```
VAR TEXT: LONGSTRING; TEXT1: STRING;

TEXT := 'WE HOLD';
TEXT1 := 'THESE TRUTHS TO BE ';
STRCAT(TEXT,TEXT1);
STRCAT(TEXT,'SELF EVIDENT');
WRITELN( TEXT );
will print
WE HOLD THESE TRUTHS TO BE SELF EVIDENT
```

A run time error will occur if the length of the concatenation is greater than the declared length of L.

3.65 SUCC

```
FUNCTION SUCC(C: xxxx): xxxx;
```

xxxx is INTEGER, LONGINT, CHAR, BOOLEAN, or any user-defined scalar type.

SUCC returns the value wich immediately follows the parameter in the enumeration of values of type xxxx.

Examples:

```
SUCC('B')='C'      SUCC(FALSE)=TRUE      SUCC(17)=18
```

Given the declaration

```
TYPE T=(UN, DEUX, TROIS, QUATRE, CINQ);
```

we have

```
SUCC(TROIS)=QUATRE
```

3.66 TAN

FUNCTION TAN(A: REAL): REAL;

This function returns the tangent of the value of the parameter. The parameter is given in radians.

3.67 TIME

FUNCTION TIME: INTEGER;

This function returns the current value of the system clock in centi-seconds (10 millisecond units) as an unsigned 16-bit integer. The resolution the time will, however, generally be coarser than 10 milli-seconds, normally 40 milliseconds. Clock overflow is ignored. The aboslute value of the returned integer is without significance; however, the difference between the integers returned by two subsequent calls of TIME may be used as a measure of the time elapsed between the two calls provided that this time does not exceed 65535 centiseconds (10 minutes and 55.35 seconds).

Example:

Assuming that I is an integer:

```
I:=TIME;
....         (* Perform the operations to be timed *)
I:=TIME-I;   (* Calculate the time difference *)
WRITELN( 'Time spent: ', I/100, ' seconds.' );
```

3.68 TRUNC and TRUNCL

FUNCTION TRUNC(A: REAL): INTEGER;
or
FUNCTION TRUNCL(A: REAL): LONGINT;

This function converts the parameter into an integer/longint by discarding the fraction part of the number. A run time error occurs if

the parameter lies outside the range of integer/longint values.

Example:

TRUNC(1.9)=1   TRUNCL(34501.9)=34501

3.69 UNLOCK

PROCEDURE UNLOCK(F: FILE OF xxx; RECNO: INTEGER);

UNLOCK unlocks the record in F specified by RECNO.

3.70 WRITE

PROCEDURE WRITE(VAR F: FILE OF xxx; A, B, C, ...);
or
PROCEDURE WRITE(VAR F: TEXT; A, B, C, ...);
or
PROCEDURE WRITE(A, B, C, ...);

This procedure is used to write integers, reals, characters, and strings onto sequential files in ASCII format.

A run time error occurs if an attempt is made to output more than 255 characters.

The specified file may be a disk file, the ERROR standard file, or the OUTPUT standard file. OUTPUT is assumed if no file is specified.

When writing to a disk file WRITE simply adds the items to the current output buffer, but the actual writing does not take place before a WRITELN call is executed (see section 3.71).

When writing to the OUTPUT, or ERROR standard files, the output is performed, leaving the cursor/printer head immediately after the last character printed.

For a description of the parameters, please refer to section 3.71.

3.71 WRITELN

```
PROCEDURE WRITELN(VAR F: FILE OF xxx; A, B, C, ...);
or
PROCEDURE WRITELN(VAR F: TEXT; A, B, C, ...);
or
PROCEDURE WRITELN(A, B, C, ...);
```

This procedure works exactly as WRITE with the following exception:

If the specified file is a disk file, the output buffer is written as a new record to the file at the end of the WRITELN operation. If the output buffer is empty when WRITELN is called, a record of length one containing a space character is written to the file.

If the specified file is OUTPUT, or ERROR the cursor/print head moves to the beginning of the next line at the end of the WRITELN operation.

A WRITELN call with only a file specification or without parameters altogether simply consitutes the writing of the output buffer onto the file or the moving of the cursor or print head to the beginning of the next line.

The parameters to be written by WRITE or WRITELN may be expressions of the type INTEGER, LONGINT, REAL, CHAR, or some STRING, LONGSTRING, CSTRING, or PACKED ARRAY OF CHAR type. These expressions may, optionally, be followed by a colon and an integer expression specifying a field width, which may again be followed by a colon and a field specification if the item to be output is of type real.

Integers:

An integer without a field width specification is output in a field of the exact length required to write that integer.

An integer with a field width specification greater than or equal to the number characters required to write that integer is output right-justified in a field of the specified length.

An integer with a field width specification smaller than the number of characters required to write that integer is output in a field of the exact length required to write that integer.

If the field length specifier of an integer is greater than or equal to 100, the number is output in hexadecimal representation in a field being 100 less than the specified field.

Example:

```
WRITELN(1);
WRITELN(22);
WRITELN(333);
WRITELN(4444);
WRITELN(-1);
WRITELN(-22);
WRITELN(-333);
WRITELN(-4444);
```

will output

```
1
22
333
4444
-1
-22
-333
-4444
```

whereas

```
WRITELN(1:3);
WRITELN(22:3);
WRITELN(333:3);
WRITELN(4444:3);
WRITELN(-1:3);
WRITELN(-22:3);
WRITELN(-333:3);
WRITELN(-4444:3);
```

will output

```
  1
 22
333
4444
 -1
-22
-333
-4444
```

Reals

A real without a field specification or with one field specification will, regardless of the field specification, be output in a field of length 20 in the following format:

-9.999999999999E-123

A real followed by two field specifications is, if possible, output in a field of the length specified by the first field width specification and with the number of fraction digits specified by the second field specification. The number is rounded to the desired number of decimals.

One character will always be reserved in front of the number for the sign. If the specified field is too small, the field is expanded as required. If the value to be output is greater than 1E13 the standard format given above is chosen.

If the fraction specification is 0, the standard output format is chosen.

If the fraction specification is -1, the number is output as an integer followed by a decimal point.

If the fraction specification is -2, the number is output as an integer.

Examples:

```
WRITELN(123.866);
WRITELN(123.866:10);
WRITELN(123.866:10:2);
WRITELN(123.866:10:0);
WRITELN(123.866:10:-1);
WRITELN(123.866:10:-2);
WRITELN(123.866:2:2);
```

will output

```
 1.238660000000E+02
 1.238660000000E+02
    123.87
 1.234560000000E+02
      124.
       124
 123.87
```

Characters:

A character without a field width is output in a field of length one. A character with a field width is output right-justified in a field of the specified length.

Examples:

```
WRITELN('C');
WRITELN('C':3);
```

will output

```
C
  C
```

Strings (packed arrays of characters):

A string or packed array of characters without a field width is output in a field with a length equal to the dynamic length of the string or

the length of the array.

A string or packed array of characters with a field width greater than the dynamic length of the string or the length of the array is output right-justified in a field of the specified length.

If a string or packed array of characters is given with a field width smaller than the dynamic length of the string or the length of the array, only so many characters are output as are specified in the field width.

Examples:

```
WRITELN('THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG');
WRITELN('THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG':50);
WRITELN('THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG':30);
```

will output

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
       THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
THE QUICK BROWN FOX JUMPS OVER
```

4. Pascal Pre-processor.

/pbin/ppp is the Pascal language pre-processor which is invoked as the first program by the compilation control programs /pbin/ppc and /pbin/pac.

ppp takes as input pascal source code that can include pre-processor directives and outputs pascal source code acceptable to the Pascal compilers.

Directives for ppp are recognized by having the character # in position one on a line. The syntax of these lines is independent of the Pascal syntax and they can be placed anywhere in the source text.

Pre-processor directives can be used to acheive:

textual replacement (macroes)
conditional compilation
inclusion of files

Especially the substitution of "one line procedures" with a macro may speed up the resulting loadmodules run time, as calling procedures written in Pascal is time consuming.

4.1 Textual replacement.

There are two kinds of textual replacement:

#define IDENTIFIER TOKEN-STRING

#define IDENTIFIER(IDENTIFIER,...,IDENTIFIER) TOKEN-STRING

If the first kind is used the pre-processor will replace all occurrences of the given IDENTIFIER by the given TOKEN-STRING.

The other kind is a macro-definition containing arguments. Occurrences of the first IDENTIFIER followed by (, a sequence of "tokens" separated by commas, and) is replaced by the given TOKEN-STRING. Each occurence of an IDENTIFIER in the formal parameterlist of the definition is replaced by the corresponding TOKEN-STRING in the call. The

actual parameters in the call are TOKEN-STRINGS separated by commas. Commas surrounded by " do not separate arguments. The number of formal and actual parameterers must be the same. Textstrings and character constants can not be replaced.

Example:

given:

```
#define formula(a,b) a*a+b*b

i:=formula(3,4);
```

the pre-processor will output the line

```
i:=3*3+4*4;
```

to be compiled.

In both kinds of replacement the given TOKEN-STRING will be scanned to see if it contains earlier defined IDENTIFIERS. If so these are replaced. A long definition can continue on the next line by writing ! as the last character in every line that is to be continued.

A line on the form:

```
#undef IDENTIFIER
```

causes the pre-processor to cancel the previous definition of IDENTIFIER.

Definitions as #define can also be made by using a special option, -D, when calling the pre-processor, as #undef can be made using the option -U (see chapter 4.4).

4.2 Inclusion of files.

A line on the form

#include 'FILENAME'

causes the pre-processor to replace the line by the contents of the file given by FILENAME. FILENAMEs that do not start with a / are sought for in the directory containing the source code file and then in the directories mentioned when using the I option (see chapter 4.4).

It is possible for included files to contain inclusion of other files.

4.3 Conditional compilation.

When the pre-processor is given a line on the form

#if CONSTANT-EXPRESSION

it will check whether or not the constant expression is true.

A line on the form

#ifdef IDENTIFIER

causes the pre-processor to check whether or not the given IDENTIFIER for the moment is defined through an earlier #define. If so the condition is true.

#ifndef IDENTIFIER

causes the pre-processor to check whether or not the given IDENTIFIER is undefined. If so the condition is true.

All three of above can be followed by an arbitrary numer of lines, where amongst may be a line on the form

#else

The lines to be conditionally compiled must be ended by a line intended for the pre-processor on the form

#endif

If the checked condition is true the lines between #else and #endif are ignored. If the condition is false the lines between the condition and #else (or #endif if there is not #else) are ignored.

The above mentioned constructions may appear on more than one level.

The syntax for CONSTANT-EXPRESSION is:

```
CONSTANT-EXPRESSION ::= SIMPLE-EXPR {RELOP  SIMPLE-EXPR}
RELOP               ::=  = | <> | < | <= | > | >=
SIMPLE-EXPR         ::= [- | +]  TERM  {ADDOP  TERM}
ADDOP               ::=  + | - | OR
TERM                ::=  FACTOR {MULOP  FACTOR}
MULOP               ::=  * | DIV | MOD | AND
FACTOR              ::=  UNSIGNED INTEGER CONSTANT |
                         UNSIGNED LONGINT CONSTANT |
                         CHARACTER CONSTANT |
                         NOT FACTOR | (CONSTANT EXPRESSION) |
                         TRUE | FALSE
```

Type checking of constant expressions is the same as in Pascal. The result of a constant expression must be of type BOOLEAN.

4.4 Using the pre-processor.

ppp can be called using ppc or pac (the compilation control programs) or it can be called directly in parameter or dialog mode.

When +q is given as parameter the pre-processor will prompt for further parameters (user input shown underlined):

```
$ ppp +q
Source code file:        tst.p
Resulting file:          tst.pp
List option (d/q):       d
Define symbols:          version='010787'
Undefine symbols:
Searching directories:   /usr/abc
```

The corresponding parameter mode is:

```
$ ppp -i tst.p -o tst.pp -Ld -I/usr/abc -D version='010787'
```

Source code file:

The source code file is a Pascal source text that can include directives for the pre-processor. It will normally (but not necessarily) be of type p.

Resulting file:

The output from the pre-processor is the input file with all pre-processor directives interpreted, that is a Pascal source text acceptable by the Pascal compilers.

List option:

The list option may be used to control what kind of information the pre-processor writes on the output device. If listoption d is used ppp will print a dot (.) for each line of the source text read. q option causes the pre-procesor to suppress all output except error messages.

Define symbols:

Define symbols are definitions of symbols analogous to:

```
  #define IDENTIFIER TOKENSTRING
```

the corresponding syntax of a define symbol is:

```
  IDENTIFIER=TOKENSTRING
```

Several symbols can be defined separated by commas:

```
  id1=2,id2=true,id3='hallo'
```

Undefine symbols:

Undefine symbols are analogous to the #undef directive:

```
  #undef IDENTIFIER
```

the corresponding syntax is:
  IDENTIFIER
Several symbols can be undefined separated by commas:
  id1,id2,id3

Searching directories:
Searching directories is a list of directories used by the pre-proces-ser when searching for include files. If the name of an include file does not start with a then the pre-processor will look for the file first in the directory containing the source code, and if the file is not found it will search in the directories listed as searching direc-tories - if more than one directory is given they should be separated by commas.

5. Passing Parameters to the Pascal Program.

It is possible for the user to pass a sequence of characters (a parameter string) to the Pascal program when starting its execution:

Pascal assembler: $ filename parameters

Interpreted Pascal: $ inter filename parameters
or
$ filename parameters

The program may access the parameter string in the following manner:

When execution starts the address of the parameter string will be placed in the first four bytes of the data area of the Pascal program (either by the interpreter or by the Pascal assembler runtime system). Therefore the user may access the parameter string in the following manner:

Include a type declaration like, for example:

```
TYPE PARMARRAY=PACKED ARRAY (1..255) OF CHAR;
```

The length of the parameter array may not exceed 255 characters.

Next, the programmer must give as the very first variable declaration a pointer to this array. For example:

```
VAR PARM: ^PARMARRAY;
```

If this is the first VAR declaration given, PARM will be the four bytes into which the system writes the address of the parameter string, and consequently PARM^ will be a character array containing the actual parameter string.

Example:

Assume that the file TEST either contains the P-code (interpreted Pascal) or the loadmodule (compiled Pascal) of the following program:

```
PROGRAM TEST;
TYPE PARMARRAY=PACKED ARRAY (1..255) OF CHAR;
VAR PARM: ^PARMARRAY;
BEGIN
  WRITE('THE FIRST CHARACTER OF THE PARAMETER STRING IS: ');
  WRITELN( PARM^(1) );
END.
```

If this program is run in the following manner:

Pascal-assembler: $ test hello

(The parameterarray is accessible in main modules and submodules, but can only be accessed in external modules when passed as a parameter to a subroutine in the module).

Interpreted Pascal: $ test hello
or
$ inter test hello

it will output

THE FIRST CHARACTER OF THE PARAMETER STRING IS: h

Furthermore se chapter 3 (the standard functions GETENVR and GETOPT).

6. Aborting the Program.

Program execution may be aborted in two ways:

1. A 'civilized' way to abort a program is the following: the program should at a strategic place test the value of EOF(OUTPUT), which becomes true when the user presses a function key. After each input statement EOF(INPUT) should be tested to see whether the user entered ctrl d. If either of these two function values become true, appropriate action should be taken to terminate the execution of the program (for instance by using EXIT(PROGRAM)).

2. A 'brute force' method of abortion may, for example, be needed if an erroneous program runs wild, or if for some reason you want to stop a particular program immediately.

   This can be done either by pressing the ctrl-key and the c-key simultaneously or by using the program kill (see Supermax Operating System, System V, reference manual, section 1).

   When the action is taken the process running the program will die. All files are closed before the death of the process.

Appendix A. DO's and DONT's.

This appendix lists a few typical errors often made by programmers.

DO check the value of IORESULT after each RESET, REWRITE, LOCK, and CHAIN procedure call.

DO check the value of IORESULT after each file I/O operation if the automatic I/O checking is off.

DO read all the way to end-of-file in a sequential file, if you used a REWRITE call to open it, but have not written into it. When you close the file an end-of-file mark will be indicated at the current file location, so it had better be the end of the file!

DON'T specify {$R-} unless you are sure that your program works.

DON'T execute programs with syntax errors.

DON'T assume anything about the dynamic length of a string before a value has been assigned to it.

DON'T reference through a pointer value before you are sure that it's value is not NIL.

DON'T reference through an uninitialized pointer variable.

DON'T store a pointer variable in a file. There is no guarantee that the item pointed to by the pointer variable will lie in the same address when the pointer is read from the file at a later execution of the program.

DON'T pass a string as a reference parameter to a subroutine where the declared maximum length of the formal parameter is greater than the declared maximum length of the actual parameter.