nascom

NASCOM PASCAL

Programming Manual

Nascom Microcomputers
Division of Lucas Logic Limited
Welton Road Wedgnock Industrial Estate
Warwick CV34 5PZ
Tel: 0926 497733 Telex: 312333

Lucas Logic

TABLE OF CONTENTS

TABLE OF CONTENTS

0: INTRODUCTION

The NASCOM PASCAL Language System is meant to offer an alternative to BASIC. Programs written in NASCOM PASCAL will execute much faster than their BASIC equivalents, and better programming techniques can be practised, as Pascal is far more versatile than BASIC.

As the NASCOM PASCAL system is very compact (only 12K, hereof 5.5K compiler), it has not, of course, been possible to implement standard Pascal in full: The NASCOM PASCAL subset does not support user defineable types, sets and file-types. However all of the basic statement constructions are retained, and procedures and functions allow for both value and variable parameters. The fundamental data types INTEGER, REAL and BOOLEAN are likewise supported, while the type CHAR has been replaced by the type STRING, which offers a more flexible character handling.

This manual fully defines the NASCOM PASCAL subset, and should be carefully studied before any programming efforts are made.

1: BASIC ELEMENTS OF THE LANGUAGE

1.1 SYMBOLS

The basic vocabulary of Pascal consists of basic symbols classified into letters, digits, and special symbols:

Letters:	A to Z, a to z, '_' and '\'.
Digits:	0 1 2 3 4 5 6 7 8 9
Symbols:	+ - * / = < > ( ) [ ] . , ; : ' { }

The compiler does not differ between upper case and lower case letters. Some operatores and delimiters are formed using two special symbols:

1. <> <= >= := ..
2. (. and .) can be used instead of [and].
3. (* and *) can be used instead of { and }.

1.2 RESERVED WORDS AND STANDARD IDENTIFIERS

The reserved words listed below can not be used as user defined identifiers:

AND	ARRAY	BEGIN	BOOLEAN
CASE	DIV	DO	DOWNTO
ELSE	END	EXOR	EXTERNAL
FOR	FUNCTION	GOTO	IF
INIT	INTEGER	LABEL	MOD
NOT	OF	OR	OTHERS
PROGRAM	PROCEDURE	REAL	REPEAT
SHIFT	STRING	THEN	TO
UNTIL	VAR	WHILE	

Certain identifiers, called standard identifiers, are predefined (e.g. sin, cos). Unlike the reserved words these identifiers can be redefined by the user:

abs	addr	arctan	call
chr	concat	cos	empty
exp	false	frac	inp
int	keyboard	left	ln
load	maxint	mem	mid
odd	ord	out	pi
plot	point	pred	random
read	readln	right	round
save	sin	sqr	sqrt
succ	true	trunc	write
writeln			

1.3 SEPARATORS

Blanks, ends of lines, and comments are considered as separators. At least one separator most occur between any pair of consecutive identifiers, numbers or reserved words.

2: USER DEFINED ELEMENTS

2.1 IDENTIFIERS

Identifiers are names denoting constants, procedures, functions, variables, and labels. They must begin with a letter, which may be followed by any number of letters, digits, or '.'-characters. Examples:

```
PASCAL    Pascal    NAME.41.CODE
```

Note that the compiler does not differ between upper case and lower case letters. Thus, the identifier 'PASCAL' is identical to 'Pascal'.

2.2 NUMBERS

Numbers may be written in both decimal and hexadecimal notations. Hexadecimal numbers must be preceeded by a $-sign. The letter E preceeding the scale factor is pronounced as 'times 10 to the power of'. Examples:

```
1    100    $25EC    0.138    5E10    87.13556E-8
```

No separators may occur within numbers.

2.3 STRINGS

Sequences of characters enclosed by single quote marks are called strings. To include a quote mark in a string it should be written twice. Examples:

```
'NASCOM PASCAL'    'A'    'A    '    'that''s all folks'
```

2.4 COMMENTS

A comment is a sequence of characters enclosed in curly brackets (or (* and *)), which can be removed from the program text without altering its meaning. Example:

```
(* This is a comment *)
```

3: DATA TYPES

A data type defines the set of values a variable may assume. Every variable occuring in a program must be associated with one and only one data type. NASCOM PASCAL supports four basic data types: Integer, real, boolean, and string.

3.1 INTEGERS

An integer is a whole number within the range -32768 to 32767. When operating on integers overflow and underflow will not be detected.

3.2 REALS

A real is a real number within one of these ranges:

```
-1.7014118346E+38 <= R <= -2.9387358770E-39
R = 0
2.9387358770E-39 <= R <= 1.7014118346E+38
```

Reals provide 11+ significant digits. If an overflow occurs during an arithmetic operation involving reals, the program will break and display an error message. If an underflow occurs the result will be zero.

3.3 BOOLEANS

A boolean variable should only assume the predefined values true (-1) and false (0). However, as NASCOM PASCAL does not differ between integers and booleans, a boolean variable can assume other values, but this is strongly discouraged.

3.4 STRINGS

When a string variable is declared one informs the compiler of the maximum length it may assume (between 1 and 255). Examples:

```
STRING[32]
STRING[stringsize]
```

3.5 ARRAYS

An array is a structure consisting of a fixed number of components which are all of the same type, called the component type. The elements of the array are designated by indices, which are of the type integer. Upon declaration the upper and lower bound of each index is written seperated by '..'. Examples:

```
ARRAY [1..10] OF INTEGER
ARRAY [0..maxsize] OF STRING[32]
ARRAY [-5..11,29..45] OF REAL
```

Components in an n-dimensional array are designated by n integer expressions. Examples:

```
data[12]
b[i+j,7]
names[pointers[8],3]
```

3.5.1 The mem array

The mem array is a predefined one-dimensional array representing memory. Each component designates a byte, whose address is given by the index. Components of the mem array can only assume values between 0 and 255. If a value greater than 255 is assigned the actual value will only be the least significant 8 bits. Examples:

```
i:=mem[$C00] AND $16;

FOR p:=1 TO length(s) DO
mem[offset+p]:=ord(mid(s,p,1));
```

4: DECLARATIONS

A program consists of 3 parts:

1. The program header
2. The declaration part
3. The statement part

The program heading gives the program a name and lists its parameters, through which the program communicates with the environment. Examples:

```
PROGRAM conversion;

PROGRAM calculation(input,output);
```

In NASCOM PASCAL the program header is purely optional, and if it is used everything between the reserved word PROGRAM and the first semicolon is considered as a comment.

Declarations must be listed in the following order:

1. Label declaration part
2. Constant definition part
3. Variable declaration part
4. Procedure and function declaration part

None of the above mentioned parts need to be present (thus the declaration part may be empty).

4.1 LABEL DECLARATION PART

All labels used in the program must be declared in the label declaration part, which is introduced by the reserved word LABEL. A label may either be an identifier or an unsigned number. Examples:

```
LABEL 1,error,999,stop;
```

Any statement in the program may be prefixed by a label followed by a colon (making possible a reference by a goto statement). Examples:

```
999: write('Done...');
```

A label should only be referenced within the block in which it is declared.

4.2 CONSTANT DEFINITION PART

A constant definition introduces an identifier as a synonym for a constant. The symbol CONST introduces the constant definition part. Example:

```
CONST
  number=45;
  max=193.158;
  min=-max;
  name='Johnson';
```

Predifined constants are as follows:

pi	Real	3.1415926536.
true	Boolean	True (-1).
false	Boolean	False (0).
maxint	Integer	32767.
empty	String	'' (The empty string).

4.3 VARIABLE DECLARATION PART

Every variable occuring in the program must be declared in the variable declaration part, which is introduced by the reserved word VAR. A variable declaration associates an identifier and a data type to the variable. More variables of the same data type can be declared on the same line. Examples:

```
VAR
  i,j,k: INTEGER;
  xcoor,ycoor: REAL;
  names: ARRAY [1..100] OF STRING [32]
```

The variable is accessable throughout the entire block containing the declaration, unless the identifier is redefined in a subordinate block.

When entering a block all variables declared within the block will cleared, e.g. reals and integers assumes the value 0, booleans assumes the value false, and strings assumes the value empty.

4.4 PROCEDURE AND FUNCTION DECLARATION PART

The procedure declaration serves to define procedures within the current procedure or program (see chapter 7). A procedure is activated from a procedure statement (see chapter 6.1.2).

The function declaration part serves to define a program part which computes and returns a value (see chapter 8). Functions are activated by the evaluation of a function designator, which is a constituent of an expression (see chapter 5.4).

5: EXPRESSIONS

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operators and operands, i.e. variables, constants, and functions.

The rules of composition specify operator precedences according to four classes of operators. The NOT operator has the highest precedence, followed by the multiplying operators (* / DIV MOD AND SHIFT), then the adding operators (+ - OR EXOR), and, finally, with the lowest precedence, the relational operators (= <> > < >= <=). All operators allowing integers as operands will also allow booleans. Any expression enclosed within parentheses is evaluated independently of preceeding or succeeding operators.

5.1 THE NOT OPERATOR

The NOT operator denotes complementation of its operand, which must be of the type integer or of the type boolean. Examples:

```
NOT true        = false
NOT false       = true
NOT 5           = -6
```

5.2 MULTIPLYING OPERATORS

Operator	Operation	Type of operands	Type of result
*	Multiplication	real, integer	real, integer
/	Division	real, integer	real
DIV	Integer division	integer	integer
MOD	Modulus	integer	integer
SHIFT	Logical shift	integer	integer
AND	Logical AND	integer	integer

The operation I SHIFT J has the following effect: I will be shifted to the left J times, if J is positive, and -J times to the right, if J is negative. Thus the result will always equal zero if ABS(J) is greater than 15.

5.3 ADDING OPERATORS

Operator	Operation	Type of operands	Type of result
+	Addition	real, integer	real, integer
-	Subtraction	real, integer	real, integer
OR	Logical OR	integer	integer
EXOR	Logical EXOR	integer	integer

When used as operators with one operand only, - denotes sign inversion, and + denotes the identity operation.

5.4 FUNCTION DESIGNATORS

A function designator specifies the activation of a function. It consists of the identifier designating the function and a list of actual parameters. The parameters are variables or expressions, and are substituted for the corresponding formal parameters. Examples:

```
sin(y)*cos(x)
concat('Name: ',firstname,' ',lastname)
arctan(1.0)*4.0
(sum(a,100)<5) AND (z=0)
```

6: STATEMENTS

Statements denote algorithmic actions and are said to be executable. They may be prefixed by a label which can be referenced by a GOTO statement (see chapter 6.1.3).

6.1 SIMPLE STATEMENTS

A simple statement is a statement of which no part constitutes another statement. In this group are the assignment, procedure, GOTO, INIT, and empty statements.

6.1.1 Assignment statements

The assignment statement serves to replace the current value of a variable or a function identifier by a new value specified as an expression.

The variable (or function) and the expression must be of identical type, with the following exceptions being permitted:

1) If the type of the variable is real, the type of the expression may be integer.

2) A string expression need not have the same length as the maximum length of the string variable. If more characters are assigned than specified by the maximum length, only the lefmost characters will be transferred.

Example:

```
x:=y+z  {replace current value of x by sum of y and z}
```

6.1.2 Procedure statements

A procedure statement serves to execute the procedure denoted by the procedure identifier. The procedure statement may contain a list of actual parameters which are substituted in place of their corresponding formal parameters (see chapter 9) defined in the procedure declaration. Examples:

```
sort(names);
exchange(x,y);
plot(x,round(sin(x*f)*20)+24,1 ;
```

6.1.3 GOTO statements

A GOTO statement serves to indicate that further processing should continue at another part of the program, namely, at the place of the label.

The following restrictions hold concerning the applicability of labels:

1) The scope of a label is the block within which it is declared. It is, therefore, not possible to jump into or out of a procedure or a function.

2) Jumps into and out of FOR statements are not allowed.

3) Every label must be specified in a label declaration in the heading of the block in which the label marks a statement.

6.1.4 INIT statements

An INIT statement serves to initialize an array structure to a set of constant values. The constants and the components of the array must be of identical type. Example:

```
VAR
  data: ARRAY[1..6] OF INTEGER;
BEGIN
  INIT data TO 15,6,19,8,1,3;
  :
  :
END.
```

The above program is equal to:

```
VAR
  data: ARRAY[1..6] OF INTEGER;
BEGIN
  data[1]:=15; data[2]:=6;  data[3]:=19;
  data[4]:=8;  data[5]:=1;  data[6]:=3;
  :
  :
END.
```

If less constants are specified than the total number of components in the array, only the first components will be initialized. Example:

```
VAR
  numbers: ARRAY[0..9] OF STRING[5];
BEGIN
  INIT numbers TO empty,'one','two','three','four','five';
  :
  :
END.
```

When the INIT statement has been executed, the components of numbers will have the following values:

```
numbers[0]=empty       numbers[1]='one'
numbers[2]='two'       numbers[3]='three'
numbers[4]='four'      numbers[5]='five'
numbers[6]=empty       numbers[7]=empty
numbers[8]=empty       numbers[9]=empty
```

When initializing array structures with more than one dimension the components will be processed with the rightmost dimension increasing first. Example:

```
VAR
  a: ARRAY[1..3,1..3] OF INTEGER;
BEGIN
  INIT a TO 9,6,8,15,18,33,7,10,19;
  :
  :
END
```

The above program will initialize the components of a to:

```
a[1,1]=9;      a[1,2]=6;      a[1,3]=8;
a[2,1]=15;     a[2,2]=18;     a[2,3]=33;
a[3,1]=7;      a[3,2]=10;     a[3,3]=19;
```

The INIT statement can in addition serve to initialize a section of memory. Example:

```
INIT mem[base] TO $EF,$41,$42,$43,$00,$C9;
```

Assuming that the variable base equals $D00, the byte at $D00 will equal $EF, the byte at $D01 will equal $41, etc., upon completing the INIT statement.

6.1.5 Empty statements

The empty statement denotes no action and occurs whenever the syntax of Pascal requires a statement but no statement appears. Examples:

```
BEGIN END;
WHILE digit AND (a>17) DO {nothing};
REPEAT {wait} UNTIL keyboard;
```

6.2 STRUCTURED STATEMENTS

Structured statements are constructs composed of other statements which have to be executed in sequence (compound statements), conditionally (conditional statements), or repeatedly (repetitive statements).

6.2.1 Compound statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The symbols BEGIN and END act as statement brackets. Example:

```
BEGIN
  z:=x; x:=y; y:=z;  {interchange values of x and y}
END;
```

The compound statement neither forbids nor requires a semicolon succeeding the last statement.

6.2.2 Conditional statements

A conditional statement selects for execution a single of its component statements.

6.2.2.1 IF statements

The IF statement specifies that a statement be executed only if a certain condition (boolean expression) is true. If it is false, then either no statement is to be executed, or the statement following the symbol ELSE is to be executed.

The syntactic ambiguity arising from the construct

```
IF <e1> THEN IF <e2> THEN <s1> ELSE <s2>
```

is resolved by evaluating

```
IF <e1> is false, no statement is executed.
IF <e1> is true and <e2> is true, <s1> is executed.
IF <e1> is true and <e2> is false, <s2> is executed.
```

Examples:

```
IF x<1.5 THEN z:=x+y ELSE z:=1.5;
IF name=empty THEN name:='Not stated';
```

6.2.2.2 CASE statements

The CASE statement consists of an expression (the selector) and a list of statements, each labelled by a constant or a list of constants of the type of the selector. It specifies that the one statement be executed whose constant list contains the current value of the selector. If no constant equals the value of the selector, control is given to the statement succeeding the OTHERS: label, if it exists. Otherwise, no statement will be executed.

Valid selector types are integer, boolean, and string types (reals are not allowed). Examples:

```
CASE operator OF
  '+': x:=x+y;
  '-': x:=x-y;
  '*': x:=x*y;
  '/': x:=x/y
END;

CASE number OF
  1: write('one');
  2: write('two');
  3,4,5: write('some');
  OTHERS: write('several');
END;
```

The CASE statement neither forbids nor requires a semicolon succeeding the last statement.

6.2.3 Repetitive statements

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known beforehand (i.e. before the repetitions are started), the FOR statement is the appropriate construct to express this situation; otherwise, the WHILE or the REPEAT statement should be used.

6.2.3.1 WHILE statements

The expression controlling repetition must be of type boolean. The statement is repeatedly executed until the expression becomes false. If its value is false at the beginning, the statement is not executed at all. Example:

```
WHILE a<1000 DO
BEGIN
  a:=sqr(a); b:=b+1;
END;
```

6.2.3.2 REPEAT statements

The expression controlling repetition must be of type boolean. The sequence of statements between the symbols REPEAT and UNTIL is repeatedly executed (and at least once) until the expression becomes true. Example:

```
REPEAT
  read(digit); write(digit);
  number:=number*10+ord(digit)-48;
UNTIL number>1000;
```

The REPEAT statement neither forbids nor requires a semicolon succeeding the last statement.

6.2.3.3 FOR statements

The FOR statement indicates that the component statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable of the FOR statement. The progression can be up TO (succeeding) or DOWNTO (preceding) a final value.

The control variable, the initial value, and the final value must be of type integer.

If the initial value is greater than the final value when using the TO clause, or if the initial value is less than the final value when using the DOWNTO clause, the component statement is not executed at all.

Examples:

```
FOR i:=1 TO max DO writeln(i:5,sqr(i):8);

FOR i:=1 TO 100 DO FOR j:=1 TO 10 DO
BEGIN
  IF a[i,j]>5 THEN a[i,j]:=5;
  count:=count+a[i,j];
END;
```

Upon completion of a FOR statement the value of the control variable is given by:

1) If the component statement was not executed the control variable will equal the initial value.

2) When using the TO clause the control variable will equal the final value plus one.

3) When using the DOWNTO clause the control variable will equal the final value less one.

7: PROCEDURES

A procedure is a seperate program part which may be activated from a procedure statement (see chapter 6.1.2).

7.1 PROCEDURE DECLARATIONS

A procedure declaration generally consists of 3 parts:

1) The procedure heading.
2) The declaration part.
3) The statement part.

7.1.1 The procedure heading

The procedure heading specifies the identifier naming the procedure, an optional formal parameter list, and an optional EXTERNAL specification.

The paramaters are either value or variable parameters (see chapter 9).

EXTERNAL specifies that the procedure is a seperate machine code subroutine, which resides at the address given by the integer constant following the EXTERNAL symbol (see appendix E). CODE specifies that the procedure is listed in Z-80 machine code, directly following the CODE symbol (see appendix E). In the case of EXTERNAL or CODE procedures the declaration part as well as the statement part is empty.

7.1.2 The declaration part

The declaration part has the same form as that of a program. All identifiers introduced in the formal parameter list and the declaration part are local to the procedure declaration, which is called the scope of these identifiers. They are not known outside their scope. A procedure declaration may reference any constant, variable, procedure, or function identifier global to it (i.e. defined in an outer block), or it may choose to redefine the name.

7.1.3 The statement part

The statement part specifies the algorithmic actions to be executed upon activation of the procedure by a procedure statement. The statement part takes the form of a compound statement (see chapter 6.2.1). The use of a procedure identifier in a procedure statement within the statement part implies recursive execution of the procedure.

7.2 STANDARD PROCEDURES

A standard procedure need not be declared, and may be redefined by the programmer by using its name as a procedure identifier in a procedure declaration.

call(a) Generate a call to the memory address given by the integer expression a.

screen(x,y) Move the cursor to line y, coloumn x. x and y are integer expressions. If a coordinate value is illegal, the current value of this coordinate is unchanged by the procedure activation. Thus the screen procedure may be used as a tabulator by zeroing the y-coordinate.

plot(x,y,f) — x,y, and f are integer expressions. Alter the state of the semigraphic pixel at x,y, according to the value of f:

f=0: Reset x,y.
f=1: Set x,y.
f=2: Invert x,y.

The plot procedure compensates for the offset of line 16 on the NASCOM display. Hence, pixels with y-coordinates within the interval 0<=y<=2 resides on line 16. A procedure activation involving illegal coordinate values will be ignored.

out(p,d) — Output least significant 8 bits of d to the port given by the least significant 8 bits of p. p and d are integer expressions.

The standard procedures supporting input and output are described in chapter 10.

8: FUNCTIONS

A function is a program part which computes and returns a value. Functions are activated by the evaluation of a function designator (see chapter 5.5) which is a constituent of an expression.

8.1 FUNCTION DECLARATIONS

A function declaration generally consists of 3 parts:

1) The function heading.
2) The declaration part.
3) The statement part.

8.1.1 The function heading

The function heading specifies the identifier naming the function, an optional formal paramater list, the result type, and an optional EXTERNAL specification.

The paramaters are either value or variable parameters (see chapter 9).

The result type of the function can be either integer, boolean, real, or string.

EXTERNAL specifies that the function is a seperate machine code subroutine which resides at the address given by the integer constant following the EXTERNAL symbol (see appendix E). CODE specifies that the function is listed in Z-80 machine code, directly following the CODE symbol (see appendix E). In the case of EXTERNAL or CODE functions the declaration part as well as the statement part is empty.

8.1.2 The declaration part

The declaration part has the same form as that of a program. All identifiers introduced in the formal parameter list and the declaration part are local to the function declaration, which is called the scope of these identifiers. They are not known outside their scope. A function declaration may reference any constant, variable, procedure, or function identifier global to it (i.e. defined in an outer block), or it may choose to redefine the name.

8.1.3 The statement part

The statement part takes the form of a compound statement (see chapter 6.1.2). Within the statement part at least one statement assigning a value to the function identifier must occur. This assignment determines the result of the function. The appearance of the function identifier in an expression within the function itself implies recursive execution of the function.

8.2 STANDARD FUNCTIONS

A standard function need not be declared, and may be redefined by the programmer by using its name as a function identifier in a function declaration.

8.2.1 Arithmetic functions

In the functions listed below the type of x must be either real or integer, and the type of the result is the type of x.

abs(x) Computes the absolute value of x.

sqr(x) Computes x*x.

In the functions listed below the type of x must be either real or integer, and the type of the result is real.

sin(x) Sine.

cos(x) Cosine.

arctan(x) Arccus tangent.

ln(x) Natural logarithm.

exp(x) Exponential function.

sqrt(x) Square root.

int(x) The whole part of x, i.e the result is the greatest whole number less than or equal to x for x>=0, and the least whole number greater than or equal to x for x<0.

frac(x) The fractional part of x with the same sign as x, i.e. frac(x)=x-int(x).

8.2.2 Integer functions

In the functions listed below the type of i is integer.

succ(i) Computes i+1. The type of the result is integer.

pred(i) Computes i-1. The type of the result is integer.

odd(i) Returns the boolean value true if i is odd, or the boolean value false if i is even.

8.2.3 String functions

length(s) Returns the length of the string s. The type of the result is integer.

mid(s,p,n) Returns a string containing n characters copied from s starting at the p'th position in s. The type of s is string, and the type of n and p is integer.

mid(s,p) Returns the leftmost cahracters copied from s starting at the p'th position in s. The type of s is string and the type of p is integer.

left(s,n) Returns the leftmost n characters copied from s. The type of s is string and the type of n is integer.

right(s,n) Returns the rightmost n characters copied from s. The type of s is string and the type of n is integer.

concat(strs) strs is any number of string expressions separated by commas. The result is a string which is the concatenation of the parameters in the same sequence as they are written.

8.2.4 Transfer functions

trunc(x) The type of x is real; the result is the greatest integer less than or equal to x for x>=0, and the least integer graeter than or equal to x for x<0.

round(x) The type of x is real; the result, of type integer, is the value of x rounded, i.e.:

```
round(x) = trunc(x+0.5), for x>=0
           trunc(x+0.5), for x<0
```

ord(s) Returns the ASCII value of the leftmost character in the string s. If s is empty the result will be zero. The type of the result is integer.

chr(i) Returns a string containing one character whose ASCII value is i. The type of i is integer.

8.2.5 Further standard functions

addr(v) Returns the memory address of the variable v. The memory address of an array can be calculated by referring to the first element of each dimension.

random Returns a random number within the interval 0<=r<1. The type of the result is real.

random(i) Returns a random integer within the interval 0<=r<i. The type of the result is integer.

inp(p) Returns the value read from port p. p must be an integer expression within the interval 0<=p<=255. The type of the result is integer.

keyboard Scans the keyboard once, and returns the ASCII value of the currently depressed key. If no key is depressed 0 is returned. The type of the result is integer.

point(x,y) Returns the boolean value true if the semigraphic pixel x,y is set, otherwise returns the boolean value false. The type of x and y must be integer.

9: PARAMETERS

Parameters provide a substitution mechanism that allows the algorithmic actions of a procedure or a function (in this chapter referred to as a subprogram) to be repeated with a variation of its arguments.

9.1 FORMAL AND ACTUAL PARAMETERS

A procedure statement or a function designator may contain a list of actual parameters, which are substituted for the corresponding formal parameters that are defined in the heading of the subprogram. The correspondance is established by the positioning of the parameters in the lists of actual and formal parameters.

9.2 PARAMETER TYPES

BLS Pascal supports two kinds of parameters: Value parameters and variable parameters.

9.2.1 Value parameters

When no symbol heads a formal parameter part of a subprogram heading, the parameter(s) of this part are said to be value parameters. In this case the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameters represents a local variable in the subprogram. As its initial value this variable receives the current value of the corresponding actual parameter (i.e. the value of the expression at the time of the call). The subprogram may then change the value of this variable by assigning to it; this will not, however, affect the value of the actual parameter. Hence, a value parameter can never represent a result of a computation.

Consider the following procedure declaration:

```
PROCEDURE printline(width: INTEGER);
BEGIN
  FOR width:=width DOWNTO 1 DO write('*');
  writeln;
END;
```

The procedure statement "printline(a);" will have the same effect as executing

```
width:=a;
FOR width:=width DOWNTO 1 DO write('*');
writeln;
```

Although the variable width is altered during the procedure, the variable a will be left unchanged, as width is a value parameter. As mentioned above the actual parameter need not be a variable, but can be any expression, e.g. "printline(a+2*b);" and "printline(25);".

9.2.2 Variable parameters

When the symbol VAR heads a formal parameter part of a subprogram heading, the parameter(s) of this part are said to be variable parameters. In this case the actual parameter must be a variable. The corresponding formal parameter represents this variable during the entire execution of the subprogram. Any operation involving the formal parameter is preformed directly upon the actual parameter. Hence, whenever a parameter is to represent a result of the subprogram, it must be declared as a variable parameter.

Consider the following procedure declaration:

```
PROCEDURE swap(VAR x,y: REAL);
VAR temp: REAL;
BEGIN
  temp:=x; x:=y; y:=temp;
END;
```

The procedure statement "swap(a,b);" will have the same effect as executing "temp:=a; a:=b; b:=temp;". Obviously, the statement "swap(20,a+b);" will result in an error, as the statements "temp:=20; 20:=a+b; a+b:=temp;" are impossible to execute.

9.3 RULES APPLYING TO PARAMETERS

The formal parameter list and the actual parameter list must agree with respect to the total number of parameters and the type of each of the parameters respectively.

All address calculation is done at the time of the call. Thus, if a variable is a component of an array, its index expression(s) is evaluated upon activating the subprogram.

In the case of a parameter being an array structure, the actual parameter and the formal parameter must agree with respect to component type and number of components. However the lower and upper limits of each dimension, and the number of dimensions need not agree.

If a formal parameter is a variable parameter of the type real, the corresponding actual parameter may be an expression of the type integer. This does not apply to variable parameters.

If a formal parameter is a variable parmeter of the type string, the corresponding actual parameter can be a string expression of any length. However, if the length of the actual string parameter is greater than the maximum length of the formal parameter, only the leftmost characters will be transferred. This does not apply to variable parameters.

10: INPUT AND OUTPUT

NASCOM PASCAL allows for input and output by means of four standard procedures (read, readln, write, and writeln). In addition two standard procedures (load and save) allows for loading and saving of arrays from and to the tape recorder.

10.1 INPUT

Input is supported by the standard procedures read and readln.

10.1.1 The procedure read

The procedure read allows for strings and numeric values to be input. The format of the procedure statement is:

```
read(v1,v2,...,vn);
```

Which is equal to

```
BEGIN read(v1); read(v2); ... read(vn) END;
```

During data entry the following control keys are available:

<BS>	Backspace
<ESC>	Clear line
<ENTER>	Process entry

For a variable of one of the numeric types (real or integer) the read procedure expects to read a string of characters which can be interpreted as a numeric value of the same type. Numeric values should follow the rules that apply to numeric constants (see chapter 2.2). The entry must be terminated by a carriage return (i.e. <ENTER>) immediately following the last character of the numeric value. The carriage return is not echoed. If the interpretation results in an error, the entry field will be cleared, indicating that a new value must be entered. A special case of numeric input is when no value is entered, i.e. the entry consists of nothing but a carriage return. Instead of assigning a new value to the variable, the current value will be retained.

When reading strings with a maximum length greater than one, read will accept all characters up to but not including the terminating carriage return. The maximum number of characters which can be entered is given by the maximum length of the string variable (however, not more than 63 characters).

When reading strings with a maximum length of one program execution will resume the moment a key is depressed, and the character read will not be echoed.

10.1.2 The procedure readln

The procedure readln is identical to read, except that after a value has been read a carriage return is output. The format of the procedure statement is:

```
readln(v1,v2,...,vn);
```

Which is equal to

```
BEGIN readln(v1); readln(v2); ... readln(vn) END;
```

10.2 OUTPUT

Output is supported by the standard procedures write and writeln.

10.2.1 The procedure write

The procedure write allows strings and numeric values to be output. The format of the procedure statement is:

```
write(p1,p2,...,pn);
```

which is equal to

```
BEGIN write(p1); write(p2); ... write(pn) END;
```

p1,p2,...,pn denote so-called write parameters, which, according to the type of the value to be output, can take on one of the following formats (m, n, and i denote integer expressions, r denote a real expression, and s denote a string expression):

i — The decimal representation of i is output with no preceding blanks.

i:n — The decimal representation of i is output preceded by an appropriate number of blanks to make the field width n.

r — The decimal representation of r is output in floating point format in a field of 18 characters:

" sd.ddddddddddEtdd"

where s stands for either " " or "-", d stands for a digit, and t stands for either "+" or "-".

r:n — The decimal representation of r is output in floating point format. The field width and the number of significant digits depends on the value of n:

n<8: "d.dEtdd" or "-d.dEtdd"

8<=n<17: "sd.<digits>Etdd", where <digits> denotes n-6 decimal digits.

n>17: "<spaces>sd.ddddddddddEtdd", where <spaces> denotes n-17 blanks.

r:n:m — The decimal representation of r is output in fixed point format with m digits after the decimal point in a field of n characters. m must be within the interval 0<=m<=24. If not, floating point format is used.

s — s is output with no preceding blanks.

s:n — s is output preceded by an appropriate number of blanks to make the field width n.

10.2.2 The procedure writeln

The procedure writeln is identical to write, except that after the last value has been written, a carriage return is output. The format of the procedure statement is:

```
writeln(p1,p2,...,pn);
```

which is equal to

```
BEGIN write(p1); write(p2); ... writeln(pn) END;
```

To produce a single carriage return call writeln without any parameters.

10.3 SAVING AND LOADING ARRAYS

Input and output of arrays from and to the tape recorder are supported by the standard procedures load and save.

10.3.1 The procedure save

The procedure save will output arrays of any type to the tape recorder. The format of the procedure statement is:

```
save(a);
```

where a denotes an array identifier. Upon activation of the procedure the tape LED will be switched on, a brief pause will be issued, the array will be output, and the tape LED will be switched off.

10.3.2 The procedure load

The procedure load will read a tape previously written by the save procedure. The format of the procedure statement is:

```
load(a,i);
```

where a denotes an array identifier, and i denotes the identifier of an integer variable in which an error status will be returned.

Upon activation of the procedure the tape LED will be switched on. When the procedure ends the tape LED will be switched off, and the variable i will contain the error status of the procedure call:

- i=0: No errors occured.
- i=1: Type mismatch. The number of components or the component type does not agree.
- i=2: Checksum error.
- i=3: The procedure was aborted by the user pressing the <ESC> key.

APPENDIX A: NASCOM PASCAL SYNTAX

The syntax of NASCOM PASCAL is presented using BNF formalism. The following symbols are meta-symbols belonging to the BNF formalism, and not symbols of the Pascal language:

::=	Means ´is defined as´.
|	Means ´or´.
{...}	Denotes possible repetition of the enclosed symbols zero or more times.

The symbol <character> denotes any printable character, i.e. a character with an ASCII value between $20 and $FF.

```
<letter> ::=  A | B | C | D | E | F | G | H | I | J ! K | L |
        M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
        \ | _ | a | b | c | d | e | f | g | h | i | j | k | l |
        m | n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hexdigit> ::= <digit> | A | B | C | D | E | F

<empty> ::=

<program> ::= <program heading> <block> .

<block> ::= <declaration part> <statement part>

<declaration part> ::= <label declaration part>
        <constant definition part> <variable declaration part>
        <procedure and function declaration part>

<label declaration part> ::= <empty> | LABEL <label> { , <label> }

<label> ::= <unsigned integer> | <identifier>

<unsigned integer> ::= <digit> { <digit> }

<identifier> ::= <letter> { <letter or digit> }

<letter or digit> ::= <letter> | <digit> | .

<constant definition part> ::= empty |
        CONST <constant definition> ; { <constant definition> ; }

<constant definition> ::= <identifier> = <constant>

<constant> ::= <unsigned number> | <sign> <unsigned number> |
        <constant identifier> | <sign> <constant identifier> |
        <string>

<unsigned number> ::= <unsigned integer> | <unsigned real> |
        <unsigned hexinteger>

<unsigned real> ::= <unsigned integer> . <digit> { <digit> } |
        <unsigned integer> . <digit> { <digit> }  <scale factor> |
        <unsigned integer>  <scale factor
```

```
<scale factor> ::= <unsigned integer> | <sign> <unsigned integer>

<sign> ::=  + | -

<unsigned hexinteger> ::= $ <hexdigit> { <hexdigit> }

<constant identifier> ::= <identifier>

<string> ::= ' { <character> } '

<variable declaration part> ::= <empty> |
        VAR <variable declaration> ; { <variable declaration> ; }

<variable declaration> ::= <identifier> { , <identifier> } : <type>

<type> ::= <simple type> | <structured type>

<simple type> ::= INTEGER | REAL | BOOLEAN | <string type>

<string type> ::= STRING [ <constant> ]

<structured type> ::= ARRAY [ <index type> { , <index type> } ] OF
        <simple type>

<index type> ::= <constant> .. <constant>

<procedure and function declaration part> ::=
        { <procedure or function declaration> ; }

<procedure or function declaration> ::=
        <procedure declaration> | <function declaration>

<procedure declaration> ::= <procedure heading> <block>

<procedure heading> ::= PROCEDURE <identifier>
        <formal parameter list> ; | PROCEDURE <identifier>
        <formal parameter list> ; <external specification> ;

<formal parameter list> ::= <empty> |
        ( <formal parameter part> { ; <formal parameter part> } )

<formal parameter part> ::= <parameter group> |
        VAR <parameter group>

<parameter group> ::= <variable declaration>

<external specification> ::= EXTERNAL <constant>

<function declaration> ::= <function heading> <block>

<function heading> ::= FUNCTION <identifier>
        <formal parameter list> : <result type> ; | FUNCTION
        <identifier> <formal parameter list> : <result type> ;
        <external specification> ;

<result type> ::= <simple type>

<statement part> ::= <compound statement>

<compound statement> ::= BEGIN <statement> { ; <statement> } END
```

```
<statement> ::= { <label> : } <unlabelled statement>

<unlabelled statement> ::= <simple statement> |
        <structured statement>

<simple statement> ::= <assignment statetement> |
        <procedure statement> | <goto statement> |
        <init statement> | <empty statement>

<assignment statement> ::= <variable> := <expression> |
        <function identifier> := <expression>

<variable> ::= <simple variable> | <component variable>

<simple variable> ::= <identifier>

<component variable> ::= <array identifier> [ <expression>
        { , <expression> } ]

<array identifier> ::= <identifier>

<function identifier> ::= <identifier>

<expression> ::= <simple expression> | <simple expression>
        <relational operator> <simple expression>

<relational operator> ::=  = | <> | > | < | >= | <=

<simple expression> ::= <term> { <adding operator> <term> }

<adding operator> ::=  + | - | OR | EXOR

<term> ::= <factor> { <multiplying operator> <factor> }

<multiplying operator> ::= * | / | DIV | MOD | AND | SHIFT

<factor> ::= <uncomplemented factor> | NOT <uncomplemented factor>

<uncomplemented factor> ::= <unsigned factor> |
        <sign> <unsigned factor>

<unsigned factor> ::= <variable> | <unsigned constant> |
        ( <expression> ) | <function designator>

<unsigned constant> ::= <unsigned number> | <string> |
        <constant identifier>

<function designator> ::= <function identifier>
        <actual parameter list>

<actual parameter list> ::= <empty> | ( <actual parameter>
        { , <actual parameter> } )

<actual parameter> ::= <expression> | <variable> |
        <array identifier>

<procedure statement> ::= <procedure identifier>
        <actual parameter list>

<goto statement> ::= GOTO <label>
```

<init statement> ::= INIT <array identifier> TO <constant list> |
INIT MEM [<expression>] TO <constant list>

<constant list> ::= <constant> { , <constant> }

<empty statement> ::= <empty>

<structured statement> ::= <compound statement> |
<conditional statement> | <repetitive statement>

<conditional statement> ::= <if statement> | <case statement>

<if statement> ::= IF <expression> THEN <statement> |
IF <expression> THEN <statement> ELSE <statement>

<case statement> ::= CASE <expression> OF <case list> END |
CASE <expression> OF <case list> ; OTHERS: <statement> END

<case list> ::= <case list element> { ; <case list element> }

<case list element> ::= <constant list> : <statement>

<repetitive statement> ::= <while statement> | <repeat statement> |
<for statement>

<while statement> ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> { ; <statement> }
UNTIL <expression>

<for statement> ::= FOR <control variable> := <for list> DO
<statement>

<control variable> ::= <variable>

<for list> ::= <initial value> TO <final value> |
<initial value> DOWNTO <final value>

<initial value> ::= <expression>

<final value> ::= <expression>

APPENDIX B: SOME USEFUL ROUTINES

This appendix lists some useful procedures and functions. You may also refer to appendix F, which gives instructions on the routines needed to interface a printer from NASCOM PASCAL.

```
{ value will convert the decimal number contained in s into }
{ a real value                                              }

FUNCTION value(s: STRING[48]): REAL;
CONST
  zero=48;  {ASCII zero}
VAR
  r,f: REAL;
  p: INTEGER;
  ch: STRING[1];
  neg,decpoint: BOOLEAN;
PROCEDURE nextchar;
BEGIN
  p:=pred(p); ch:=mid(s,p,1)
END {of nextchar};
BEGIN {value}
  f:=1; nextchar;
  IF ch='-' THEN
  BEGIN neg:=true; nextchar END;
  WHILE (ch>='0') AND (ch<='9') DO
  BEGIN
    r:=r*10.0+(ord(ch)-zero);
    IF decpoint THEN  f:=f*10.0;
    nextchar;
    IF (ch='.') AND NOT decpoint THEN
    BEGIN decpoint:=true; nextchar END;
  END;
  IF neg THEN value:=-r/f ELSE value:=r/f;
END {of value};


{ pos will return the position of the first occurrance of   }
{ the target string t in the source string s. If t does not }
{ occur within s, a zero will be returned                   }

FUNCTION pos(t,s: STRING[48]): INTEGER;
LABEL exitpos;
VAR
  ldif,lt,p: INTEGER;
BEGIN
  lt:=length(t); ldif:=length(s)-lt;
  WHILE p<=ldif DO
    p:=succ(p);
    IF mid(s,p,lt)=t THEN
    BEGIN pos:=p; GOTO exitpos END
  END;
  exitpos:
END {of pos};
```

```
{ topline will display the string s on line 16 of the }
{ NASCOM display                                        }

PROCEDURE topline(s: STRING[48]);
CONST
  toplineaddr=$BC9;  {topline address - 1}
  blank=32;          {ASCII blank}
VAR
  p: INTEGER;
BEGIN
  FOR p:=1 TO length(s) DO
  mem[p+toplineaddr]:=ord(mid(s,p,1));
  FOR p:=p TO 48 DO
  mem[p+toplineaddr]:=blank;
END {of topline};


{ hex will return the hexadecimal representation of n with }
{ d digits as a string. Up to four digits are allowed.     }

FUNCTION hex(n,d: INTEGER): STRING[4];
CONST
  hexdigits='0123456789ABCDEF';
VAR
  s: string[4];
BEGIN
  WHILE d>0 DO
  BEGIN
    s:=concat(mid(hexdigits,succ(n and $0F),1),s);
    n:=n shift -4; d:=pred(d);
  END;
  hex:=s;
END {of hex};
```

APPENDIX C: THE SYSTEM WORKSPACE

The system workspace resides between $C80 and $D00. In this area the following addresses may be of interest to the user:

C92-C93 WSP The program workspace stack pointer. When executing a program WSP will be set to point to the end address of the program. Each time a program block is activated (the main program, a procedure, or a function), WSP will move to a higher address, thus reserving memory for the variables of that program part. When exiting the block, WSP will be altered to point to its original position.

C94-C95 PMTP The highest RAM address the currently executing program is allowed to access. Should WSP move beyond PMTP, the program will break and display a runtime error (runtime error 99).

C98-C9B RNDN The last calculated random seed. By initializing these four bytes (to an abitrary selected value) the user can obtain the same random sequence each time the program is run.

The first instruction sequence in the object code of a program is a call to the initializing routine, followed by 7 bytes of parameters:

```
CD xx xx aa bb cc dd ee ff gg
```

bbaa is the end address of the program. WSP will be initialized to this value. ddcc is the highest RAM address the program is allowed to access (ddcc is obtained from MTOP (see NASCOM PASCAL Operating Manual, appendix C) during compilation). PMTP will be initialized to this value. ffee is the address to be loaded into the stack pointer (SP), and is set to $1000 by default. gg is a byte telling the runtime package where to transfer control to, in case of a runtime error, or when completing execution of the program. If gg is zero a jump to the language system will be executed, otherwise control will be transferred to NAS-SYS.

Normally, the area between $D00 and $1000 is reserved for the system stack. However, the stack pointer initialization value can be modified (using the NAS-SYS M command), allowing for other areas to be used. The following applies concerning the use of the system stack area:

A procedure or a function call consumes two bytes of stack. An active FOR loop consumes four bytes of stack. When evaluating an expression the stack will be used to store intermediate results. Hence, a comparison of two strings, may consume as much as 512 bytes, if both strings are of length 255.

During program execution the position of the stack pointer will <u>NOT</u> be checked. Thus, the you must ensure that recursive execution of procedures or functions does not enter an endless loop.

APPENDIX D: INTERNAL DATA FORMAT

In the descriptions following below the symbol 'addr' denotes the address of the first byte a variable of the described type consumes. It is this value the standard function addr returns.

Integers and booleans:

Internally NASCOM PASCAL does not differ between integers and booleans. An integer is stored as a 2's complement 16 bit number, thus consuming 2 bytes. The least significant byte is stored first, as the Z-80 standard specifies:

addr	Least significant byte.
addr+1	Most significant byte.

Reals:

A real is stored as a 40 bit mantissa and an 8 bit 2's exponent, thus consuming 6 bytes:

addr	Most significant byte of mantissa.
:	
:	
addr+4	Least significant byte of mantissa.
addr+5	2's exponent.

The exponent is in binary format with an offset of $80. Hence, an exponent of $84 means that the value of the mantissa is to be multiplied by 2^($84-$80) = 2^4 = 16. An exponent value of zero indicates that the the value of the variable is zero. The value of the mantissa can be obtained by dividing the unsigned integer, consisting of the first five bytes, by 2^40. The mantissa is always normalized, i.e. the most siginificant bit should be interpreted is a 1. However, the sign of the mantissa is stored in this bit, a 1 indicating that the value is negative, and a 0 indicating that the value is positive.

Strings:

A string will consume its maximum length plus one bytes of storage. The first byte contains the current length of the string (called n), the second byte contains the n'th character of the string, the third byte contains the n-1'th character, etc.:

addr	Current length (n).
addr+1	n'th character.
addr+2	n-1'th character.
:	
:	
addr+n	First character.

If the current length of the string is less than the maximum length, the contents of the unused bytes are unknown.

Arrays:

A component of an array uses the same internal format as a simple variable of that specific type. The components with the lowest index values will be stored first. An array with more than one dimension will be stored with the rightmost dimension increasing first. E.g. an array declared as:

```
    a: ARRAY[1..3,1..3]
```

will be stored in this order:

```
    lowest addr.    a[1,1]
                    a[1,2]
                    a[1,3]
                    a[2,1]
                    a[2,2]
                    :
                    :
    highest addr.   a[3,3]
```

APPENDIX E: EXTERNAL AND CODE SUBPROGRAMS

Declaring procedures and functions with the EXTERNAL or the CODE specification allows the user to call seperate machine code subroutines.

Parameters are transferred to the subroutine using the program workspace stack. Each parameter value is 'pushed' onto the stack, in the same order as they appear. When evaluating a function designator, memory space for the result value is reserved, before any parameters are pushed. The machine code routine may access the parameters by indexing from the value contained in WSP (see appendix C).

The format of a value parameter is described in appendix D. In the case of a variable parameter a word (2 bytes) will be pushed containing the absolute address of the first byte of the referenced variable. If the variable parameter is an array, the absolute address of the first component will be pushed.

Assume that the following function declaration has been made:

```
FUNCTION test(VAR i: INTEGER; r: REAL): STRING[16];
EXTERNAL $D00;
```

When evaluating the function designator a call will be placed to $D00, and the top of the workspace stack will be organised in the following manner:

lowest addr.	WSP-25	17 bytes reserved for the result
	:	value (of type string). These
	:	bytes are cleared at the time of
	WSP-9	the call.
	WSP-8	A word containing the address of
	WSP-9	the integer variable.
	WSP-6	Value of type real.
	:	
	:	
highest addr.	WSP-1	

The address of the first byte of the locations reserved for the result may be calculated like this:

```
WSP:    EQU  0C92H
        :
        :
        LD   HL,(WSP)
        LD   DE,-25
        ADD  HL,DE
```

When executing the code HL will point to the first byte. The address of the integer variable can be obtained by executing:

```
        LD   HL (WSP)
        LD   DE,-8
        ADD  HL,DE
             A,(HL)
        INC  HL
        LD   H,(HL)
        LD   L,A
```

As an example of user written machine code subroutines two routines are shown below which will input and output values from and to the data ports (NOTE: These routines are predeclared in NASCOM PASCAL, see chapters 8.2.5 and 7.2). In the main program the following declarations should be made:

```
CONST
  outaddr=$D00;
  inpaddr=$D0D;
:
:
PROCEDURE out(port,data: INTEGER); EXTERNAL outaddr;
FUNCTION inp(port: INTEGER): INTEGER; EXTERNAL inpaddr;
```

The machine code subroutines could be like this:

```
0001 0D00                   ORG  0D00H
0002
0003 0C92          WSP:     EQU  0C92H
0004
0005 0D00 DD2A920C OUTP:    LD   IX,(WSP)
0006 0D04 DD7EFE            LD   A,(IX-2)
0007 0D07 DD4EFC            LD   C,(IX-4)
0008 0D0A ED79              OUT  (C),A
0009 0D0C C9                RET
0010
0011 0D0D DD2A920C INP:     LD   IX,(WSP)
0012 0D11 DD4EFE            LD   C,(IX-2)
0013 0D14 ED78              IN   A,(C)
0014 0D16 DD77FC            LD   (IX-4),A
0015 0D19 C9                RET
0016
0017 0D1A                   END
```

The above routines can also be implemented using the CODE specification:

```
PROCEDURE out(port,data: INTEGER);
CODE $DD,$2A,$92,$0C,$DD,$7E,$FE,$DD,$4E,$FC,$ED,$79;

FUNCTION inp(port: INTEGER): INTEGER;
CODE $DD,$2A,$92,$0C,$DD,$4E,$FE,$ED,$78,$DD,$77,$FC;
```

It is important to note that only fully relocateable routines can be implemented using the CODE specification. Also note that the RET instruction ($C9) ending an EXTERNAL routine should not be given in a CODE routine.

All RAM between WSP and PMTP can be used as workspace by the machine code routine.

The object code produced by the compiler, as well as the runtime package routines, are fully interruptable. If interrupts are used, the interrupt service routine must save all registers to be used, and restore them before returning.

APPENDIX F: PRINTER INTERFACING

This appendix describes the routines needed to interface a printer (or actually any user defined output device) from NASCOM PASCAL.

To interface a printer you must create a machine code routine, which will perform the actions needed to output the accumulator (the A register) to the printer. In the NASCOM PASCAL Operating Manual, appendix D, is shown an example of such a routine. Note that the output routine must be fully relocateable. To implement the output driver, a CODE procedure is created, which contains the actual code, as well as the code needed to insert a jump vector to the routine in addresses $C77-$C79 (referred to as $UOUT in the NAS-SYS workspace):

```
PROCEDURE prepareprinter;
CODE $D7,nn,xx,xx,xx,.....,xx,xx,$E1,
     $22,$78,$0C,$3E,$C3,$32,$77,$0C;
```

where nn is the length (in bytes) of the output routine, and the xx's are the actual machine code. Thus, to implement the output routine shown in the Operating Manual, the following CODE procedure must be declared:

```
PROCEDURE prepareprinter;
CODE $D7,$0A,$F5,$DB,$00,$17,$30,$FB,
     $F1,$DF,$6F,$C9,$E1,$22,$78,$0C,
     $3E,$C3,$32,$77,$0C;
```

In addition to the prepareprinter procedure, two CODE procedures must be declared, which will switch on and off the printer:

```
PROCEDURE pron; CODE $D7,$02,$75,$00,$E1,$DF,$71;
PROCEDURE proff; CODE $DF,$77;
```

Below is shown a skeleton program which uses the printer interfacing routines:

```
PROGRAM usesprinter;
:
PROCEDURE prepareprinter; CODE .....;
PROCEDURE pron; CODE .....;
PROCEDURE proff; CODE .....;
:
BEGIN
  prepareprinter;
  :
  pron;
  writeln('This should be output to the printer');
  proff;
  writeln('This should be output on the screen');
  :
END.
```

The prepareprinter procedure is only called once, in the beginning of the program. When the pron procedure is called, output will be directed to the printer, until proff is called, which restores normal operation.

APPENDIX G: COMPILER ERROR MESSAGES

00 FIND address found.

01 Syntax error (e.g. missing ';' in the line above).
02 '=' expected.
03 ':' expected.
04 '[' expected.
05 ']' expected.
06 '(' expected.
07 ')' expected.
08 ',' expected.
09 '.' expected.
10 '..' expected.
11 ':=' expected.

20 Lower limit greater than upper limit in array declaration.
21 Overflow in array declaration.
22 'OF' missing in array declaration.
23 Illegal character in identifier.
24 String length cannot be zero.
25 Unknown data type.

30 Constant of type integer expected.
31 Constant of type string expected.
32 Constant of type real expected.
33 Integer constant should be within the range 0<=i<=255.

40 'BEGIN' expected.
41 'THEN' missing in if statement.
42 Case selector must be of type integer or of type string.
43 'OF' missing in case statement.
44 'END' missing in case statement.
45 'DO' missing in while statement.
46 Varible of type integer expected.
47 'TO' or 'DOWNTO' missing in for statement.
48 'DO' missing in for statement.
49 Label identifier has not been declared.
50 'TO' missing in init statement.

60 Type string not allowed here.
61 Expression of type integer expected.
62 Expression of type string expected.
63 Type mismatch in expression.
64 Unknown identifier in expression.
65 Syntax error or overflow in numeric constant, or string constant contains a carriage return.
66 String constant too long.

70 Type mismatch in assignment or parameter list.
71 Unknown variable identifier.
72 Unknown array identifier.

80 Label declared and referenced but not defined.

99 Unexpected end of source text.

APPENDIX H: RUNTIME ERROR MESSAGES

01 Floating point overflow.
02 Division by zero attempted.
03 Attempt to calculate the square root of a negative number.
04 Attempt to calculate the natural logarithm of a negative or zero number.
05 Attempt to convert a real value outside the integer range into an integer.

10 The resulting string at a concat function call is longer than 255 characters, or the position at a mid function call is less than or equal to zero.

20 An array index is outside range.

99 Workspace overflow. All available RAM has been used.

Tonder Offsettryk