Blue Label Software

Pascal

User Manual

PolyData
microcenter

INDEX

0: INTRODUCTION

Blue Label Software Pascal is a complete 12 K Pascal language system developed for use on the NASCOM range of microcomputers. The minimum computer system required to take full advantage of the BLS Pascal is:

    NAS-SYS 1 or NAS-SYS 3 monitor.
    16 K RAM (ROM version) or 32 K RAM (Tape version).

This manual describes how to operate the language system. In programming matters the user should refer to the BLS Pascal Programming Manual.

1: THE COMMAND MODE

When started as described in APPENDIX A, the system will prompt:

```
BLS Pascal version x.x
Copyright (C) 1981
Poly-Data microcenter ApS
>_
```

where x.x is the version number. The '>' character is the system prompt and indicates that the system is ready for command entry. When entering commands the following control-keys may be used:

<BS>	Backspace.
<ESC>	Clear line.
<ENTER>	Process command line.

A command consists of a command word eventually followed by a command parameter. At least one blank is required between the command word and the parameter. A command need not be written fully, but may be abbreviated to the first character, e.g. the command:

```
LOAD game
```

can be abbreviated to

```
L game
```

The operating system recognizes 11 commands, which, according to their function, can be divided into 4 groups:

1. Loading and saving sourcetexts
2. The editor
3. The compiler
4. Miscellaneous commands

2: LOADING AND SAVING SOURCETEXTS

Sourcetexts are written to tape using the NAS-SYS block format. Thus, if a checksum error occurs during a load, the user can rewind the tape and read the erroneous block once more.

2.1 The SAVE command

The SAVE command writes the current sourcetext to a cassette tape. The command line format is:

    SAVE filename   or   S filename

The filename can have any length and may contain blanks.

2.2 The LOAD command

The LOAD command reads a sourcetext from a cassette tape. The command line format is:

    LOAD filename   or   L filename

If the filename is omitted, the first file found will be loaded. Each time a file is found, the system will print:

    File filename found

When loading a sourcetext it will be placed after the current sourcetext, thus allowing the user to load several seperate subroutines. If a new sourcetext is to be loaded the current sourcetext must be deleted first e.g. by issuing a ZAP command. A LOAD command can be aborted at any time by pressing <ESC>.

2.3 The VERIFY command

The VERIFY command is identical to the LOAD command, except that the sourcetext read from the cassette tape is not loaded into memory. The purpose of the command is to check that the sourcetext can be read from the tape without error. The command line format is:

    VERIFY filename   or   V filename

If the filename is omitted, the first file found will be verified.

3: THE EDITOR

The system editor is an on-screen editor, which means that the display may be likened to a window, which can be moved about over the sourcetext. The cursor always reside within the window and its position determines where characters or lines are to be edited, deleted or inserted.

The maximum line length is 80 characters. As the display is only 48 characters wide the text window can, apart from moving up and down, move to the left and to the right. If one enters more than 48 characters on a line the cursor will not move to the next line, but instead the display will scroll to the left and the leftmost characters will 'disappear'. This may seen confusing, but when writing Pascal programs it is often preferable to have a line length greater than 48 characters. Also it enables one to take full advantage of an 80-coloumn printer.

The editor is invoked by the command line:

    EDIT   or   E

When entering the editor the cursor will be placed in the same spot it left previously, or, if it is the first activation after a cold start or a ZAP command, the display will be cleared and the cursor will be moved to the top left corner.

The editor recognizes 27 commands which uses the ASCII values between Ø1H and 1BH, i.e. the control characters. All other characters will, when entered, be inserted in the sourcetext at the current cursor position.

If all available RAM has been used, the system will return to the command mode and print:

    Overflow

The sourcetext is undamaged, but any attempt to enter more text will be denied. If possible one has to expand the buffer area, by moving MTOP to a higher adress (see APPENDIX B), before continuing.

In the description of the editor commands the following notations will be used:

CTRL/ or SHFT/ followed by a character indicates that the character is to be entered while depressing either <CTRL> or <SHIFT>.

<RI> means right arrow, <LE> means left arrow, <UP> means up arrow, and <DO> means down arrow.

3.1 Editing commands

The editing commands are used to edit the sourcetext.

<BS> Move the cursor left and blank the cursor position. If the cursor is in the first column of a line, move

it to column 79 in the line above.

<ENTER> Move the cursor to the first column in the next line and insert an empty line.

<ESC> Delete the current line and move the cursor to the first column in the line above.

SHFT/<RI> Insert blank at the cursor and move rest of line to the right. CTRL/V may be used instead of SHFT/<RI>.

SHFT/<LE> Delete character at cursor and move rest of line to the left. CTRL/U may be used instead of SHFT/<LE>.

SHFT/<DO> Insert a blank line, and move the cursor to the first column. CTRL/Z may be used instead of SHFT/<DO>

SHFT/<UP> Delete current line, and move the cursor to the first column. CTRL/Y may be used instead of SHFT/<UP>.

3.2 Cursor movement commands.

The cursor movement commands are used to move the cursor without altering the sourcetext.

<RI> Move the cursor right. If the cursor is in column 79 move it to first column in the next line. CTRL/R may be used instead of <RI>.

<LE> Move the cursor left. If the cursor is in the first column move it to column 79 in the line above. CTRL/Q may be used instead of <LE>.

<DO> Move the cursor down. If the cursor is at the bottom line scroll the display up. CTRL/T may be used instead of <DO>.

<UP> Move the cursor up. If the cursor is at the top line scroll the display down. CTRL/S may be used instead of <UP>.

CTRL/B Move the cursor to the first line of the sourcetext.

CTRL/E Move the cursor to the last line of the sourcetext.

CTRL/N Move the cursor 14 lines down.

CTRL/O Move the cursor 14 lines up.

<LF> Move the cursor to the first column in the current line. CTRL/J may be used instead of <LF>.

<CS> Move the cursor to the column after the last character on the current line. CTRL/L may be used instead of <CS>.

3.3 Block commands

The block commands affect blocks of the sourcetext. A block is marked by block markers which can be inserted using the CTRL/A command. A block command only affect the first marked block in the sourcetext. If no blocks are marked all block commands (except CTRL/A) will be ignored.

CTRL/A This command must be followed by a character. A 'B' indicates that a begin-block marker is to be inserted, an 'E' indicates that an end-block marker is to be inserted. Block markers are always inserted in front of the first character in the current line. If the current line already contains a block marker the CTRL/A command is ignored.

CTRL/D Delete the first marked block (including block markers) from the sourcetext and move the cursor to the line which contained the end-block marker.

CTRL/I Insert the first marked block (excluding block markers) before the current line. If the cursor is within the first marked block CTRL/I is ignored.

CTRL/P Print the first marked block to the user defined output routine (see APPENDIX D). The CTRL/P command must be followed by a character. 'L' indicates that the listing should include line numbers and any other character indicates that no line numbers should be issued.

3.4 Search commands

The search commands are used to locate a target string in the sourcetext.

CTRL/F Find the first occurance of a target string of maximum 40 characters. When CTRL/F is typed an empty line is inserted and, as a promt character, a right arrow is printed. The target string is entered using the same control-keys as when entering command lines. When <ENTER> is pressed the target string will disappear. If the string searched for is found the cursor will be placed at the first character. If not found, the cursor does not move. The search always starts at the next line.

CTRL/C Continue searching for the last entered target string.

3.5 Tabulator commands

CTRL/K This command is used to alter the tabulator length. The command must be followed by a character, which determines the length. The character 'A' denotes the length 1, 'B' denotes the length 2, etc., which means that the length will be the ASCII value of the character less 64. The maximum length is 63. If one selects a length of zero (by typing CTRL/K followed by '@'), the tabulator enters the indent mode. In this mode, when activating the tabualtor, the cursor will move to the position beneath the first character in the line above.

<CH> Move the cursor to the next tabulator position, or, if the tabulator is in the indent mode, to that column in the current line which corresponds to the column of the first character in the previous line. CTRL/W may be used instead of <CH>.

3.6 Other editor commands

CTRL/G This command is used to alter the <GRAPH> key function. The command must be followed by a character. An 'A' means that the <GRAPH> key is to function as an ALPHA-LOCK key: Each time it is depressed it will reverse the function of the <SHIFT> key (for the letters A-Z only). A 'G' means that the <GRAPH> key is to function normally.

CTRL/X Clear the display and return to the command mode. In addition delete all block markers.

4: THE COMPILER

The compiler is the heart of the language system. It is capable of translating the sourcetext into executable Z-80 machine code.

The compiler can be invoked in several different modes:

1) Using the COMPILE/RUN commands the object code will be placed directly into memory after the sourcetext. This method is the fastest, but also requires the most RAM space as both the sourcetext and the object code must reside in memory at the same time.

2) When the compiler is activated from a TAPE command the object code will be dumped to the cassette recorder using NAS-SYS block format. Of course this method is somewhat slower than the above, but it saves memory, and allows the user to direct the object code to any address.

3) When activated from a FIND command the compiler can be used to locate a statement in the sourcetext which corresponds to a certain address in the object code, e.g. the address of a runtime error. This mode is extremely useful for easy debugging of programs.

When locating an error the compiler will automatically invoke the editor, and place the cursor in the erroneous statement.

Let us assume that the following program has been entered:

```
VAR number: REAL;
BEGIN
  readln(numbr);
  writeln('The square root is',sqrt(number));
END.
```

The program contains an error, as the identifier number is misspelled in the readln statement. If a compilation is attempted, this is what will happen:

```
Compilation error 64              Press <SPACE>
  readln(numbr);
  writeln('The square root is',sqrt(number));
END.
```

To indicate the error the cursor is placed at the 'n' in the misspelled identifier. When the spacebar is pressed the top line will be cleared and the user may edit the sourcetext in the same way as usually.

If the buffer overflows during a compilation the compiler will abort, and print:

```
Overflow
```

If it is possible the user must expand the buffer area, using one of two methods:

1) If there is more RAM available MTOP should be moved to a

higher address (see APPENDIX B).

2) If the compiler was activated from a COMPILE or a RUN command, the TAPE command should be used instead.

4.1 The COMPILE command.

Activating the compiler from a COMPILE command will place the object code directly into memory in succession of the sourcetext. The command line format is:

```
COMPILE            or            C
```

When the command line is entered the compiler will print:

```
Compiling
```

If no errors occur the following will be printed when the compilation is completed:

```
Compiling  OK
Text: $aaaa $bbbb <xxxxx>
Code: $cccc $dddd <yyyyy>
```

aaaa and bbbb are the start and end address of the sourcetext (in hex) and xxxxx is the size in bytes. cccc, dddd, and yyyyy are the corresponding parameters of the object code.

4.2 The RUN command

This command is used to execute a program. The command line format is:

```
RUN            or            R
```

If no object code is present the compiler will be activated prior to executing the program. Assuming no errors occured during compilation, or if the object code was already present, the system will print:

```
Running
```

and control will be transferred to the program. When the program ends the control will be transferred back to the language system.

If a runtime error occurs during program execution the system will print:

```
Runtime error xx at $nnnn
```

and control will be transferred to the language system (or to NAS-SYS if the program was compiled using the TAPE command; see chapter 4.3). xx is the error number and nnnn is the error address (in hex). The error address is not an absolute address but an offset address from the start address. By issuing a FIND command (see chapter 4.4) the user may locate the statement that caused the runtime error.

4.3 The TAPE command

When activating the compiler from a TAPE command the object code will be dumped to the cassette recorder using NAS-SYS block format. The command line format is:

```
TAPE nnnn            or            T nnnn
```

where nnnn is the absolute start address (in hex) of the program. If nnnn is omitted the system will choose $2180 ($1000 for the ROM version) as start address ($2180 is the end address of the runtime package in the tape version). When compilation is complete the system will print:

```
xxxx End.
```

where xxxx is the end address of the object code.

When the tape is loaded (using the R command in NAS-SYS) the program can be executed by entering the NAS-SYS command Exxxx. The program requires the runtime package to be present between $1000 and $2180 ($D000 and $E180 for the ROM version). However the rest of the language system is not needed during program execution. Thus, when a program is thoroughly tested it can be compiled using the TAPE command (and, if you are using the tape version, merged to the runtime package) to form a directly executeable object code.

4.4 The FIND command

The FIND command is used to locate a statement in the sourcetext which corresponds to an offset address in the object code. In this mode the compiler will generete no object code. The command line format is:

```
FIND nnnn            or            F nnnn
```

where nnnn is the offset address. The offset address is calculated by subtracting the start address from the address one wishes to locate. If a program starts at $2180 the command:

```
FIND 115
```

will locate the statement, which origins at $2295. If nnnn is omitted the address of the last runtime error is substituted. When activated from a FIND command the compiler will print:

```
Searching
```

If the offset address is reached during compilation the editor will be invoked and the top line will display:

```
Compilation error 00            Press <SPACE>
```

The cursor will be placed at or just after the relevant text. When the spacebar is pressed the top line will be cleared and the user may edit the sourcetext in the same way as usual. If the offset address is not reached the system will print:

```
Searching ?
```

5: MISCELLANEOUS COMMANDS

5.1 The MEMORY command

This command displays the start and end addresses and the size of the sourcetext, and the same parameters of the object code if it is present. The command line format is:

MEMORY or M

The command will print:

Text; $aaaa $bbbb <xxxxx>

and, if the object code is present, in addition:

Code: $cccc $dddd <yyyyy>

aaaa and bbbb are the start and end address of the sourcetext (in hex) and xxxxx is the size in bytes. cccc, dddd, and yyyyy are the corresponding parameters of the object code.

5.2 The ZAP command

This command deletes the sourcetext as well as the object code. The command line format is:

ZAP

NOTE: To secure that the ZAP command is not invoked accidentally, command word abbreviation does not apply here.

5.3 The QUIT command

This command transfers the control to NAS-SYS. The command line format is:

QUIT or Q

The language system may be warmstarted later, using the method described in appendix A.

APPENDIX A: SYSTEM STARTUP

Tape version:

The BLS Pascal tape version is recorded at 1200 baud using the NAS-SYS block format. The tape is loaded using the R command. The system is coldstarted by entering:

E2180 aaaa

where aaaa is the highest RAM address the system is allowed to access. If aaaa is omitted all available RAM will be used.

The system is warmstarted by entering:

E2182

ROM version:

The system is coldstarted by entering:

J aaaa

where aaaa is the highest RAM address the system is allowed to access. If aaaa is omitted all available RAM will be used.

The system is warmstarted by entering:

Z

APPENDIX B: SYSTEM WORKSPACE

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

```
C80-C81   MTOP    The highest RAM address the system is allowed
                  to access.
C82-C83   EOFP    The end address of the sourcetext.
C84-C85   PEND    The end address of the object code.
```


APPENDIX C: MEMORY MAPS

Tape version:

```
0C80 +--------------------------+
     !    system workspace      !
0D00 +--------------------------+
     !      system stack        !
1000 +--------------------------+
     !     runtime package      !
2180 +--------------------------+
     !    operating system      !
2980 +--------------------------+
     !        compiler          !
4000 +--------------------------+
     !       sourcetext         !
EOFP +--------------------------+
     !      object code         !
PEND +--------------------------+
     !   program workspace      !
MTOP +--------------------------+
```

ROM version:

```
0C80 +--------------------------+
     !    system workspace      !
0D00 +--------------------------+
     !      system stack        !
1000 +--------------------------+
     !       sourcetext         !
EOFP +--------------------------+
     !      object code         !
PEND +--------------------------+
     !   program workspace      !
MTOP +--------------------------+
     !                          !
D000 +--------------------------+
     !     runtime package      !
E180 +--------------------------+
     !    operating system      !
E980 +--------------------------+
     !        compiler          !
FFFF +--------------------------+
```

APPENDIX D: THE USER DEFINED OUTPUT ROUTINE

When using the editor command CTRL/P, output will be directed to the NAS-SYS user routine. A jump vector to this routine should be placed in $UOUT ($C77-$C79). Listed below is a routine to control a printer connected to the serial port with a BUSY line (active high) connected to bit 7 of port 0:

```
0D00                      ORG  0D00H

0D00 F5        PRINT:     PUSH AF
0D01 DB00      P1:        IN   A,(0)
0D03 17                   RLA
0D04 30FB                 JR   C,P1
0D06 F1                   POP  AF
0D07 DF6F                 SCAL 6FH
0D09 C9                   RET

0D10                      END
```


APPENDIX E: COMMAND SUMMARY

Command mode:

SAVE filename	Write sourcetext to cassette.
LOAD filename	Read sourcetext from cassette.
VERIFY filename	Verify.
EDIT	Activate editor.
COMPILE	Compile sourcetext.
RUN	Execute object code.
TAPE nnnn	Compile and dump object code to cassette.
FIND nnnn	Locate address in sourcetext.
MEMORY	Display program parameters.
ZAP	Delete sourcetext and object code.
QUIT	Return to NAS-SYS.

The editor:

<BS>	Backspace.
<ENTER>	Move cursor down and insert line.
<ESC>	Delete line and move cursor up.
SHFT/<RI>	Insert blank.
SHFT/<LE>	Delete character.
SHFT/<DO>	Insert line.
SHFT/<UP>	Delete line.
<RI>	Move cursor right.
<LE>	Move cursor left.
<DO>	Move cursor down.
<UP>	Move cursor up.
CTRL/B	Move cursor to beginning of sourcetext.
CTRL/E	Move cursor to end of sourcetext.
CTRL/N	Move cursor down 14 lines.
CTRL/O	Move cursor up 14 lines.
<LF>	Move cursor to first coloumn.
<CS>	Move cursor to last character.
CTRL/A (B,E)	Insert block marker.
CTRL/D	Delete first marked block.
CTRL/I	Insert first marked block.
CTRL/P (L,?)	Print first marked block.
CTRL/F	Find target string.
CTRL/C	Continue searching.
CTRL/K (char)	Alter tabulator length.
<CH>	Move cursor to next tabulator position.
CTRL/G (A,G)	Alter <GRAPH> key function.
CTRL/X	Return to command mode.

Blue Label Software

Pascal

Programming Manual

PolyData

INDEX

0: INTRODUCTION

The Blue Label Software Pascal Language System is meant to offer an alternative to BASIC. Not only will the user gain execution speed, but he can also practise better programming techniques, as Pascal is far more versatile than BASIC.

As the BLS 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 BLS 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 BLS 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: Ø 1 2 3 4 5 6 7 8 9
Symbols: + - * / = < > ( ) [ ] . , ; : ' { }

The compiler does not differ between capital and non capital 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	EXTERNAL	OTHERS
ARRAY	FOR	PROCEDURE
BEGIN	FUNCTION	PROGRAM
BOOLEAN	GOTO	REAL
CASE	IF	REPEAT
CODE	INIT	SHIFT
DIV	INTEGER	STRING
DO	LABEL	THEN
DOWNTO	MOD	TO
ELSE	NOT	UNTIL
END	OF	VAR
EXOR	OR	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	left	read
addr	ln	readln
arctan	load	right
call	maxint	round
chr	mem	save
concat	mid	sin
cos	odd	sqr
empty	ord	sqrt
exp	out	succ
false	pi	true
frac	plot	trunc
inp	point	write
int	pred	writeln
keyboard	random	

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
```

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:

```
'BLS 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. BLS 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 BLS 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 BLS 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,' ',surname)
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 or CODE 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 and 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=Ø: 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 Ø<=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 1Ø.

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 or CODE 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-8Ø machine code, directly following the CODE symbol. In the case of EXTERNAL and 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 procedure (see chapter 7.1.2).

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>=Ø, and the least whole number greater than or equal to x for x<Ø.
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 and returns the 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

BLS 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 to the user:

<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. Leading spaces are allowed. The numeric value 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 the user is to re-enter the value.

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 the user depresses a key. 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>d.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 the user may 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: BLS PASCAL SYNTAX

The syntax of BLS 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> .

<program heading> ::= <empty> | PROGRAM { <character> } ;

<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> } E <scale factor> |
        <unsigned integer> E <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/code specification> ;

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

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

<parameter group> ::= <variable declaration>

<external/code specification> ::= <external specification> |
        <code specification>

<external specification> ::= EXTERNAL <constant>

<code specification> ::= CODE <constant> { , <constant> }

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

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

```
{ 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;

BEGIN
  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;
```

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 5 bytes of parameters:

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

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 BLS Pascal User Manual, appendix C) during compilation). PMTP will be initialized to this value. ee 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 ee is zero a jump to the language system will be executed, otherwise control will be transferred to NAS-SYS.

The area between $D00 and $1000 is reserved for the system stack. Upon initialization the stack pointer will be loaded with $1000. 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 not be checked. Thus, the user must be shure that recursive execution of procedures or functions does not enter a loop with no exits.

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 BLS 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: MACHINE CODE SUBROUTINES

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 $DØØ;
```

When evaluating the function designator a call will be placed to $DØØ, 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  ØC92H
        :
        :
        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
        LD   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 BLS Pascal, see chapters 8.2.5 and 7.2). In the main program the following declarations should be made:

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

The machine code subroutines could be like this:

```
0D00                     ORG   0D00H

          =0C92   WSP:   EQU   0C92H

0D00 DD2A920C OUTP:      LD    IX,(WSP)
0D04 DD7EFE              LD    A,(IX-2)
0D07 DD4EFC              LD    C,(IX-4)
0D0A ED79                OUT   (C),A
0D0C C9                  RET

0D0D DD2A920C INP:       LD    IX,(WSP)
0D11 DD4EFE              LD    C,(IX-2)
0D14 ED78                IN    A,(C)
0D16 DD77FC              LD    (IX-4),A
0D19 C9                  RET

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 must not be used in the case of 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 using interrupts, the interrupt service routine must save all registers to be used on the stack.

APPENDIX E: BENCHMARK TESTS

On the following pages the 15 Pascal benchmark tests, as proposed in Personal Computer World december 1980 issue, are listed. The timings obtained using a NASCOM 2 (Z-80 microprocessor, 4 MHz 1 waitstate), are listed below, and, for comparison, the corresponding timings obtained on a Heathkit H-11A (LSI 112 16 bit processor), and on an APPLE 2 (6502 microprocessor), both running UCSD Pascal. All timings are listed in seconds:

TEST	BLS Pascal	H-11A	APPLE 2
magnifier	0.8	3.9	6.4
forloop	8.6	42.8	74.3
whileloop	23.0	40.1	70.0
repeatloop	20.8	35.0	63.3
litteralassign	11.7	50.0	88.5
memoryaccess	15.1	52.0	91.0
realarithmetic	59.8	61.7	93.0
realalgebra	58.5	40.6	83.4
vector	62.2	102.9	203.3
equalif	24.3	66.8	116.7
unequalif	24.2	65.8	115.3
noparameters	6.8	26.4	50.2
value	12.5	29.3	54.4
reference	12.1	29.7	55.3
maths	65.3	25.8	66.0

It should be noted that UCSD Pascal provides only 6+ significant digits when operating on reals, while BLS Pascal provides 11+ significant digits.

```
PROGRAM magnifier;
VAR k: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO;
END.

PROGRAM forloop;
VAR j,k: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO FOR j:=1 TO 10 DO;
END.

PROGRAM whileloop;
VAR j,k: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO
  BEGIN
    j:=1; WHILE j<=10 DO j:=j+1
  END
END.

PROGRAM repeatloop;
VAR j,k: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO
  BEGIN
    j:=1; REPEAT j:=j+1 UNTIL j>10;
  END;
END.

PROGRAM litteralassign;
VAR j,k,l: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO FOR j:=1 TO 10 DO l:=0
END.

PROGRAM memoryaccess;
VAR j,k,l: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO FOR j:=1 TO 10 DO l:=j
END.

PROGRAM realarithmetic;
VAR k: INTEGER; x: REAL;
BEGIN
  FOR k:=1 TO 10000 DO x:=k/2*3+4-5;
END.

PROGRAM realalgebra;
VAR k: INTEGER; x: REAL;
BEGIN
  FOR k:=1 TO 10000 DO x:=k/k*k+k-k;
END.

PROGRAM vector;
VAR k,j: INTEGER; matrix: ARRAY[0..10] OF INTEGER;
BEGIN
  matrix[0]:=1;
  FOR k:=1 TO 10000 DO FOR j:=1 T0 10 DO
  matrix[j]:=matrix[j-1]
```



```
END.

PROGRAM equalif;
VAR j,k,l: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO FOR j:=1 TO 10 DO
  IF j<6 THEN l:=1 ELSE l:=0
END.

PROGRAM unequalif;
VAR j,k,l: INTEGER;
BEGIN
  FOR k:=1 TO 10000 DO FOR j:=1 TO 10 DO
  IF j<2 THEN l:=1 ELSE l:=0
END.

PROGRAM noparameters;
VAR j,k: INTEGER;
PROCEDURE none5; BEGIN j:=1 END;
PROCEDURE none4; BEGIN none5 END;
PROCEDURE none3; BEGIN none4 END;
PROCEDURE none2; BEGIN none3 END;
PROCEDURE none1; BEGIN none2 END;
BEGIN
  FOR k:=1 TO 10000 DO none1;
END.

PROGRAM value;
VAR j,k: INTEGER;
PROCEDURE value5(i: INTEGER); BEGIN i:=1 END;
PROCEDURE value4(i: INTEGER); BEGIN value5(i) END;
PROCEDURE value3(i: INTEGER); BEGIN value4(i) END;
PROCEDURE value2(i: INTEGER); BEGIN value3(i) END;
PROCEDURE value1(i: INTEGER); BEGIN value2(i) END;
BEGIN
  FOR k:=1 TO 10000 DO value1(j)
END.

PROGRAM reference;
VAR j,k: INTEGER;
PROCEDURE refer5(VAR i: INTEGER); BEGIN i:=1 END;
PROCEDURE refer4(VAR i: INTEGER); BEGIN refer5(i) END;
PROCEDURE refer3(VAR i: INTEGER); BEGIN refer4(i) END;
PROCEDURE refer2(VAR i: INTEGER); BEGIN refer3(i) END;
PROCEDURE refer1(VAR i: INTEGER); BEGIN refer2(i) END;
BEGIN
  FOR k:=1 TO 10000 DO refer1(j)
END.

PROGRAM maths;
VAR k: INTEGER; x,y: REAL;
BEGIN
  FOR k:=1 TO 1000 DO
  BEGIN
    x:=sin(k); y:=exp(x)
  END
END.
```

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 interval 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 that 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.

The Blue Label Software Pascal Language System, version _______ , serial number _______ , is copyrighted and all rights are reserved by Poly-Data microcenter ApS.

Name and address:

____

____

____

____

hereby agrees not to sell, rent, or otherwise distribute the above mentioned program, or any part hereof, in any form, without prior written consent of Poly-Data microcenter ApS.

SIGNED AND AGREED:

Signature: ______________________________ Date: ____________

Dealer:

ELECTROVALUE LTD.
28 St. Judes Road, Englefield Green
EGHAM, SURREY TW20 0HB
Telephone: Egham 33603 Telex: 264475
Reg'd in England No. 1047769
VAT Registration No. 211 5797 71

NASCOM

Software

NASCOM PASCAL Language System

NASCOM PASCAL is a complete 12K Pascal language system, designed specially for the NASCOM 1 or 2 with NAS-SYS 1 or NAS-SYS 3 monitor. NASCOM PASCAL is based on the high-level programming language Pascal, widely recognized as the programming language of the future.

NASCOM PASCAL basically consists of a runtime package (4.5K), a control program (0.5K), an on-screen editor (1.5K) and a compiler (5.5K).

The compiler is a one pass compiler which directly produces Z–80 machine code. This architecture not only provide very fast compilation (2000 lines pr. minute), but also results in program execution speeds 3 to 20 times faster than equivalent BASIC programs.

In 5.5K only it is, of course, not possible to implement standard Pascal. The NASCOM PASCAL subset does not support user defineable types, sets, and file types. However, all basic statement constructions are retained, and procedures/functions are fully recursive and support both variable and value 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.

Briefly, the NASCOM PASCAL subset includes:

Statements:	BEGIN .. END; IF .. THEN .. ELSE; WHILE .. DO; FOR .. TO/DOWNTO .. DO; REPEAT .. UNTIL; GOTO; CASE .. OF .. OTHERS; INIT .. TO; Assignment (:=); Procedure statements
Data types:	REAL; INTEGER; STRING; BOOLEAN; ARRAY
Constants:	MAXINT; PI; TRUE; FALSE; EMPTY
Operators:	+ — * / DIV MOD SHIFT AND OR EXOR = <> > < >= <=
Procedures:	WRITE; WRITELN; READ; READLN; LOAD; SAVE; CALL; SCREEN; PLOT; OUT
Functions:	ABS; SQR; SQRT; SIN; COS; ARCTAN; LN; EXP; INT; FRAC; SUCC; PRED; ODD; TRUNC; ROUND; ORD; CHR; LENGTH; MID; LEFT; RIGHT; CONCAT; RANDOM; ADDR; POINT; INP; KEYBOARD
Declarations:	LABEL; CONST; VAR; PROCEDURE; FUNCTION

Reals provide 11.5 significant digits. Integers are within the range –32768 to 32767 (16 bits). Strings can be up to 255 characters long. Arrays may have any number of dimensions, and can be of any of the types INTEGER, REAL, BOOLEAN, or STRING. Constants may be presented in either decimal or hex notation. User written machine code subroutines are supported using procedures/functions declared as EXTERNAL or CODE. Thus, a machine code subprogram is treated by the compiler as a normal procedure or function. The procedure WRITELN allows for numbers or strings to be output using a specific format.

NASCOM PASCAL Language System

The compiler can be invoked in several different modes. The COMPILE and the RUN commands will load the object code directly into memory after the source text, allowing you to execute your programs almost immediately. The TAPE command will output the object code to the tape recorder, using NAS-SYS block format. When the compiler is invoked from a FIND command it will locate the statement that caused the most recent runtime error. The object code produced by NASCOM PASCAL requires only the runtime package to be present in memory during execution. Once a program is tested it can be merged to the runtime package to form a directly executeable machine code program.

The NASCOM PASCAL editor is a very powerful on-screen editor. Apart from being able to scroll up and down over the text, the display can scroll to the left and to the right, allowing lines to be up to 80 characters in length. Blocks can be marked and deleted or copied to any other location in the source text. A build-in tabulator eases source text entry, and the GRAPH key can be selected to operate as a CAPS-LOCK key, which, when depressed, reverts the SHIFT key function. The find command will locate any target string in the source text. Optionally, the continue command can be used to find further occurrances. The editor reacts to 27 different commands, all of which are control-characters, i.e. characters produced by depressing CTRL and another key, or by depressing ENTER, BS, ESC, etc. This greatly simplifies command entry.

Program texts can be saved using file names of up to 60 characters. When a program is loaded it is merged to the end of the current program, thus allowing you to maintain a library of separate subroutines.

NASCOM PASCAL is meant to offer an alternative to BASIC. Programs written in NASCOM PASCAL will execute much faster than their BASIC counterparts, and better programming techniques can be practised, as Pascal is far more versatile than BASIC. Compared to other Pascals the NASCOM PASCAL offers a lot more features in the same amount of memory, and shows Benchmark timings comparable to those obtained on 16-bit mini computers.

NASCOM PASCAL is available in two versions: A tape version, which resides in memory from 1000H to 3FFFH, and an EPROM version, which is situated between D000H and FFFFH. The EPROM version is supplied in 6 2716 EPROMs, together with instructions to fit the EPROMs on the NASCOM 2 main PCB by paging the top 12K of memory into two banks (NASCOM PASCAL in one bank and NASCOM BASIC plus an assembler in another bank). The documentation consists of two printed manuals: An Operating Manual (17 pages), which describes how to operate the system, and a Programming Manual (40 pages), which describes the NASCOM PASCAL subset.

Lucas Logic

Lucas Logic Limited
Welton Road Wedgnock Industrial Estate
Warwick CV34 5PZ

Tel: Warwick (0926) 497733 Telex: 312333

Publication No. 4117 Printed in England 10M/482/DL