MITTEILUNG NR. 37

# PASCAL FOR THE DECSYSTEM-10

BY

E. KISICKI AND H.-H. NAGEL

IFI - HH - M - 37 / 76

NOVEMBER 1976

(UPL)PASCAL.INF

# TYMCOM X

# PASCAL

# USER'S MANUAL

12-19-78

RUN (UPL) PASCAL.INF
to produce these notes

TYMSHARE PASCAL-10 NOTES.
D. E. GRIMES.
PHONE: (408)446-7173.
LAST REVISION: 22 MARCH 1979

## 1.0 HOW TO EXECUTE THE COMPILER.

PASCAL CAN BE INVOKED EITHER DIRECTLY:

```
.RUN (UPL)PASCAL
```

OR VIA THE CONCISE COMMAND LANGUAGE:

```
.COMPILE (OR .LOAD OR .EXECUTE) FAUX
```

do all pascal prog programs have & use this name?

WHEN YOU HAVE A FILE NAMED FAUX.PAS IN YOUR DIRECTORY. UNFORTUNATELY, THE LOCAL COMMAND LINE INTERPRETER DOES NOT AUTOMATICALLY RECOGNIZE THE ".PAS" EXTENSION. YOU CAN EASILY "CUSTOMIZE" IT, HOWEVER, WITH THE FOLLOWING COMMANDS:

```
.CTEST SETNON (UPL)PASCAL OUT=REL
```

one time only

THIS TEMPORARILY ADDS ".PAS" TO THE TABLE OF KNOWN EXTENSIONS THAT THE COMMAND LINE INTERPRETER CONSULTS WHEN PROCESSING A COMPIL-CLASS COMMAND AND TELLS IT WHAT PROCESSOR TO INVOKE AND WHAT EXTENSION TO GIVE ITS OUTPUT FILE. THE ADDITION WILL GO AWAY WHEN YOU LOG OFF UNLESS YOU FOLLOW THE ABOVE COMMAND WITH:

```
.CTEST MAKINI
```

to permanent

THIS BUILDS THE TEMPORARY TABLE OF KNOWN EXTENSIONS INTO A FILE NAMED RPG.INI IN YOUR DIRECTORY. THE COMMAND LINE INTERPRETER USES THIS FILE, IF IT EXISTS, TO INITIALIZE THE TABLE EACH TIME YOU LOG IN.

## 2.0 HOW TO EXECUTE THE CROSS-REFERENCER.

TO CROSS-REFERENCE AND REFORMAT YOUR PASCAL PROGRAM WITHOUT COMPILING IT, TYPE

```
.RUN (UPL)CROSS <CORE>
```

THE CROSS-REFERENCER WILL PROMPT FOR THE NAMES OF YOUR SOURCE, REFORMATTED SOURCE, AND CROSS-REFERENCE FILES. A <CORE> OF 50 SUFFICES TO CROSS-REFERENCE THE COMPILER; SMALLER PROGRAMS WILL NEED CORRESPONDINGLY LESS.

## 3.0 REFERENCES.

1. JENSEN, K., AND N. WIRTH, PASCAL USER MANUAL AND REPORT (SECOND EDITION); SPRINGER-VERLAG, NEW YORK, 1978. THIS IS THE DEFINING DOCUMENT FOR PASCAL.

2. (UPL)PASCAL.MAN. A 55-PAGE ADDENDUM TO THE ABOVE DOCUMENT, DESCRIBING THE PDP-10 IMPLEMENTATION. SPOOL IT ON TTY PAPER WITHOUT HEADINGS FOR THE BEST APPEARANCE.

3. PASCAL NEWS. A QUARTERLY PUBLICATION OF THE PASCAL USERS' GROUP. ANYONE CAN JOIN; DUES ARE $6 PER YEAR. CONTACT:

   PASCAL USERS' GROUP, %ANDY MICKEL
   UNIVERSITY COMPUTER CENTER: 227 EX
   208 SE UNION STREET
   UNIVERSITY OF MINNESOTA
   MINNEAPOLIS, MN 55455

## 4.0 LOCAL MODIFICATIONS.

1. THE CLOCK BUILT-IN FUNCTION RETURNS ACCUMULATED TRU'S, IN UNITS OF 10-4 TRU, RATHER THAN CPU MILLISECONDS.

2. THE 'P' COMPILER OPTION, WHICH FORMERLY WAS EQUIVALENT TO 'D', NOW CAUSES A PAGE SKIP IN THE LISTING FILE AFTER THE LINE IN WHICH IT APPEARS. NOTE THAT, ACCORDING TO THE SYNTAX FOR COMPILER OPTIONS, THE 'P' MUST BE FOLLOWED BY '+', '-', OR A DIGIT; ALL OF WHICH ARE IGNORED. NOTE ALSO THAT PASCAL LISTINGS PRESERVE THE STRUCTURE OF SOURCE FILES WITH STANDARD LINE NUMBERS AND PAGE MARKS, AS PRODUCED BY EDIT10.

3. THE 'U' COMPILER OPTION (AND THE 'CARD' COMPILER SWITCH) NOW ACCEPT AN OPTIONAL NUMERIC ARGUMENT. POSSIBLE FORMS ARE:

| SWITCH | OPTION | COLUMNS READ |
|---|---|---|
| CARD | U+ | 72 |
| NOCARD | U- | 132 |
| CARD:N | UN | N = 0 -> 72 |
| | | 0 < N <= 132 -> N |
| | | N > 132 -> 132 |

4. THE RESET PROCEDURE NORMALLY HAS THE EFFECT OF A REWIND FOLLOWED BY A GET. IF THE DEVICE IS TTY, HOWEVER, THE GET IS SUPPRESSED AND INPUT OF A NULL LINE IS SIMULATED: AFTER THE RESET, EOF IS FALSE, EOLN IS TRUE, AND THE BUFFER VARIABLE IS A BLANK.

## 5.0 CHANGES INTRODUCED IN THE 22 MARCH 1979 VERSION.

### 5.1 Bug Fixes.

OBSCURE BUGS HAVE BEEN FIXED IN MAX/MIN, NEW/DISPOSE, AND PACK/UNPACK.

### 5.2 New Predeclared Routines.

THE FOLLOWING BUILT-IN ROUTINES ARE NOW AVAILABLE:

1. FUNCTION DEC_DATE: INTEGER; RETURNS THE DATE IN DEC STANDARD FORMAT:
   [(YEAR-1964)*12 + (MONTH-1)]*31 + (DAY-1).

2. PROCEDURE SAVERANDOM(VAR I: INTEGER); SETS ITS ARGUMENT TO THE LAST RANDOM NUMBER (INTERPRETED AS AN INTEGER) THAT HAS BEEN GENERATED BY FUNCTION RANDOM.

3. PROCEDURE SETRANDOM(I: INTEGER); I MUST BE A NON-NEGATIVE INTEGER < 2^31. THE STARTING VALUE OF THE RANDOM FUNCTION IS SET TO
   I, IF I IS NONZERO;
   ITS NORMAL STARTING VALUE, IF I IS ZERO.

### 5.3 Listing Format.

1. EACH ERROR MESSAGE NOW POINTS TO THE PREVIOUS ERROR MESSAGE, IF ANY.

2. THE LOCATION COUNTER IS ALWAYS LISTED, WHETHER OR NOT THE CODE OPTION IS SELECTED.

### 5.4 Case Error Checking.

IF THE CASE SELECTOR VALUE MATCHES NO CASE LABEL, NO OTHERS: CLAUSE EXISTS, AND RUN-TIME CHECKING IS ENABLED, A RUN-TIME ERROR OCCURS. FORMERLY, THERE WAS ALWAYS AN IMPLICIT NULL OTHERS: CLAUSE.

### 5.5 Type Compatibility.

TWO PACKED ARRAY TYPES ARE NOW COMPATIBLE ONLY IF THEIR COMPONENTS OCCUPY THE SAME AMOUNT OF STORAGE. SINCE ARRAY ASSIGNMENTS ARE DONE BY BLOCK COPIES, THE RESULT IS GARBAGE OTHERWISE.

## 6.0 NOTES ON FILE HANDLING.

THESE NOTES ATTEMPT TO CLARIFY CHAPTERS 2 AND 4 OF PASCAL.MAN. PASCAL PROVIDES THREE LEVELS OF AUTOMATIC OPENING AND NAME SUBSTITUTION:

### 6.1 PREDECLARED FILES.

YOUR PASCAL PROGRAM EFFECTIVELY CONTAINS THE FOLLOWING DECLARATION:

```
VAR INPUT, OUTPUT, TTY, TTYOUTPUT: TEXT;
```

THUS YOU NEED NOT (AND SHOULD NOT) DECLARE THESE FILES EXPLICITLY. NOTE THAT THE PSEUDO FILE VARIABLE TTY ACTUALLY REFERS TO TWO FILES, ONE FOR INPUT AND ONE FOR OUTPUT; THE COMPILER TRANSLATES OUTPUT OPERATIONS ON TTY TO THE SAME OPERATION ON TTYOUTPUT, WHICH YOU NEVER SEE.

YOUR PASCAL PROGRAM EFFECTIVELY BEGINS WITH THE FOLLOWING STATEMENTS:

```
RESET(TTY); REWRITE(TTYOUTPUT);
```

IN ADDITION, IF THE PROGRAM HAS NO PARAMETERS (SEE BELOW), THE FOLLOWING STATEMENTS ARE EFFECTIVELY PRESENT:

```
RESET(INPUT); REWRITE(OUTPUT);
```

(THIS EXPLAINS WHY THE COMPILER, WHICH IS ITSELF A PASCAL PROGRAM WITH NO PARAMETERS, OPENS A NULL FILE NAMED OUTPUT IN YOUR DIRECTORY.)

I/O via TTY needs no correspondg file name in header -

### 6.2 FILES NAMED IN THE PROGRAM HEADER.

THESE FILES ARE NOT AUTOMATICALLY DECLARED; YOU MUST INCLUDE A DECLARATION FOR THEM. FOR EXAMPLE:

```
PROGRAM EXAMPLE(INFILE*, OUTFILE);
...
VAR INFILE, OUTFILE: FILE OF INTEGER;
```

PASCAL WILL, HOWEVER, OPEN THESE FILES FOR YOU AT THE SAME TIME IT OPENS THE PREDECLARED FILES, AND ALSO ALLOWS YOU TO SPECIFY THE NAME OF THE PHYSICAL FILE WITH WHICH THE FILE VARIABLE IS TO BE ASSOCIATED. IT LOOKS FOR THIS INFORMATION IN THREE PLACES, IN THIS ORDER:

1. A TEMPCORE FILE WHOSE NAME IS THE FIRST THREE CHARACTERS OF THE PROGRAM NAME (EXA IN OUR EXAMPLE).

2. A DISK FILE WHOSE NAME IS THE FIRST SIX CHARACTERS OF THE PROGRAM NAME WITH THE EXTENSION CMD (EXAMPL.CMD, IN OUR EXAMPLE).

3. THE TERMINAL, WITH PROMPTS. IN OUR EXAMPLE:

```
INFILE    = (YOUR RESPONSE)
OUTFILE   = (YOUR RESPONSE)
```

THIS MEANS THAT YOUR PROGRAM CAN GET ITS PARAMETERS FROM THE COMMAND LINE INTERPRETER JUST AS THE COMPILER DOES.

NOTE

A WORD OF WARNING: PASCAL DECIDES WHETHER TO OPEN THE FILE FOR READING OR WRITING (RESET OR REWRITE) BY THE PRESENCE OR ABSENCE OF AN ASTERISK FOLLOWING THE FILE NAME IN THE PROGRAM HEADER. IF YOU ACCIDENTALLY OMIT THE ASTERISK, PASCAL WILL DISCARD YOUR INPUT FILE WITHOUT WARNING.

## 6.3 ALL OTHER FILES.

IF A FILE DOES NOT HAVE ONE OF THE "MAGIC" NAMES INPUT, OUTPUT, OR TTY, AND IS NOT NAMED IN THE PROGRAM HEADER, YOU MUST BOTH DECLARE IT AND OPEN IT EXPLICITLY. YOU CAN STILL SPECIFY THE NAME OF THE PHYSICAL FILE WITH WHICH IT IS TO BE ASSOCIATED AT THE TIME OF THE OPEN BY MEANS OF OPTIONAL PARAMETERS TO RESET/REWRITE.

FINALLY, NOTE THAT ALL THREE KINDS OF FILE VARIABLES CAN BE REOPENED ON THE SAME OR DIFFERENT PHYSICAL FILES BY SUBSEQUENT CALLS TO RESET/REWRITE.

-TYPE (UPL) PASCAL.MAN

# Tymnet Pascal-10 Notes.

Last revision: 6 October 78

Pascal can now be invoked either directly:

```
@RUN(UPL)PASCAL
```

or via the Concise Command Language:

```
.COMPILE (or .LOAD or .EXECUTE) FAUX
```

when you have a file named FAUX.PAS in your directory. Unfortunately, the local command line interpreter does not automatically recognize the ".PAS" extension. You can easily "customize" it, however, with the following commands:

```
@CTEST SETNON (UPL)PASCAL OUT=REL
```

This temporarily adds ".PAS" to the table of known extensions that the command line interpreter consults when processing a COMPIL-class command and tells it what processor to invoke and what extension to give its output file. The addition will go away when you log off unless you follow the above command with:

```
@CTEST MAKINI
```

This builds the temporary table of known extensions into a file named RPG.INI in your directory. The command line interpreter uses this file, if it exists, to initialize the table each time you log in.

Temporary restrictions.

(none)

Local peculiarities.

- The CLOCK built-in function returns accumulated TRU's, in units of 10^-4 TRU, rather than CPU milliseconds.

- The 'P' compiler option, which formerly was equivalent to 'D', now causes a page skip in the listing file after the line in which it appears. Note that, according to the syntax for compiler options, the 'p' must be followed by '+', '-', or a digit, all of which are ignored. Note also that Pascal listings preserve the structure of source files with standard line numbers and page marks, as produced by EDIT10.

- The 'U' compiler option (and the 'CARD' compiler switch) now accept an optional numeric argument. Possible forms are:

| switch | option | columns read |
|---|---|---|
| CARD | U+ | 72 |
| NOCARD | U- | 132 |
| CARD:n | Un | n = 0 => 72 |
| | | 0 < n <= 132 => n |
| | | n > 132 => 132 |

- The RESET procedure normally has the effect of a rewind followed by a GET. If the device is TTY, however, the GET is suppressed and input

of a null line is simulated: after the RESET, EOF is FALSE, EOLN is TRUE, and the buffer variable is a blank.

To cross-reference and reformat your Pascal program without compiling it:

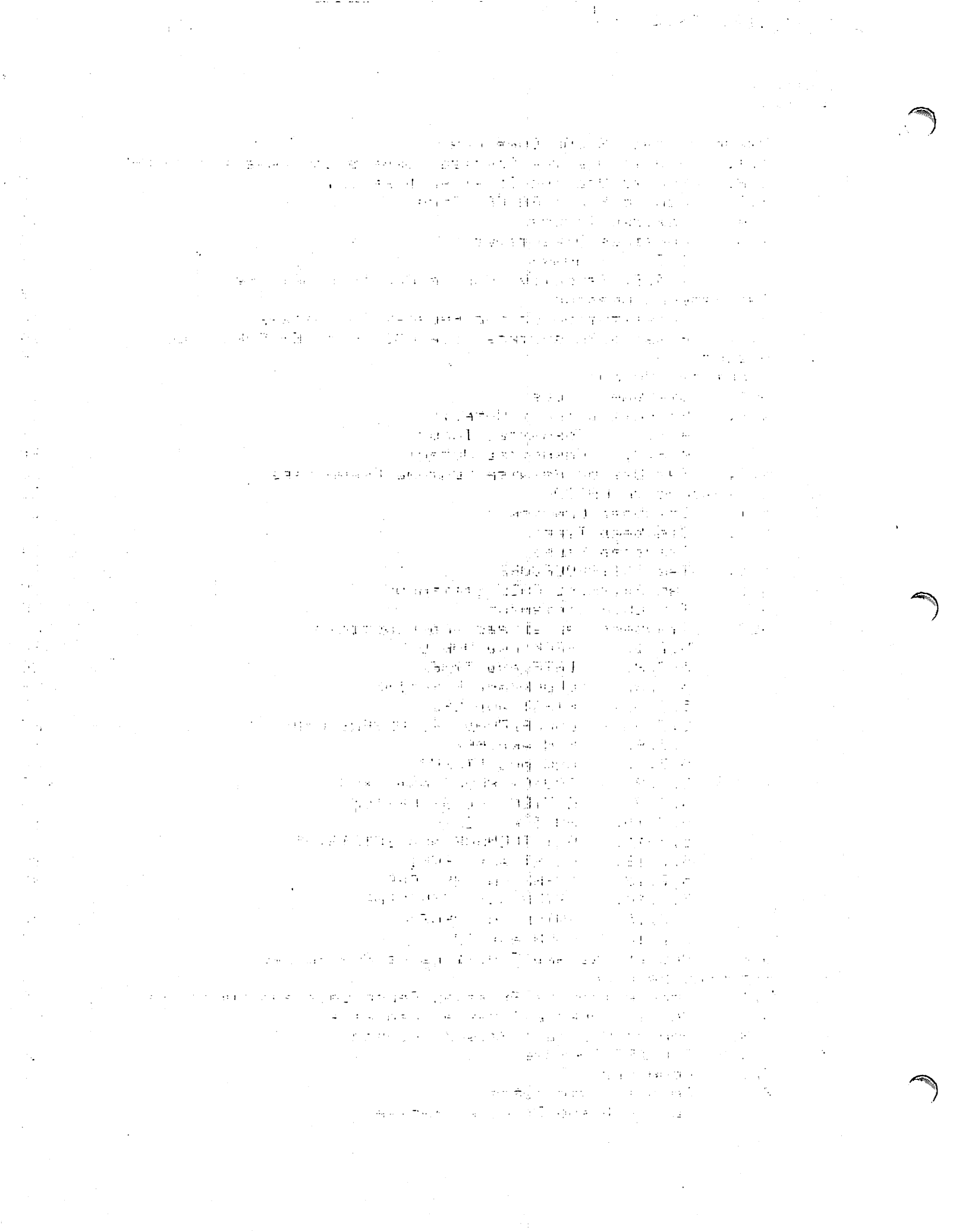
```
-RUN(UPL)CROSS
```

The cross-referencer will prompt for the names of your source, reformatted source, and cross-reference files.

- TYPE (DEPL) PASCAL.MAN

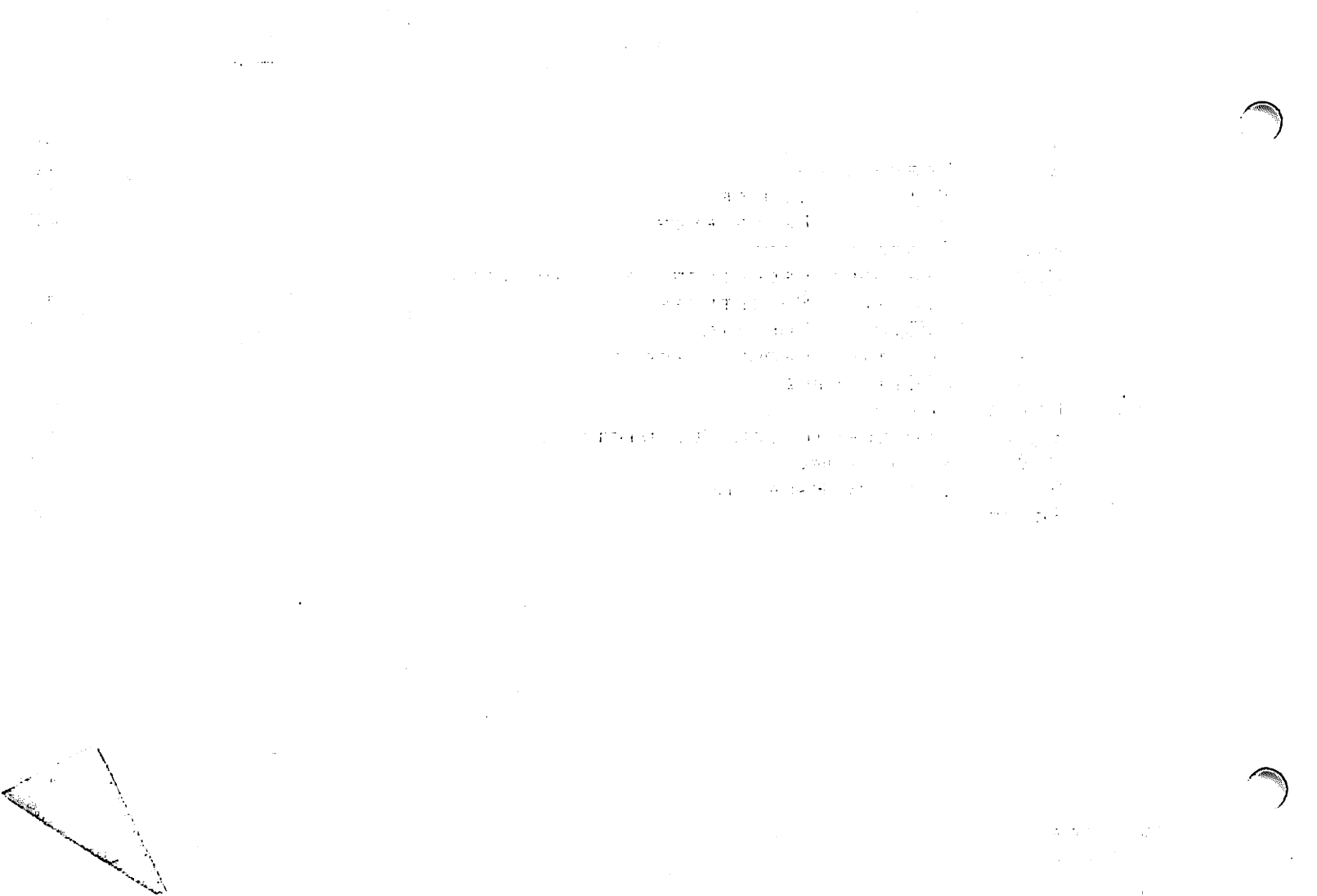


CONTENTS
========



Contents
========

## Preface



## Preface

The PASCAL compilers for the DECSystem-10 were developed at the university of Hamburg, Germany, based on an early version of the transportable PASCAL-P compiler [4,5,6]. Work still continues on improving this compiler with respect to efficiency, standardization and the addition of still missing or desirable language features [7].

This report is intended to familiarize the reader with the PASCAL dialect on the DECSystem-10. This description refers to a DECSystem-10 compiler with the name PASCAL that will accept STANDARD PASCAL (and some features of PASCAL 6000-3.4) as defined by Wirth in [2] as a subset.

It should be noted that earlier compiler versions or special purpose, Load-and-Go PASCAL compilers might accept a modified PASCAL language not compatible with STANDARD PASCAL and its extensions as accepted by this compiler.

It is assumed that the reader has a basic knowledge about PASCAL.

Within syntax definitions, terms enclosed in "[]" are optional and terms enclosed in "[]*" may appear zero or more times. The word "or" or a vertical bar "|" indicate alternative terms.

This report evolved from a PASCAL HELP file originally written in German which has been translated into English by Burger [11]. This version has been completely reworked from the english version. We gratefully acknowledge the help of B.Gisch and H.Linde in editing this report.

# 1. Usage of the PASCAL Compiler

## 1.1. How to Use the Concise Command Language for PASCAL

If the installation has modified the DECSystem-10 Concise Command Language (CCL) handler [8] to detect the extension PAS and activate the PASCAL compiler, one simply has to use the standard CCL commands. Options, as described in Section 1.5., may be appended according to the CCL conventions. These commands, called "COMPIL-class" commands, are defined as follows:

```
<compil-class command> ::= <command name>
                           [<file name>=]
                           <file description>
                           [@<file name>]
                           [<compiler options>]
                           [<command options>]
<command name>         ::= COMPILE or LOAD or EXECUTE
<compiler options>     ::= (<option sequence>)
<command options>      ::= /<option sequence>
<option sequence>      ::= <option> [/<option>]*
<option>               ::= <identifier> [:<unsigned integer>]
<file description>     ::= [<device>:] <filename>
                           [.<extension>] [<ppn>]
                           [<protection>]
```

Example:

```
.COMPILE FILNAM( CODESIZE:1200/DEBUG )/LIST
.LOAD FILNAM( RUNCORE:16/CODE )/CREF
.EXECUTE FILNAM( FORTIO/NOCHECK )/COMPILE
```

The so-called == and @-constructions [8] are also supported.

```
.COMPILE NAMFIL=FILNAM( EXTERN )/LIST
```

generates relocatable output on NAMFIL.REL and a listing on LPT:NAMFIL.

If SWITCH.CMD contains the line

```
( CODESIZE:1200/DEBUG/RUNCORE:20 )
```

the command

```
.COMPILE FILNAM@SWITCH
```

is equivalent to

```
.COMPILE FILNAM( CODESIZE:1200/DEBUG/RUNCORE:20 )
```

Depending on the command the program is compiled and/or loaded and/or executed. The program is compiled if the PAS-file has a younger creation date than the REL-file -if existent-. The compilation can be enforced with the COMPILE-option (refer to 1.5.).

## 1.2. How to Use the PASCAL Compiler Directly

The compiler is executed by

```
.R PASCAL
```

In this case the compiler will prompt the user at his terminal to provide file descriptions and compiler-options for source, list and relocatable object code files (refer to Section 2.1. or 5.7.9. for more detailed information). The file description has the following form:

```
DEVICE: FILNAM.PAS [PROJECT, PROGRAMMER] <PROTECTION>
```

The underlined parts may be omitted. They are, by default:

| | |
|---|---|
| DEVICE | DSK |
| [PROJECT, PROGRAMMER] | own PPN |
| <PROTECTION> | installation default (usually <057>) |

The compiler-options may be appended to each of the following file descriptions according to the conventions described in Section 1.5..

Example:

```
.r pascal
OBJECT = filnam.rel/codesize:800        relocatable object
LIST = filnam.lst/code                  listing
SOURCE = filnam.pas/nocheck/debug       source file
```

If e.g. the file description for OBJECT is defaulted, the compiler generates relocatable object code on a file named OBJECT (refer to 2.1. and 4.1.). A listing is generated if either the LIST-, CODE- or CREF-option is specified or just a file description for LIST is provided.

## 1.3. How to Run a PASCAL Program

Compilation starts when the following message is output:

```
PASCAL: FILNAM [<program name>: <entry>, ...]
```

After successful compilation the following size information is output:

```
HIGHSEG: u K + m WORD(S)
LOWSEG : v K + n WORD(S)
```

The first line denotes the core requirement for the high-segment (code) in K, the next line denotes the core requirement for the low-segment (data) in K.

If the compiler has generated relocatable code on FILNAM.REL, the program is loaded by

```
.LOAD FILNAM,REL or .LOAD FILNAM
```

An executable program is obtained by

```
.SAVE FILNAM w
```

after loading where w is the total core requirement for the program. It should be at least u + v + 5. A sharable high-segment is obtained by

```
.SSAVE FILNAM w
```

SSAVE must not be used if the program has been compiled using the DEBUG-option.

Another way to provide the core requirement for a program is to compile it with the RUNCORE-option. The value of the option denotes the maximum core requirement for the low-segment in K words. The value has to be v + 2 at least.

Example:

```
.load filnam( runcore:16 )/compile
.ssave filnam
.run filnam
```

In this case the program itself allocates 16K for its low-segment each time it is executed and the user need not to take care of the size of the high-segment.

The program is executed by

```
.RUN FILNAM
```

Should more core be required than has been specified when saving or compiling the program, the command

```
.RUN FILNAM n
```

may be used where n stands for the larger core requirement.

If several programs are linked together or it is necessary to get the correct size of low- and high-segment, the program(s) should be loaded with the MAP-option like

```
.LOAD FILNAM,PROC1,PROC2,/SEARCH MYLIB[302,3015]/MAP
```

The "loader map" can be obtained by

```
.PRINT <program name>.MAP
```

After inspection of the loader map one can specify w as the total size of the low-segment plus the total size of the high-segment plus the core needed for Stack and Heap. The value of the RUNCORE-option for later compilations can be evaluated as the total length of the low-segment plus the size of the heap.

## 1.4. Lexical Issues

The PASCAL compiler accepts only a subset of the ASCII characters, namely the 64 characters with octal values from 40 to 137. Characters with octal codes between 140 and 177 (essentially the "lower case" characters) are converted to "upper case" by subtracting 40 octal from their code. The TABulator-character is expanded on input from text-files to the appropriate number of blanks. Lines are ended by a Line-feed. All other characters appearing in the input text are ignored (refer to 4.).

Next we shall describe language elements which use special characters.

Comments are enclosed in (* and *), e.g.:

```
(*THIS IS A COMMENT*)
```

Identifiers must differ over their first 10 characters. They may be written using the underline-character (which is sometimes printed as left-arrow) to improve readability, e.g.:

```
NEW_NAME
```

String-constants are character sequences enclosed in single-quotes, e.g.:

```
'THIS IS A STRING'
```

If a quote is to appear in the string it must be repeated, e.g.:

```
'ISN''T PASCAL FUN?'
```

(PACKED) arrays of CHAR or subranges of CHAR are referred to as strings throughout this report.

An INTEGER-constant is represented in octal form if it consists of octal digits followed by B. An INTEGER is represented in hexadecimal form if it consists of a " (i.e. double-quote) followed by hexadecimal digits. The following representations have the same value:

```
63      77B      "3F
```

A REAL-constant is defined as

```
<real>   ::= [<sign>]<number>[.<number>][E[<sign>]<number>]
<number> ::= <digit> [<digit>]*
```

SET-constants are element lists enclosed in "[]", e.g.:

```
['A', '+'] or [1, 15]
```

Subrange notation is allowed inside of SET-constants, e.g.:

```
['A'..'Z', '0'..'9'] or [1..7, 12..17]
```

SETs within a program may contain variables - compatible with the type of the SET elements - , e.g.:

```
...
VAR S: SET OF CHAR;
    C: CHAR;
...
S := ['A'..'C', C, '+','-'];
...
```

## 1.5. Compiler Directives

Compiler-options are written as comments and are designated as such by a '$'-character as the first character of the comment, e.g.:

```
(*$T-,E+,D+,R15   the rest is comment   *)
```

An option is turned on if it is followed by '+' and it is turned off if it is followed by '-'. Some options must be followed by an INTEGER. Options may alternatively be specified at translation time, either with the COMPILE-, LOAD-, or EXECUTE-command or by appending the options to the file names if the compiler is executed directly (refer to 1.1-2.). Any compiler-option explicitly requested in the source-program overrides compiler-options provided at translation time.

### 1.5.1. Summary

There are certain compiler-options which can only be specified at translation time. For these, no source-program switch is indicated.

Other compiler-options -marked with (*)- have to be specified for the first time BEFORE the program heading. Some of these -marked with (**)- cannot be reset anymore inside the program.

In the following table the inverse option -if possible- is given below the positive option. The options are divided into three groups:

(1) Compiler-options
(2) "COMPIL-class" Command-options
(3) Loader Command-options

| Function | Specification in source program at translation time | | Default |
|---|---|---|---|
| **----------Compiler-options----------** | | | |
| List object code as MACRO-10. | L+<br>L- | (CODE)<br>NOCODE | off |
| Perform runtime tests,<br>1) array indices<br>2) assignments to subranges<br>3) zero-divide<br>4) arithmetic overflow<br>5) variables in SET-constants<br>6) input to file variables or variables of subrange types | T+<br>T- | (CHECK)<br>NOCHECK | on |
| Enable debugging including Post-Mortem dump. (*)<br>P is accepted in addition to D to maintain compatibility with PASCAL 6000. | D+<br>D-<br>P+<br>P- | (DEBUG)<br>NODEBUG | off |
| All level-1 procedures or functions of a program may be activated by other programs. Refer to 6. for detailed information. (**) | E+<br>E- | (EXTERN)<br>NOEXTERN | off |
| Only the first 72 characters of a source program line are accepted for compilation (card format). (**) | U+<br>U- | (CARD)<br>NOCARD | off |
| Maximum number of instructions that may be generated for the statement part of a "main progam", procedure or function. (**) | Sn | (CODESIZE:n) | n=1000 |
| Size of low-segment in K words. (**) | Rn | (RUNCORE:n) | n=static core requirement for total low-segment |
| Enable FORTRAN-I/O in external FORTRAN subroutines. (**) | I+<br>I- | (FORTIO)<br>NOFORTIO | off |
| Highest register used to pass parameters, with n in [2..12]. External procedures must be compiled with the same value that is assumed in the main program. (**) | Xn | REGISTER:n | n=6 |

| Function | Specification in source program at translation time | Default |
|---|---|---|
| Refer to 6.2, for more detailed information about this option, (**) | Fn | FILE;n | n=1 |
| Compile and load, | | (LINK) NOLINK | off |
| Compile, load and execute, | | EXECUTE NOEXECUTE | off |
| Compile, | | COMPILE NOCOMPILE | on |
| Generate listing, | | (LIST) NOLIST | on |
| Generate cross-reference list, | | CREF NOCREF | off |
| **"COMPIL-class" Command-options** | | | |
| Enforce compilation, | | /COMPILE NOCOMPILE | depends on creation date of REL- and PAS-file |
| Generate cross-reference list, | | /CREF NOCREF | off |
| **Loader Command-options** | | | |
| Provide information about all programs loaded, | | /MAP NOMAP | off |
| Put a program library on top of the library search chain, (<lib> ::= <file description>) | | SEARCH <lib> | PASLIB and FORLIB |

## 1.5.2. Selective Use of Compiler-options

The Compiler-options L and D (or P) can be used to selectively list object code resp. generate information for the DEBUG-system. Selective use of options, however, is effective only if the option is not switched off before the END of a procedure or function (because the DEBUG information for a block is generated at the end of its statement part).

Example:

```
        .
        .
        .
       (*$L+*)
       PROCEDURE P1;
        .
        .
        .
       END (* P1 *);
       (*$L- MACRO-10 code is output for P1 *)
        .
        .
        .
       (*$D+*)
       PROCEDURE P2;
        .
        .
        .
       END (* P2 *);
       (*$D- local variables of P2 may be investigated and
             breakpoints may be set in P2 *)
        .
        .
        .
       PROCEDURE P3;
        .
        .
        .
       (*$D+*)
       BEGIN (* P3 *)
        .
        .
        .
       END (* P3 *);
       (*$D- only breakpoints may be set in P3 since
             the debug option remained off during analysis of
             the declaration part for this procedure *)
        .
        .
        .
```

The L-option in the source code is effective only if a listing is requested (refer to 1.2.).

## 2. The Program Heading

Every program consists of a heading and a block. The block contains a declaration part in which all objects local to the program are defined, and a statement part which specifies the actions to be executed upon these objects.

```
<program> ::= <program heading> <block> .
<block>   ::= <label decl. part>
              <constant decl. part>
              <type decl. part>
              <variable decl. part>
              <procedure/function decl. part>
              <statement part>
```

(for INITPROCEDURE-declaration see 5.4.)

The program heading gives the program a name. This name is not significant inside the program. A program, however, is usually referred to by the name of the file(s) containing the executable code (SHR- and LOW-file). If the program is compiled with the EXTERN-option, the names of all level-1 procedures and functions which are assigned to be called from other programs have to be listed after the program name. Notice that only six characters are significant for all these names. They must not contain '_'-characters. The program heading also lists the program parameters through which the program communicates with it's enviroment.

```
<program heading> ::= PROGRAM
                      <program name>
                      [,<procedure or function name>]*
                      [(<parameter> [,<parameter>]*)];
```

### 2.1. Substitution of the Program Parameters

Possible parameters are files. A PASCAL file variable is implemented as a file in the DECSystem-10. These files exist outside the program (before or after execution on disk or tape) and can be made available to the program by two alternatives:

(1) An external file can be connected to a PASCAL file variable by use of the standard procedures RESET or REWRITE. These procedures allow to specify the properties of the external file (refer to 5.7.12.).

(2) Another way is the substitution for the formal parameters specified in the program heading.

Thus, a program parameter is defined as follows:

```
<parameter> ::= <file variable> or
                <file variable>*
```

These formal parameters -except INPUT and OUTPUT (TTY MUST NOT be specified)- must be declared as files inside the program. Every external file is automatically "opened" when the program is started by the RUN-command. If the file is to be "opened" only for input, this has to be indicated by appending an asterisk to the file parameter. However, the asterisk itself does not constitute any protection against writing on the file (by a subsequent REWRITE). The substitution is performed by prompting the user for the external file specification for each file variable in the same sequence as specified in the program heading.

Thus, it is not necessary to "open" these files inside the program with RESET or REWRITE.

Example:

Let FILNAM.PAS contain the following program:

```
PROGRAM COMPARE_FILES( FILE1*, FILE2*, OUTPUT );
VAR FILE1, FILE2: FILE OF INTEGER;
  ...
END.
```

Execution could start as follows:

```
.run filnam
FILE1 = dske:test.pas[302,3015]
FILE2 =
OUTPUT = filnam.lst<333>
  ...
EXIT
```

If the request for a file description is defaulted with a Carriage-Return, it is assumed -in this case- that a file named FILE2 does already exist. This is quite analogous to RESET(FILE2).

Programs which only make use of the standard files INPUT and OUTPUT may omit specification of the parameter list because INPUT and OUTPUT are automatically "opened" if the parameter list is empty. Thus,

```
PROGRAM ALPHA( INPUT*, OUTPUT );
```

is equivalent to

```
PROGRAM ALPHA;
```

All these files may still be re-defined by RESET or REWRITE -as mentioned under (1)- later in the program.

## 2.2. Passing Parameters with CMD- and TEMPCORE-files

Another way to provide the actual program parameters is to create a file (TEXT-file only) named XXXXXX.CMD where XXXXXX are the first 6 characters of the program name. For the example above the file COMPAR.CMD could contain the following line:

```
DSK: TEST.PAS[302,3015],,FILNAM.LST<057>
```

No prompting is done now:

```
.run filnam
 ...
EXIT
```

At last, it is possible to pass the information in a TEMPCORE-file (TEMPorary CORE file). These files are allocated in core storage and can be accessed/created without time-consuming disk reads/writes. In PASCAL, a file is assumed to be a TEMPCORE-file if it is specified as

```
DSK:XXX.TMP
```

XXX are the first 3 characters of the program name (COM.TMP in the example above). If a core file cannot be allocated, the disk file

```
DSK:NNNXXX.TMP
```

is created where NNN is the "job number". So, if a TEMPCORE-file cannot be found in core, such a disk file is "looked up".

Example:

```
 ...
VAR TEMPORARY_FILE: TEXT;
 ...
REWRITE( TEMPORARY_FILE,'ABC   TMP' );
WRITE( TEMPORARY_FILE, ...
 ...
RESET( TEMPORARY_FILE );
WHILE NOT EOF( TEMPORARY_FILE ) DO
 BEGIN
  IF NOT EOLN( TEMPORARY_FILE )
  THEN READ( TEMPORARY_FILE, ...
 ...
```

Notice that a temporary file is assigned to contain short information (e.g. pass program parameters to other programs called by CALL, refer to 5.7.10.) and cannot exceed 128 words or 640 ASCII characters (one DECSystem-10 disk block).

If a program is executed with the RUN-command, it will try to obtain its program parameters from either a TEMPCORE-file or a disk file with the extension CMD. The general input format for program parameters is listed below.

CMD-file or TEMPCORE-file:

[<file description> [, <file description>]*]

TTY:

[<file description>]

If both files do not exist, it prompts the user to enter the parameters directly at his terminal.

This technique is used by the DECSystem-10 COMPIL-program to pass parameters to the different compilers.

## 3. Labels

Any executable statement in a program may be marked by prefixing it with a label followed by a colon. This label must be declared in the label declaration part before its use.

<label declaration part> ::= LABEL <label> [,<label>] ;

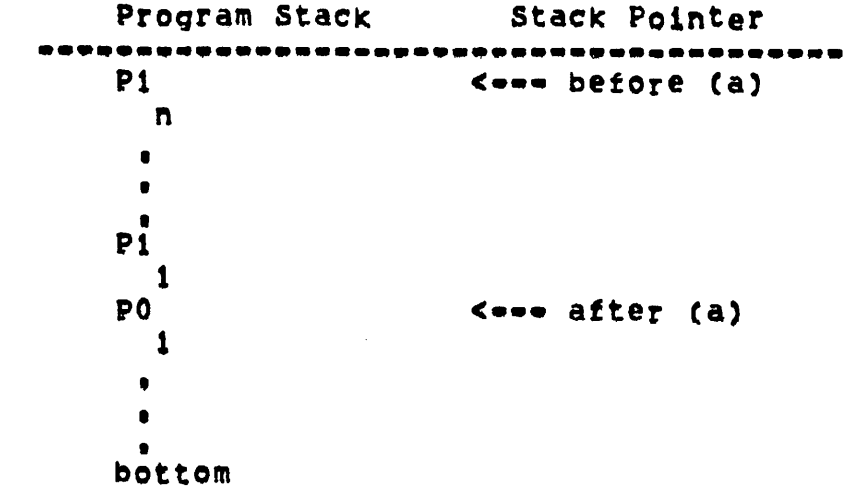
A label must be an unsigned INTEGER-constant of at most 4 digits. The scope of a label is the entire block wherein it is declared. That is, a statement of the statement-part of this block may be prefixed with the label. This label may be referenced by GOTO-statements inside the same statement part or all other statement parts of procedures or functions declared inside this block.

Example:

```
     ...
 ------------------------------------------
 I    PROCEDURE P0;                       I
 I     LABEL 11, 12, 13;                  I
 I      ...                               I
 I -------------------------------------- I
 I I   PROCEDURE P1;                    I I
 I I    LABEL 21;                       I I
 I I    BEGIN (* P1 *)                  I I
 I I      ...                           I I
 I I     21: P1;                        I I
 I I     IF P THEN GOTO 11        (a)   I I
 I I          ELSE GOTO 21        (b)   I I
 I I      ...                           I I
 I I    END (* P1 *);                   I I
 I -------------------------------------- I
 I     BEGIN (* P0 *)                     I
 I      ...                               I
 I      IF Q THEN GOTO 21          (c)    I   (*wrong*)
 I           ELSE GOTO 12;         (d)    I   (*wrong*)
 I      ...                               I
 I      IF P                              I
 I      THEN                              I
 I       BEGIN                            I
 I        12: R;                          I
 I         ...                            I
 I       END                              I
 I      ELSE GOTO 13;              (e)    I   (*wrong*)
 I      ...                               I
 I      WHILE S DO                        I
 I       BEGIN                            I
 I        13: T;                          I
 I         ...                            I
 I       END;                             I
 I      11: U;                            I
 I      ...                               I
 I     END (* P0 *);                      I
 ------------------------------------------
        ...
```

(a) This is a valid GOTO from an "inner" procedure of P0. The label 11 is defined (there is a statement prefixed with it) in its scope P0. Such a construction provides an "exit" from P1 to P0.

(b) In this case the GOTO refers to a label which is declared in the same block (P1). Execution of this GOTO involves recursive activation of P1, since the statement prefixed with LABEL 21 is a (recursive) procedurecall of P1. If in the n-th recursion step (a) is executed, the Stack is reset as follows:




(c) This GOTO is invalid because it references a label outside its declaration scope.

(d,e) Of course it is not allowed to jump into conditional (d) or any other structured (e) statement because the result could be undefined.

# 4. Input and Output

Input and output is performed by the standard procedures GET, GETLN, PUT and PUTLN (for GETLN and PUTLN refer to 5.7.14-15.). Input and output to TEXT-files (PACKED FILE OF CHAR) should be done with the standard procedures READ, READLN, WRITE, WRITELN and PAGE as described in "PASCAL - USER MANUAL AND REPORT" [1,2]. The latter is called "formatted"-I/O throughout this chapter.

The DECSystem-10 mainly distinguishes two kinds of data modes for files, namely

(1) ASCII-mode with five 7-bit characters packed into one word

(2) binary-mode with a 1:1 core-to-file mapping

The following table illustrates how PASCAL file types are implemented in the DECSystem-10:

| File Type | Packed | Unpacked |
|---|---|---|
| Subrange of CHAR or CHAR | ASCII-mode, "upper case", formatted-I/O, linenumbers and pagemarks | binary-mode, subrange as specified in file declaration standard-I/O |
| Subrange of ASCII or ASCII | ASCII-mode, complete 7-bit-ASCII, standard-I/O | as above |
| other | treated as unpacked | binary-mode, standard-I/O |

During input from TEXT-files (PACKED FILE OF CHAR or subrange of CHAR) all characters less than ' ' -except LF and HT- are ignored and all characters greater than '_' are converted to "upper case" as follows: UPPER_CHAR := CHR( ORD( INPUT_CHAR ) - 40B );

HT (TAB) is expanded to the appropriate number of blanks and LF marks the end of a line (sets EOLN to TRUE).

## 4.1. Standard Files

The standard files INPUT, OUTPUT and TTY as all the files specified as program parameters in the program heading can be used directly for input and output without having to "open" them by use of RESET or REWRITE. All file variables are assigned by default to the external file

DSK: XXXXXX,YYY

where XXXXXX are the first 6 characters of the file identifier and

YYY are the next 3 characters of it. Blanks are used if there are not enough characters.

Example:

| File Identifier | Default Assignation |
|---|---|
| INPUT | DSK: INPUT |
| OUTPUT | DSK: OUTPUT |
| TTY | TTY: TTY |
| AUSGABE | DSK: AUSGAB.E |

## 4.2. Formatted Input/Output

TEXT-files (PACKED FILE OF CHAR) can be accessed, as any other file, by the standard procedures GET and PUT. This is of course quite cumbersome as these procedures are defined for single character manipulation. The procedures READ and WRITE imply a set of transformation routines which are designed to recognize the pattern of e.g. an INTEGER-, REAL-, BOOLEAN-, SET- or string-constant in the "TEXT" and to convert it into the installation dependent internal representation, resp. to transform the internal representation into an appropriate text.

### 4.2.1. Formatted Input

The procedures READ and READLN have non-standard parameter lists. If the first parameter is a file identifier, input is done from the external file currently assigned to this file variable. Otherwise INPUT is assumed. The number of parameters is unlimited. They must obey the restrictions on VAR-parameters, especially they must not be constants or elements of packed data structures. The following types of variables are accepted:

| Type | Restriction |
|---|---|
| INTEGER | -MAXINT <= value <= MAXINT |
| REAL | SMALLREAL <= ABS( value ) <= MAXREAL |
| CHAR | none |
| ASCII | none |
| BOOLEAN | TRUE or FALSE |
| scalar | type must be declared inside the program ≤ 10 char |
| subrange | FIRST(parameter) <= value <= LAST(parameter) |
| string | must be a correct string-constant (refer to 1.4.) |
| SET | must be a correct SET-constant (refer to 1.4.) |

Example:

```
PROGRAM READER;
TYPE Z = (EINS,ZWEI,DREI);
VAR R: -13..34..13..34;
  S: Z;
  T: SET OF Z;
BEGIN
  READ( T, R, S);
    ...
END.
```

If the file assigned to INPUT contains

[] 15,0 EINS or [EINS,ZWEI] 13,0 ALPHA

the following error message is given before termination:

```
%? INPUT ERROR: SCALAR UNDEFINED OR OUT OF RANGE
   *** 1.5000000000E+01 *** AT USER PC ...
```

resp.

```
%? INPUT ERROR: SCALAR UNDEFINED OR OUT OF RANGE
   *** ALPHA *** AT USER PC ...
```

The following input is correct:

[EINS,,DREI] 13,0 ZWEI

### 4,2,2, Formatted Output

If the first parameter to WRITE or WRITELN is not a file identifier, OUTPUT is assumed, The number of parameters for WRITE and WRITELN is unlimited, Expressions of various types might be specified, The parameters to WRITE and WRITELN may be followed by a "format specification", A parameter with format has one of the following forms:

```
X : E1
X : E1 : E2
X : E1 : O
X : E1 : H
```

E1 is called the field width, It must be an expression of type INTEGER yielding a non-negative value, For SETs the field width applies to the SET elements, If no format is given, the default value for E1 is for type

| | |
|---|---|
| INTEGER | 12 |
| BOOLEAN | 6 |
| CHAR | 1 |
| ASCII | 1 |
| REAL | 16 |
| ALFA | 10 |
| scalar | 10 |
| string | the length of the string |
| SET | the default field width of the SET element type |

Blanks precede the value to be printed if the field width is larger than necessary to print the value. Depending on the type involved the following is printed if the field width is smaller than necessary to print the value:

| | |
|---|---|
| INTEGER | E1 asterisks |
| REAL | E1 asterisks |
| BOOLEAN | T or F preceded by the appropriate number of blanks |
| SET | the default action for the SET element type is performed |
| string | the leftmost characters fitting the field width |
| scalar | same as string |

Example:

```
WRITELN( MAXINT:1, [10,20]:1 );
```

produces the following output:

```
*[*,*]
```

In the following examples, colons usually stand for blanks.

No characters are printed if the field width is 0. The minimal field width for values of type REAL is 8.

Example:

```
WRITELN('STR':4, 'STR', 'STR':2, -12.0:10);
WRITELN(['A'..'Z','1'..'9','+','_'],['A','B','C']:5);
WRITELN(15:9, TRUE, FALSE:4, 'X':3);
```

The following character sequence will be printed:

```
:STRSTRST-1.200E+01
['+','1'..'9','A'..'Z','_'][:'A'..:'C']
:::::::15::TRUE:::F::X
```

A value of type REAL can be printed as a fixed point number if the format with expression E2 is used. E2 must be of type INTEGER and yield a non-negative value. It specifies the number of digits following the decimal point. There must be enough room for the fractional part, otherwise asterisks will be printed.

Example:

```
WRITE(1.23:5:2, 1.23:4:1, 1.23:6:0, 1.23:4:3);
```

The following character sequence will be printed:

```
:1.23:1.2:::::1.****
```

A value of type INTEGER can be printed in octal representation if the format with letter O is used. The octal representation consists of 12 digits. If the field width is smaller than 12, the rightmost digits are used to fill the field width. If the field width is larger than 12, the appropriate number of blanks precedes the digits.

Example:

```
WRITE(12345B:2:0, 12345B:6:0, 12345B:15:0);
The following character sequence will be printed:
45012345:::000000012345
```

A value of type INTEGER can also be printed in hexadecimal representation if the format with letter H is used. The hexadecimal representation consists of 9 digits. Using the format with letter H the following character sequence will be printed for the example above:

```
E50014E5:::::0000014E5
```

## 4.3. The Use of Printer Control Characters

In some installations the first character of each line is used as a printer control character when a text file is send to the printer. The first character is not printed but instead interpreted as controlling the paper feed mechanism of the printer. The following (FORTRAN-) conventions are in wide use [9]:

| Character | ASCII | Action before Printing |
|---|---|---|
| ' ' | LF | skip to next line, form feed after 60 lines |
| '0' | LF,LF | skip one line |
| '1' | FF | go to top of next page |
| '2' | DLE | space 1/2 of a page |
| '3' | VT | space 1/3 of a page |
| '+' | | overprint the line |
| '*' | DC3 | skip to next line, no form feed |
| '-' | LF,LF,LF | skip two lines |
| '/' | DC4 | space 1/6 of a page |
| '.' | DC2 | triple space, form feed after 20 lines |
| ',' | DC1 | double space, form feed after 30 lines |

A file containing such control characters must be printed with the following command:

```
.PRINT <file description>/FILE:FORTRAN
```

Example:

```
WRITELN('1','OVER');
WRITELN('+','    PRINT');
```

The following output will be printed on the first line of the next page:

```
OVERPRINT
```

# 5. Extension to STANDARD PASCAL

## 5.1. Standard Constants

The INTEGER constants MININT and MAXINT are defined as

```
CONST MININT = 400000000000B (* -34,359,738,368 *);
      MAXINT = 377777777777B (* +34,359,738,367 *);
```

They represent the smallest and greatest INTEGER in the DECSystem-10.

The REAL constants MAXREAL and SMALLREAL are defined as:

```
CONST MAXREAL   = 1.7014118432E+38;
      SMALLREAL = 1.4693680107E-39;
```

They represent the absolutely smallest and greatest REAL in the DECSystem-10.

The ASCII constants NUL..SP and DEL (ASCII mnemonics for special control characters) are primarily intended to ease special I/O-operations.

Example: WRITE(TTY,BEL) instead of WRITE(TTY,CHR(07B))

They do not belong to the PASCAL character set (the type CHAR) which is only a subset of the ASCII character set (refer to 8.5.).

## 5.2. Standard Types

The type ASCII represents the total 7-bit-ASCII character set. It is a superset of CHAR.

```
TYPE ASCII = NUL..DEL;
     CHAR  = ' '..'_';
```

Thus, ASCII and CHAR are type-compatible and assignments from CHAR to ASCII and reverse are legal provided only values belonging to the subrange CHAR are assigned to a variable of that type. Refer to 4. and 9.1.(e) for closer information about SETs and files of ASCII.

Example:

```
...
VAR A: PACKED FILE OF ASCII;
    ASC: ASCII;
...
WHILE NOT ( EOF(A) OR (A^=LF) ) DO
 BEGIN
  ASC := A^; GET(A); P(ASC);
   ...
```

## 5.3. Standard Files

In addition to the standard TEXT-files INPUT and OUTPUT the standard TEXT-file TTY is available in DECSystem-10 PASCAL. This file is used to communicate with the terminal. The first parameter of the standard I/O-procedures must be TTY if they are to be applied to this file.

TTY is "opened" at the beginning of the program. An asterisk is typed out on the terminal to indicate that input is expected (to assign a value to TTY^). If the program does not need any input from the terminal, a Carriage-Return should be entered.

TTY should only be a parameter to the standard procedures GET, GETLN, PUT, PUTLN, READ, READLN, WRITE, WRITELN, PAGE, BREAK, GETFILENAME and GETSTATUS. It must not be used as an actual parameter (VAR-parameter) of type TEXT to either declared procedures and functions.

Example:
```
...
VAR CH: CHAR;
...
CH := TTY^; GET( TTY^ );...;TTY^ := 'A'; PUT( TTY^ );
...
```

The standard procedure BREAK (refer to 5.7.13.) is provided in order to force the output to the terminal even if the interna buffer is not yet full.

## 5.4. The INITPROCEDURE

Variables of type scalar, subrange, pointer, ARRAY or RECORD declared in the outermost block of a program may be initialized by an INITPROCEDURE. The body of an INITPROCEDURE contains only assignment statements. Indices as well as the assigned values must be constants. INITPROCEDUREs can be defined ONLY in the outermost block prior to the first procedure or function definition with executable body. Assignment to components of packed structures is ONLY possible if the components occupy a full word.
The definition of the INITPROCEDURE is as follows:

```
<block>              ::= <declaration part>
                         <statement part>
<declaration part>   ::= <label decl. part>
                         <constant decl. part>
                         <type decl. part>
                         <variable decl. part>
                         <init part>
                         <procedure/function decl. part>
<init part>          ::= [<initprocedure>]*
<initprocedure>      ::= INITPROCEDURE;
                          BEGIN
                           <assignments>
                          END;
```

## 5.5. The Extended CASE Statement

The CASE statement may be extended with the case OTHERS which then must appear as the last case in the CASE statement. The statement associated with OTHERS will be executed if the expression of the CASE statement does not evaluate to one of the explicitly given case labels.

Example:

```
PROGRAM CASES;
VAR X: CHAR;
 ...
BEGIN
  ...
 CASE X OF
      'A':    P(X);
      'B':    Q(X);
       ...
      OTHERS: Z(X)
 END;
  ...
END.
```

## 5.6. The LOOP Statement

The LOOP statement is an additional control statement which combines the effects of the WHILE and the REPEAT statement.

The LOOP statement is defined as follows:

```
<loop statement> ::= LOOP
                       <statement part>
                     EXIT IF <expression>;
                       <statement part>
                     END
<statement part> ::= <statement> [;<statement>]*
```

The expression must result in a Boolean value.

## 5.7. Standard Procedures and Functions

In output examples, colons stand for blanks throughout this Chapter.

### 5.7.1. PACK and UNPACK

In addition to the first three parameters for these procedures described in [1, 2] two more optional parameters may be specified for PACK and UNPACK, namely to indicate where in the packed array the transfer should begin and how many items should be packed or unpacked.

```
PACK(A, I, Z [, J [, L] ])      is equivalent to

  for k := 0 to L-1 do Z[ J + k ] := A[ I + k ] ;

UNPACK(Z, A, I [, J [, L] ])    is equivalent to

  for k := 0 to L-1 do A[ I + k ] := Z[ J + k ] ;
```

where the default values are

J = LOWERBOUND( Z ) and
L = 1 + MIN( UPPERBOUND( Z ) - J, UPPERBOUND( A ) - I )

### 5.7.2. DATE and TIME

The procedure

```
DATE( <alfa variable> )
```

assigns the current date in the format 'dd-mmm-yy ' to the parameter which must be of type ALFA.

The procedure

```
TIME( <alfa variable> )
```

assigns the daytime in the format 'hh:mm:ss ' to the parameter which must be of type ALFA.

### 5.7.3. CLOCK and REALTIME

The function CLOCK returns the elapsed CPU time in msec and the function REALTIME returns the current time in msec. Both the functions have an INTEGER result type.

### 5.7.4. FIRST and LAST

The functions

```
FIRST( <variable> ) and
LAST ( <variable> )
```

return the lowest resp. the highest value of the (installation dependent) range of the variable type. The result type of these functions depends on the type of the parameter. Variables of any scalar type -except REAL- are accepted.

Example:
```
PROGRAM SUBRANGE;
TYPE SCAL = (EINS,ZWEI,DREI,VIER);
VAR I: INTEGER; SI: 0..15;
  C: CHAR; SC: 'A'..'Z';
  S: SCAL; SS: ZWEI..DREI;
BEGIN
  ...
 WRITE( FIRST(C), LAST(I):12:0, LAST(S):5,
        FIRST(SI), LAST(SC):2, FIRST(SS):10 );
  ...
END.
```

The following output is produced by this program:

```
:377777777777:VIER:::::::::::0:Z::::::ZWEI
```

### 5.7.5. LOWERBOUND and UPPERBOUND

The functions

```
LOWERBOUND( <array variable> ) and
UPPERBOUND( <array variable> )
```

return the lowest resp. the highest value of the range of the index type of the array. The result type depends on the index type.

Example:
```
PROGRAM BOUNDS;
TYPE FARBE = (RED,YELLOW,GREEN);
VAR FELD: ARRAY[ 'A'..'Z', YELLOW..GREEN, 10..50 ] OF CHAR;
BEGIN
  ...
 WRITE( LOWERBOUND( FELD ):2,
        UPPERBOUND( FELD[ 'A' ] ):6,
        LOWERBOUND( FELD[ 'X', YELLOW ] ):3 );
  ...
END.
```

The following output is produced:

```
:A:GREEN:10
```

### 5.7.6. MIN and MAX

The functions

```
MIN( <expression>, <expression> [,<expression>]* )
MAX( <expression>, <expression> [,<expression>]* )
```

return the minimum resp. the maximum of the expression list. Up to 72 expressions may be specified. Expressions of any scalar type -except BOOLEAN- are accepted. INTEGER and REAL may be mixed. If so, the INTEGER-expressions are converted to REAL. The result type depends on the type of the expressions.

Example:

```
PROGRAM MINMAX;
TYPE SCAL = (KIND,VATER,MUTTER,OPA,OMA);
VAR C: CHAR; I: INTEGER; R: REAL; S: SCAL;
BEGIN
  ...
 S := VATER; C := 'X'; I := 15; R := 13.4E4;
 WRITE( MIN(S,MUTTER),
        MAX(C,'+'):2,
        MIN(I,3):2,
        MAX(13.0,I,R):9 );
 ...
 END.
```

The following is printed:

```
VATER:X:3:1.34E+05
```

### 5.7.7. NEW and DISPOSE

The procedures NEW and DISPOSE have the following parameter list:

```
( <pointer variable> [,<tagfield constant>]*
                     [:<integer expression>] )
```

NEW allocates variables in the heap. The address of the newly allocated variable is assigned to <pointer variable>. If the first argument is pointing to a record with variant parts, the tagfield constants specifying the desired record variant may be set. In this case core will be reserved exactly as required for this record variant. (Later assignments to this record must correspond to the variant reserved, otherwise the heap may be overwritten in difficult to detect cases ). If the last component is an array, the "actual" size of the array (the number of components required for this reservation) may be appended - preceeded by a colon - to the last tagfield-constant.

The corresponding de-allocation of the variable by DISPOSE must have the IDENTICAL parameter list. If the pointer is undefined or out of the heap, an appropriate error message (refer to 8.4.) is output before program termination.

Example:

```
  ...
CONST MAX_DIM = 1000;
TYPE  PTR = ^REC
      REC = RECORD
              CASE INTEGER OF
               1: ( CASE CHAR OF
                      'A': ( FIELD: ARRAY[1..MAX_DIM] OF REAL ));
                  .
                  .
                  .
              END (*REC*);
  ...
VAR   PTR_VAR: PTR;
  ...
NEW( PTR_VAR, 1, 'A': 200 ); (* occupies 200 words *)
  ...
DISPOSE( PTR_VAR, 1, 'A': 200 );
  ...
```

The Heap is organised like the Stack. Thus, on DISPOSE, all variables allocated later than the disposed one are de-allocated too.

```
  Heap            Heap Pointer
-------------------------------------------
  V                 <- before DISPOSE( V   )
   n                                    1
   .
   .
   .
   V
    1
   V                <- after DISPOSE( V   )
    0                                   1
   .
   .
   .
   bottom
```

## 5.7.8. TRUNC, ROUND and EXPO

The function

TRUNC( <real or integer expression> )

returns the greatest INTEGER less than or equal to the argument.

Examples:

```
TRUNC( -13.35)   is -14
TRUNC( +13.35)   is +13
```

The function

ROUND( <real or integer expression> )

returns the INTEGER nearest to the argument:

ROUND(E) := TRUNC(E+0.5);

Examples: ROUND( -0.5 ) is 0
ROUND( +0.5 ) is +1

The function

EXPO( <real or integer expression> )

returns the INTEGER exponent of the floating point representation of the argument. EXPO is defined as:

expo := trunc(ld(abs(<real or integer expression>))) + 1

Examples: EXPO( -13.78E-22 ) is -69
EXPO( +1.38E15 ) is 51
EXPO( 1 ) is 1
EXPO( 0 ) is 0

## 5.7.9. OPTION and GETOPTION

The function

OPTION( <alfa variable or alfa constant> )

returns the value TRUE if the option indicated by the argument has been specified by the user during program initialization (substitution of the program parameters, refer to 2. and 5.7.10.), otherwise FALSE is returned.

Example:

Let FILNAM.PAS contain the following program:

```
PROGRAM DECIDER ( INPUT*, OUTPUT );
 ...
IF OPTION('STEP2     ')
THEN P
ELSE Q;
 ...
```

If the program parameters are entered like

```
.run filnam
INPUT = data[100,100]/step2
OUTPUT = dskc:out.dat<055>
```

P will be executed.

The procedure

GETOPTION(<alfa constant or variable>,<integer variable>)

can be used to "read" options qualified by numerical attributes. The value of the option is assigned to the INTEGER variable (second parameter).

Example:

Let FILNAM.PAS contain:

```
PROGRAM CHOOSER ( INPUT* );
VAR CHOOSE: INTEGER;
 ...
IF OPTION('TYPE      ')
THEN
 BEGIN
  GETOPTION('TYPE      ',CHOOSE);
  CASE CHOOSE OF
      0: P(CHOOSE);
       ...
  END
 END;
 ...
```

If the program parameter is entered like

```
.run filnam
INPUT = /type:0
```

P(0) will be executed.

## 5.7.10. HALT and CALL

The procedure

HALT

allows to enter the DEBUG-system from any statement of a program. If the program is not compiled with the DEBUG-option, a HALT-instruction [8] is executed. The DEBUG-system outputs the message

```
$STOP BY HALT IN <program name>
$STOP IN <line>:<line>
$
```

The procedure

CALL(<file name and extension>,[,<device>[,<ppn>[,<core>]]])

enables the PASCAL programmer to start the execution of another (main) program. This must not be confounded with activating a procedure or function. The execution of the current program is terminated and all variables local to this program are lost.

Execution of CALL has the same effect as issuing a RUN-command on the called program. There is no return to the calling program other than issuing a CALL for it in the called program. Notice that executio then will start again at the beginning of the program. Only the first parameter is mandatory.

The parameters are used as follows:

<file name and extension>

This parameter must be of type PACKED ARRAY[1..9] OF CHAR. It denotes the file containing the executable code of the program to be called. The first 6 characters represent the filename and the last 3 characters the extension. If the latter are blanks, SAV (resp. SHR or HGH and LOW) is assumed (usual case).

<device> - must be specified - 'DSK   '

The parameter must be of type PACKED ARRAY[1..6] OF CHAR. If it is omitted, 'SYS   ' is assumed.

<ppn>

Refer to 5.7.12. for this parameter. If <device> is specified as 'SYS   ', <ppn> must be 0 or defaulted.

<core>

This parameter has to be of type INTEGER. It specifies the amount of core for the low-segment of the called program in K words.

Example:

```
  ...
IF OPTION('CREF      ')
THEN CALL('CROSS     ','SYS   ',0,20);
  ...
```

Temporary core files can be used to pass parameters to called programs (refer to 2.2.).

Example:

```
PROGRAM ONE;
  ...
REWRITE(OUTPUT,'TWO   TMP');
WRITE('DATEN/RESTART'); (* here creating file TWO.TMP *)
  ...
REWRITE(OUTPUT,'DATEN    ');
WRITE(... (* here creating file DATEN *)
  ...
CALL('TWO       ','DSKD  ',222002000B,20);
  ...
END.

PROGRAM TWO( INPUT* );
  ...
READ(... (* here reading file DATEN *)
  ...
IF OPTION('RESTART   ')
THEN CALL('ONE       ','DSKE  ',302003015B,10);
  ...
END (* TWO *).
```

## 5.7.11. GETFILENAME and GETSTATUS

The procedure

```
GETFILENAME( <file variable>, <file name and extension>,
             <protection>, <project-programmer number>,
             <device mnemonic>, <alfa variable or constant> )
```

has 6 mandatory parameters.

GETFILENAME reads a DECSystem-10 file description (as described under 1.2.) from the external file assigned to <file variable> and assigns appropiate values to the following four (VAR-) parameters (refer to 5.7.12.). The format used for the file description is the same as used by the DECSystem-10 Concise Command Language and for program parameters. The first parameter to GETFILENAME must be a TEXT-file which has already been "opened" for input.

If <device> is 'TTY ', the user is prompted with

```
XXXXXXXXXX=
```

to enter the file description where XXXXXXXXXX is the string provided by the sixth parameter. Notice that a Carriage-Return/Line-Feed terminates the file description.

The procedure

```
GETSTATUS( <file variable>, <file name and extension>,
           <protection>, <project-programmer number>,
           <device mnemonic> )
```

can be used to assign appropriate values from the "file control block" of <file variable> to the other parameters (refer to 5.7.12.).

Example; If FILNAM.PAS contains

```
PROGRAM WORK_IT_OUT( INPUT* );
 ...
GETFILENAME(TTY, FIL, PROT, PPN, DEV, 'SOME_FILE ');
RESET(SOME_FILE, FIL, PROT, PPN, DEV);
 ...
GETSTATUS(SOME_FILE, FIL, PROT, PPN, DEV);
IF DEV = 'DSK   '
THEN P
ELSE Q;
 ...
```

the execution could be like

```
.r filnam
INPUT     = data;data.dat
*
 ...
SOME_FILE = other.dat[7,7]
 ...
```

## 5.7.12. RESET and REWRITE

A file must be "opened" with the standard procedure RESET when it is to be used for reading. It must be "opened" with the standard procedure REWRITE when it is to be used for writing.

RESET and REWRITE have the following parameter list:

(<file variable>, <file name and extension>, <protection>, <project-programmer number>, <device mnemonic>)

Only the first parameter is required. The other parameters are optional and used as follows:

<file name and extension>

This parameter must be of type PACKED ARRAY [1..9] OF CHAR. The first 6 characters are used as file name, the next 3 characters as file extension. The parameter is used to overwrite the current file name assigned to <file variable>.

<protection>

This parameter must be of type INTEGER. It is not necessary for input. If 0 is specified, the installation default value is taken (usually <057>). In octal representation this parameter may have values from 0..777B.

<project-programmer number>

This parameter must be of type INTEGER. For the PPN "1023,7777" it would have the following form in octal representation: 1023007777B. The job's PPN is used if the value is 0 or the parameter is missing.

<device mnemonic>

This parameter must be of type PACKED ARRAY [1..6] OF CHAR. It defines the device where the file resides. If this parameter is missing, 'DSK ' is assumed.
If this parameter is specified, new buffer space will be reserved at the first free location possibly after extension of the low segment by the monitor. Therefore the actual core requirement may grow during execution - potentially resulting in program termination due to insufficient core.

In the following example REWRITE is used to assign the actual file TEST.LST to the file variable OUTPUT. The file is created with protection <055>.

Example:

```
REWRITE(OUTPUT,'TEST  LST',55B)
```

### 5.7.13. BREAK and MESSAGE

The standard procedure BREAK is provided in order to force output even if the internal buffer is not yet full. BREAK itself does not insert a Carriage-Return/Line-Feed when used with textfiles e.g. during output to the user terminal. This allows the user to type in on the same line where the output appeared. The line will be ended by a Carriage-Return/Line-Feed only if WRITELN(TTY) was used before the call of BREAK(TTY) or BREAK (because TTY is the default argument of BREAK).

If another file identifier is indicated as argument to the standard procedure BREAK, output to the appropriate file will be enforced even if the output buffer is not yet full. This may be advantageous if a file identifier refers to a computer-computer communication channel as output device. If the file is situated on a directory device like DSK or DTA, the rest of the internal buffer is filled up with 0's. Thus, when reading the file the user must be aware of these 0's.

The procedure

MESSAGE( <argument list> )

is equivalent to

WRITELN(TTY,); WRITELN(TTY, <argument list> ); BREAK(TTY);

### 5.7.14. GETLN and GETLINENR

These procedures are to be used exclusively for TEXT-files.

The procedure

GETLN( <file variable> )

first advances the file to the next Line-Feed. Then it assigns the first character of the next line to the variable <file variable>^. If the TEXT-file contains DECSystem-10 standard line numbering and page marks [8], the new line number is placed into a dedicated location in the "file control block".

The procedure

GETLINENR( [<file variable>,] <line number> )

enables the user to "read" this line number. The number is assigned to the second parameter which must be a variable of type PACKED ARRAY[1..5] OF CHAR. If the first parameter is omitted, INPUT is assumed. A page mark appears to the program as an empty line with no line number. In this case the value ' ' is assigned to <line number>. An additional GETLN is necessary to get the first relevant character of the next line. If no line numbering is

provided by the input file at all, GETLINENR will return '-----' constantly.

Example:

```
PROGRAM LINENUMBERS;
VAR CH: CHAR;
    LINE_NUMBER: PACKED ARRAY[1..5] OF CHAR;
    PAGE_NUMBER: INTEGER;
 ...
PAGE_NUMBER := 1;
GETLINENR(LINE_NUMBER);
 ...
IF EOLN
THEN
 LOOP
  GETLN(INPUT);
  GETLINENR(LINE_NUMBER)
 EXIT IF EOF OR (LINE_NUMBER <> '     ');
  PAGE_NUMBER := PAGE_NUMBER + 1
 END;
 ...
```

### 5.7.15. PUTLN and PAGE

The procedure PUTLN( <file variable> )

writes a Carriage-Return/Line-Feed to the output file.

The procedure PAGE [( <file variable> )]

writes a Carriage-Return/Form-Feed to the output file. The file variable must be of type TEXT. If the parameter is defaulted, OUTPUT is assumed.

### 5.7.16. EOF and EOLN

The function EOF [( <file variable> )]

returns the value TRUE if the user tries to read beyond the physical end of a file, otherwise FALSE is returned. The contents of <file variable>* is undefined if EOF is TRUE. INPUT is assumed if the parameter is defaulted.

If a Line-Feed has been encountered in the input file, a blank is assigned to <file variable>* and the function EOLN [( <file variable> )] returns TRUE otherwise FALSE is returned. INPUT is assumed if no argument is provided. If the user does not care about lines, linenumbers and pagemarks in his file, he does not have to use GETLN or READLN to get the next line. GET or READ invoke GETLN if EOLN is TRUE. EOLN applies to TEXT-files only.

## 5.8. Procedures and Functions as parameters

(This feature was implemented in accordance to G.V.D. KRAATS, Technische Hogeschool Twente)

In DECSystem-10-PASCAL it is necessary to declare the formal parameters of a formal procedure or function. An example may illustrate this:

```
PROGRAM INTEGRATION;

FUNCTION INTEGRATE ( FUNCTION F ( REAL ) : REAL;
                     LOWBOUND,
                     HIGHBOUND,
                     DELTA_X : REAL ) : REAL;

VAR X,SUM : REAL;

BEGIN (*INTEGRATE*)
  (* ASSUME : LOWBOUND < HIGHBOUND AND DELTA_X > 0 *)
  SUM := 0;
  X := LOWBOUND;
  WHILE X < HIGHBOUND DO
   BEGIN
    SUM := SUM + F(X) * DELTA_X;
    X := X + DELTA_X
   END;
  INTEGRATE := SUM
END (*INTEGRATE*);

FUNCTION SINUS (ARGUMENT : REAL) : REAL;

BEGIN (*SINUS*)
  SINUS := SIN(ARGUMENT)
END (*SINUS*);

BEGIN (*INTEGRATION*)
  WRITELN( INTEGRATE(SINUS, 0, 3.14, 0.01) : 6 : 3)
END (*INTEGRATION*).
```

Note that:

a) at the declaration of "F" no identifier is required for the formal parameter of "F", only the type(s) of the argument(s) of formal procedures and functions must be specified.

b) at a call no actual parameters must be specified for procedures and functions which serve as procedure/function parameters. (Obviously, a function which appears at the position of a value parameter must have parameters .)

The declaration of "formal formal parameters" is necessary for the compiler to check the actual procedure- and function- parameters with respect to their parameters and types. This avoids a considerable overhead of parameter- and type-checking at runtime of a PASCAL-program.

The syntax of DECSystem-10-PASCAL is extended as follows:

```
<procedure heading>                ::=
  PROCEDURE <identifier> [<formal parameterlist>] ;
<function heading>                 ::=
  FUNCTION  <identifier> [<formal parameterlist>] :
                                       <type identifier> ;
<formal parameterlist>             ::=
  ( <formal parametersection> [ ; <formal parametersection> ]*)
<formal parametersection>          ::=
  [VAR] <identifierlist> : <type identifier> |
  PROCEDURE <identifierlist> [<formal formal parameterlist>] |
  FUNCTION <identifierlist> [<formal formal parameterlist>] :
                                                <type identifier>
<identifierlist>                   ::=
  <identifier> [ , <identifier> ]*
<formal formal parameterlist>      ::=
  ( <formal formal parametersection>
                    [ ; <formal formal parametersection> ]* )
<formal formal parametersection>   ::=
  [VAR :] <type identifier> |
  PROCEDURE [<formal formal parameterlist>] |
  FUNCTION  [<formal formal parameterlist>] : <type identifier>
```

Important Notes:
---

a) External procedures and functions which are not written in PASCAL must not be used as actual parameters to formal procedures or functions since they obey different parameterconventions,
b) Standard-procedures and -functions are not allowed as actual parameters to formal procedure and function parameters because most of them either
   *) have parameterlists of variable length (e.g. WRITE) or
   *) have parameters of varying types (e.g. PUT) or
   *) are implemented as FORTRAN- or MACRO10-subroutines (e.g. SIN) or
   *) generate in-line code that cannot be passed as an actual parameter(e.g. PRED).

In these cases one may declare a PASCAL-procedure which only calls the appropriate standard or external (non-PASCAL) procedure (as in the example above).

# 6. External Programs

DECSystem-10 PASCAL provides a facility to access external procedures and functions that exist outside the user program and have been separately compiled. This enables the PASCAL programmer to access program libraries.

## 6.1. Declaration of External Procedures or Functions

The declaration of such a procedure or function consists of a heading followed by the word EXTERN or FORTRAN. Thus the procedure/function declaration is extended to the following:

```
<proc./func. decl.> ::= <proc./func. heading>;
                        <proc./func. block>;
<proc./func. block> ::= <block> or
                        EXTERN or
                        FORTRAN
```

## 6.2. How to Compile External Programs

External programs must be compiled with the EXTERN-option. The statement part of an external program (usually referred to as "main program") should only consist of

```
BEGIN END
```

because it cannot be executed if the program is compiled with the EXTERN-option. Example:

```
PROGRAM MAIN;
        VAR R: REAL;
         ...
        FUNCTION SINUS( X: REAL ): REAL; EXTERN;
        BEGIN (* MAIN *)
         ...
        WRITE( TTY, SINUS( R ) );
         ...
        END.

        (*$E+*)
        PROGRAM FUNCTIONS, SINUS, COSINUS...
         ...
        FUNCTION SINUS( ARG: REAL ): REAL;
         ...
        BEGIN (* SINUS *)
         ...
        END;
         ...
        BEGIN (* FUNCTIONS *)
        END.
```

It is also possible to call external FORTRAN subroutines. If the FORTRAN I/O-routines are needed, the calling PASCAL main program must be compiled with the FORTIO-option.

Example:

The program

```
PROGRAM MAIN;
 ...
PROCEDURE FORPRO( ... ); FORTRAN;
 ...
END.
```

must be compiled like

```
.compile main( fortio/runcore:10 )/list
.compile forpro.for/list
.load main,forpro/map
```

Notice that the FORTRAN I/O-routines require at least 6K of the high-segment core.

Inside external PASCAL programs the standard file variables INPUT, OUTPUT and TTY are also pre-declared. They access the same files as they do in the PASCAL main program.

A list of program parameters is ignored for external programs but on principle it is allowed to declare file variables in such programs. In this case the program must be compiled with the FILE-option to guarantee a unique assignation of "data channels" to the files. If there are e.g. 3 files -except standard files- already declared in the calling main program, the FILE-option has to be specified as

```
/FILE:4
```

The value of the FILE-option -default 1- must not be greater than 12.

## 6.3. How to Create a Program Library

A program library containing the REL-files of external PASCAL programs can be easily created just by providing a file containing the source code of all these programs to the compiler. The PASCAL compiler accepts a program library for compilation.

```
<program library> ::= <program> [<program>]*
<program>         ::= <program heading> <block> .
```

Other modules like FORTRAN subroutines or MACRO-10 routines (such routines must correspond in their calling sequence either to the conventions for a PASCAL procedure or function or to those for a FORTRAN subroutine) can be added to the library with the FUDGE2-program [8].

Each of the programs must start on a separate line because a READLN is performed by the compiler when the program end has been encountered.

The E-option cannot be reset inside a program library.

Example:

Let PROGLB.PAS contain

```
(*$E+*)
PROGRAM P1, E11, E12;
 ...
BEGIN (* P1 *)
 ...
END (* P1 *) .
PROGRAM P2, E21, E22;
 ...
BEGIN (* P2 *)
 ...
END (* P2 *) .
```

The program library is created with

```
.compile proglb( extern  )/list
PASCAL: PROGLB [P1: E11, E12]
 ...
PASCAL: PROGLB [P2: E21, E22]
 ...
```

The library can be examined and modified with the FUDGE2-program

```
.r fudge2
*tty:=proglb.rel/ss
P1      E11 E12
P2      E21 E22
*^C
```

Now programs can be loaded in "library search mode" like

```
.compile filnam/list
.load filnam,/search proglb/map
```

# 7. The PASCAL DEBUG-system [10]

The PASCAL DEBUG-system is only accessible to programs which have been compiled with the DEBUG-option. The system can be used to set breakpoints at specified linenumbers. When a breakpoint is encountered, program execution is suspended and variables (using normal PASCAL notation) may be examined and new values may be assigned to them. Also additional breakpoints may be set or breakpoints may be cleared. It is helpful to have a listing of the program available as the system is linenumber oriented. The program should be saved with w having a value of at least u + v + 15 (see Section 1.3.).

## 7.1. Commands

The commands described here can be used when the system enters a breakpoint. When the program is executed it responds with an asterisk if input from TTY is required. After a Carriage-Return has been typed, the initial breakpoint (set by the system) will be entered with the message

```
$DEBUG: <program name>
$
```

Additional breakpoints are set by

```
STOP <line>
```

where <line> is of the form linenumber/pagenumber or just linenumber which is equivalent to linenumber/1 . e.g.: 120/3 . A maximum of 20 breakpoints may be set simultaneously.

The breakpoint is cleared by

```
STOP NOT <line>
```

The breakpoints set may be listed by

```
STOP LIST
```

Variables may be examined by the command

```
<variable> =
```

<variable> may be any variable as allowed by the PASCAL definition. In particular it may be just a component of a structured variable or the whole structure itself. The buffervariable FILNAM^ connected with the file identifier FILNAM as well as the file identifier itself can be accessed. If the file identifier is examined, the contents of the "file control block" are given.

The command [12]

STACKDUMP

will generate a TEXT-file with a source-level dump of the current stack content. If the DEBUG-option has been switched off locally in the program text, the error message "There is no information about this part of the program" may be output. Similarly , the command

HEAPDUMP

will generate a TEXT-file with a source-level dump of the current heap content. If the DEBUG-option has been switched off locally, the error messages "can't continue the heap dump" or "type of referenced variable not known" might appear. After STACKDUMP or HEAPDUMP the Debug-system outputs the message

$LOOK FOR DUMP ON FILE XXXXXX.PMD

where XXXXXX is some fantasy name.

A new value may be assigned to a variable by

<variable> := <variable or constant>

The assignment follows the usual type rules of PASCAL.

The currently active call sequence of procedures and functions i obtained by

TRACE

The names of the procedures and functions together with line numbers of their activation are printed in reverse order of their activation.

Program execution is continued by the command

END

The program will run until another breakpoint is encountered. The breakpoint is announced by

$STOP AT <line>
$

## 7.2. Program Interrupts

If a program -compiled with the DEBUG-option- is interrupted by any run-time error (refer to 8.4.) or by

^C^C

the DEBUG-system is usually automatically entered or -if not- it is possible to (re-) enter the DEBUG-system with the monitor command DDT. The DEBUG-system outputs the message

```
$STOP BY RUNTIME ERROR IN <program name>
$STOP IN <line>:<line>
$
```

or

```
$STOP BY DDT COMMAND IN <program name>
$STOP IN <line>:<line>
$
```

If the program is running in "batch mode", the DEBUG-system is automatically entered and a post-Mortem dump is generated.

## 7.3. How to Debug External Programs

If the main program and/or several external programs have been compiled with the DEBUG-option, it is the loading sequence -the sequence in which the programs are specified in the LOAD-command- that indicates which program is to be debugged. ONLY the first program in the loading sequence may be debugged provided it has been compiled with the DEBUG-option.

Example:

```
.LOAD MAIN, P1, P2          MAIN can be debugged
.LOAD P1, MAIN, P2          now P1 can be debugged
```

# 8. Tables

## 8.1. Operations

### 8.1.1. Summary

| Operator | Operation | Type of Operand(s) | Result Type |
|---|---|---|---|
| **----------Arithmetic---------- | | | |
| + (unary) | identity | INTEGER or REAL | same as operand |
| - (unary) | sign inversion | | |
| + | addition | | INTEGER or REAL |
| - | subtraction | | |
| * | multiplication | | |
| / | REAL division | | REAL |
| DIV | INTEGER div, | INTEGER | INTEGER |
| MOD | modulus | INTEGER | |
| **----------Relational---------- | | | |
| = | equality | any scalar, string, SET or pointer | BOOLEAN |
| <> | inequality | | |
| < | less-than | any scalar or string | |
| > | greater-than | | |
| <= | less-equal, set inclusion | any scalar, string, or SET | |
| >= | greater-equal, set inclusion | | |
| IN | set membership | 1st op. scalar, 2nd op. its SET type | |
| **----------Logical---------- | | | |
| NOT | negation | BOOLEAN | BOOLEAN |
| OR | disjunction | | |
| AND | conjunction | | |
| **----------SET---------- | | | |
| + | set union | SET | same type |
| - | set difference | | |
| * | set intersection | | |
| **----------Assignment---------- | | | |
| := | assignment | any compatible types except file types | --- |

If both INTEGER and REAL values appear in expressions, the result type is always REAL.

### 8.1.2. Precedence

| | Arithmetic | Logical | SET |
|---|---|---|---|
| 1st | + (unary)<br>- (unary) | NOT | |
| 2nd | MOD, DIV,<br>*, / | AND | * |
| 3rd | +, - | OR | +, - |
| 4th | =, <, <=,<br><>, >, >= | =, <, <=,<br><>, >, >= | <=, >=, IN |

## 8.2. Reserved Words

IF, DO, OF, TO, IN, OR,

END, FOR, VAR, DIV, MOD, SET, AND, NOT,

THEN, ELSE, WITH, GOTO, LOOP, CASE, TYPE, FILE, EXIT,

BEGIN, UNTIL, WHILE, ARRAY, CONST, LABEL,

EXTERN, RECORD, DOWNTO, PACKED, OTHERS, REPEAT,

FORTRAN, FORWARD, PROGRAM,

FUNCTION,

PROCEDURE, SEGMENTED,

INITPROCEDURE

## 8.3. Standard Procedures and Functions

### 8.3.1. Procedures

Procedures marked with (*) are also defined in PASCAL 6000-3.4, those marked with (**) only in DECSystem-10 PASCAL. The marked procedures and functions are not part of STANDARD PASCAL.

Input/Output:
RESET, REWRITE, GET, PUT, PAGE, READ, READLN, WRITE, WRITELN, MESSAGE (*), GETLN (**), PUTLN (**), GETLINENR (**), BREAK (**)

Execution Control:
HALT (*), CALL (**)

Allocation of Dynamic Storage:
NEW, DISPOSE (*)

Communication with the Enviroment:
DATE (*), TIME (*), GETFILENAME (**), GETSTATUS (**), GETOPTION (**)

### 8.3.2. Functions

| Function | Type of Argument(s) | Result Type |
|---|---|---|
| **----------STANDARD PASCAL----------** | | |
| ARCTAN | INTEGER or REAL | REAL |
| COS | | |
| EXP | | |
| LN | | |
| SIN | | |
| SQRT | | |
| ABS | | INTEGER or REAL |
| SQR | | |
| ROUND | | INTEGER |
| TRUNC | | INTEGER |
| ODD | INTEGER | BOOLEAN |
| EOF | any file | |
| EOLN | TEXT | |
| PRED | any scalar except REAL | argument type |
| SUCC | | |
| CHR | INTEGER | CHAR |
| ORD | any scalar or pointer | INTEGER |

| Function | Type of Argument(s) | Result Type |
|---|---|---|
| **----------PASCAL 6000-3,4---------- | | |
| CARD | any SET | INTEGER |
| CLOCK | --- | |
| EXPO | REAL | |
| **----------DECSystem-10 PASCAL---------- | | |
| ARCCOS | INTEGER or REAL | REAL |
| ARCSIN | | |
| COSD | | |
| COSH | | |
| LOG | | |
| RANDOM | | 0.0..1.0 |
| SIND | | REAL |
| SINH | | |
| TANH | | |
| REALTIME | --- | INTEGER |
| OPTION | ALFA | BOOLEAN |
| FIRST | any scalar except REAL | argument type |
| LAST | | |
| LOWERBOUND | array | array index type |
| UPPERBOUND | | |
| MIN | any scalar except BOOLEAN | scalar type |
| MAX | | |

If both INTEGER and REAL values appear in the argument list for MIN or MAX, the result type is REAL.

## 8.4. Run-time Error Messages

The run-time error messages of the PASCAL run-time support have the general format

```
%? <message text> AT USER PC <octal address>
```

The following is a list of all run-time messages that might be output during execution of a PASCAL program.

Address Checks:

```
POINTER OUT OF BOUNDS; CANNOT RETAIN VARIABLE
HEAP OVERRUNS STACK; RETRY WITH MORE CORE
NOT ENOUGH CORE TO READ TEMPCORE-FILE <file name>
STACK OVERRUNS HEAP; RETRY WITH MORE CORE
CORE REQUIREMENT GREATER THAN "CORMAX"
ARRAY INDEX OUT OF BOUNDS
```

DEBUG-system:

```
PROGRAMS COMPILED WITH THE DEBUG-OPTION MUST NOT BE SHARABLE:
   RETRY WITH .SAVE INSTEAD OF .SSAVE
ILLEGAL MEMORY REFERENCE
TIME LIMIT EXCEEDED
DEBUG-SYSTEM ERROR: <error number>
```

Input/Output:

```
INPUT ERROR: INVALID SCALAR SPECIFICATION *** <scalar> ***
INPUT ERROR; SCALAR UNDEFINED OR OUT OF RANGE
   *** <scalar> ***
INPUT ERROR: INVALID SET SPECIFICATION
INPUT ERROR: SET ELEMENT SPECIFIED TWICE *** <scalar> ***
NO ACCESS TO OR NO DISK SPACE FOR FILE <file name>:
   ERROR IN REWRITE
REWRITE FOR FILE <file name> REQUIRED
INPUT ERROR: ATTEMPT TO READ BEYOND EOF OF <file name>
INPUT ERROR: RESET REQUIRED FOR <file name>
INPUT DATA ERROR IN FILE <file name>
SCALAR OUT OF RANGE IN FILE <file name>
OUTPUT ERROR: DISK SPACE EXHAUSTED FOR FILE <file name>
```

Arithmetic Operations:

```
ARITHMETIC OVERFLOW OR ZERODIVIDE
SCALAR OUT OF RANGE
MORE THAN 72 SET ELEMENTS
```

Program Parameters:

```
NO ACCESS TO <file name> OR NOT FOUND: REENTER
SYNTAX ERROR: REENTER
```

Program Execution:

```
CANNOT RUN <file name>
```

## 8.5. ASCII Table

The PASCAL character set (the type CHAR) is encircled in the following table.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|---|
| | 00 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | |
| | 01 | BS | HT | LF | VT | FF | CR | SO | SI | |
| | 02 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | |
| | 03 | CAN | EM | SUB | ESC | FS | GS | RS | US | |
| I | 04 | SP | ! | " | # | $ | % | & | ' | I |
| I | 05 | ( | ) | * | + | , | - | . | / | I |
| I | 06 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | I |
| I | 07 | 8 | 9 | : | ; | < | = | > | ? | I |
| I | 10 | @ | A | B | C | D | E | F | G | I |
| I | 11 | H | I | J | K | L | M | N | O | I |
| I | 12 | P | Q | R | S | T | U | V | W | I |
| I | 13 | X | Y | Z | [ | \ | ] | ^ | _ | I |
| | 14 | ` | a | b | c | d | e | f | g | |
| | 15 | h | i | j | k | l | m | n | o | |
| | 16 | p | q | r | s | t | u | v | w | |
| | 17 | x | y | z | { | \| | | | DEL | |

# 9. Miscellaneous

## 9.1. Implementation Restrictions

(a) A maximum of 12 files may be declared by the user.

(b) Arrays of files and records with files as components are not implemented.

(c) Segmented files are not implemented.

(d) Call of external COBOL or ALGOL procedures or functions are not implemented.

(e) A SET may contain a maximum of 72 elements of scalar or subrange types -except REAL-. Thus, SETs of ASCII or INTEGER are not possible. Only subranges consisting of not more than 72 values are allowed (e.g. CHAR). The following rules are valid for these subranges:

| Range Type | Restriction |
|---|---|
| ASCII or CHAR | ORD(FIRST(<subrange>)) >= 40B and ORD(LAST(<subrange>)) <= 147B |
| other | ORD(FIRST(<subrange>)) >= 0 and ORD(LAST(<subrange>)) <= 71 |

(f) A range of SET-elements in a set must be given using constant bounds only (no expression as bound is allowed).

(g) No explicit runtime check for NIL as pointervalue is performed yet.

## 9.2. Known Bugs

(a) Comparison of entire variables of type PACKED RECORD or PACKED ARRRAY may cause errors if these variables are used with a variant part.

(b) Alphabetical ordering of PACKED ARRAY [1..n] OF CHAR may result in a different ordering from that obtained with the same character sequences represented simply as ARRAY [1..n] OF CHAR. (This is due to the fact that the most significant bit of the first character in a word represents the sign bit during the arithmetic word-compare used for PACKED ARRAY"s whereas it has no such influence for ARRAY [1..n] OF CHAR which are compared character by character).

## 9.3. Utility Programs

CROSS is a program which formats a PASCAL program, produces a cross-reference list of identifiers, indicates the nesting of statements and reports the static nesting of procedures. The program can be executed once by providing the CREF-option in the COMPILE-command.

```
.COMPILE FILNAM/CREF
```

The program then will create the newly formatted source file on FILNAM.NEW and the cross-reference list on FILNAM.CRL.

Another way is to execute CROSS directly with

```
.R CROSS [<core assignment>]
```

Now, like PASCAL does if executed directly, the program asks for the names of the following files:

```
OLDSOURCE = filnam.pas          old source code
NEWSOURCE = filnam.new          formatted source code
CROSSLIST = filnam.crl          cross-reference list
```

## 10. References

[1] N. Wirth
The Programming Language PASCAL
Acta Informatica 1, 35 (1971)
and
Revised Report
Bericht Nr. 5
Berichte der Fachgruppe Computer-Wissenschaften
ETH Zuerich, July 1973

[2] K. Jensen, N. Wirth
PASCAL - User Manual and Report
Lecture Notes in Computer Science, Vol 18
Springer Verlag Berlin, Heidelberg, New York, 1974

[3] C.A.R. Hoare and N. Wirth
An Axiomatic Definition of the Programming Language PASCAL,
Bericht Nr. 6
Berichte der Fachgruppe Computer-Wissenschaften
ETH Zurich, November 1972

[4] K.V. Nori, U. Ammann, K. Jensen, H.H. Naegeli
The PASCAL-P Compiler: Implementation Notes
Bericht Nr. 10
Berichte der Fachgruppe Computer-Wissenschaften
ETH Zurich, December 1974

[5] C.-O. Grosse-Lindemann, H.-H. Nagel
Postlude to a PASCAL-Compiler bootstrap on a DEC System-10
Bericht Nr. 11
Institut fuer Informatik der Universitaet Hamburg, October 1974
and
Software - practice and Experience 6, 29-42 (1976)

[6] G. Friesland, C.-O. Grosse-Lindemann, F.H. Lorenz
H.-H. Nagel, P.-J. Stirl
A PASCAL compiler bootstrapped on a DECSystem-10
in 3. GI-Fachtagung ueber Programmiersprachen
Lecture Notes in Computer Science, Vol 7, page 101
Springer Verlag Berlin, Heidelberg, New York, 1974

[7] H.-H. Nagel
PASCAL for the DECSystem-10, Experiences and further Plans
Mitteilung Nr. 21
Institut fuer Informatik der Universitaet Hamburg, November 1975

[8] DECSystem-10 Operating System Commands
Digital Equipment Corporation, Maynard/ Massachussetts,1971(1974)

[9] DECSystem-10 FORTRAN-10 Language Manual
Digital Equipment Corporation, Maynard/ Massachussetts,1967(1974)

[10]P.Putfarken
Testhilfen fuer PASCAL-Programme
Diplomarbeit
Institut fuer Informatik der Universitaet Hamburg
November 1976

[11]W.F. Burger, H.-H. Nagel
PASCAL on the DEC10
Technical Report No. 22B
The University of Texas at Austin
Austin, Texas 78712

[12]B.Nebel, B.Pretschner
Erweiterung des DECSystem-10 PASCAL-Compilers um eine
Moeglichkeit zur Erzeugung eines Post-Mortem-Dump
Mitteilung Nr. 34
Institut fuer Informatik der Universitaet Hamburg, Juni 1976