HP 3000 Computer Systems

HEWLETT PACKARD

PASCAL/3000 Reference Manual

HP 3000 Computer System

PASCAL/3000

Reference Manual

19420 HOMESTEAD RD., CUPERTINO, CALIFORNIA 95014

Part No. 32106-90001

Printed in U.S.A. 10/83
E1083

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition and the dates when pages were changed in updates to that edition. Within the manual, any page changed since the last edition has the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. When an update is incorporated in a subsequent reprinting of the manual, these bars are removed.

First Edition	Dec 1981
Second Edition	Oct 1983

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

The software product part number printed alongside the date indicates the version and update level of the software product at the time the manual edition or update was issued. Many product updates and fixes do not require manual changes, and conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition	Dec 1981	32106A.00.00
Second Edition	Oct 1983	32106A.00.00

CONTENTS

SECTION 1 - THE PASCAL/3000 LANGUAGE

SECTION 2 - DECLARATIONS

CONTENTS

SECTION 3 - STATEMENTS

SECTION 4 - EXPRESSIONS

SECTION 5 - TOKENS

CONTENTS

SECTION 6 - I/O

CONTENTS

SECTION 7 - STANDARD PROCEDURES AND FUNCTIONS

CONTENTS

SECTION 7 (Continued)

SECTION 8 - COMPILER OPTIONS

CONTENTS

SECTION 8 (Continued)

SECTION 9 - STORAGE AND EXECUTION EFFICIENCY

SECTION 10 - USING PASCAL/3000

CONTENTS

APPENDIX A - PASCAL/3000 SYNTAX DIAGRAMS

APPENDIX B - RESERVED WORDS AND STANDARD IDENTIFIERS

APPENDIX C - COMPILE-TIME ERRORS

APPENDIX D - RUN-TIME ERRORS

APPENDIX E - UNDETECTED ERRORS

APPENDIX F - USING INTRINSICS

APPENDIX G - PASCAL/3000 AND OTHER LANGUAGES

CONTENTS

APPENDIX H - PASCAL/3000 AND HP3000 SUBSYSTEMS

APPENDIX I - I/O DEFINITIONS

TABLES AND FIGURES

THE PASCAL/3000 LANGUAGE

SECTION I

INTRODUCTION

> On rare occasions in programming language development there appears a programming language which is widely recognized as superior, and which propagates itself among discerning implementors and users solely by its merits, and without any political or commercial backing. ALGOL 60 was such a language. Pascal is another.
>
> —Welsh, Sneeringer, and Hoare quoted in *Tutorial: Programming Language Design* by Wasserman, p. 284

Niklaus Wirth designed the programming language Pascal in 1968 as a vehicle for teaching the fundamentals of structured programming and as a demonstration that it was possible to efficiently and reliably implement a 'non-trivial' high level language.

Since then, Pascal has established itself as the dominant programming language in university-level computer science courses. It has also become an important language in commercial software projects, especially in systems programming.

Pascal/3000 is a version of Pascal intended for the HP3000 computer. The Pascal/3000 compiler implements Pascal/3000 by compiling Pascal/3000 source code into HP3000 object code and storing this code in a user subprogram library (USL). The MPE Segmenter may subsequently prepare the USL into an executable program file.

Pascal/3000 is a superset of Hewlett-Packard Standard Pascal, a company-standard language currently implemented on several Hewlett-Packard computers. HP Standard Pascal, in turn, is a superset of American National Standards Institute (ANSI) Pascal.

Subsequent pages of this section outline the organization of this manual and summarize the HP Standard Pascal and Pascal/3000 extensions. The experienced Pascal programmer may use these summaries as a guide for further study of unfamiliar features.

MANUAL ORGANIZATION

This manual fully describes Pascal/3000. The reader wishing to learn Pascal should refer to an introductory text.

Sections 2 through 5 of this manual discuss the features of Pascal/3000 in top-down fashion, starting with programs and concluding with lexical tokens.

Section 6 explains Pascal/3000 files and the various procedures and functions which the programmer may use to manipulate them.

Section 7 presents the standard operations and functions supported by Pascal/3000.

Section 8 discusses the Pascal/3000 compiler options.

Section 9 explains the storage requirements of the various Pascal/3000 data types and shows how the programmer can optimize storage and execution efficiency.

Section 10 discusses ways to invoke the Pascal/3000 compiler using various MPE commands.

Finally, several appendices present supplementary information.

Throughout this document, Pascal/3000 reserved words, compiler options, and directives appear in upper case, e.g. BEGIN, USLINIT, FORWARD. Standard identifiers appear in italics, e.g. *readln, maxint, text.*

Appendix B lists the Pascal/3000 reserved words and standard identifiers.

In the original Jensen and Wirth *Pascal Report,* the term 'string' refers to any packed array of *char* with a starting index of 1. Pascal/3000, however, supports the standard type *string.* To avoid confusion, the term PAC is used for the type packed array of *char.*

The following is a list of the HP Standard Pascal features which are extensions of ANSI Standard Pascal. For the full description of a feature, the reader should refer to the appropriate pages in subsequent sections.

Identifiers

The underscore character (_) may appear in identifiers, but not as the first character (see Section 5).

Longreal Numbers

The type *longreal* is identical with the type *real* except that it provides greater precision (see Section 2). The letter 'L' precedes the scale factor in a longreal literal (see Section 5).

String Literals

HP Standard Pascal permits the encoding of control characters or any other single ASCII character after the sharp symbol (#) (see Section 5). For example, the string literal #G represents CTRL-G, i.e. the bell.

Constructors (Structured Constants)

The programmer can specify the value of a declared constant with a constructor. In general, a constructor establishes values for the components of a previously declared array, record, string or set type (see Section 2). Record, array, and string constructors may only appear in a CONST section of a declaration part of a block. Set constructors, on the other hand, may also appear in expressions in executable statements and their typing is optional (see Section 4).

Constant Expressions

The programmer may also specify the value of a declared constant with a constant expression. A constant expression returns an ordinal value and may contain only declared constants, literals, calls to the functions *ord, chr, pred, succ, hex, octal, binary,* and the operators +, -, *, DIV, and MOD (see Section 2).

A constant expression may appear anywhere that a constant may appear.

Minint

The standard constant *minint* is defined in the Pascal/3000 system as the integer value -2147483648.

String Type

HP Standard Pascal supports the predefined type *string*. A *string* type is a packed array of *char* with a declared maximum length (see Section 2) and an actual length that may vary at run time.

The programmer may compare a variable of type *string* with a similar variable or a string literal (see Section 4), or assign a string or string literal to a string (see Section 3).

Several standard procedures and functions manipulate strings (see Section 7). *Strlen* returns the current length of a string; *strmax* the maximum length. *Strwrite* writes one or more values to a string; *strread* reads values from a string. *Strpos* returns the position of the first occurrence of a specified string within another string. *Strltrim* and *strrtrim* trim leading and trailing blanks, respectively, from a string. *Strrpt* returns a string composed of a designated string repeated a specified number of times. *Strappend* appends one string to another. *Str* returns a specified portion of a string, i.e. a substring. *Setstrlen* sets the current length of a string without changing its contents. *Strmove* copies a substring from a source string to a destination string. *Strinset* inserts one string into another. *Strdelete* deletes a specified number of characters from a string.

Record Variant Declaration

The variant part of a record field list may have a subrange as a case constant (see Section 2).

Declaration Part

In the declaration part of a block, the programmer can repeat and intermix the CONST, TYPE, and VAR sections (see Section 2). The LABEL section must still precede and the PROCEDURE and FUNCTION sections follow the CONST, TYPE, and VAR sections.

Assignment Compatibility

If T1 is a PAC variable and T2 is a string literal, then T2 is assignment compatible with T1 provided that T2 is not longer than T1. If T2 is shorter than T1, the system will pad T1 with blanks.

If T1 is *real* and T2 is *longreal*, the system truncates T2 to *real* before assignment.

CASE Statement

The reserved word OTHERWISE may precede a list of statements and the reserved word END in a CASE statement. If the case selector evaluates to a value not specified in the case constant list, the system executes the statements between OTHERWISE and END (see Section 3). Also, subranges may appear as case constants.

WITH Statement

The record list in a WITH statement may include a call to a function which returns a record as its result (see Section 3).

Function Return

A function may return a structured type, except the type file. That is, a function may return an array, record, set or string (see Section 2).

I/O

The programmer may open a file which is not a textfile for direct access with the procedure *open*. Direct access files have a maximum number of components, indicated by the function *maxpos*. The procedure *seek* places the current position of a direct access file at a specified component. The programmer can read from a direct access file or write to it with the procedures *readdir* or *writedir*, which are combinations of *seek* and the standard procedures *read* or *write*.

The programmer may open any file in the 'write-only' state without altering its contents using the procedure *append*. The current position after *append* is the end of the file.

The programmer may explicitly close any file with the procedure *close*.

To permit interactive input, the system defines the primitive file operation *get* as 'deferred get' (see Appendix I).

The procedure *read* accepts any simple type as input. Thus, it is possible to read a *boolean* or enumerated value from a file. It is also possible to read a value which is a packed array of *char* or *string*.

The procedure *write* accepts identifiers of an enumerated type as parameters. The programmer may write an enumerated constant directly to a file.

The function *position* returns the index of the current position for any file which is not a textfile. The function *linepos* returns the integer number of characters which the program has read from or written to a textfile since the last line marker.

The procedures *page, overprint,* and *prompt* operate on textfiles. *Page* causes a page eject when a text file is printed. *Overprint* causes the printer to perform a carriage return without a line feed, effectively overprinting a line. *Prompt* flushes the output buffer without writing a line marker. This allows the cursor to remain on the same screen line when output is directed to a terminal.

Section 6 describes files and I/O operations in detail.

Heap Procedures

The procedure *mark* marks the state of the heap. The procedure *release* restores the state of the heap to a state previously marked. This has the effect of deallocating all storage allocated by the new procedure since the program called a particular *mark* (see Section 7).

Halt Procedure

The *halt* procedure causes an abnormal termination of a program (see Section 7).

Numeric Conversion Functions

The functions *binary, octal,* and *hex* convert a parameter of type string or PAC, or a string literal, to an integer. *Binary* interprets the parameter as a binary value; *octal* as an octal value; *hex* as a hexadecimal value (see Section 7).

Compiler Options

Compiler options appear between dollar signs ($). HP Standard Pascal has five options: ANSI, PARTIAL__EVAL, LIST, PAGE, and INCLUDE. ANSI sets the compiler to issue warnings in the listing when source code includes features which are not legal in ANSI Standard Pascal. PARTIAL__EVAL permits the partial evalution of boolean expressions. LIST allows the programmer to suppress the compiler listing. PAGE causes the listing to resume on the top of the next page. INCLUDE specifies a source file which the compiler will process at the current position in the program.

The following is a list of Pascal/3000 features which are extensions of HP Standard Pascal. For a full description of these features, the reader should refer to the relevant pages in subsequent sections.

Directives

EXTERNAL and INTRINSIC are legal Pascal/3000 directives fully described in Section 2.

EXTERNAL indicates that the system will find a procedure or function in an external compilation unit. The programmer may qualify EXTERNAL with the terms SPL, SPL VARIABLE, FORTRAN, or COBOL. SPL indicates the external procedure or function is in SPL without option variable parameters; SPL VARIABLE that it is in SPL and has option variable parameters; FORTRAN that it is in FORTRAN; and COBOL that it is in COBOL 68 or COBOL II.

INTRINSIC indicates the declared procedure or function is a MPE or user-defined intrinsic. The formal parameter list of a procedure or function declared with the INTRINSIC directive is optional. That is, the call to the procedure may contain actual parameters even if no formal parameters appear in the declaration. Furthermore, the system will perform certain conversions of the actual parameters (see Appendix F).

Procedure and Function Calls

Calls of a procedure or function declared EXTERNAL SPL VARIABLE or INTRINSIC (where there are option variable parameters) may omit actual parameters. The programmer must specify empty option variable parameters with the comma (,) (see Section 3).

Ccode Function

The *ccode* function returns an integer in the range 0..2 which indicates the condition code after an intrinsic call (see Section 7).

Fnum Function

The *fnum* function returns an integer which indicates the value of the MPE file number of the physical file associated with a logical file.

Sizeof Function

The *sizeof* function returns the size in bytes of the storage required for a variable (see Section 7).

Waddress and Baddress Functions

The *waddress* function returns the DB relative word address of a variable, or the external P label of a procedure or function (see Section 7). The *baddress* function returns the DB relative byte address of a variable (see Section 7).

Assert Procedure

The *assert* procedure evaluates a boolean expression and, provided the compiler option ASSERT_HALT is ON, aborts a program when the expression is *false*. It is also possible to specify an optional procedural parameter. When the expression is *false*, the system will execute this procedure before terminating the program (see Section 7).

Compiler Options

Pascal/3000 supports a number of compiler options fully described in Section 8 and briefly summarized here.

ALIAS permits the programmer to specify an external name for a procedure or function which is different from its declared name.

ASSERT_HALT causes program termination when the boolean expression in a call to the *assert* procedure is *false*.

CHECK_ACTUAL_PARM or CHECK_FORMAL_PARM specify the level of checking the system will perform for actual or formal parameters.

CODE permits the programmer to suppress the generation of object code for a portion of source code.

CODE_OFFSETS shows the p register offsets for statements.

COPYRIGHT inserts a copyright notice in the USL and program files.

EXTERNAL and GLOBAL permit the separate compilation of procedures and functions.

HEAP_COMPACT causes the system to concatenate free space in the heap. HEAP_DISPOSE lets the system reallocate disposed areas in the heap.

LINES sets the number of listing lines per page.

LIST_CODE produces a mnemonic listing of the HP3000 machine code generated by the compiler.

PRIVATE_PROC permits non level 1 procedures or functions to be compiled into separate relocatable binary modules with 'public' entry points.

RANGE causes the compiler to generate range checking code for assignments, array indexing, parameter passing, and pointer dereferencing.

SEGMENT changes the current segment name to a specified name.

SKIP_TEXT causes the compiler to ignore source code.

SPLINTR specifies a file which the system will search when a program declares an intrinsic.

STANDARD_LEVEL specifies the level of legal Pascal syntax. The compiler issues a warning when encountering a feature not permitted at the specified level.

SUBPROGRAM allows independent compilation of specified level 1 procedures or functions.

TABLES produces an identifier map for each compilation block.

TITLE places a specified title at the top of the listing page.

USLINIT causes the compiler to initialize the designated USL to empty before placing any object code in it.

WIDTH instructs the compiler to process a specified number of columns of source text.

XREF tells the compiler to prepare and issue a cross reference of a compilation block.

As well as these language features, Pascal/3000 provides three support library routines which are accessible from other HP3000 subsystems or languages. GETHEAP allocates a region of the DL-DB area of the stack; RTNHEAP deallocates a region of the DL-DB area; HP32106 returns the version name for the currently installed Pascal/3000 support library. Appendix F describes these three procedures.

When called by Pascal, subsystems such as VPLUS which use the DL-DB area of the stack call GETHEAP and RTNHEAP to avoid possible conflict with the Pascal heap (see Appendix H).

COMPILING PASCAL/3000 PROGRAMS

An Overview

The Pascal/3000 compiler scans and parses Pascal/3000 source code and then emits HP3000 object code into a USL file. The compiler produces object code for one procedure, function, or outer block at a time and the programmer may control this code generation with various compiler options. For example, the SUBPROGRAM compiler option instructs the compiler to issue object code only for specified level 1 procedures or functions and their nested procedures or functions, and to suppress code generation for the outer block (see Section 8). Also, the compiler options EXTERNAL and GLOBAL permit the separate compilation of procedures or functions without redeclaring all global variables (see Section 8).

As it processes source code, the Pascal/3000 compiler produces a program listing. The programmer may suppress this listing by setting the LIST compiler option OFF, or enhance it with several optional features such as an identifier map, a cross reference, or code offsets (see Section 8).

To invoke the Pascal/3000 compiler, the programmer may select one of four MPE commands described in Section 10. The MPE command :PASCAL processes a source text into a USL file; the command :PASCALPREP compiles the source text and then prepares the resulting USL into a program file; the command :PASCALGO compiles, prepares, and then executes a program; finally, the :RUN commmand can directly invoke the compiler, which is a program file fully specified by the name PASCAL.PUB.SYS.

Since the compiler opens the source file "read only" with semiexclusive access to insure that no one writes to the file during compilation, the use of semiexclusive access requires that the group have LOCK access.

DEVELOPING PASCAL/3000 PROGRAMS WITH HPTOOLSET

Pascal programs can be developed, compiled, prepped, run and symbolically debugged using the program development utility HPToolset. TOOLSET contains an Editor, a Program key to translate and run your code, and a symbolic debugger that alleviates having to know memory locations or convert source statements into code statements.

See the HPToolset Reference Manual for details on how to run TOOLSET and how to use each of its features.

DECLARATIONS

SECTION II

PROGRAM FORM

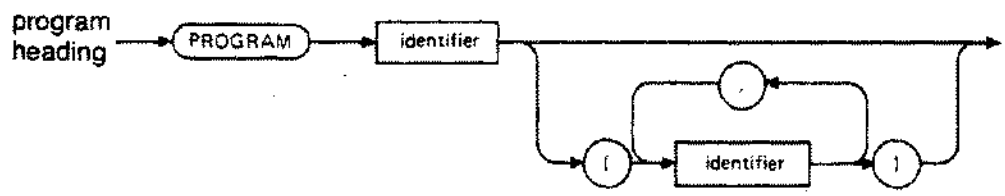
The Pascal/3000 compiler will successfully compile any Pascal/3000 source code which conforms to the syntax and semantics of a Pascal/3000 program. The form of a Pascal/3000 program consists of a program heading, a semi-colon (;), an outer block, and a period.

Syntax

Compilation fails when any of these elements are missing.

The program heading consists of the reserved word PROGRAM, an identifier (the program name) and an optional parameter list.

Syntax



The identifiers in the parameter list are variables which the programmer must declare in the outer block, except for the standard textfiles *input* and *output*.

Input and *output* are standard file variables which the system associates by default with the MPE files $STDIN and $STDLIST and which it opens automatically at the beginning of program execution (see Section 6). In Pascal/3000, *input* or *output* need only appear as program parameters if some file operation, e.g. *read* or *write*, refers to them explicitly or by default.

Program parameters are often the names of file variables, but a logical file, i.e. a file declared in the program, need not necessarily appear as a program parameter. The advantage of putting the name of the logical file in the program parameter list is that the system will use the first 8 characters of this name as the default name for the MPE file associated with the program's logical file (see Section 6).

PROGRAM FORM

Other types of variables may appear in the parameter list of the program heading. In particular, a variable of type *integer*, subrange of *integer*, PAC, or *string* may occur. Such a variable will capture the value of the PARM or INFO parameter of the MPE :RUN command. In other words, the programmer may pass the integer value of the PARM parameter or the character string value of the INFO parameter to a Pascal/3000 program at run time (see Section 10). For example, the INFO parameter can pass the name or names of physical files which the programmer wishes to associate with the logical files in a program (see Section 6).

The outer block of a program consists of an optional declaration part and a required statement part.

Syntax

The declaration part consists of definitions of labels, constants and types, and declarations of variables, procedures and functions. The statement part is made up of a compound statement which may be empty or may contain several simple or structured statements (see Section 3). The statement part is also termed the 'body' or 'executable portion' of the block.

The outer block of a program is identical with the block of a procedure declaration, except that it terminates with a period (.).

Examples

```
PROGRAM minimum;              {The minimum program the Pascal/3000 }
BEGIN                         {compiler will process successfully: }
END.                          {no program parameters;              }
```

```
PROGRAM show_form1 (output); {Uses the standard textfile output}
BEGIN                         {and the standard procedure       }
  writeln ('Greetings!')      {writeln.                         }
END.
```

```
PROGRAM show_form2 (i,f);  {The program parameters are declared}
VAR                        {in the declaration part. The PARM  }
  i: integer;              {parameter of the :RUN command will }
  f: FILE OF integer;      {pass a value to i. The second      }
BEGIN                      {executable statement in the body of}
  append (f);              {outer block writes this value on   }
  write (f,i);             {the file f.                        }
END.
```

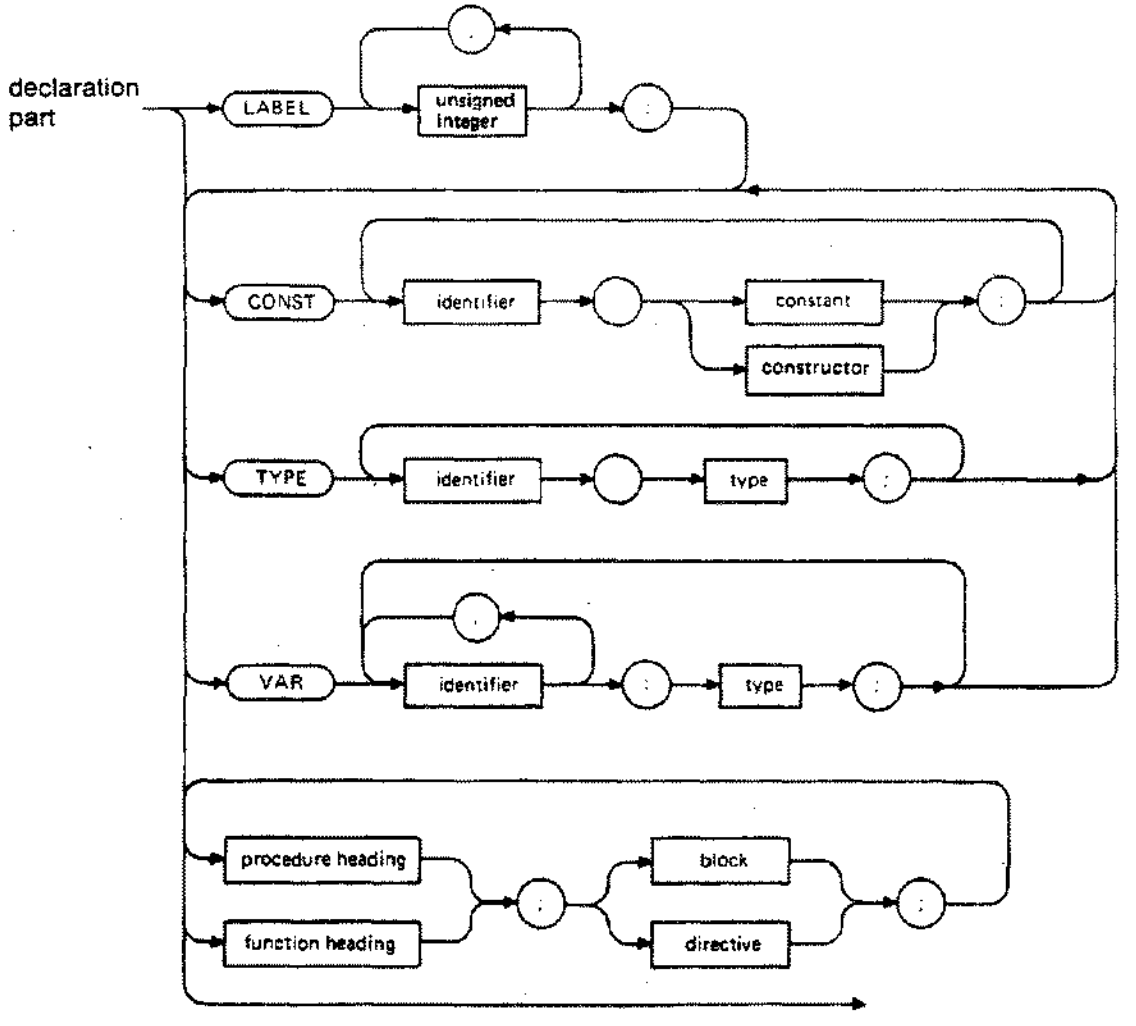
```
PROGRAM show_form3 (input,output);
VAR
  a,b,total: integer;
FUNCTION sum (i,j: integer): integer;   (Function declaration  )
  BEGIN                                 {with an inner block   }
    sum:= i + j                         {which is not part of  }
  END;                                  {outer block.          }
BEGIN
  write ('Enter two integers: ');
  prompt;
  readln (a,b);
  total:= sum (a,b);
  writeln ('The total is: ', total)
END.
```

DECLARATION PART

The declaration part of an HP Pascal block defines the labels, declared constants, data types, variables, procedures, and functions which will appear in the executable statements in the body of the block.

The reserved word LABEL precedes the declaration of labels; CONST or TYPE the definition of declared constants or types; VAR the declaration of variables; PROCEDURE or FUNCTION the declaration of a procedure or a function.

Syntax



Within a declaration part, label declarations must come first; procedure or function declarations last. The programmer, however, may intermix and repeat CONST and TYPE definition sections and VAR declaration sections (see example below). This is an HP Standard Pascal extension of ANSI Standard Pascal.

ANSI Standard Pascal does not allow any of the reserved words LABEL, CONST, TYPE or VAR to be used more than once.

The programmer can, but usually will not, redeclare or redefine a standard declared constant, type, variable, procedure or function in the declaration part.

Example

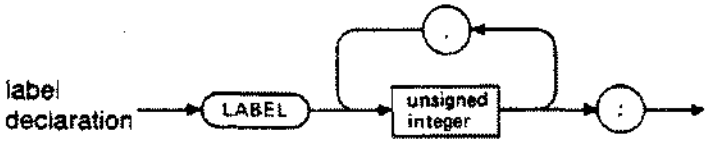
```
PROGRAM show_declarepart;
LABEL  25;
VAR
  birthday: integer;
TYPE
  friends = (Joe, Simon, Leslie, Jill);
CONST
  maxnuminvitee = 3;
VAR
  invitee: friends;
PROCEDURE celebrate; EXTERNAL;     {End of declaration part.}

BEGIN                              {Beginning of body.      }
  .
  .
END.
```

LABEL DECLARATION

A label declaration specifies integer labels which mark executable statements in the body of the block. The GOTO statement transfers control to a labeled statement (see Section 4).

The reserved word LABEL precedes one or more integers separated by commas.

Syntax



Integers must be in the range 0 to 9999. Leading zeros are not significant. For example, the labels 9 and 00009 are identical.

Label declarations must come first in the declaration part of a block.

The programmer cannot use a label to mark a statement in a procedure or function nested within the procedure, function, or outer block where the label is declared. This means a GOTO statement may jump out of but not into a procedure.

Example

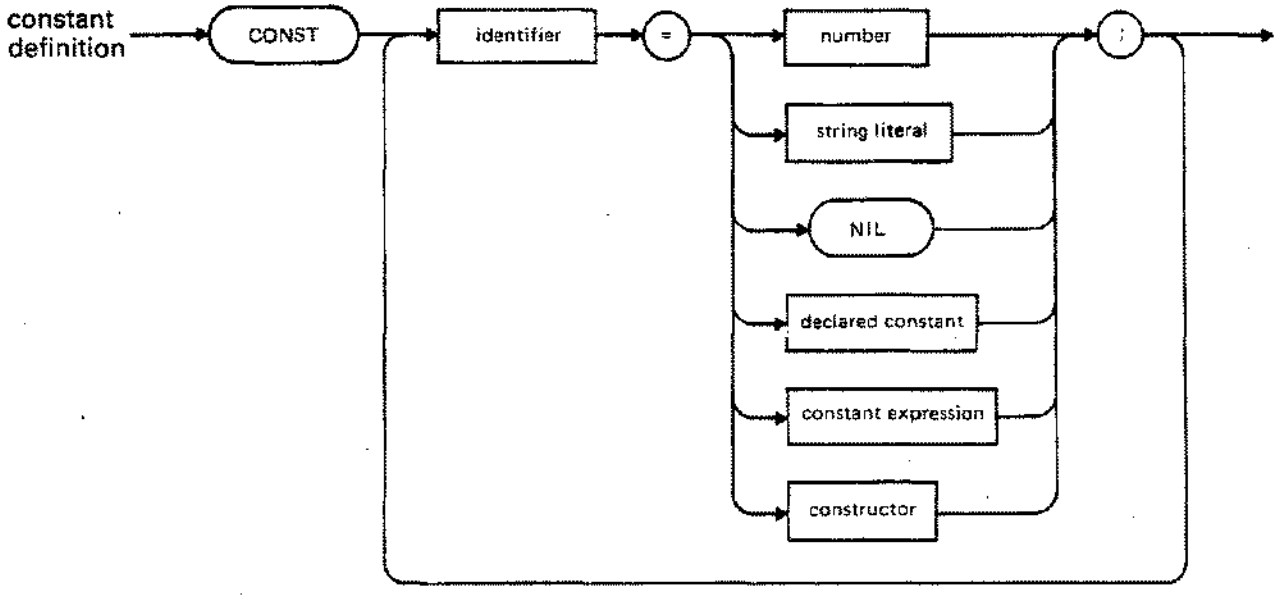
```
LABEL 9, 19, 40;
```

CONSTANT DEFINITION

A constant definition establishes an identifier as a synonym for a constant value. The programmer may then use the identifier in place of the value.

The reserved word CONST precedes one or more constant definitions. A constant definition consists of an identifier, the equals sign (=), and a constant value.

Syntax



Section 5 explains the form of numbers and string literals. The reserved word NIL is a pointer value. Declared constants include the standard constants *maxint* and *minint* as well as the standard enumerated constants *true* and *false*.

Constant expressions are a restricted class of Pascal/3000 expressions. They must return an ordinal value which is computable at compile time. Consequently, operands in constant expressions must be integers or ordinal declared constants. Operators must be +, -, *, DIV, or MOD. All other operators are excluded. Furthermore, only calls to the standard functions *ord, chr, pred, succ, abs, hex, octal,* and *binary* are legal.

CONSTANT DEFINITION

One exception to the restrictions on constant expressions is permitted: the programmer may change the sign of a real or longreal declared constant using the negative real unary operator (-). The positive operator (+) is legal but has no effect.

A constructor specifies values for a previously declared array, *string*, record, or set type. Subsequent pages describe constructors and the structured declared constants they define.

Constant definitions must follow label declarations and precede function or procedure declarations. The programmer can repeat and intermix CONST sections with TYPE and VAR sections.

Example

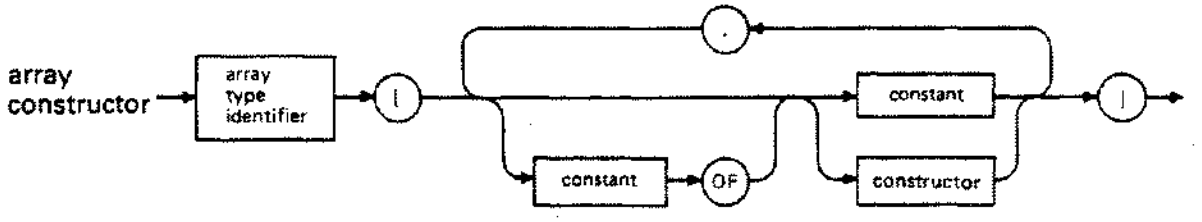
```
CONST
  fingers   = 10;              {Unsigned integer.          }

  pi        = 3.1415;          {Unsigned real.             }

  message   = 'Use a fork!';   {String literal.            }

  nothing   = NIL;

  delicious = true;            {Standard constant.         }

  neg_pi    = -pi;             {Real unary operator.       }

  hands     = fingers DIV 5;   {Constant expression.       }

  numforks  = pred(hands);     {Constant expression with   }
                               {call to standard function. }
```

ARRAY CONSTANT
(Array Constructor)

An array constant is a declared constant defined with an array constructor which specifies values for the components of an array type.

An array constructor consists of a previously defined array type identifier and a list of values in square brackets. Each component of the array type must receive a value which is assignment compatible with the component type.

Syntax



Within the square brackets, the reserved word OF indicates that a value occurs repeatedly. For example, 3 OF 5 assigns the integer value 5 to three successive array components. The symbols (. and .) may replace the left and right square brackets, respectively. An array constant may not contain files.

Array constructors are only legal in a CONST section of a declaration part. They cannot appear in other sections or in executable statements.

The programmer may use an array constant to initialize a variable in the executable part of a block. The programmer may also access individual components of an array constant in the body of a block, but not in the definition of other constants (see Selectors in Section 4).

ARRAY CONSTANT

Examples

```
TYPE
  boolean_table = ARRAY [1..5] OF boolean;
  table         = ARRAY [1..100] OF integer;
  row           = ARRAY [1..5] OF integer;
  matrix        = ARRAY [1..5] OF row;
  color         = (red, yellow, blue);
  color_string  = PACKED ARRAY [1..6] OF char;
  color_array   = ARRAY [color] OF color_string;

CONST
  true_values   = boolean_table [5 OF true];
  init_values1  = table [100 OF 0];
  init_values2  = table [60 OF 0, 40 OF 1];
  identity      = matrix [row [1, 0, 0, 0, 0],
                          row [0, 1, 0, 0, 0],
                          row [0, 0, 1, 0, 0],
                          row [0, 0, 0, 1, 0],
                          row [0, 0, 0, 0, 1]];
  colors        = color_array [color_string ['RED', 3 OF ' '],
                               color_string ['YELLOW'],
                               color_string ['BLUE', 2 OF ' ']];
```

In the last example, the type of the array component is *char*, yet both string literals and characters appear in the constructor. This is one case where a value (string literal) is not assignment compatible with the component type *(char)*. Alternatively, the programmer could write

```
  colors        = color_array['RED','YELLOW','BLUE'];
```

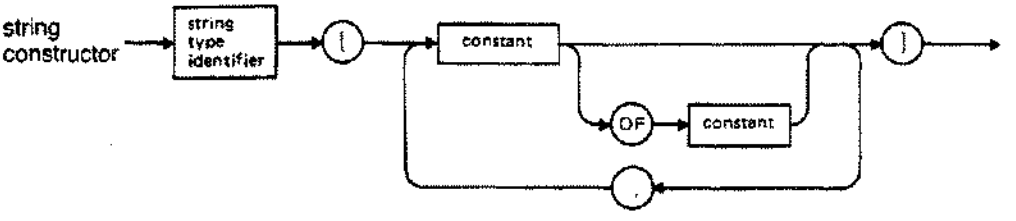
for the last constant definition.

STRING CONSTANT
(String Constructor)

A string constant is a declared constant defined with a string constructor which specifies values for a *string* type.

A string constructor consists of a previously defined string type identifier and a list of values in square brackets.

Syntax



Within the square brackets, the reserved word OF indicates that a value occurs repeatedly. For example 3 OF 'a' assigns the character 'a' to three successive string components. The symbols (. and .) may replace the left and right brackets, respectively. String literals of more than one character may appear as values.

The length of the string constant may not exceed the maximum length of the *string* type used in its definition.

String constructors are only legal in a CONST section of a declaration part. They cannot appear in other sections or in executable statements.

The programmer may use a string constant to initialize a variable in the statement part of a block. The programmer may also access individual components of a string constant in the body of the block, but not in the definition of other declared constants (see Selectors in Section 4).

STRING CONSTANTS

Examples

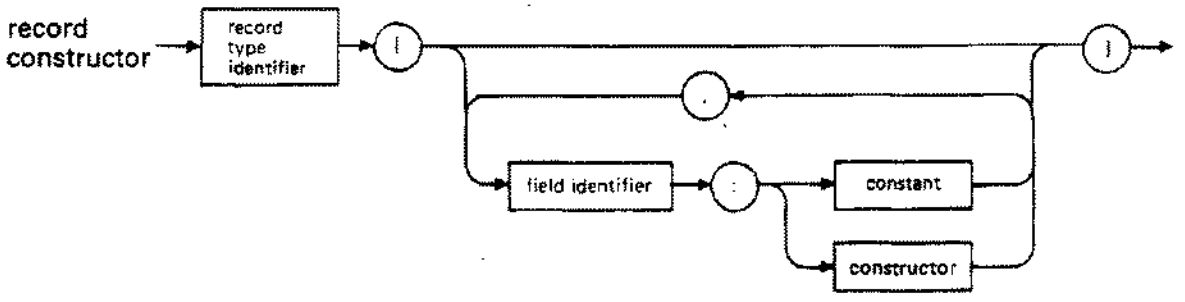
```
TYPE
  s = string[80];

CONST
  blank = ' ';
  greeting = s['Hello!'];
  farewell = s['G',2 OF 'o','d','bye'];
  blank_string = s[10 OF blank];
```

RECORD CONSTANT
(Record Constructor)

A record constant is a declared constant defined with a record constructor which specifies values for the fields of a record type.

A record constructor consists of a previously declared record type identifier and a list in square brackets of fields and values. All fields of the record type must appear, but not necessarily in the order of their declaration. Values in the constructor must be assignment compatible with the fields.

Syntax



For records with variants, the contructor must specify the tag field before any variant fields. Then only the variant fields associated with the value of the tag may appear. For free union variant records, i.e. tagless variants, the initial variant field selects the variant.

The values may be constant values or constructors. To use a constructor as a value, the programmer must define the field in the record type with a type identifier. A record constant may not contain a file.

A record constructor is only legal in the CONST section of a declaration part. It cannot appear in other sections or in an executable statement.

The programmer may use a record constant to initialize a variable in the body of a block. The programmer can also select individual fields of a record constant in the body of a block, but not when defining other constants.

RECORD CONSTANT

Examples

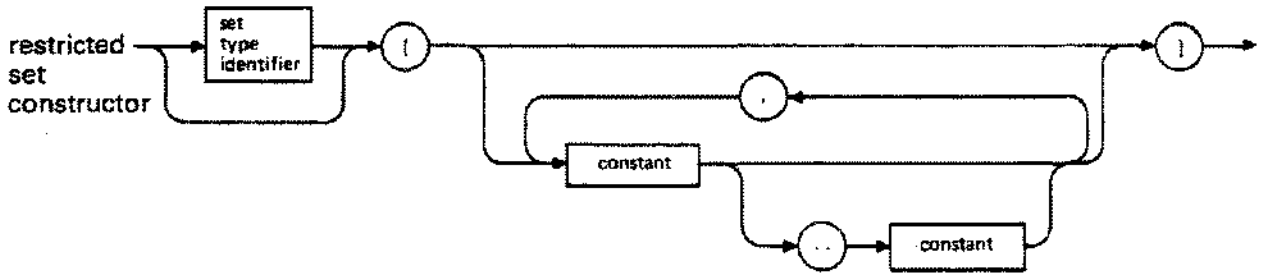
```
TYPE
  securtype = (light, medium, heavy);
  counter   = RECORD
                pages: integer;
                lines: integer;
                characters: integer;
              END;
  report    = RECORD
                revision: char;
                price:    real;
                info:     counter;
                CASE securtag: securtype OF
                  light:   ();
                  medium:  (mcode: integer);
                  heavy:   (hcode:    integer;
                            password: string[10]);
              END;

CONST
  no_count   = counter [pages: 0, characters: 0, lines: 0];
  big_report = report [revision: 'B',
                       price:    19.00,
                       info:     counter [pages:      19,
                                          lines:      25,
                                          characters: 900],
                       securtag: heavy,
                       hcode:    999,
                       password: 'unity'];
```

SET CONSTANT
(Restricted Set Constructor)

A set constant is a declared constant defined with a restricted set constructor which specifies set values.

A restricted set constructor consists of an optional previously declared set type identifier and a list of constant values in square brackets. Subranges may appear in this list.

Syntax



A value must be an ordinal constant value or an ordinal subrange. A constant expression is legal as a value. The symbols (. and .) may replace the left and right square brackets, respectively.

Restricted set constructors may appear in a CONST section of a declaration part or in executable statements. Unrestricted set constructors permit variables to appear as values within the brackets (see Section 4).

The programmer can use a set constant to initialize a set variable in the body of a block.

Examples

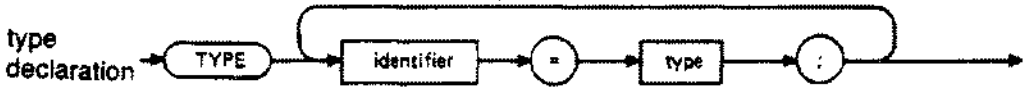
```
TYPE
  digits  = SET OF 0..9;
  charset = SET OF char;
CONST
  all_digits = digits [0..9];                (Subrange.)
  odd_digits = digits [1, 1+2, 5, 7, 9];
  letters    = charset ['a'..'z', 'A'..'Z'];
  no_chars   = charset [];
  no_iden    = [2, 4, 6, 8]          {No set identifier.}
```

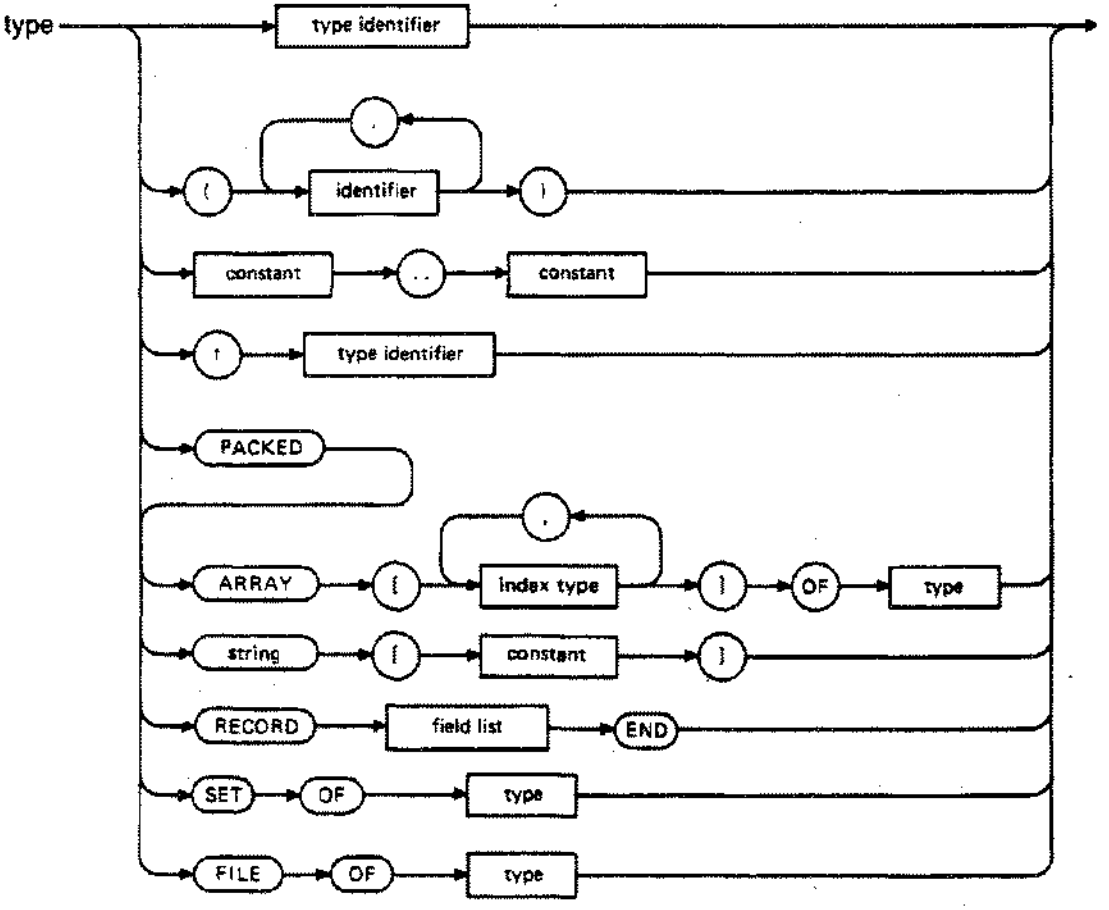
TYPE DEFINITIONS
(Data Types)

A type definition establishes an identifier as a synonym for a data type. The identifier may then appear in subsequent type or constant definitions, or in variable declarations.

The reserved word TYPE precedes one or more type definitions. A type definition consists of an identifier, the equals sign (=), and a data type.

Syntax





A data type determines a set of attributes which include:

- the set of permissible values
- the set of permissible operations
- the amount of storage required

Subsequent pages explain the permissible values and operations for the various data types. Section 9 discusses storage.

The three most general catagories of data type are simple, structured, and pointer.

Simple data types are the types ordinal, *real*, or *longreal*. Ordinal types include the standard types *integer*, *char*, and *boolean*, as well as enumerated and subrange types defined by the programmer.

Structured data types are the types array, record, set, or file. The standard type *string* is also a structured data type. The standard type *text* is a variant of the file type.

Pointer data types define pointer variables which point to dynamically allocated variables on the heap.

Figure 2-1 shows the relation of these various categories.

TYPE DEFINITIONS

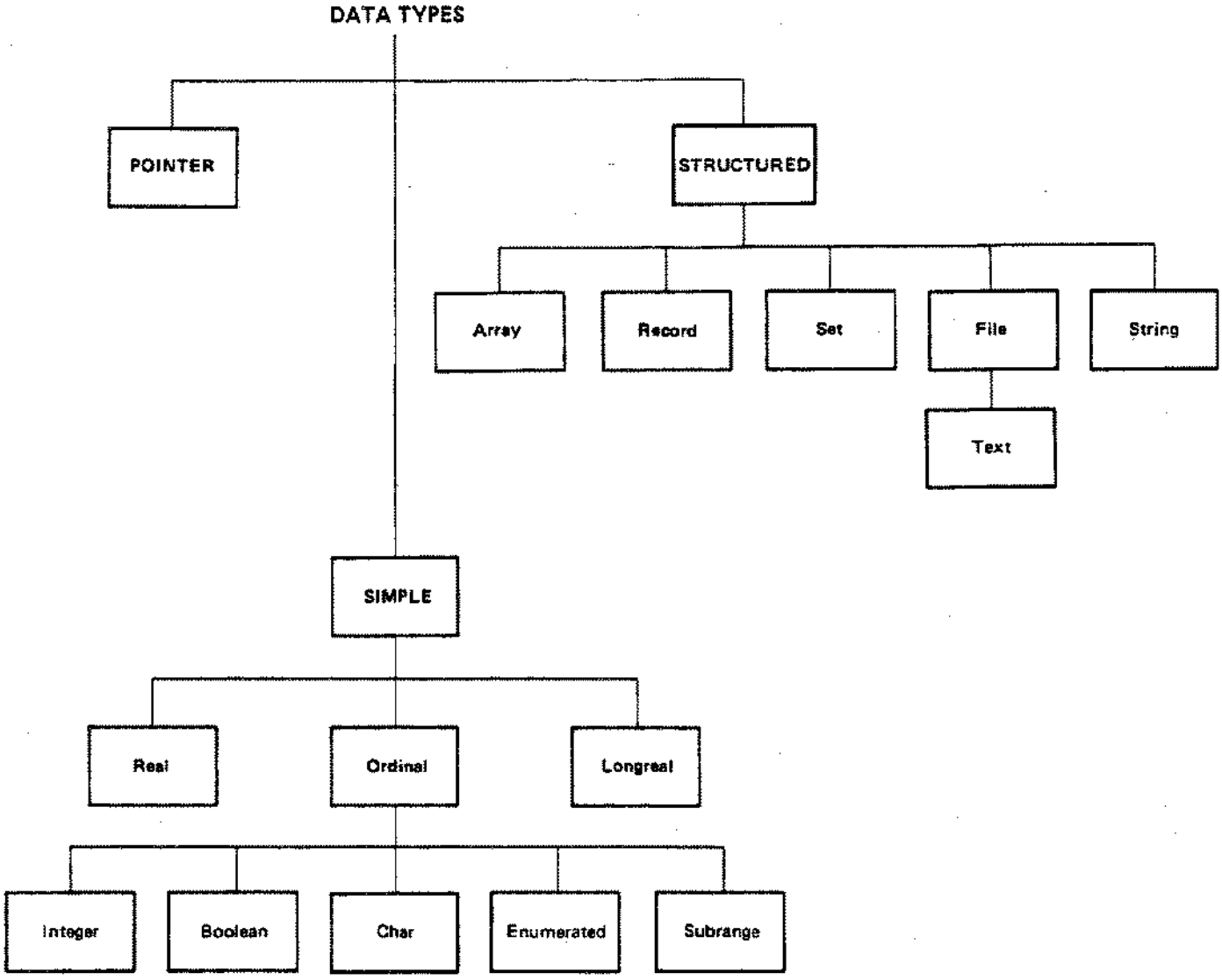


Fig. 2-1. PASCAL/3000 DATA TYPES

Pascal/3000 predefines the type *boolean* as:

```
TYPE boolean = (false, true);
```

The identifiers *false* and *true* are standard identifiers, where *true* > *false*.

Boolean is a standard simple ordinal type.

Permissible Operators

assignment	- :=
boolean	- AND, OR, NOT
relational	- <, <=, =, <>, >=, >, IN

Standard Functions

boolean argument	- *ord, pred, succ*
boolean return	- *eof, eoln, odd*

Standard Procedure

boolean parameter	- *assert*

Example

```
VAR
  loves_me: boolean;
```

CHAR TYPE

The 8-bit ASCII character set comprises the type *char,* which is a simple ordinal standard type.

A pair of single quote marks encloses a char literal (see Section 5).

Permissible Operators

assignment	- :=
relational	- <, <=, =, <>, >=, >, IN

Standard Functions

char argument	- *ord*
char return	- *chr, pred, succ*

Example

```
VAR
  do_you:  char;
```

INTEGER TYPE

The type *integer* is a subrange whose lower bound is the standard constant *minint* and whose upper bound is the standard constant *maxinit*. Pascal/3000 defines *minint*, *maxint*, and *integer* like this:

```
CONST
  minint = -2147483648;
  maxint = 2147483647;

TYPE
  integer = minint..maxint;
```

The value of the standard constant *minint* may not appear as an integer literal, although it may be input from a file.

Integer is a standard simple ordinal type.

Section 5 describes the form of an integer literal.

Permissible Operators

assignment	- :=
relational	- <, <=, =, <>, >, >=, IN
arithmetic	- +, -, *, /, DIV, MOD

Standard Functions

integer argument	- *abs, arctan, chr, cos, exp, ln, odd, ord, pred, sin, sqr, sqrt, succ*
integer return	- *abs, binary, ccode, fnum, hex, linepos, maxpos, octal, ord, position, pred, round, sizeof, strlen, strmax, strpos, sqr, trunc*

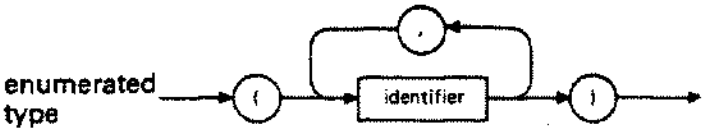
Example

```
VAR
  wholenum:  integer;
```

ENUMERATED TYPE

An enumerated type is an ordered list of identifiers in parentheses. The sequence in which the identifiers appear determines the ordering. The *ord* function returns 0 for the first identifier; 1 for the second identifier; 2 for the third identifier; and so on (see Section 7).

Syntax



There is no arbitrary limit on the number of identifiers that may appear in an enumerated type.

Enumerated types are simple ordinal types defined by the programmer.

Permissible Operators

assignment - :=

relational - <, <=, =, <>, >=, >, IN

Standard Functions

enumerated argument - *ord, pred, succ*
enumerated return - *pred, succ*

Example

```
TYPE
  workdays = (monday, tuesday, wednesday, thursday, friday);
  weekend  = (saturday, sunday);
```

SUBRANGE TYPE

A subrange type is a sequential subset of an ordinal host type. A subrange type consists of a lower bound, the special symbol .., and an upper bound. The upper and lower bounds must be constant values of the same ordinal type and the lower bound cannot be greater than the upper bound.

Syntax

A constant expression may appear as an upper or lower bound.

A subrange type is a simple ordinal type.

Permissible Operations and Standard Functions

A variable of a subrange type possesses all the attributes of the host type of the subrange, but its values are restricted to the specified closed range.

Example

```
TYPE
  day_of_year = 1..366;

  lowercase   = 'a'..'z';           {Host type is char.    }

  earlyweek   = Monday..Wednesday   {Identifiers from      }
                                    {enumerated host type.}
                                    {Monday < Wednesday.   }

  e_type      = 1..maxsize - 1      {Upper bound is con-   }
                                    {stant expression.     }
                                    {Maxsize is declared   }
                                    {constant.             }
```

REAL TYPE

The type *real* represents a subset of the real numbers. For Pascal/3000, this subset covers the ranges:

-1.15792E+77 to -8.63617E-78
0.0
8.63617E-78 to 1.15792E+77

The type *real* is a standard simple type.

Section 5 describes the form of a real literal.

Permissible Operators

assignment	- :=
relational	- <, <=, =, <>, >=, >
arithmetic	- +, -, *, /

Standard Functions

real argument	- *abs, arctan, cos, exp, ln, round, sin, sqr, sqrt, trunc*
real return	- *abs, arctan, cos, exp, ln, sin, sqr, sqrt*

Example

```
VAR
  realnum: real;
```

The type *longreal* represents a subset of the real numbers. In Pascal/3000, this subset covers the ranges:

-1.157920892373162L+77 to -8.636168555094445L-78
0.0
8.636168555094445L-78 to 1.157920892373162L+77

The type *longreal* is a standard simple type.

Section 5 describes the form of a longreal literal.

Permissible Operators

assignment	- :=
relational	- <, <=, =, <>, >=, >
arithmetic	- +, -, *, /

Standard Functions

longreal argument - *abs, arctan, cos, exp, ln, round, sin, sqr, sqrt, trunc*

longreal return - *abs, arctan, cos, exp, ln, sin, sqr sqrt*

Example

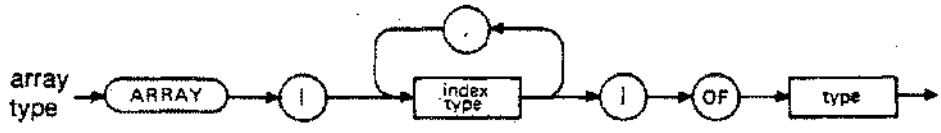
```
VAR
  precisenum: longreal;
```

ARRAY TYPE

An array is a fixed number of components which are all the same type. A computable index designates each component of an array.

An array type definition consists of the reserved word ARRAY, an index type in square brackets, the reserved word OF, and the component type. The reserved word PACKED may precede ARRAY. It instructs the compiler to optimize storage space for the array components (see Section 9).

Syntax



The index type must be an ordinal type. The component type may be any simple, structured, or pointer type, including a file type. The symbols (. and .) may replace the left and right square brackets, respectively.

An array type is a structured type defined by the programmer.

The programmer may access a component of an array using the index of the component in a selector (see Section 4).

In ANSI Standard Pascal, the term 'string' designates a packed array of *char* with a starting index of 1. HP Standard Pascal and Pascal/3000, however, define a standard type *string* which is identical with a packed array of *char* except that its actual length may vary at run time. To distinguish these two data types, the acronym PAC will denote

PACKED ARRAY [1..n] OF *char*;

throughout this manual.

Permissible Operators

assignment - :=

relational (PAC only) - <, <=, =, <>, >=, >

Standard Procedures

array parameters - *pack, unpack*

Examples

```
TYPE
  name    = PACKED ARRAY [1..30] OF char;   {PAC type}
  list    = ARRAY [1..100] OF integer;
  strange = ARRAY [boolean] OF char;
  flag    = ARRAY [(red, white, blue)] OF 1..50;
  files   = ARRAY [1..10] OF text;
```

Multiply-dimensioned Arrays

If an array definition specifies more than one index type or if the components of an array are themselves arrays, then the array is said to be multiply-dimensioned. There is no arbitrary limit on the number of array dimensions.

Examples

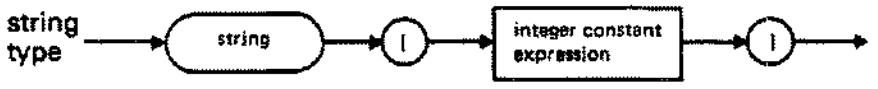
```
TYPE
  { equivalent definitions of truth }
  truth  = ARRAY [1..20] OF
             ARRAY [1..5] OF
               ARRAY [1..10] OF boolean;
  truth  = ARRAY [1..20] OF
             ARRAY [1..5, 1..10] OF boolean;
  truth  = ARRAY [1..20, 1..5] OF
             ARRAY [1..10] OF boolean;
  truth  = ARRAY [1..20, 1..5, 1..10] OF boolean;
```

STRING TYPE

In Pascal/3000, a string is a packed array of *char* whose maximum length is set at compile time and whose actual length may vary dynamically at run time.

A *string* type consists of the standard identifier *string* and an integer constant expression in square brackets which specifies the maximum length.

Syntax



The maximum length must be in the range 1..32767. The symbols (. and .) may replace the left and right square brackets, respectively.

A *string* type is a standard structured type.

Characters enclosed in single quotes are string literals. The compiler interprets a string literal as type PAC, *string*, or *char*, depending on context.

Integer constant expressions are constant expressions which return an integer value, an unsigned integer being the simple case (see Constant Definition above).

When a formal reference parameter is type *"string"*, the programmer may not specify the maximum length (see example below). This allows actual string parameters to have various maximum lengths.

The programmer may access a single component of a string using an integer expression in square brackets as a selector (see Section 4). The standard function *str* selects a substring of a string (see Section 7).

NOTE: Variables of *string* type, as other Pascal variables, are NOT initialized. The current string length contains meaningless information until the user initializes the string.

Permissible Operators

assignment - :=

concatenation - +

relational - =, <>, <=, >=, >,

Standard Functions

string argument - *str, strlen, strltrim, strmax, strpos, strrpt, strrtrim*
string return - *str, strltrim, strrpt, strrtrim*

Standard Procedures

string parameter - *setstrlen, strappend, strdelete, strinsert, strmove, strread, strwrite*

Examples

```
CONST
  maxlength = 100;

TYPE
  name   = string[30];
  remark = string[maxlength * 2];

PROCEDURE proc1 (VAR s: string); EXTERNAL; {Maximum length }
                                           {not required.  }
```

RECORD TYPE

A record is a collection of components which are not necessarily the same type. Each component is termed a field of the record and has its own identifier.

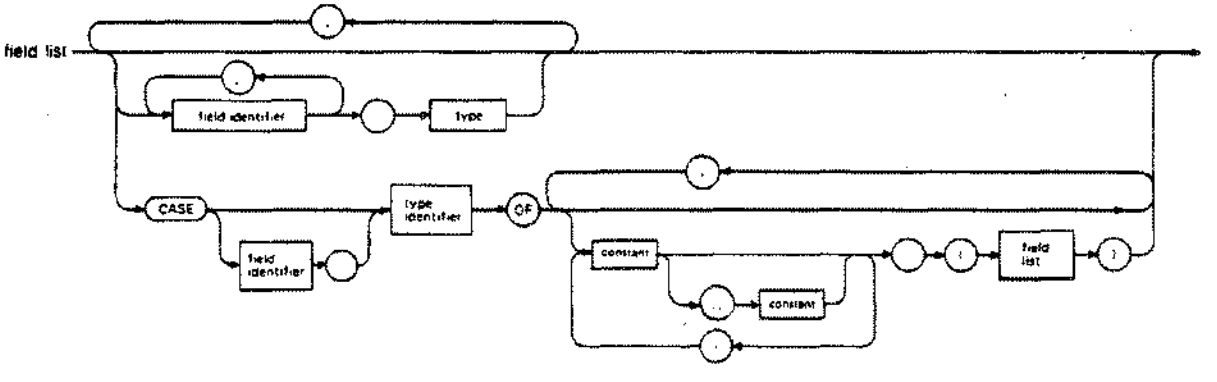
A record type consists of the reserved word RECORD, a field list, and the reserved word END.

The reserved word PACKED may precede the reserved word RECORD. It instructs the compiler to optimize storage of the record fields (see Section 9).

Syntax

The field list has a fixed part and an optional variant part.

Syntax



In the fixed part of the field list, a field definition consists of an identifier, a colon (:), and a type. Any simple, structured, or pointer type is legal. The programmer may define several fields of the same type by listing identifiers separated by commas.

In the variant part, the reserved word CASE introduces an optional tag field identifier and a required ordinal type identifier. Then the reserved word OF precedes a list of case constants and alternative field lists. Fields of type file or of a type which contains files are not legal in the variant part of a record.

Case constants must be type compatible with the tag. The programmer may associate several case constants with a single field list. The various constants appear separated by commas. Subranges are also legal case constants. HP Pascal does NOT require that you specify all possible tag values. This is an extension to the ANSI Standard Pascal. The programmer may use the empty field list to indicate that a variant doesn't exist (see example below).

The programmer may not use the OTHERWISE construction in the variant part of the field list. OTHERWISE is only legal in CASE statements (see Section 3).

Variant parts allow variables of the same record type to exhibit structures that differ in the number and type of their component parts. The value of the tag field, if any, indicates which variant is currently valid. When the tag is assigned another value, previous variants cease to exist.

The programmer may access a field of a record using the appropriate field selector (see Section 4).

A record is a structured type defined by the programmer.

Permissible Operator

assignment (entire record) - :=

RECORD TYPE

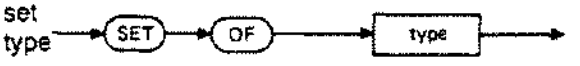
Examples

```
TYPE
  word_type = (int, ch);
  word      = RECORD                (variant part only with tag)
                CASE word_tag: word_type OF
                  int: (number: integer);
                  ch : (chars : PACKED ARRAY [1..2] OF char);
              END;

  polys   = (circle, square, rectangle, triangle);
  polygon = RECORD        (fixed part and tagless variant part)
              poly_color: (red, yellow, blue);
              CASE polys OF
                circle:    (radius: integer);
                square:    (side: integer);
                rectangle: (length, width: integer);
                triangle:  (base, height: integer);
            END;

  name_string  = PACKED ARRAY [1..30] OF char;
  date_info    = PACKED RECORD              (fixed part only)
                   mo: (jan, feb, mar, apr, may, jun,
                        jul, aug, sep, oct, nov, dec);
                   da: 1..31;
                   yr: 1900..2001;

                 END;
  marital_status = (married, separated, divorced, single);
  person_info    = RECORD                 (nested variant parts)
                     name: name_string;
                     born: date_info;
                     CASE status: marital_status OF
                       married..divorced:
                                 (when:   date_info;
                                  CASE has_kids: boolean OF
                                    true: (how_many: 1..50);
                                    false: (); (Empty variant)
                                 )
                       single: ();
                   END;
```

SET TYPE

A set is the powerset, i.e. the set of all subsets, of a base type. A set type consists of the reserved words SET OF and an ordinal base type.

Syntax



The base type may be any ordinal type and may contain up to 32767 elements.

If the standard type *integer* appears as the base type, the compiler uses the integer subrange 0..255 as the actual base type. Thus, the programmer cannot associate a value outside this range with such a set.

It is legal to declared a packed set, but this does not affect storage.

A set type is a structured type defined by the programmer.

Permissible Operators

assignment	- :=
union	- +
intersection	- *
difference	- -
subset	- <=
superset	- >=
equality	- =, <>
inclusion	- IN

Examples

```
TYPE
  charset   = SET OF char;
  fruit     = (apple, banana, cherry, peach, pear, pineapple);
  somefruit = SET OF apple..cherry;
  poets     = SET OF (Blake, Frost, Brecht);
  big_set   = SET OF 1..10000;
```

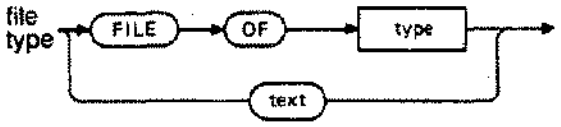
FILE TYPE

A logical file is a declared data structure in a Pascal/3000 program. A physical file is an independent entity controlled by the MPE Operating System. At run time, logical files are associated with physical files, allowing a program to manipulate data in the external environment (see Section 6).

A logical file is a sequence of components of the same type, which may be any type except a file type or a structured type with a file type component.

A file type consists of the reserved words FILE OF and a component type.

Syntax



The programmer may access file components sequentially or directly using a variety of Pascal/3000 standard procedures and functions fully described in Section 6.

It is legal to declare a packed file, but this has no effect on storage.

The standard file type *text* is described on the next page.

Examples

```
TYPE
  person      = RECORD
                  name: PACKED ARRAY [1..30] OF char;
                  age:  1..100;
                END;
  bit_vector  = PACKED ARRAY [1..100] OF boolean;
  person_file = FILE OF person;
  data_file   = FILE OF integer;
  vector_file = FILE OF bit_vector;
```

TEXT FILE TYPE

The standard file type *text* permits ordinary input and output oriented to characters and lines. *Text* type files have two important features: (1) the components are type *char*; (2) the file is subdivided into lines by special end-of-line markers.

Text type variables are termed 'textfiles'.

The programmer cannot open textfiles for direct access, i.e. with the procedure *open*. Textfiles may be sequentially accessed, however, with the procedures *reset, rewrite*, or *append*. All standard procedures that are legal for sequentially accessed files are also legal for textfiles (see Section 6).

Certain standard procedures and functions, on the other hand, are legal only for textfiles: *readln, writeln, page, prompt, overprint, eoln, and linepos.*

Textfiles permit conversion from the internal form of certain types to an ASCII character representation and vice versa.

Subsequent pages in this chapter and in Section 6 describe two standard textfiles, *input* and *output*.

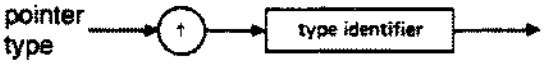
Example

```
VAR
  myfile: text;
```

POINTER TYPE

A pointer references a dynamically allocated variable on the heap. A pointer type consists of the caret (^) and a type identifier.

Syntax



The type may be any type, including file types. The @ symbol may replace the caret.

The programmer need not have previously defined the type appearing after the caret. This is an exception to the general rule that Pascal identifiers are first defined and then used. However, the programmer must define the identifier after the caret within the same declaration part, although not necessarily within the same TYPE section.

The pointer value NIL belongs to every pointer type; it points to no variable on the heap.

Permissible Operators

assignment - :=
equality - =, <>

Standard Procedures

pointer parameters - *new, dispose, mark, release*

Examples

```
TYPE
  ptr1   = ^rec1;
  ptr2   = ^rec2;
  rec1   = RECORD
             f1, f2: integer;
             link:   ptr2;
           END;
  rec2   = RECORD
             f1, f2: real;
             link:   ptr1;
           END;
```

TYPE COMPATIBILITY

Relative to each other, two Pascal/3000 types can be identical, type compatible, assignment compatible, or incompatible.

Identical Types

Two types are identical if either of the following is true:

(1) Their types have the same type identifier.

(2) If A and B are their two type identifiers, and they have been made equivalent by a definition of the form

```
TYPE A = B
```

Type Compatible Types

Two types T1 and T2 are type compatible if any of the following is true.

(1) T1 and T2 are identical types.

(2) T1 and T2 are subranges of the same host type, or T1 is a subrange of T2, or T2 is a subrange of T1.

(3) T1 and T2 are set types with compatible base types and both T1 and T2 or neither are packed.

(4) T1 and T2 are PAC types with the same number of components, or if T2 is a string literal no longer than T1.

(5) T1 and T2 are both *string* types.

(6) T1 and T2 are both real types, i.e. *real* or *longreal*.

Assignment Compatible Types

Section 3 describes assignment compatible types.

Incompatible Types

Two types are incompatible if they are not identical, type compatible, or assignment compatibile.

Examples

```
TYPE
   interval = 0..10;
   range = interval;

VAR
   v1 : 0..10;
   v2, v3: 0..10;
   v4 : interval;
   v5 : interval;
   v6 : range;
```

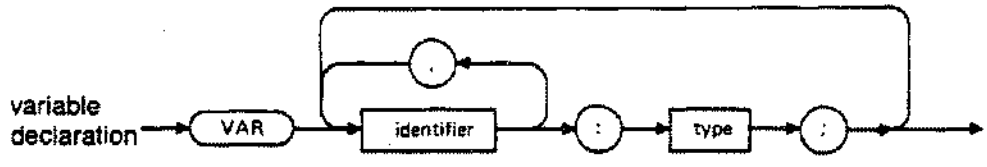
All of the variables are type compatible, but only v4, v5, and v6, as well as the pair v2 and v3, have identical types.

VARIABLE DECLARATION

A variable declaration associates an identifier with a type. The identifier may then appear as a variable in executable statements.

The reserved word VAR precedes one or more variable declarations. A variable declaration consists of an identifier, a colon (:), and a type. The programmer may list any number of identifiers separated by commas. These identifiers will then be variables of the same type.

Syntax



The type may be any simple, structured, or pointer type. The form of the type may be a standard identifier, a declared type identifier, or a data type (see example below).

Variable declarations must follow label declarations and precede function and procedure declarations. The programmer may repeat VAR sections and intermix them with CONST and TYPE sections.

The programmer may access components of a structured variable using an appropiate selector. Pointer variable dereferencing accesses dynamic variables on the heap. (see Section 4).

Pascal/3000 predefines two standard variables, *input* and *output*, which are textfiles. Formally,

```
VAR
  input, output: text;
```

Section 6 discusses these standard variables in detail. They commonly appear as program parameters and serve as default files for various file operations.

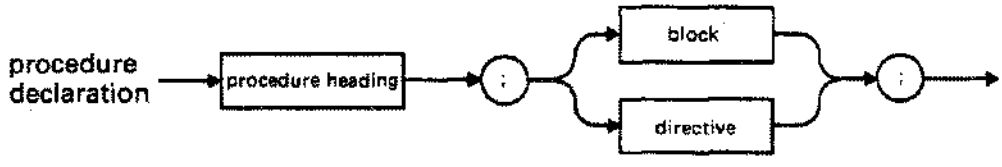
Examples

```
TYPE
  answer = (yes, no, maybe);
VAR
  pagecount,
  linecount,
  charcount: integer;              {Standard identifier.      }

  whats_the: answer;               {User-declared identifier.}

  album    : RECORD                {Data type.                }
               speed: (lp, for5, sev8);
               price: real;
               name : string[20];
             END;
```

PROCEDURE DECLARATION

A procedure is a block which the programmer may activate with a procedure statement. A procedure declaration consists of a procedure heading, a semi-colon (;), and a block or a directive followed by a semi-colon.

Syntax



The procedure heading consists of the reserved word PROCEDURE, an identifier (the procedure name), and, optionally, a formal parameter list. For level-1 procedures, the procedure name must be unique within fifteen characters (see below).

Syntax

A directive can replace the procedure block. The directives are FORWARD, EXTERNAL, and INTRINSIC (see below).

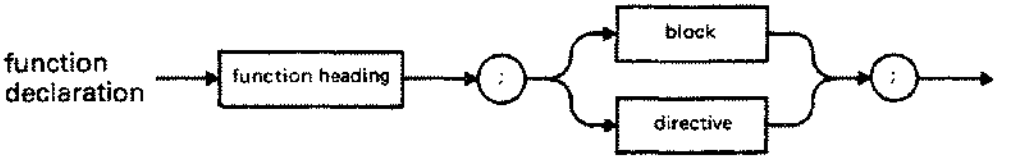
A procedure block is syntactically identical with the block described in Sections 2 and 3 of this manual. It consists of an optional declaration part and a statement part.

Procedure declarations must occur at the end of a declaration part after label, constant, type, and variable declarations. The programmer may repeat procedure declarations and intermix them with function declarations.

FUNCTION DECLARATION

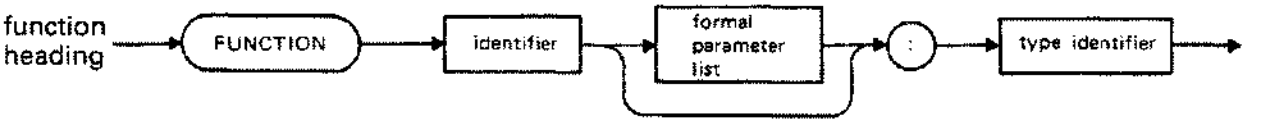
A function is a block which the programmer may activate with a function call and which returns a value. A function declaration consists of a function heading and a block or a directive.

Syntax



A function heading consists of the reserved word FUNCTION, an identifier (function name), an optional formal parameter list, and a result type. For level 1 functions, the function name must be unique within fifteen characters (see below). The result type may be any type, except a file type or a structured type containing a file.

Syntax



A directive can replace the function block. The directives are FORWARD, EXTERNAL, and INTRINSIC (see below).

A function block is syntactically identical with the block described in Sections 2 and 3 of this manual. However, in the body of a function block there must be at least one statement assigning a value to the function identifier. This assignment statement determines the function result. If the function result is a structured type, the programmer must assign a value to each of its components using an appropriate selector (see Section 4).

FUNCTION DECLARATION

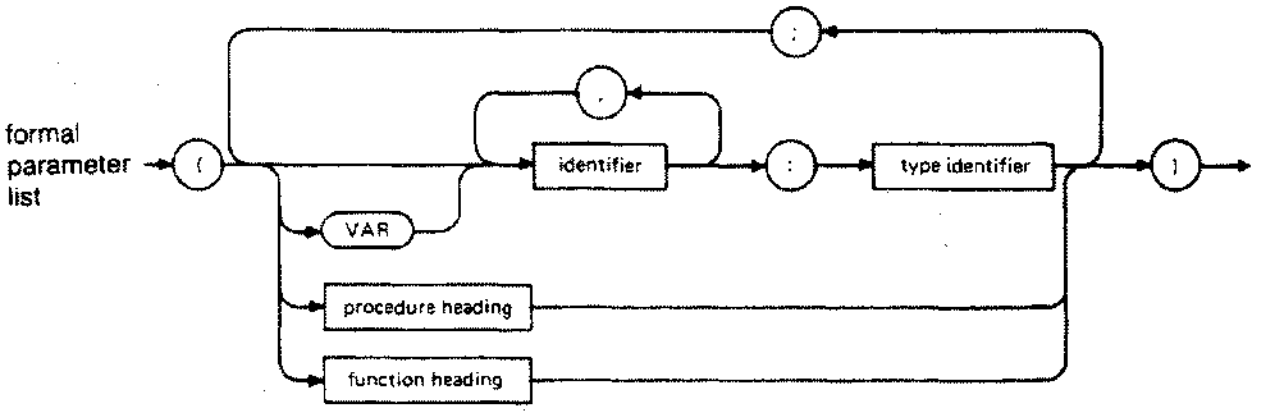
Function declarations may occur at the end of a declaration section after label, constant, type, and variable declarations. The programmer may repeat function declarations and intermix them with procedure declarations.

FORMAL PARAMETER LIST

A formal parameter list appears optionally in a procedure or function heading and specifies the formal parameters for a procedure or function. A procedure statement or function call in the body of a block provides the matching actual parameters.

The four sorts of formal parameters are value, variable, functional, and procedural parameters. Value parameters are identifiers followed by a colon (:) and a type identifier. Variable parameters are identical with value parameters except they are preceded by the reserved word VAR. Functional or procedural parameters are function or procedure headings.

Syntax



The programmer may repeat and intermix the four types of formal parameters. Several identifiers may appear separated by commas. These identifiers will then represent formal variable or value parameters of the same type.

A formal value parameter functions as a local variable during execution of the procedure or function. It receives its initial value from the matching actual parameter. Execution of the procedure or function doesn't affect the actual parameter, which, therefore, may be an expression.

A formal variable parameter represents the actual parameter during execution of the procedure. Any changes in the value of the formal variable parameter will alter the value of the actual parameter, which, therefore, must be a variable. A *string* type formal variable parameter need not specify a maximum length.

FORMAL PARAMETER LIST

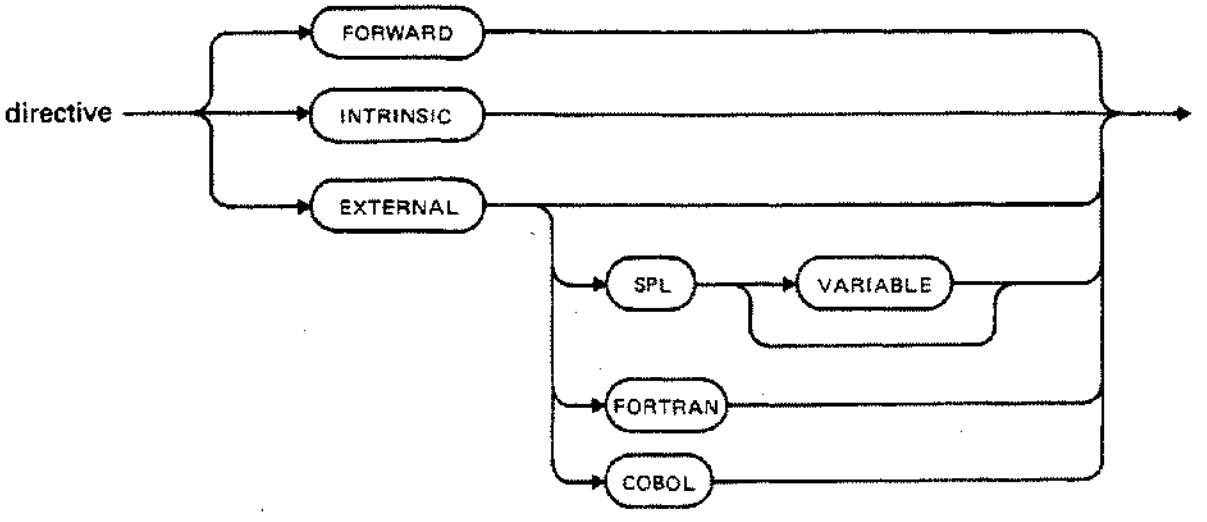
A formal procedural or functional parameter is a synonym for the actual procedural or functional parameter. The parameter lists, if any, of the actual and formal procedural or functional parameters must be congruent (see Section 3).

Examples

```
PROGRAM show_formparm;
VAR
  test: boolean;

FUNCTION chek1 (x, y, z: real): boolean;
  BEGIN
    {Perform some type of validity check on x, y, z }
    {and return appropriate value.                   }
  END;

FUNCTION chek2 (x, y, z: real): boolean;
  BEGIN
    {Perform an alternate validity check on x, y, z }
    {and return appropriate value.                   }
  END;

PROCEDURE read_data (FUNCTION check (a, b, c: real): boolean);
  VAR p, q, r: real;
  BEGIN
    {read and validate data}
    readln (p, q, r);
    IF check (p, q, r) THEN ...
  END;

BEGIN {show_formparm}
  .
  IF test THEN read_data (chek1)
    ELSE read_data (chek2);
  .
END.
```

A directive may replace a block in a procedure or function declaration. In Pascal/3000, the three directives are FORWARD, EXTERNAL and INTRINSIC. The programmer may qualify the EXTERNAL directive with the terms SPL, FORTRAN, or COBOL. Furthermore, the term VARIABLE may appear after SPL.

Syntax



The FORWARD directive makes it possible to postpone full declaration of a procedure or function; the EXTERNAL directive to declare Pascal or non-Pascal procedures or functions in other compilation units; the INTRINSIC directive to declare MPE or programmer-created intrinsics.

The terms FORWARD, EXTERNAL, SPL, VARIABLE, FORTRAN, COBOL, and INTRINSIC may appear as programmer-defined identifiers in source code and, at the same time, as directives.

Subsequent pages describe each directive in detail.

FORWARD DIRECTIVE

The FORWARD directive permits the full declaration of a procedure or function to follow the first call of the procedure or function. For example, suppose a programmer declares procedures A and B on the same level. Both A and B cannot call each other without using the FORWARD directive:

```
PROCEDURE A; FORWARD;
PROCEDURE B;
  BEGIN
    .
    A;    {calls A}
    .
  END;
PROCEDURE A;  {full declaration of A}
  BEGIN
    .
    B;    {calls B}
    .
  END;
```

After using the FORWARD directive, the programmer must fully declare the function or procedure in the same declaration part of the block. Formal parameters, if any, and the function result type must appear with the FORWARD declaration. The programmer may omit these formal parameters or result type, however, when making the subsequent full declaration (see example below). If repeated, they must be identical with the original formal parameters or result type.

The FORWARD directive may appear with a procedure or function at any level.

Example

```
FUNCTION exclusive_or (x,y: boolean): boolean;
  FORWARD;
  .
  .
FUNCTION exclusive_or;          {Parameters not repeated.}
  BEGIN
    exclusive_or:= (x AND NOT y) OR (NOT x AND y);
  END;
```

EXTERNAL DIRECTIVE

The EXTERNAL directive permits the programmer to call Pascal or non-Pascal procedures or functions in other compilation units. These external procedures and functions may be part of a segmented library, a relocatable library, or a separately compiled subprogram; their source code may be Pascal/3000, SPL, FORTRAN, COBOL 68, or COBOL II.

The EXTERNAL directive may appear with a procedure or function declaration at any level. The actual external procedure or function referenced, however, must be a level 1 procedure or function.

In general, the programmer is responsible for matching the formal parameters or result type of a procedure or function declared EXTERNAL with the formal parameters or result type of the external procedure or function (see Appendix G). In contrast, the INTRINSIC directive requires little or no matching.

There are five possible forms of an EXTERNAL directive,

```
EXTERNAL
EXTERNAL SPL
EXTERNAL SPL VARIABLE
EXTERNAL FORTRAN
EXTERNAL COBOL
```

which we examine in turn.

EXTERNAL - The source code of the external procedure or function is Pascal/3000. The formal parameters of the declaration, if any, must match the formal parameters of the external procedure or function in number, order, and type, i.e. they must be type identical. They need not have the same name. Also, the result type of a function must be identical with the result type of the external function.

EXTERNAL SPL - The source code of the external procedure or function is SPL without option variable parameters. Formal parameters need not have the same name as the external formal parameters. They must, however, match the external formal parameters in number and order. Furthermore, the Pascal/3000 type of the formal parameters or the function result must satisfactorily conform to the SPL type of the external formal parameters or result type (see Appendix G and SPL Reference Manual).

EXTERNAL DIRECTIVE

EXTERNAL SPL VARIABLE - The source code of the external procedure or function is SPL with option variable parameters. The programmer must use this form of the EXTERNAL directive even if no parameters are omitted when calling the external SPL procedure or function. The formal parameters must match the formal parameters of the external SPL procedure in number, order and type, but not necessarily in name. The Pascal/3000 type of the formal parameters or result type must satisfactorily conform to the SPL type of the external formal parameters or function result (see Appendix G and the SPL Reference Manual).

EXTERNAL FORTRAN - The source code of the external procedure or function is FORTRAN. The formal parameters, if any, must match the external formal parameters in order and number, but not necessarily in name. The Pascal/3000 type of the formal parameters or function result must satisfactorily conform with the FORTRAN type of the external formal parameters or function result (see Appendix G and the FORTRAN/3000 Reference Manual).

EXTERNAL COBOL - The source code of the external procedure or function is COBOL 68 or COBOL II. The declared formal parameters must match the external formal parameters in order and number, but not necessarily in name. Again, the Pascal/3000 type of the declared formal parameters or function result must satisfactorily conform with the COBOL type of the external formal parameters or function result (see Appendix G and the COBOL or COBOL II Reference Manuals).

Examples

See Appendix G.

The INTRINSIC directive permits the programmer to call MPE or user-created intrinsics with great flexibility. For example, the programmer can declare an intrinsic procedure or function with a full or partial formal parameter list, or no formal parameter list at all. Also, the programmer may use the ALIAS option to declare an intrinsic in more than one way.

Formal Parameter List

In a procedure or function declared with the INTRINSIC directive, the formal parameter list is optional. A subsequent procedure statement or function call may pass actual parameters to the intrinsic even if no formal parameter list appeared. A formal parameter list for an intrinsic only provides strong type checking of actual parameters. When formal parameters appear, the actual parameters must match in the normal manner. When formal parameters are absent, the actual parameters may be of any type as long as reasonable conversion to the intrinsic parameter is possible (see Appendix F).

Furthermore, partial formal parameter lists are legal. The MPE intrinsic FOPEN, for example, is an option variable intrinsic with up to 13 parameters. The programmer could declare FOPEN with only 3 formal parameters, and these parameters would correspond to the first 3 parameters of FOPEN. Then the compiler will strongly type check the first 3 actual parameters against the specified formal parameters. The system will convert succeeding actual parameters to whatever FOPEN requires.

There is one restriction on the formal parameters in an INTRINSIC declaration: if a formal parameter appears for the nth intrinsic parameter, then formal parameters must also appear for the 1st to n-1st intrinsic parameters.

Specifying formal parameters does not affect the use of empty actual parameters in calls to option variable intrinsics. The programmer is still free to pass empty actual parameters to the option variable intrinsic (see Section 3).

Alternative Intrinsic Declarations

The programmer must declare an intrinsic with no functional return as a procedure. On the other hand, an intrinsic with a functional return may be declared as a procedure or as a function, depending on the way the programmer wishes to use it in the Pascal/3000 program. Furthermore, the ALIAS option makes it possible to declare the same intrinsic in both ways (see example below).

To use the intrinsic as a function, the programmer declares it as a function with a Pascal/3000 result type. The system cannot handle the intrinsic function return without having a Pascal/3000 type. Once declared as a function, the intrinsic cannot appear as a procedure in executable statements.

To use the intrinsic as a procedure, the programmer declares it as a procedure in the usual way. The system will discard the intrinsic function return. Once declared as a procedure, the intrinsic cannot appear as a function in the body of the program.

The programmer may also use the ALIAS option to declare an intrinsic which does not have a legal Pascal/3000 name, e.g. there are single quote marks in the name.

Examples

```
TYPE
  smallint = -32768..32767;

PROCEDURE pfileinfo; $ALIAS 'PRINT''FILE''INFO'$  {System name.}
  INTRINSIC;


PROCEDURE fopen_p(VAR form_desg: barr;
                      foptions: smallint;
                      aoptions: smallint
                 );
                  $ALIAS 'FOPEN'$    {FOPEN used as procedure. }
                  INTRINSIC;

FUNCTION fopen_f(VAR form_desg: name_rec
                ): smallint;
                  $ALIAS 'FOPEN'$     {FOPEN used as function. }
                  INTRINSIC;
```

LEVEL 1 PROCEDURES AND FUNCTIONS

Level 1 procedures and functions are procedures and functions which the programmer declares at the main program level. That is, other procedures or functions do not contain them. The Pascal/3000 compiler creates entry points for level 1 procedures and functions. This means they are accessible from outside the compilation block in which the programmer declares them. Since they appear as distinct entries in a USL directory, the MPE Segmenter requires that names of level 1 procedures and functions be unique within the first fifteen characters.

When the compiler option PRIVATE_PROC is OFF, the compiler makes the names of all procedures and functions from any level known to the Segmenter, i.e. the names appear in the USL directory. Thus, all procedure or function names must be unique within 15 characters. When PRIVATE_PROC is ON (the default setting), however, names of non-level 1 procedures or functions need not be unique. This conforms with the usual scope conventions for Pascal identifiers.

Example

```
PROGRAM show_level;

PROCEDURE proc1;              {Level 1 procedure.  }
  PROCEDURE subproc1;         {Level 2 procedure.  }
    BEGIN
    .
    END;
  BEGIN {proc1}
  .
  END;

BEGIN {show_level}
 .
END.
```

RECURSIVE PROCEDURES AND FUNCTIONS

A recursive procedure or function is a procedure or function that calls itself. It is also legal for procedure A to call procedure B which in turn calls procedure A. This is indirect recursion and is often an instance when the FORWARD directive is useful.

When a program uses extensive recursion, the stack space allocated by the system may not be sufficient. The programmer can overcome this problem using the STACK or MAXDATA parameters of the MPE :PREP or :RUN commands.

Example

```
FUNCTION factorial (n: integer): integer;
{Calculates factorial recursively}
  BEGIN
    IF n = 0 THEN
      factorial := 1
    ELSE
      factorial := n * factorial(n-1);
  END;
```

The scope of an identifier is its domain of accessibility, i.e. the region of a program in which the programmer may use it.

In general, a programmer-defined identifier may appear anywhere in a block after its definition. Furthermore, the identifier may appear in a block nested within the block in which it is defined.

If the programmer redefines an identifier in a nested block, however, this new definition takes precedence. The object defined at the outer level will no longer be accessible from the inner level (see example below).

Once defined at a particular level, an identifier may not be redefined at the same level (except for field names).

Labels are not identifiers and their scope is restricted. They cannot mark statements in blocks nested within the block where they are declared.

Identifiers defined at the main program level are 'global'. Identifiers defined in a function or procedure block are 'local' to the function or procedure.

Example

```
PROGRAM show_scope (output);
CONST
  asterisk = '*';
VAR
  x: char;
PROCEDURE writeit;
  CONST
    x = 'LOCAL AND GLOBAL IDENTIFIERS DON'T CONFLICT';
  BEGIN
    write (x)
  END;
BEGIN (show_scope)
  x:= asterisk;
  write (x);
  writeit;
  write (x)
END.  (show_scope)
```

STATEMENTS

SECTION III

INTRODUCTION

A statement is a sequence of special symbols, reserved words, and expressions which either performs a specific set of actions on data or controls program flow.

Pascal/3000 statement types and purposes include:

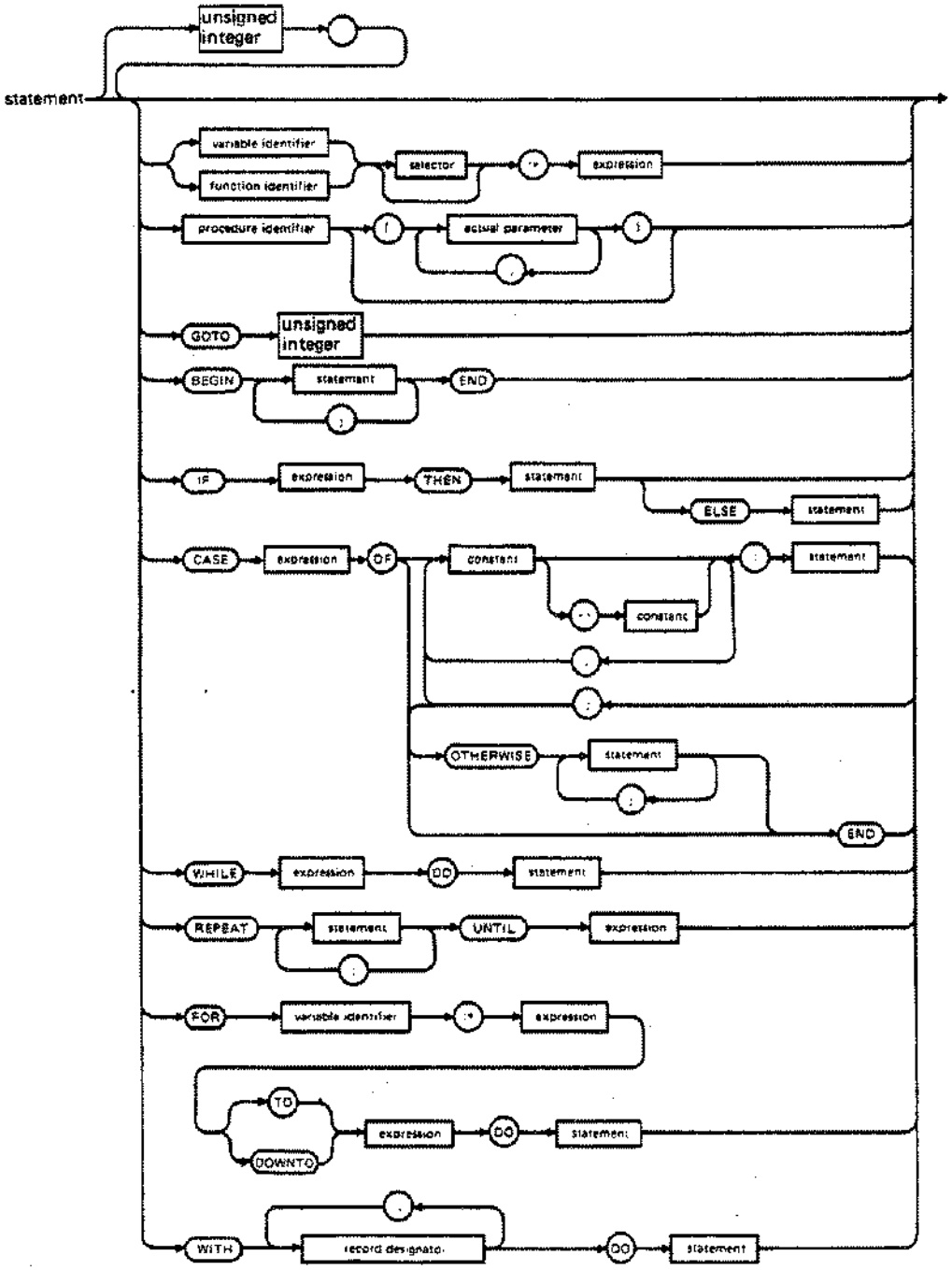
STATEMENT TYPE	PURPOSE
compound	group statements
empty	do nothing
assignment	assign a value to a variable
procedure	activate a procedure
GOTO	transfer control unconditionally
IF, CASE	conditional selection
WHILE, REPEAT, FOR	repeat a group of statements
WITH	manipulate record fields

Empty, assignment, procedure, and GOTO statements are 'simple' statements. IF, CASE, WHILE, REPEAT, FOR, and WITH statements are 'structured' statements because they themselves may contain other statements.

An integer label declared in the declaration section of the block may mark a statement (see Section 2). This label is the object of a GOTO statement.

The following pages describe each type of statement.

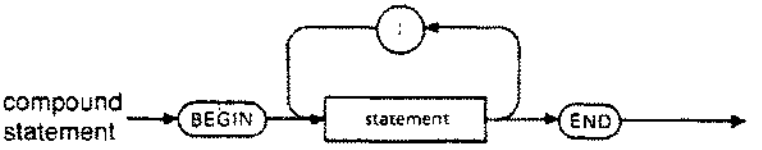
Syntax


statement
unsigned integer
variable identifier
function identifier
selector
:=
expression
procedure identifier
(
actual parameter
)
,
GOTO
unsigned integer
BEGIN
statement
END
;
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
:
statement
..
constant
,
;
OTHERWISE
statement
;
END
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement
WITH
,
record designator
DO
statement

COMPOUND STATEMENT

A compound statement is a sequence of statements bracketed by the reserved words BEGIN and END. A semi-colon (;) delimits one statement from the next. The system executes the statements in the sequence in order.

Syntax



A compound statement has two primary uses: (1) it defines the statement part of a block; (2) it replaces a single statement within a structured statement. A compound statement may also serve to logically group a series of statements.

Examples

```
PROCEDURE check_min;
  BEGIN                                                  {This        }
    IF min > max THEN                                    {compound    }
      BEGIN                                 {Compound    } {statement   }
        writeln('Min is wrong.');           {statement is} {is          }
        min := 0;                           {part of IF  } {the         }
      END;                                  {statement.  } {procedure's}
  END;                                                   {body.       }
. . .

BEGIN                 {Nested compound statements          }
  BEGIN               {for logically grouping statements.}
    start_part_1;
    finish_part_1;
  END;

  BEGIN
    start_part_2;
    finish_part_2;
  END;
END;
```

EMPTY STATEMENT

An empty statement performs no action and is denoted by no symbol. It is often useful for indicating that nothing should occur or for inserting extra semi-colons in code.

These two statements, for example, explicitly specify no action when i is 2,3,4,6,7,8,9, or 10:

```
CASE i OF
  0    : start;
  1    : continue;
  2..4 : ;
  5    : report_error;
  6..10: ;
  11   : stop;
  OTHERWISE fatal_error;
END;
```

```
IF i IN [2..4, 6..10] THEN
  (do nothing)
  ELSE continue;
```

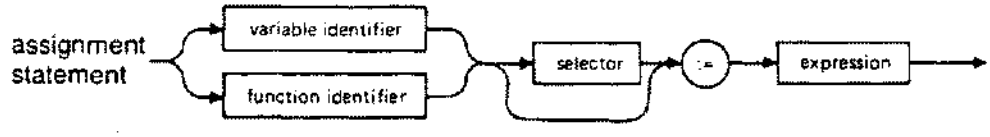
In this compound statement, there is an empty statement before END:

```
BEGIN
  I:= J + 1;
  K:= I + J;
END
```

ASSIGNMENT STATEMENT

An assignment statement assigns a value to a variable or a function result. The assignment statement consists of a variable or function identifier, an optional selector, a special symbol (:=), and an expression which computes a value.

Syntax



The receiving element may be of any type except file, or a structured type containing a file type component. An appropriate selector permits assignment to a component of a structured variable or structured function result.

The type of the expression must be assignment compatible with the type of the receiving element (see below).

ASSIGNMENT STATEMENT

Example

```
FUNCTION show_assign: integer;

  TYPE
    rec = RECORD
            f: integer;
            g: real;
          END;

    index = 1..3;
    table = ARRAY [index] OF integer;

  CONST
    ct = table [10, 20, 30];
    cr = rec [f:2, g:3.0];

  VAR
    s: integer;
    a: table;
    i: index;
    r: rec;
    p1,
    p: ^integer;
    str: string[10];

  FUNCTION show_structured: rec;
    BEGIN                          {Assign to a          }
       show_structured.f := 20;    {part of the record,  }
       show_structured := cr;      {whole record,        }
       show_assign := 50;          {outer function.      }
    END;


  BEGIN {show_assign}      {Assign to a                    }
    s := 5; i:= 3;         {simple variable,               }
    a := ct;               {array variable,                }
    a [i] := s + 5;        {subscripted array variable,    }
    r := cr;               {record variable,               }
    r.f := 5;              {selected record variable,      }
    p := p1;               {pointer variable,              }
    p^ := r.f - a [i];     {dynamic variable,              }
    str := 'Hi!';          {string variable,               }
    show_assign := p^;     {function result variable.      }
  END;  {show_assign}
```

ASSIGNMENT COMPATIBILITY

Section 2 defines type identity and type compatibility. We now define assignment compatibility.

The programmer may only assign a value of type T2 to a variable or function result of type T1 if T2 is assignment compatible with T1. For T2 to be assignment compatible with T1, any of the following conditions must be true:

(1) T1 and T2 are type compatible types which are neither files nor structures that contain files.

(2) T1 is *real* or *longreal* and T2 is *integer* or an integer subrange. The compiler converts T2 to *real* or *longreal* prior to assignment.

(3) T1 is *longreal* and T2 is *real*. The compiler converts T2 to *longreal* prior to assignment.

(4) T1 is *real* and T2 is *longreal*. The compiler rounds T2 to the precision of T1 prior to assignment.

Furthermore, a run-time or compile-time error will occur if the following restrictions are not observed:

If T1 and T2 are type compatible ordinal types, the value of type T2 must be in the closed interval specified by T1.

If T1 and T2 are type compatible set types, all the members of the value of type T2 must be in the closed interval specified by the base type of T1.

A special set of restrictions applies to assignment of string literals or variables of type *string*, PAC, or *char* (see below).

Special Cases

The pointer constant NIL is both type compatible and assignment compatible with any pointer type.

The empty set [] is both type compatible and assignment compatible with any set type.

ASSIGNMENT COMPATIBILITY

String Assignment Compatibility

Certain restrictions apply to the assignment of string literals or variables of the type *string*, packed array of *char* (PAC), or *char*.

If T1 is a string variable, T2 must be a string variable or a string literal whose length is equal to or less than the maximum length of T1. T2 cannot be a PAC or char variable. Assignment sets the current length of T1.

If T1 is a PAC variable, T2 must be a PAC of equal length or a string literal whose length is less than or equal to the length of T1. T1 will be blank filled if T2 is a string literal which is shorter than T1. T2 cannot be a string or a char variable.

If T1 is a char variable, T2 may be a char variable or a string literal with a single character. T2 cannot be a string or PAC variable.

Table 3-1 summarizes these rules. The standard function *strmax* (s) returns the maximum length of the string s. The standard function *strlen* (s) returns the current length of the string s.

String constants are considered string literals when they appear on the right side of an assignment statement.

Any string operation on two string literals, such as the concatenation of two string literals, results in a string of a *string* type.

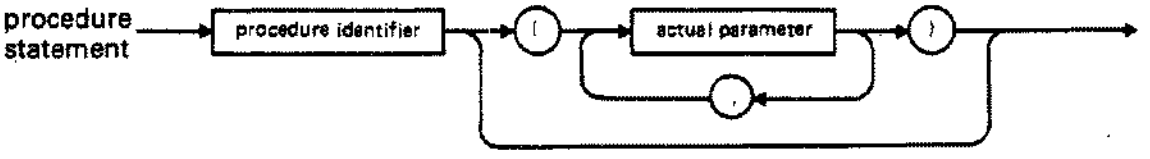
Table 3-1. STRING, PAC, AND STRING LITERAL ASSIGNMENT

T1 | :=| T2	*string*	PAC	*char*	String Literal
string	Only if *strmax* (T1) >= *strlen* (T2) *strlen* (T1) := *strlen* (T2)	Not allowed	Not allowed	Only if *strmax* (T1) >= *strlen* (T2) *strlen* (T1) := strlen (T2)
PAC	Not allowed	Only if T1 length = T2 length	Not allowed	Only if T1 length >= *strlen* (T2) T1 is padded if necessary
char	Not allowed	Not allowed	Yes	Only if *strlen* (T2) = 1

PROCEDURE STATEMENT

A procedure statement transfers program control to the block of a declared or standard procedure. A procedure statement consists of a procedure identifier and, if required, a list of actual parameters in parentheses.

Syntax



The procedure identifier must be the name of a standard procedure or a procedure declared by the programmer.

If a procedure declaration includes a formal parameter list (see Section 2), the procedure statement must supply the actual parameters. The actual parameters must match the formal parameters in number and order, except in the case of a procedure declared with the directive INTRINSIC or the directive EXTERNAL SPL VARIABLE (see Section 2). Such a procedure has option variable parameters which the programmer may omit by specifying the empty actual parameter with a comma (,) (see example below). Furthermore, the programmer may pass actual parameters to a procedure declared INTRINSIC even if no formal parameters appear in the declaration. Appendices F and G discuss the details of calling intrinsics and procedures or functions written in languages other than Pascal/3000.

Actual value parameters are expressions which must be assignment compatible with the formal value parameters.

Actual variable parameters are variables which must be type identical with the formal variable parameters. Components of a packed structure cannot appear as actual variable parameters.

Actual procedural or functional parameters are the names of procedures or functions declared by the programmer. Standard procedures or functions are not legal actual parameters.

If a procedure or function passed as an actual parameter accesses any entity non-locally upon activation, then the entity accessed is one which was accessible to the procedure or function when it was passed as a parameter. For example, suppose Procedure A uses the non-local variable x. If A is then passed as an actual procedural parameter to Procedure B, it will still be able to use x, even if x is not otherwise accessible from B. Technically, the compiler preserves the static link when A is passed.

The formal parameters, if any, of an actual procedural or functional parameter must be congruent with the formal parameters of the formal procedural or functional parameter. Two formal parameter lists are congruent if they contain an equal number of parameters and the parameters in corresponding positions are equivalent. Two parameters are equivalent if

(1) They are both value parameters of the identical type. Assignment compatibility is not legal.

(2) They are both variable parameters of the identical type.

(3) They are both procedural parameters with congruent parameter lists.

(4) They are both functional parameters with congruent parameter lists and identical result types.

After a procedure executes, control returns to the statement after the procedure statement.

PROCEDURE STATEMENT

Example

```
PROGRAM show_pstate (output);

  PROCEDURE external_proc                    {External declaration. }
     (e1: integer;
      e2: real);  EXTERNAL SPL VARIABLE; {Parameters are option }
                                         {variable.             }

  PROCEDURE actual_proc              {Actual procedure declaration.}
     (a1: integer;
      a2: real);
    BEGIN
       IF a2 < a1 THEN
          actual_proc (a1, a2-a1)  {recursive call}
       ...
    END;

  PROCEDURE outer                     {Another actual declaration. }
     (a: integer;
      PROCEDURE proc_parm
      (p1: integer; p2 : real));

    PROCEDURE inner;   (nested procedure)
      BEGIN
        actual_proc (50, 50.0);
      END;

    BEGIN {outer}                {Calling a                      }
       writeln ('Hi');           {predefined procedure,          }
       inner;                    {inner procedure,               }
       external_proc (,2.2);     {external procedure with actual }
                                 {parameter omitted,             }
       proc_parm (2, 4.0);       {procedural parameter.          }
    END;  {outer}

BEGIN {show_pstate}
  outer (10, external_proc);       {Procedure statements with    }
  outer (30, actual_proc);         {procedural parameters.       }
END.  {show_pstate}
```

GOTO STATEMENT

A GOTO statement transfers control unconditionally to a statement marked by a label. A GOTO statement consists of the reserved word GOTO and the specified label.

Syntax

The scope of labels is restricted. Labels may only mark statements appearing in the executable portion of the block where they are declared. They cannot mark statements in inner blocks. GOTO statements, however, may appear in inner blocks and reference labels in an outer block. Thus, it is possible to jump out of a procedure or function but not into one. It is also possible to jump across segment boundaries.

A GOTO statement may not lead into a component statement of a structured statement from outside that statement or from another component statement of that statement. For example, it is illegal to branch to the ELSE part of an IF statement from either the THEN part, or from outside the IF statement.

GOTO STATEMENT

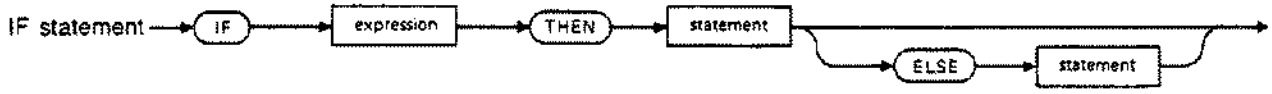
Example

```
PROGRAM show_goto;
LABEL 500, 501;
TYPE
   index = 1..10;
VAR
   i: index;
   target: integer;
   a: ARRAY[index] OF integer;
PROCEDURE check;
  VAR
    answer: string [10];
  BEGIN
    .
    {ask user if OK to search}
    IF answer= 'no' THEN GOTO 501; {jumping out of procedure}
    .
  END;

BEGIN {show_goto}
   .
  check;
   .
  FOR i := 1 TO 10 DO
    IF target = a[i] THEN GOTO 500;
  writeln (' Not found');
  GOTO 501;
500:
  writeln (' Found');
501:
END.  {show_goto}
```

An IF statement specifies a statement the system will execute provided that a particular condition is true. If the condition is false, then the system doesn't execute the statement, or, optionally, it executes another statement.

The IF statement consists of the reserved word IF, a boolean expression, the reserved word THEN, a statement, and, optionally, the reserved word ELSE and another statement.

Syntax



The statements after THEN or ELSE may be any Pascal/3000 statements, including other IF statements or compound statements. No semicolon separates the first statement and the reserved word ELSE.

The following IF statements are equivalent:

```
IF a = b THEN
  IF c = d THEN
    a := c
  ELSE
    a := e;
```

```
IF a = b THEN
  BEGIN
    IF c = d THEN
      a := c
    ELSE
      a := e;
  END;
```

That is, ELSE parts that appear to belong to more than one IF statement are always associated with the nearest IF statement.

IF STATEMENT

A common use of the IF statement is to select an action from several choices. This often appears in the following form:

```
IF e1 THEN
 ...
ELSE IF e2 THEN
 ...
ELSE IF e3 THEN
 ...
ELSE
 ...
```

This form is particularly useful to test for conditions involving real numbers or string literals of more than one character, since these types are not legal in CASE statements.

Depending on the nesting level of statements in a program, a large number of chained ELSE-IF's may cause the compiler to exceed an internal limit and not complete compilation.

Example

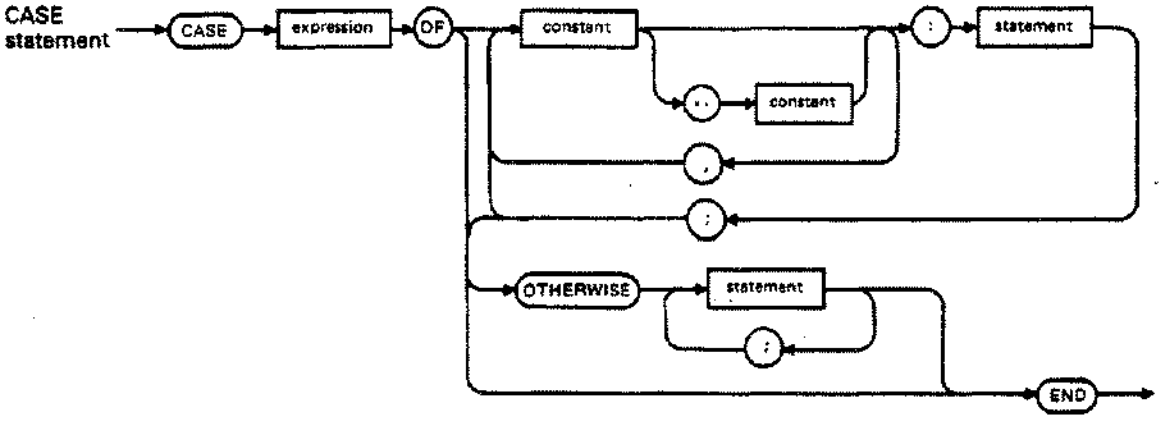
```
PROGRAM show_if (input, output);

VAR
  i,j   :  integer;
  s     :  PACKED ARRAY [1..5] OF char;
  found:  boolean;

BEGIN
   .
   .
  IF i = 0 THEN writeln ('i = 0');    {IF with no ELSE.       }
  IF found THEN                       {IF with an ELSE part.  }
    writeln ('Found it')
  ELSE
    writeln ('Still looking');
   .
   .
  IF i = j THEN                       {Select among different}
    writeln ('i = j')                 {boolean expressions.   }
  ELSE IF i < j THEN
    writeln ('i < j')
  ELSE {i > j}
    writeln ('i > j');
   .
   .
  IF s = 'RED' THEN                   {This IF statement      }
    i := 1                            {cannot be rewritten as}
  ELSE IF s = 'GREEN' THEN            {a CASE statement       }
    i := 2
  ELSE IF s = 'BLUE' THEN
    i := 3;
END.
```

CASE STATEMENT

The CASE statement selects a certain action based upon the value of an ordinal expression.

The CASE statement consists of the reserved word CASE, an ordinal expression (the selector), the reserved word OF, a list of case constants and statements, and the reserved word END. Optionally, the reserved word OTHERWISE and a list of statements may appear after the last constant and its statement.

Syntax



The selector must be an ordinal expression, i.e. it must return an ordinal value. A case constant may be a literal, a constant identifier, or a constant expression which is type compatible with the selector. Subranges may also appear as case constants. Separate ranges may not overlap.

A case constant cannot appear more than once in a list of case constants.

The programmer may associate several constants with a particular statement by listing them separated by commas.

The programmer need not bracket the statements between OTHERWISE and END with BEGIN..END.

When the system executes a CASE statement:

(1) It evaluates the selector.

(2) If the value corresponds to a specified case constant, it executes the statement associated with that constant. Control then passes to the statement following the CASE statement.

(3) If the value does not correspond to a specified case constant, it executes the statements between OTHERWISE and END. Control then passes to the statement after the CASE statement. A run time error occurs if the programmer has not used the OTHERWISE construction and the compiler has processed the CASE statement with the RANGE option ON.

CASE STATEMENT

Examples

```
PROCEDURE scanner;
  BEGIN
    get_next_char;
    CASE current_char OF
      'a'..'z',                    (Subrange label. )
      'A'..'Z':
        scan_word;

      '0'..'9':
        scan_number;

      OTHERWISE scan_special;
    END;
  END;
. . . .

FUNCTION octal_digit
   (d: digit): boolean;    {TYPE digit = 0..9}
  BEGIN
    CASE d OF
      0..7: octal_digit := true;
      8..9: octal_digit := false;
    END;
  END;
. . . .

FUNCTION op   (TYPE operators=(plus,minus,times,divide))
  (operator: operators;
   operand1,
   operand2: real)
  : real;
  BEGIN
    CASE operator OF
      plus:   op := operand1 + operand2;
      minus:  op := operand1 - operand2;
      times:  op := operand1 * operand2;
      divide: op := operand1 / operand2;
    END;
  END;
```

WHILE STATEMENT

The WHILE statement executes a statement repeatedly as long as a given condition is true. The WHILE statement consists of the reserved word WHILE, a boolean expression (the condition), the reserved word DO, and a statement.

Syntax

When the system executes a WHILE statement, it first evaluates the condition. If the condition is true, it executes the statement after DO and then re-evaluates the condition. When the condition becomes false, execution resumes at the statement after the WHILE statement. If the condition is false at the beginning, the system never executes the statement after DO.

The statement

```
WHILE condition DO statement
```

is equivalent to the following:

```
1: IF condition THEN BEGIN
      statement;
      GOTO 1;
   END;
```

Usually a program will modify data at some point so that the condition becomes false. Otherwise, the statement will repeat indefinitely. It is also possible, of course, to branch unconditionally out of a WHILE statement using a GOTO statement.

WHILE STATEMENT

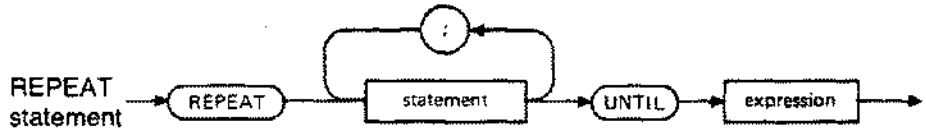
Examples

```
WHILE index <= limit DO
  BEGIN
    writeln (real_array [index]);
    index := index + 1;
  END;
  .
  .

WHILE NOT eof (f) DO
  BEGIN
    read (f, ch);
    writeln (ch);
  END;
```

REPEAT STATEMENT

A REPEAT statement executes a statement or group of statements repeatedly until a given condition is true. A REPEAT statement consists of the reserved word REPEAT, one or more statements, the reserved word UNTIL, and a boolean expression (the condition).

Syntax



The programmer need not bracket the statements between REPEAT and UNTIL with BEGIN..END.

When the system executes a REPEAT statment, it first executes the statement sequence and then evaluates the condition. If it is false, it executes the statement sequence again. If it is true, control passes to the statement after the REPEAT statement.

The statement

```
REPEAT
  statement;
UNTIL condition
```

is equivalent to the following:

```
1: statement;
   IF NOT condition THEN GOTO 1;
```

Usually the statement sequence will modify data at some point so that the condition becomes false. Otherwise, the REPEAT statment will loop forever. Of course, it is possible to branch unconditionally out of a REPEAT statement using a GOTO statement.

REPEAT STATEMENT

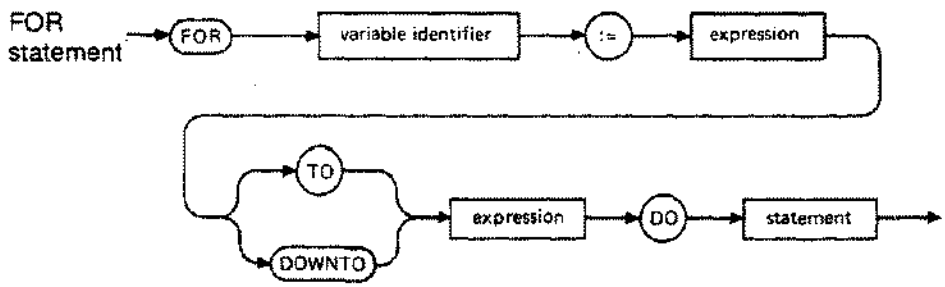
Examples

```
IF NOT eof(num_file) THEN
  REPEAT
    read (num_file, value);
    sum := sum + value;
    count := count + 1;
    average := sum / count;
    writeln ('value =', value, '   average =', average)
  UNTIL eof (num_file) OR (count >= 100);
  .
  .
REPEAT
  writeln (real_array [index]);
  index := index + 1;
UNTIL index > limit;
```

FOR STATEMENT

The FOR statement executes a statement a predetermined number of times. The FOR statement consists of the reserved word FOR and a control variable initialized by an ordinal expression (the initial value); either the reserved word TO indicating an increment or the reserved word DOWNTO indicating a decrement; another ordinal expression (the final value); the reserved word DO; and a statement.

Syntax



The control variable must be a local ordinal variable. It may not be a component of a structured variable or a locally declared procedure or function parameter. The initial and final values must be type compatible with the control variable. They must also be in range with the control variable when the initial value is first assigned. The statement after DO, of course, may be a compound statement.

When the system executes a FOR statement, it evaluates the initial and final values and assigns the initial value to the control variable. Then it executes the statement after DO. Next, it repeatedly tests the current value of the control variable and the final value for inequality, increments or decrements the control variable, and executes the statement after DO. After completion of the FOR statement, the control variable is undefined.

In a FOR..TO construction, the system never executes the statement after DO if the initial value is greater than the final value in a FOR..DOWNTO construction, it never executes the statement if the initial value is less than the final value.

FOR STATEMENT

The FOR statement

```
FOR control_var := initial TO final DO
   statement
```

is equivalent to the statement

```
BEGIN
  temp1 := initial;
  temp2 := final;
  IF temp1 <= temp2 THEN
   BEGIN
     control_var := temp1;
     statement;
     WHILE control_var <> temp2 DO
       BEGIN
         control_var := succ(control_var);  (increment)
         statement;
       END;
   END
  ELSE BEGIN END;          (Don't execute statement at all;)
END                        (control_var now undefined.     )
```

The FOR statement

```
FOR control_var := initial DOWNTO final DO
   statement
```

is equivalent to the statement

```
BEGIN
  temp1 := initial;
  temp2 := final;
  IF temp1 >= temp2 THEN
    BEGIN
      control_var := temp1;
      statement;
      WHILE control_var <> temp2 DO
        BEGIN
          control_var := pred(control_var); (decrement)
          statement;
        END;
    END
  ELSE BEGIN END;          (Don't execute statement at all;)
END                        (control_var now undefined.     )
```

In the statement after DO, the compiler protects the control variable from assignment. The programmer cannot pass the control variable as a variable parameter or use it as the control variable of a second FOR statement nested within the first. Furthermore, it may not appear as a parameter for the standard procedures *read* or *readln*. Also, the statement cannot call a procedure or function which changes the value of the control variable.

The system determines the range of values for the control variable by evaluating the two ordinal expressions once, and only once, before making any assignment to the control variable. So the statement sequence

```
i := 5;
FOR i := pred(i) TO succ(i) DO writeln('i=',i:1);
```

will write

```
i=4
i=5
i=6
```

instead of

```
i=4
i=5
```

The system will not execute the statement after DO if the initial value is greater than the final value when the FOR..TO construction appears, or less than the final value with FOR..DOWNTO.

FOR STATEMENT

If a FOR statement occurs in a section of a program with the RANGE compiler option OFF, the result of execution will not be predictable if a range error occurs. Suppose:

```
VAR
  i       : 0..10;
  initial,
  final   : 0..32767;
    .
    .
$RANGE OFF$
  initial := 1;
  final := 20;
  FOR i := initial TO final DO {The result of this FOR state-}
    writeln (i);               {ment is unpredictable, since }
                               {final is out of i's range.   }
```

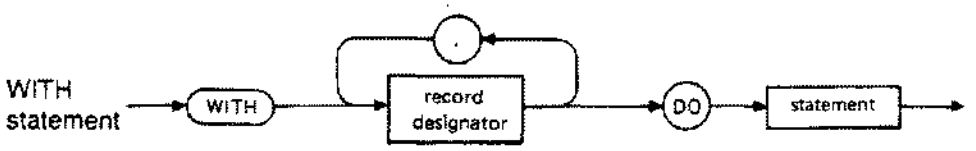
Examples

```
{VAR color: (red, green, blue, yellow);}
FOR color := red TO blue DO
  writeln ('Color is ', color);
  .
  .
FOR i := 10 DOWNTO 0 DO
  writeln (i);
writeln ('Blast Off');
  .
  .
FOR i := (a[j] * 15) TO (f(x) DIV 40) DO
  IF odd(i) THEN
    x[i] := cos(i)
  ELSE
    x[i] := sin(i);
```

WITH STATEMENT

A WITH statement allows the programmer to refer to record fields by field name alone. A WITH statement consists of the reserved word WITH, one or more record designators, the reserved word DO, and a statement.

Syntax



A record designator may be a record identifier, a function call which returns a record, or a selected record component.

The statement after DO may be a compound statement. In this statement, the programmer can refer to a record field without mention of the record to which it belongs. However, the programmer may not assign a new value to a field of a record constant or a field of a record returned by a function. In certain cases, the WITH statement saves execution time since the system need not recalculate the offset of a record field (see Section 9).

When the system executes a WITH statement, it evaluates the record designators and then executes the statement after DO.

The following statements are equivalent:

```
WITH rec DO
  BEGIN
    field1 := e1;
    writeln(field1 * field2);
  END;
```

```
BEGIN
  rec.field1 := e1;
  writeln(rec.field1
          * rec.field2);
END;
```

WITH STATEMENT

Since the sytem evaluates a record designator once and only once before it executes the statement, the statement sequence, where f is a field,

```
i := 1;
WITH a[i] DO
  BEGIN
    writeln(f);
    i:=2;
    writeln(f)
  END;
```

produces the same effect as:

```
writeln(a[1].f);
writeln(a[1].f);
```

Records with identical field names may appear in the same WITH statement. The following interpretation resolves any ambiguity:

The statement

```
WITH record1, record2, ..., recordn DO
  BEGIN
    statement;
  END;
```

is equivalent to

```
WITH record1 DO
  BEGIN
    WITH record2 DO
      BEGIN
        ...
        WITH recordn DO
          BEGIN
            statement;
          END;
        ...
      END;
  END;
```

Thus, if field f is a component of both record1 and record2, the compiler interprets an unselected reference to f as a reference to record2.f. The programmer may access the synonymous field in record1 using normal field selection, i.e. record1.f.

This interpretation also means that if r and f are records, and f is a field of r, then the statement

```
WITH r DO
  BEGIN
    WITH r.f DO
      BEGIN
        statement;
      END;
  END;
```

is equivalent to

```
WITH r,f DO
  BEGIN
    statement;
  END;
```

If a local or global identifier has the same name as a field of a designated record in a WITH statement, then the appearance of the identifier in the statement after DO is always a reference to the record field. The local or global identifier is inaccessible.

WITH STATEMENT

Examples

```
PROGRAM show_with;

TYPE
  status = (married, widowed, divorced, single);
  date   = RECORD
             month  : (jan, feb, mar, apr, may, jun,
                       july, aug, sept, oct, nov, dec);
             day    : 1..31;
             year   : integer;
           END;
  person = RECORD
             name   : RECORD
                        first, last: string[10]
                      END;
             ss     : integer;
             sex    : (male, female);
             birth  : date;
             ms     : status;
             salary : real
           END;

VAR
  employee : person;

BEGIN {show_with}
  .
  WITH employee, name, birth DO
    BEGIN
      last := 'Hacker';
      first := 'Harry';
      ss := 2147483647;
      sex := male;
      month := feb;
      day := 29;
      year := 1952;
      ms := single;
      salary := 32767.0
    END;
  .
END {show_with}
```

EXPRESSIONS

SECTION IV

INTRODUCTION

An expression is a construct which computes a result of a particular type. An expression is composed of operators and operands. An operator performs an action on objects denoted by operands and produces a value.

Operators are classified as arithmetic, boolean, relational, set, or concatenation operators. An operand may be a literal, constant identifier, set constructor, or variable. Function calls are also operands in the sense that they return a result which an operator can use to compute another value.

The type of the result of an expression is determined when the programmer writes the expression. It never changes. The actual result, however, may not be known until the sytem evaluates the expression at run time. It may differ for each evaluation. A constant expression is a restricted expression whose actual result is computable at compile time (see Section 2).

In the simplest case, an expression consists of a single operand with no operator.

Examples

```
x:= 19;                  {Simplest case.                          }

y:= 100 + x;             {Arithmetic operator with literal and    }
                         {variable operands.                      }

IF (A AND B)             {Boolean operator with boolean           }
  OR (C AND D) THEN...;  {operands.                               }

IF x > y THEN ... ;      {Relational operator with variable       }
                         {operands.                               }

setC:= setA * setB;      {Set operator with variable operands.    }

dessert:='ice'+'cream';{Concatenation operator with string       }
                         {literal operands.                       }
```

OPERATORS

An operator performs an action on one or more operands and produces a value.

Operators are classified as arithmetic, boolean, set, relational, and concatenation operators. A particular symbol may occur in more than one class of operators. For example, the symbol '+' is an arithmetic, set and concatenation operator representing numeric addition, set union, and string concatenation, respectively.

Precedence ranking determines the order in which the compiler evaluates a sequence of operators (see below).

The value resulting from the action of an operator may in turn serve as an operand for another operator.

Table 4-1 lists each Pascal/3000 operator together with its actions, permissible operands, and type of results. In the table, the term 'real' indicates both *real* and *longreal* types. Subsequent pages describe each operator in detail.

Table 4-1. PASCAL/3000 OPERATORS

Operator	Actions	Type of Operands	Type of Results
+	additions set union concatenation	*real, integer* any set type T *string*, string lit.	*real, integer* T *string*
−	subtraction set difference	*real, integer* any set type T	*real, integer* T
*	multiplication set intersection	*real, integer* any set type T	*real, integer* T
/	division	*real, integer*	*real*

Table 4-1. PASCAL/3000 OPERATORS (Continued)

DIV	division with truncation	*integer*	*integer*
MOD	modulus	*integer*	*integer*
AND	logical 'and'	*boolean*	*boolean*
OR	logical 'or'	*boolean*	*boolean*
NOT	logical negation	*boolean*	*boolean*
<	less than	any simple type *string*, or PAC	*boolean* *boolean*
>	more than	any simple type *string*, or PAC	*boolean* *boolean*
<=	less than or equal set subset	any simple type *string*, or PAC any set	*boolean* *boolean* *boolean*
>=	more than or equal, set superset	any simple type *string*, or PAC any set	*boolean* *boolean* *boolean*
=	equal to	any simple type *string*, or PAC any set type pointer	*boolean* *boolean* *boolean* *boolean*
<>	not equal to	any simple type *string*, or PAC any set type pointer	*boolean* *boolean* *boolean* *boolean*
IN	set membership	left operand: any ordinal type T right operand: set of T	*boolean*

PRECEDENCE

The precedence ranking of a Pascal/3000 operator determines the order of its evaluation in an unparenthesized sequence of operators. The four levels of ranking are:

Precedence	Operators
highest	NOT
.	*, /, DIV, MOD, AND
.	+, -, OR
lowest	<, <=, <>, =, >=, >

The compiler evaluates higher precedence operators first. For example, since * ranks above +, it evaluates these expressions identically:

(x + y * z) and (x + (y * z))

When a sequence of operators has equal precedence, evaluation proceeds in a left-to-right manner. For example, the compiler evaluates these expressions the same way:

(x + y - z) and ((x + y) - z)

If an operator is commutative (e.g. *), the compiler may choose to evaluate the operands in any order.

Within a parenthesized expression, of course, the compiler evaluates the operators and operands without regard for any operators outside the parentheses.

ARITHMETIC OPERATORS

Arithmetic operators perform integer and real arithmetic. They include +, -, *, /, DIV, and MOD.

Most arithmetic operators permit real, longreal, integer, or integer subrange operands. DIV and MOD, however, only accept integer operands.

In general, the type of its operands determines the result type of an arithmetic operator. In certain cases, the compiler implicitly converts an operand to another type (see below).

Operator	Result
+ (unary)	The value of a single operand which may be any numeric type.
- (unary)	The negated value of a single operand which may be any numeric type.
+ (addition)	The sum of two operands which may be any but not necessarily the same numeric type.
- (subtraction)	The difference of two operands which may be any but not necessarily the same numeric type.
* (multiplication)	The product of two operands which may be any but not necessarily the same numeric type.
/ (division)	The quotient of two operands which may be any but not necessarily the same numeric type. If both operands are type integer, the result is, nevertheless, real.
DIV (division with truncation)	The truncated quotient of two operands which both must be type integer. The sign of the result is positive if the signs of the operands are the same, negative otherwise. The result is zero if the first operand is zero.

ARITHMETIC OPERATORS

MOD (modulus)	The remainder when the right operand divides the left operand. Both operands must be integers, but an error occurs if the right operand is negative or zero. The result is always positive, regardless of the sign of the left operand, which must be parenthesized if it is a negative literal (see example). The result is zero if the left operand is zero. Formally, MOD is defined as i MOD j = i - ((i DIV j) * j) where i > 0 and j > 0. Or i MOD j = i - ((i DIV j) * j) + j where i < 0 and j > 0.

Implicit Conversion

The operators +, -, *, and / permit operands with different numeric types. For example, it is possible to add an integer and a real number. The compiler converts the integer to a real number and the result of the addition is real.

This implicit conversion of operands relies on a ranking of numeric types:

Rank	Type
highest	*longreal*
..	*real*
lowest	*integer*

If two operands associated with an operator are not the same rank, the compiler converts the lower to the higher prior to the operation. The result will have the type of the higher rank operand. In sum:

One operand type	Other operand type	Result type
integer	*real*	*real*
integer	*longreal*	*longreal*
real	*longreal*	*longreal*

Real division (/) is an exception. If both operands are integers, the compiler changes both to real numbers prior to the division and the result is real.

Integer values require 1 or 2 words of storage in memory depending on their size (see Section 9). If a 1-word integer and a 2-word integer are operands for a particular arithmetic operator, the compiler converts the storage for the 1-word operand to 2 words prior to the operation. The result is a 2-word integer.

Examples

```
 Expression      Result
------------    ---------

-(+10)          -10       {Unary -.                                  }
5 + 2           7         {Addition with integer operands.           }
5 - 2.0         3.0       {Subtraction with implicit conversion.     }
5 * 2           10        {Multiplication with integer operands.     }
5.0 / 2.0       2.5       {Division with real operands.              }
5 / 2           2.5       {Division with integer operands, real      }
                          {result.                                   }
5.0L0 / 2       2.5L0     {Division with implicit conversion.        }

5 DIV 2         +2        {Division with truncation.                 }
5 DIV (-2)      -2
-5 DIV 2        -2
-5 DIV (-2)     +2

5 MOD 2         +1        {Modulus.                                  }
5 MOD (-2)      error     {Right operand must be positive.           }
(-5) MOD 2      +1        {Result is positive regardless of          }
                          {sign of left operand, which is            }
                          {parenthesized since MOD has higher        }
                          {precedence than -.                        }
```

BOOLEAN OPERATORS

The boolean operators perform logical functions on boolean type operands and produce boolean results. The boolean operators are NOT, AND, and OR.

Operator	Result
NOT (logical negation)	The logical negation of a single boolean operand according to the following table:

a	NOT a
T (rue)	F (alse)
F	T

Operator	Result
AND (logical and)	The evaluation of two boolean operands produces a boolean result according to the following table:

a	b	a AND b
T	T	T
T	F	F
F	T	F
F	F	F

Operator	Result
OR (inclusive or)	The evaluation of two boolean operands produces a boolean result according to the following table:

a	b	a OR b
T	T	T
T	F	T
F	T	T
F	F	F

Examples

```
IF NOT possible THEN forget_it;

WHILE time AND money DO your_thing;

REPEAT...UNTIL tired OR bored;
```

SET OPERATORS

The set operators perform set operations on two set operands. The result is a third set. The set operators are +, -, and *.

Operator	Result
+ (union)	A set whose members are all the elements present in the left set operand and those in the right, including members present in both sets.
- (difference)	A set whose members are the elements which are members of the left set but are not members of the right set.
* (intersection)	A set whose members are only those elements present in both of the set operands.

Operands used with set operators may be variables, constant identifiers, or set constructors (see below). The base types of the set operands must be type compatibile with each other.

Examples

```
PROGRAM show_setops;
VAR
  a, b, c: SET OF 1..10;
  x : 1..10;
BEGIN
  .
  .
  a:= [1, 3, 5];
  b:= [2, 4];
  c:= [1..10];
  x:= 9;
  a:= a + b       {Union; a is now [1, 2, 3, 4, 5].      }
  b:= c - a       {Difference; b is now [6, 7, 8, 9, 10].}
  c:= a * b       {Intersection; c is now [].            }
  c:= [2, 5] + [x]  {Set constructor operands; c is now  }
END.                {[2, 5, 9].                          }
```

RELATIONAL OPERATORS

Relational operators compare two operands and return a boolean result. The relational operators are <, <=, =, <>, >=, >, and IN. The < operator means 'less than'; <= 'less than or equal'; = 'equal'; <> 'not equal'; >= 'more than or equal'; > 'more than'; and IN indicates set membership.

Depending on the type of their operands, relational operators are classified as simple, set, pointer, or string relational operators.

Simple Relational Operators

A simple relational operator has operands of any simple type, i.e. *integer, boolean, char, real, longreal,* enumerated, or subrange. All the operators listed above except IN may be simple relational operators. The operands must be type compatible, but the compiler may implicitly convert numeric types before evaluation (see Arithmetic Operators above).

For numeric operands, simple relational operators impose the ordinary definition of ordering. For char operands, the ASCII collating sequence defines the ordering. For enumerated operands, the sequence in which the constant identifiers appear in the type definition defines the ordering. Thus the predefinition of *boolean* as

```
TYPE boolean = (false, true);
```

means that *false* < *true*.

If both operands are boolean, the operator = denotes equivalence, <= implication, and <> exclusive or.

Set Relational Operators

A set relational operator has set operands. The set relational operators are =, <>, >=, <=, and IN.

The operators = and <> compare two sets for equality or inequality, respectively. The <= operator denotes the subset operation, while >= indicates the superset operation. Set A is a subset of Set B if every element of A is also a member of B. When this is true, B is said to be the superset of A.

The IN operator determines if the left operand is a member of the set specified by the right operand. When the right operand has the type SET OF T, the left operand must be type compatible with T. To test the negative of the IN operator, the programmer must use the following form:

```
NOT (element IN set)
```

Pointer Relational Operators

The programmer can use the operators = and <> to compare two pointer variables for equality or inequality, respectively. Two pointer variables are equal only if they point to exactly the same object on the heap. The programmer may compare two pointers of the same type or the constant NIL to a pointer of any type.

RELATIONAL OPERATORS

String Relational Operators

The programmer may use the string relational operators =, <>, <, <=, >, or >= to compare operands of type *string,* PAC, *char,* or string literals.

The system performs the comparison character by character using the order defined by the ASCII collating sequence.

If one operand is a string variable, the other operand may be a string variable or string literal. If the operands are not the same length and the two are equal up to the length of the shorter, the shorter operand is less. For example, if the current value of S1 is 'abc' and the current value of S2 is 'ab', then S1 > S2 is *true.* It is not possible to compare a string variable with a PAC or char variable.

If one operand is a PAC variable, the other may be a PAC of equal length or a string literal no longer than the first operand. If shorter, the string literal is blank filled prior to comparison. It is not possible to compare a PAC with a string or char variable.

If one operand is a char variable, the other may be a char variable or a single-character string literal. It is not possible to compare a char variable with a string or PAC variable.

If one operand is a string literal, the other may be a string variable, a PAC which is the same length or longer, a string literal, or a char variable provided that the string literal is only of length 1.

Table 4-2 summarizes these rules. The standard function *strmax* (s) returns the maximum length of the string variable s. The standard function *strlen* (s) returns the current length of the string expression s (see Section 6).

A string constant is considered a string literal when it appears on either side of a relational operator.

Table 4-2. STRING, PAC, CHAR, STRING LITERAL COMPARISON

A /<relop>/ B	string	PAC	char	string literal
string	Length of comparison based on smaller *strlen*	Not allowed	Not allowed	If they are not = Length of comparison based on smaller *strlen*
PAC	Not allowed	Only if A length = B length	Not allowed	Only if A length >= *strlen*(B) B is blank filled if necessary
char	Not allowed	Not allowed	Yes	Only *strlen*(B) = 1
string literal	Length of comparison based on smaller *strlen*	Only if B length >= *strlen*(A) A is blank filled if necessary	Only if *strlen*(A) = 1	Yes A or B is blank filled if necessary

RELATIONAL OPERATORS

Examples

```
PROGRAM show_relational;
TYPE
  color = (red, yellow, blue);
VAR
  a,b,c: boolean;
  p,q: ^boolean;
  s,t: SET OF color;
  col: color;
  stg: string[10];
BEGIN
   .
   .                          {Types of relational operators:    }
  b := 5 > 2;                                 {simple,      }
  b := 5 < 25.0L+1;                           {simple,      }
  b := a AND (b OR (NOT c AND (b <= a)));     {implication, }
  IF (p = q) AND (p <> NIL) THEN p^:= a = b;  {pointer,     }
  b := s <> t;                                {set,         }
  b := s <= t;                                {set, subset, }
  b := col IN [yellow, blue];                 {set, IN,     }
  stg := 'alpha';
  c := stg > 'beta';                          {string.      }
END.
```

CONCATENATION OPERATOR

The concatenation operator + concatenates two operands. The operands may be string variables, string literals, function results of type *string*, or some combination of these types.

If one of the operands is type *string*, the result of the concatenation is also type *string*. If both operands are string literals, the result is a string.

It is not legal to use the concatenation operator in a constant definition.

Examples

```
VAR
  s1,s2: string[80];
BEGIN
    .
  s1:= 'abc';
  s2:= 'def';
  s1:= s1 + s2;     (S1 is now 'abcdef')
    .
  s2:= 'The first six letters are ' + s1;
    .
END.
```

OPERANDS

An operand denotes an object which an operator acts on to produce a value. An operand may be a literal, a declared constant, a variable, a set constructor, a function call, a dereferenced pointer, or the value of another expression.

Section 5 describes the form of literals; Section 2 declared constants and variables. Subsequent pages in this chapter discuss set constructors, function calls, and pointer dereferencing.

A component of a structured type may appear as an operand. This requires use of an appropriate selector (see below).

The programmer must declare and initialize a variable before it can appear as an operand in an expression.

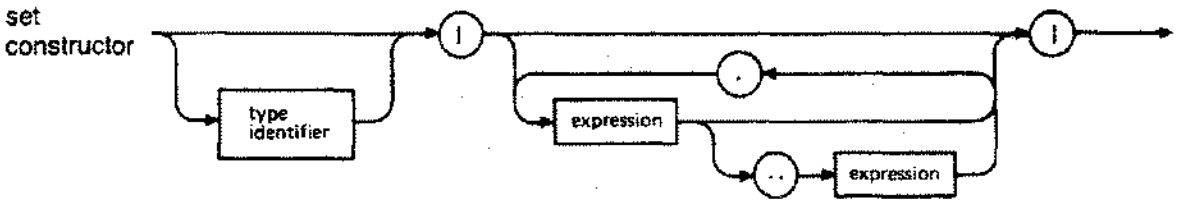
Example

```
3 + 2                        {Literals.            }
pi * 3                       {Declared constant.   }
IF day = monday THEN...      {Enumerated constants.}
x / y                        {Variables.           }
[5..9] + dolly[p]            {Set constructors.    }
succ(a) * b                  {Function call.       }
p^ DIV 2                     {Dereferenced pointer.}
(be + bop) > (dixieland)     {Result of expression.}
```

SET CONSTRUCTOR

A set constructor designates one or more values as members of a set whose type may or may not have been previously declared. A set constructor consists of an optional set type identifier and one or more ordinal expressions in square brackets. Two expressions may serve as the lower and upper bound of a subrange.

Syntax



If the programmer specifies the set type identifier, the values in the brackets must be type compatible with the base type of the set. If no set type identifier appears, the values must be type compatible with each other. The symbols (. and .) may replace the left and right square brackets, respectively.

Set constructors may appear as operands in expressions in executable statements. Set constructors with constant values are legal in the definition of constants (see Section 2).

If untyped set constructors appear as operands for a set operator, the compiler may not be able to construct the sets. Suppose, for example, that k, i, and j are integer variables. Then the expression

IF k IN [i] + [j] THEN <statement>

produces a compile time error. The compiler cannot construct a set with a cardinality greater than 32768 (see Section 9) and, since k, i, and j are in the subrange *minint..maxint*, the compilation fails.

SET CONSTRUCTOR

Examples

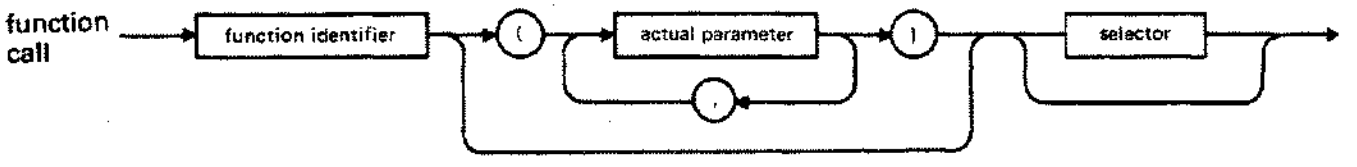
```
PROGRAM show_setconstructor;
TYPE
  int_set = SET OF 1..100;
  cap_set = SET OF 'A'..'Z';
VAR
  a,b: 0..255;
  s1: SET OF integer;
  s2: SET OF char;
BEGIN
  .
  .
  s1:= int_set[(a MOD 100) + (b MOD 100)]
  s2:= cap_set['B'..'T', 'X', 'Z'];
END.
```

FUNCTION CALL

A function call activates the block of a standard or declared function. The function returns a value to the calling point of the program. An operator can perform some action on this value and, for this reason, a function call is a sort of operand.

A function call consists of a function identifier, an optional list of actual parameters in parentheses, and an optional selector.

Syntax



The actual parameters must match the formal parameters in number and order, except for a function declared with the directives INTRINSIC or EXTERNAL SPL VARIABLE. Such a function has option variable parameters which the programmer may omit using the empty actual parameter specified by a comma (,) (see Appendix F). It is also possible to pass actual parameters to functions declared INTRINSIC even when the declaration specifies no formal parameters (see Section 2).

Actual value parameters are expressions which must be assignment compatible with the formal value parameters.

Actual variable parameters are variables which must be type identical with the formal variable parameters. Components of a packed structure may not appear as actual variable parameters.

Actual procedural or functional parameters are the names of declared procedures or functions. Standard functions or procedures are not legal actual parameters.

The parameter list, if any, of an actual procedural or functional parameter must be congruent with the parameter list of the formal procedural or functional parameter (see Procedure Statement in Section 3).

FUNCTION CALL

If an actual functional or procedural parameter, upon activation, accesses any entity non-locally, then the entity accessed is one which was accessible to the function or procedure when its identifier was passed. For example, suppose Procedure A uses the non-local variable x. If A is passed as a parameter to Function B, then it still has access to x, even if x is otherwise inaccessible in B. Technically, the compiler preserves the static link when A is passed.

If the function result is a structured type, then the function call may select a particular component as the result. This requires the use of an appropriate selector (see below). The programmer, however, should avoid inefficient use of this alternative. It is usually better to copy the entire result of such a function into a local variable before selecting a component.

Example

```
PROGRAM show_function (input,output);
VAR
  n,
  coef,
  answer: integer;

FUNCTION fact (p: integer) : integer;
  BEGIN
    IF p > 1 THEN
      fact := p * fact (p-1)
    ELSE fact := 1
  END;

FUNCTION binomial_coef (n, r: integer) : integer;
  BEGIN
    binomial_coef := fact (n) DIV (fact (r) * fact (n-r))
  END;


BEGIN {show_function}
  read(n);
  FOR coef := 0 TO n DO
    writeln (binomial_coef (n, coef));
END.  {show_function}
```

POINTER DEREFERENCING

A pointer variable points to a dynamically-allocated variable on the heap. The programmer may access the current value of this variable by dereferencing its pointer.

Pointer dereferencing occurs when the caret symbol (^) appears after a pointer designator in source code.

Syntax

The pointer designator may be the name of a pointer or selected component of a structured variable which is a pointer. The @ symbol may replace the caret.

Unless the RANGE compiler option is OFF, the compiler processes a program so that the system will check the value of the pointer at run-time. This value must be negative since dynamic variables are in the DL-DB area of the data stack. If it is NIL or some other positive value, dereferencing causes an error.

A dereferenced pointer can be an operand in an expression.

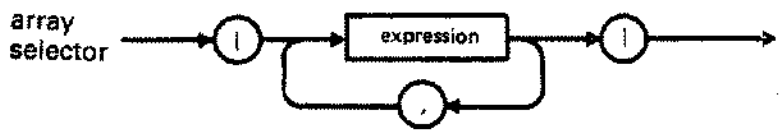
Examples

```
PROGRAM show_pointerderef (output);
TYPE
  p = ^integer;
VAR
  a,b     : integer;
  p_array : ARRAY [1..10] OF p;
  ptr     : p;
BEGIN
  .
  p_array[a]^:= a + b;
  .
  writeln(ptr^ * 2);        {Dereferenced pointer is operand. }
  .
END.
```

ARRAY SELECTOR

An array selector accesses a component of an array. The selector follows an array designator and consists of an ordinal expression in square brackets.

Syntax



The expression must be assignment compatible with the index type of the array. An array designator can be the name of an array, the selected component of a structure which is an array, or a function call which returns an array. The symbols (. and .) may replace the left and right brackets, respectively. The programmer may select a component of a multiply-dimensioned array in different ways (see example).

An array selector accesses a single component of a string variable, i.e. a character. The standard function *str*, on the other hand, returns a selected sequence of characters from a string (see Section 7).

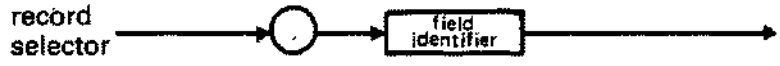
Examples

```
PROGRAM show_arrayselector;
TYPE
  a_type = ARRAY [1..10] OF integer;
VAR
  m,n          : integer;
  simp_array   : ARRAY [1..3] OF 1..100;
  multi-array  : ARRAY [1..5,1..10] OF integer;
  p            : ^a_type;
BEGIN
   .
  m:= simp_array[2];        {Assigns current value of 2nd     }
   .                        {component of simp_array to m.    }
  multi_array[2,9]:= m;     {These are                        }
  multi_array[2][9]:= m;    {equivalent.                      }
   .
  n:= p^[m MOD 10 + 1] * m {Dynamic array variable selection.}
END.
```

RECORD SELECTOR

A record selector accesses a field of a record. The record selector follows a record designator and consists of a period and the name of a field.

Syntax



A record designator is the name of a record, the selected component of a structure which is a record, or a function call which returns a record.

The WITH statement 'opens the scope' of a record, making it unnecessary for the programmer to use a record selector (see Section 3).

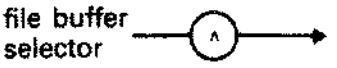
Examples

```
PROGRAM show_recordselector;
TYPE
  r_type = RECORD
             f1: integer;
             f2: char;
           END;
VAR
  a,b       : integer;
  ch        : char;
  r         : r_type;
  rec_array : ARRAY [1..10] OF r_type;
BEGIN
   .
  a:= r.f1 + b;       {Assigns current value of integer field  }
   .                  {of r plus b to a.                       }
   .
  rec_array[a].f2:= ch; {Assigns current value of ch to char  }
   .                    {field of a'th component of rec_array.}
END.
```

FILE BUFFER SELECTOR

A file buffer selector accesses the contents, if any, of the file buffer variable associated with the current position of a file. The selector follows a file designator and consists of the caret symbol (^).

Syntax



A file designator is the name of a file or the selected component of a structure which is a file. The @ symbol may replace the caret.

If the file buffer variable is not defined at the time of selection, a run time error occurs. Section 6 describes the file buffer variable and its possible state in detail.

Examples

```
PROGRAM show_bufferselector;
VAR
 f:FILE OF integer,
 a,b:integer,
BEGIN
   .
 a:= f^ + 2;                    {Assigns current contents of file}
   .                            {buffer plus 2 to a.             }
   .
 f^:=a + b;                     {Assigns sum of a and b to buffer}
   .                            {variable.                       }
END.
```

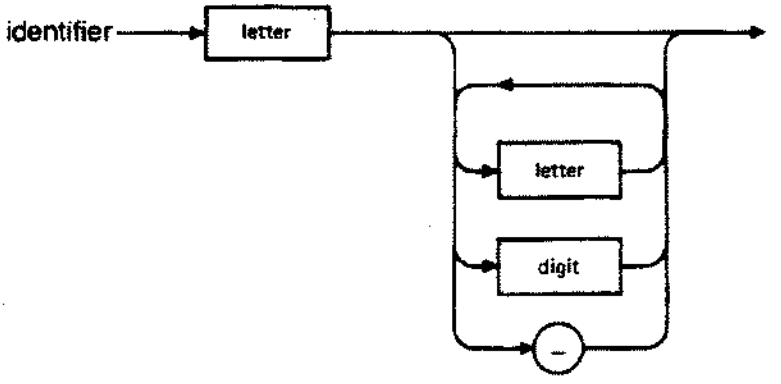
TOKENS

SECTION V

IDENTIFIERS

A Pascal/3000 identifier consists of a letter preceding an optional character sequence of letters, digits, or the underscore character (_). Identifiers denote declared constants, types, variables, procedures, functions, and programs.

Syntax:



A letter may be any of the letters in the subranges A..Z or a..z. The compiler makes no distinction between upper and lower case in identifiers. A digit may be any of the digits 0 through 9. The underscore (_) is an HP Standard Pascal extension of ANSI Standard Pascal.

An identifier may be up to a source line in length with all characters significant. Because of the requirements of the MPE Segmenter, however, the names of level 1 procedures or functions, or global variables appearing with the GLOBAL or EXTERNAL compiler options must be unique within 15 characters.

IDENTIFIERS

In general, the programmer must define an identifier before using it. Two exceptions are identifiers which define pointer types and are themselves defined later in the same declaration part, and identifiers which appear as program parameters and are declared subsequently as variables. Also, the programmer need not define an identifier which is a program, procedure, or function name, or one of the identifiers defining an enumerated type. Its initial appearance is the 'defining occurrence'. Finally, Pascal/3000 has a number of standard identifiers which the programmer may redeclare. These standard identifiers include names of standard procedures and functions, standard file variables, standard types, and procedure or function directives. Appendix B lists the Pascal/3000 standard identifiers.

Appendix B also lists the Pascal/3000 reserved words. These are system defined symbols whose meaning may never change. That is, the programmer cannot declare an identifier which has the same spelling as a reserved word.

The symbol OTHERWISE is a reserved word in the HP Standard Pascal or Pascal/3000 modes, i.e. when the compiler option STANDARD_LEVEL is set to HP or HP3000. When the STANDARD_LEVEL option is set to ANSI, or when the ANSI option is ON, however, it is no longer a reserved word and may appear as a declared identifier.

Examples

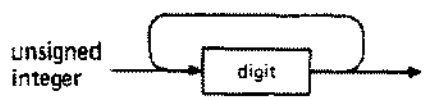
```
GOOD_TIME_9          {These identifiers    }
good_time_9          {are                  }
gOOd_TIme_9          {equivalent.          }

x2_GO
a_long_identifier
boolean              {Standard identifier.}
```

Pascal/3000 recognizes three sorts of numeric literals: integer, real, and longreal.

Integer Literals

An integer literal consists of a sequence of digits from the subrange 0..9. No spaces may separate the digits for a single literal and leading zeroes are not significant. The compiler interprets unsigned integer literals as positive values. The maximum unsigned integer literal is 2147483647, which is the value of the standard constant *maxint*. The minimum signed integer literal is −2147483648, which is the value of the standard constant *minint*.

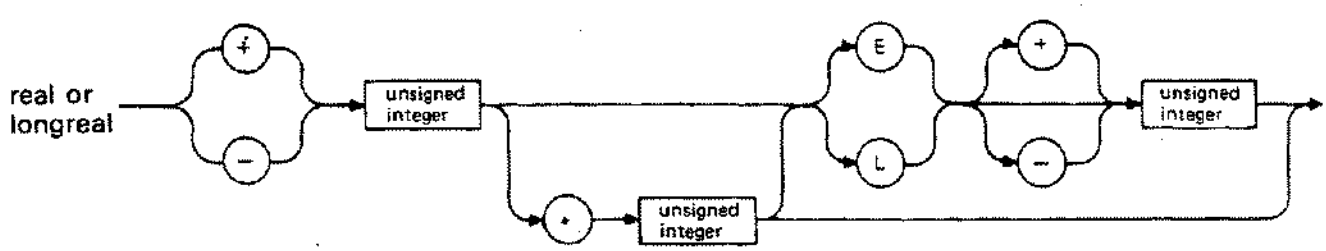
Syntax



Real and Longreal Literals

A real or longreal literal consists of a coefficient and a scale factor. An 'E' preceding the scale factor is read as 'times ten to the power of' and specifies a real literal. An 'L' preceding the scale factor also means 'times ten to the power of', but specifies a longreal literal.

Syntax



Lowercase 'e' and 'l' are legal. At least one digit must precede and follow a decimal point. A number containing a decimal point and no scale factor is considered a real literal.

NUMBERS

Real literals must be elements of the set defined by the ranges:

-1.15792E+77 to -8.63617E-78
0.0
8.63617E-78 to 1.15792E+77

Longreal literals must be members of the set defined by the ranges:

-1.157920892373162L+77 to -8.636168555094445L-78
0.0
8.636168555094445L-78 to 1.157920892373162L+77

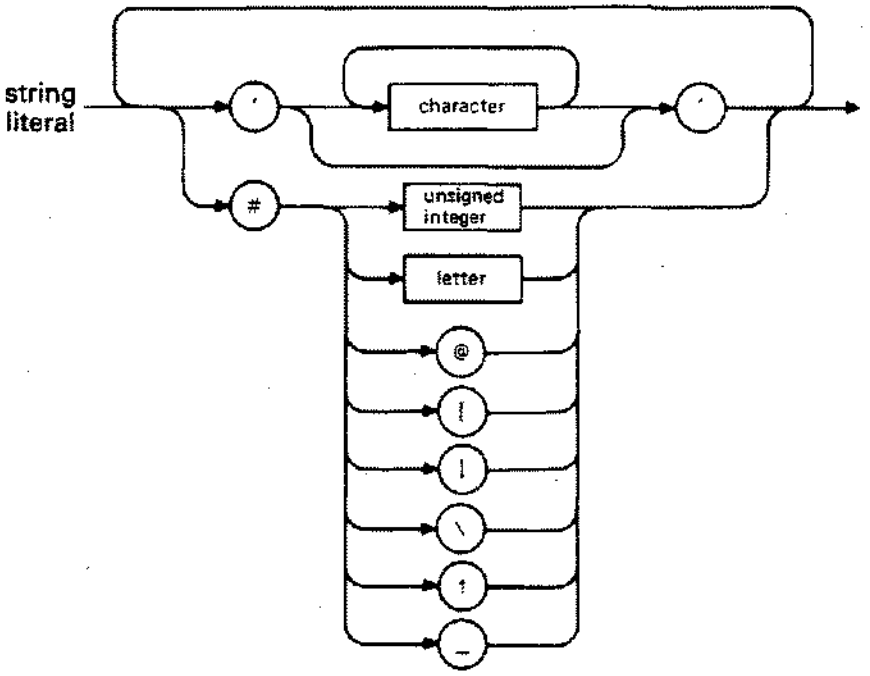
Examples

```
100                 {Integer.                       }
0.1                 {Real with no scale factor.     }
5E-3                {Real with no decimal point.    }
3.14159265358979L0  {Longreal.                      }
87.35e+8            {Real.                          }
```

STRING LITERALS

A string literal consists of a sequence of ASCII printable characters enclosed in single quote marks, a single character after a sharp symbol (#), or some combination of the two.

Syntax



The printable characters appearing between the single quotes are those ASCII characters assigned graphics and encoded by ordinal values 32 through 126.

An letter or symbol after a sharp symbol is equivalent to a control character. For example, #G or #g encodes CTRL-G, the bell character. The compiler interprets the letter or symbol according to the expression *chr* (*ord* (letter) MOD 32). Thus, the ordinal value of G is 71; modulus 32 of 71 is 7; and the ASCII value of 7 is the bell.

A number after a sharp symbol must be in the range 0..255. It directly encodes any ASCII character, printing or non-printing. For example, the string literal #80#65#83#67#65#76 is equivalent to the string literal 'PASCAL'.

A string literal is type *char*, PAC, or *string*, depending on the context.

STRING LITERALS

If a single quote is a character in a string literal, it must appear twice.

A string literal may not be longer than a single line of source code, nor may it contain separators, except for spaces (blanks) within the quotes.

Two consecutive quote marks ('') specify the null or empty string literal. Assigning this value to a string variable sets the length of the variable to zero. Assigning it to a PAC variable blank-fills the variable.

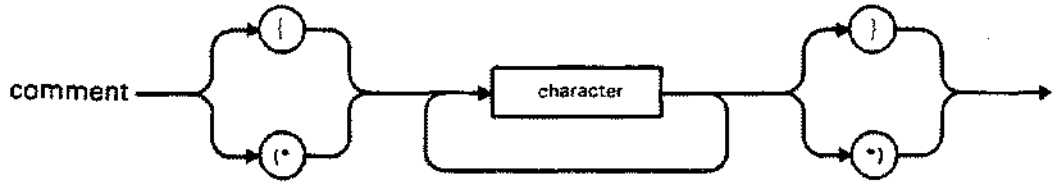
Examples

```
'Please don''t!'                    {Single quote character.}
'A'
''                                  {Null string.           }
#F
#243#H
#27'that was an ESC char, and this is also'#[
'this string has five bells'#g#g#g#7#7' in it'
```

COMMENTS

Comments consist of a sequence of characters delimited by the special symbols ¦ and ¦, or the symbols (* and *). The compiler ignores all the characters between these symbols. Comments usually document a program.

Syntax



A comment is a separator and may appear anywhere in a program a separator may appear. A comment may begin with ¦ and close with *), or begin with (* and close with ¦.

Nested comments are not legal.

A comment may cross a line boundary in source code.

Examples

```
{comment}
(*comment*)
{comment*)
{This comment
 occupies more than one line.}
```

SEPARATORS

A separator is a blank, an end-of-line marker, a comment, or a compiler option.

At least one separator must appear between any pair of consecutive identifiers, numbers, or reserved words. When one or both elements are special symbols, however, the separator is optional.

Examples

```
IF eof THEN GOTO 99          {Required separators.}
x := x + 1                   {Optional separators.}
x:=x+1                       {No separators.      }
```

SPECIAL SYMBOLS

Table 5-1 lists the special symbols valid in Pascal/3000.

Table 5-1. PASCAL/3000 SPECIAL SYMBOLS

Symbol	Purpose
+	add, set union, concatenate strings
−	subtract, set difference
*	multiply, set intersection
/	divide (real results)
=	equal to
<	less than
>	greater than
(	delimit a parameter list or a subexpression
)	
[	delimit an array index or a constructor. May be replaced by (. or .)
]	
.	select field, decimal point
,	separate listed identifiers
;	delimit statements
:	delimit list of identifiers
^	define or deference pointers, access file buffer. May be replaced by @.

SPECIAL SYMBOLS

Table 5-1. PASCAL/3000 SPECIAL SYMBOLS
(Continued)

<>	not equal
<=	less than or equal, subset
>=	greater than or equal, superset
:=	assign value to a variable
. .	subrange
{	delimit a comment. May be replaced by (* or *.)
}	
#	encode a control character
$	delimit a compiler option
'	delimit a string literal
_	separate words in an identifier

Separators may not appear within special symbols having more than one component (e.g. :=).

Certain special symbols have synonyms. In particular, (. and .) may replace the left and right brackets [and]. The symbol @ may substitute for the up-arrow ^, and (* and *) may take the place of the left and right braces { and }.

INTRODUCTION

Files are the means by which a program receives input and produces output. A file is a sequence of components of the same type. This type may be any type, except a file type or a structured type with a file type component.

Logical files are files declared in a Pascal/3000 program. Physical files are files which exist independently of a program and are controlled by the MPE operating system. The programmer may associate logical and physical files so that a program manipulates data objects external to itself.

The components of a file are indexed starting at component 1. Each file has a current component. The standard procedure *read* (f,x) copies the contents of the current component into x and advances the current position to the next component. The procedure *write* (f,x) copies x into the current component and, like *read*, advances the current position.

Each file has a buffer variable on the stack or heap whose contents, if defined, are accessible using a selector (see Section 5).

One of the standard procedures *reset, rewrite, append,* or *open* opens a file for input or output. The manner of opening a file determines the permissible operations. In particular, *reset* opens a file in the read-only state, i.e. writing is prohibited; *rewrite* and *append* open a file in the write-only state, i.e. reading is prohibited; and *open* opens a file in the read-write state, i.e. both reading and writing are legal.

Files opened with *reset, rewrite,* or *append* are sequential files. The current position advances only one component at a time. Files opened with *open* are direct access files. The programmer may relocate the current position anywhere in the file using the procedure *seek*. Direct access files have a maximum number of components determinable with the standard function *maxpos*. The maximum number of components of a sequential file, on the other hand, is not determinable with a Pascal function.

If a temporary nameless file is reopened via a *rewrite, reset, open* or *append* command and the parameter for carriage control, file disposition or file access is used, a new file is created and any association with the old file is lost.

INTRODUCTION

Textfiles are sequential files with *char* type components. Furthermore, end-of-line markers substructure textfiles into lines. The standard procedure *writeln* creates these markers. The standard files *input* and *output* are textfiles which the system automatically associates with the MPE files $STDIN and $STDLIST. The programmer cannot open textfiles for direct access.

Table 6-1 lists each Pascal/3000 file procedure or function together with a brief description of its action. The third column of the table indicates the permissible categories of files which a procedure or function may reference.

Subsequent pages describe the file procedures and functions in full.

Table 6-1. FILE PROCEDURES AND FUNCTIONS

Procedure or Function	Action	Permissible Files
append	Opens file in write-only state. Current position is after last component and eof is true.	any
close	Closes a file.	any
eof	Returns true if file is write-only, if no component exists for sequential input, or if current position in direct access file is greater than the highest-indexed component ever written to the file.	any

Table 6-1. FILE PROCEDURE AND FUNCTIONS (Continued)

eoln	Returns true if the current position of a text file is at a line marker	read-only textfiles
fnum	Returns the MPE file number of the physical file associated with a logical file.	any
get	Allows assignment of current component to buffer and, in some cases, advances current position.	read-only or read-write files
linepos	Returns number of characters read from or written to textfile since last line marker	textfiles
maxpos	Returns index of last possible component of direct access file.	direct access files
open	Opens file in read-write state. Current position is 1 and eof is false.	any file except a textfile
overprint	A form of write which causes the next line of a textfile to print over the current line.	write-only textfiles
page	Causes skip to top of new page when a textfile is printed.	write-only textfiles

Table 6-1. FILE PROCEDURE AND FUNCTIONS (Continued)

position	Returns integer indicating the current component of a non-text file.	any file except a textfile
prompt	A form of write which assures textfile buffers have been written to the device. No line marker is written.	write-only textfiles
put	Assigns the value of the buffer variable to the current component and advances the current position.	write-only or read-write files
read	Copies current component into specified variable parameter and advances current position.	read-only or read-write files
readdir	Moves current position of a direct access file to designated component and then performs read.	direct access files
readln	Performs read on textfile and then skips to next line	read-only textfiles
reset	Opens file in read-only state. Current position is 1 and eof is false.	any
rewrite	Opens file in write-only state. Current position is 1 and eof is true. Old components discarded.	any

Table 6-1. FILE PROCEDURE AND FUNCTIONS (Continued)

seek	Places current position of direct access file at specified component number.	direct access files
write	Assigns parameter value to current file component and advances current position.	write-only or read-write files
writedir	Advances current position in direct access file to be designated component and performs a write.	direct access files
writeln	Assigns parameter value to current textfile component, appends a line marker, and advances current position.	write-only textfiles

APPEND

Usage

append (f)
append (f,s)
append (f,s,t)

Parameters

f The name of a logical file. f may not be omitted.

s The name of an MPE physical file which the system will associate with f. s may be a string expression or PAC variable.

t Parameters specifying carriage control and file access. These are:

CCTL - specifies that textfile f has carriage control.

NOCCTL - specifies that textfile f has no carriage control.

SHARED - specifies that f may be open to more than one process.

EXCLUS - specifies that f may be open to only one process at a time.

T may be a string or PAC variable, or a string literal. Two parameters may appear separated by a comma. The compiler ignores leading and trailing blanks and considers upper and lower case equivalent.

The default file access for all files is EXCLUS; the default carriage control for textfiles is CCTL.

Description

The procedure *append*(f) opens file f in the write-only state and places the current position immediately after the last component. All previous contents of f remain unchanged. *EOF*(f) returns true and the file buffer f^ is undefined. The programmer may now write data on f.

If f is already open, *append* closes and then reopens it. If the parameter s is specified, the system closes any physical file previously associated with f.

When f appears as a program parameter and s is not specified, the system uses an MPE file with a name made up of the first 8 characters of the f identifier. It associates this physical file with f. If a physical file with the default name doesn't already exist, the system creates a temporary MPE file with the default name. The programmer may subsequently save this file using the *close* procedure and the SAVE parameter.

When f does not appear as a program parameter and s is not specified, the system maintains any previous association of an MPE file with f. If there is no such association, it creates a temporary nameless MPE file and opens it in the write-only state.

The programmer cannot save a temporary nameless MPE file and it becomes inaccessible after the process terminates or the physical-to-logical file association changes.

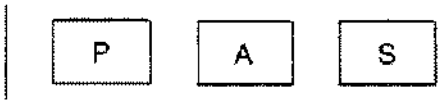
Append is an HP Standard Pascal extension of ANSI Pascal.

APPEND

Illustration

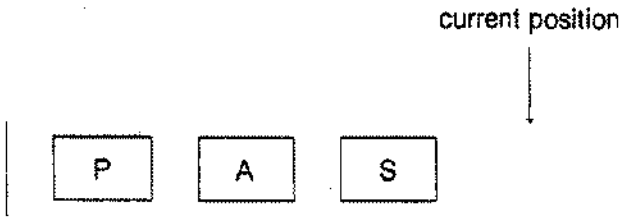
Suppose examp__file is a closed file of *char* containing three components. In order to open it and write additional material without disturbing its contents, we call *append*.

|initial condition|



state: closed

append (examp__file);



state: write-only
examp__file^: undefined
eof (examp__file): *true*

Usage

close (f)
close (f,t)

Parameters

f The name of a logical file. f may not be omitted

t A parameter specifying the disposition of any physical file associated with f. The possibilities are:

SAVE - The system will save the file as a permanent file.

PURGE - The system will destroy the file.

TEMP - The system will save the file as a temporary file which disappears at the end of the current session or job.

t may be a string or PAC variable, or a string literal. The compiler ignores leading and trailing blanks and considers upper and lower case equivalent.

If t is not specified, the file will retain its current status.

Description

The procedure *close* (f) closes the file f so that it is no longer accessible. After *close,* the function *eof* (f) is *true*, the buffer variable f^ is undefined, and any association of f with a physical file is dissolved.

When closing a direct access file, the last component of the file will be the highest-indexed component ever written to the file. The value of *maxpos* for the file, however, remains unchanged

Close is an HP Standard Pascal extension of ANSI Pascal.

EOF

Usage

eof (f)
eof

Argument

f The name of a logical file. If f is omitted, the system uses the standard file *input*.

Description

The boolean function *EOF*(f) returns *true* when f is in the write-only state, when f is in the direct access state and its current position is greater than the highest-indexed component ever written to f, or when no component remains for sequential input. Otherwise, *EOF*(f) returns *false*.

When reading a non-char value from a textfile, *EOF* may return *false* even if no other value of that type remains in the file for input, e.g. the components after the current position are all blanks.

Usage

eoln (f)
eoln

Argument

f The name of a logical textfile opened in the read-only state. If f is omitted, the system uses the standard file *input*.

Description

The boolean function *eoln* (f) returns *true* if the current position of textfile f is at an end-of-line marker. The function references the buffer variable f^, possibly causing an input operation to occur. For example, after *readln*, a call to *eoln* will place the first character of the new line in the buffer variable.

FNUM

Usage

fnum (f)

Argument

f The name of a logical file. f may not be omitted. The programmer must specify the standard files *input* or *output* by name.

Description

The function *fnum* (f) returns the MPE file number of the physical file currently associated with the logical file f. The programmer may use this number in calls to MPE File System intrinsics. If no associated physical file exists, an error occurs.

Fnum is a Pascal/3000 extension of HP Standard Pascal.

Fnum returns an integer in the range 0..255.

Usage

get (f)
get

Parameter

f The name of a logical file which must be in the read-only or read-write state. If f is omitted, the system uses the standard file *input*.

Description

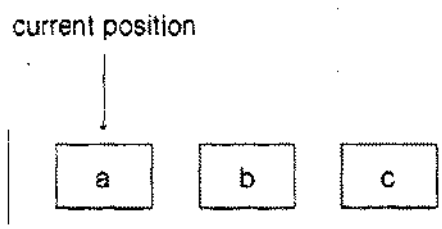
The procedure *get* (f) causes a subsequent reference to the buffer variable f^ to actually load the buffer with the current component. This is the 'deferred' get described in detail in Appendix I. In certain circumstances, namely after a call to *read*, *get* also advances the current position.

If the current component does not exist when *get* is called, f^ will be undefined and *EOF*(f) will return *true*. An error occurs if f is in the write-only state or if *EOF*(f) is *true* prior to the call to *get*.

Illustration

Suppose examp_file is a file of *char* with three components which has just been opened in the read-write state. The current position is the first component and examp_file^ is undefined. To inspect the first component, we call *get*:

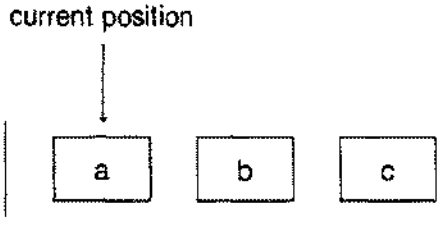
|initial condition|



state : read-write
examp_file^ : undefined
eof(examp_file) : *false*

GET

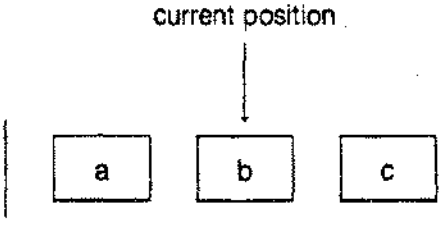
get (examp__file);



state : read-write
examp__file^ (deferred) : a
eof (examp__file) : *false*

The current position is unchanged. Now, however, a reference to examp__file^ loads the first component into the buffer and advances the current position. We assign the buffer to a variable.

char__var:= examp__file^



state : read-write
examp__file^ : a
eof (examp__file) : *false*

Usage

linepos (f)

Argument

f The name of a logical textfile. f may not be omitted. The programmer must specify the standard files *input* or *output* by name.

Description

The function *linepos* (f) returns the integer number of characters read from or written to the textfile f since the last end-of-line marker. This does not include the character in the buffer variable f↑. The result is zero after reading a line marker, or immediately after a call to *readln* or *writeln*.

MAXPOS

Usage

maxpos (f)

Argument

f The name of a logical file in the read-write state. f may not be omitted.

Description

The function *maxpos* (f) returns the integer index of the last component of f which the program may access. An error occurs if f is not opened as a direct access file.

The value of *maxpos* (f) is the limit of the physical file associated with f. If a Pascal program creates a physical file, this limit is 1023 records by default. The programmer may change this limit using the MPE :BUILD or :FILE commands.

Usage

open (f)
open (f,s)
open (f,s,t)

Parameters

f The name of a logical file which is not a textfile. f may not be omitted.

s The name of an MPE physical file which the system will associate with f. s may be a string expression or PAC variable.

t File access specification. The possibilities are:

SHARED - specifies that file may be open to more than one process.

EXCLUS - specifies that file may be open to only one process at a time.

t may be a string or PAC variable, or a string literal. The compiler ignores leading and trailing blanks and considers upper and lower case equivalent.

If t is omitted, the default is EXCLUS.

Description

The procedure *open*(f) opens f in the read-write state and places the current position at the beginning of the file. The function *EOF*(f) returns *false*, unless the file is empty. The buffer variable f↑ is undefined.

After a call to *open*, f is said to be a direct access file. The programmer may read or write data using the procedures *read*, *write*, *readdir*, or *writedir*. The procedure *seek* and the function *maxpos* are also legal. *Eof* (f) becomes *true* when the current position is greater than the index of the higest-indexed component ever written to f.

OPEN

Direct access files have a maximum number of components. The function *maxpos* returns this number.

The programmer cannot open a textfile for direct access since its format is incompatible with direct access operations.

If f is already open, the system will close it automatically and then reopen it. If the parameter s is specified, the system will close any physical file previously associated with f.

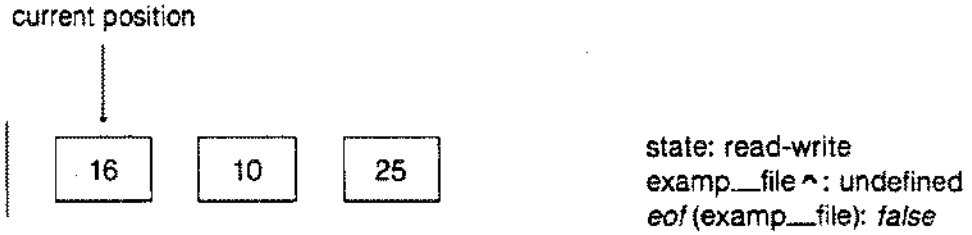
When f appears as a program parameter and s is not specified, the system uses an MPE file with a name consisting of the first 8 characters of f's identifier as the associated physical file. If a physical file with the appropriate name doesn't exist, the system creates a temporary MPE file with the default name. The programmer may save this file using the procedure *close* with the SAVE parameter.

When f does not appear as a program parameter and s is not specified, the system maintains any previous association of a physical file with f. If there is no such association, it opens a temporary nameless MPE file. The programmer cannot save this file. It becomes inaccessible after the process terminates or the physical-to-logical file association changes.

Illustration

Suppose examp_file is a file of *integer* with three components. To perform both input and output, we call *open*:

open (examp_file);



Usage

overprint (f)
overprint (f,e)
overprint (f,e1,...,en)
overprint (e)
overprint (e1,...,en)
overprint

Parameters

f The name of a logical textfile. If f is omitted, the system uses the standard file *output*.

e An expression of any simple, *string*, or PAC type, or a string literal. The system writes the value of e on f according to the formatting conventions described for the procedure *write*. Several expressions may appear separated by commas.

Description

The procedure *overprint* (f) writes a special line marker on the textfile f and advances the current position. When f is printed, this special marker causes a carriage return but suppresses the line feed. This means the printer will print the line after the special marker over the line preceding it.

After the execution of *overprint* (f), the buffer variable f^ is undefined and *eoln* (f) is false.

The expression parameter e behaves exactly like the equivalent parameter for the procedure *write* (see below).

PAGE

Usage

page (f)
page

Parameter

f The name of a logical textfile. If f is omitted, the system uses the standard file *output.*

Description

The procedure *page* (f) writes a special character to the textfile f which causes the printer to skip to the top of form when f is printed. The current position in f advances and the buffer variable f^ is undefined.

The form feed only works if the file (f) has been associated to the lineprinter with a file equation, or the "CCTL" option is used.

Usage

position (f)

Argument

f The name of a logical file which is not a textfile.

Description

The function *position*(f) returns the integer index of the current component of f, starting from 1. Input or output operations will reference this component. f must not be a textfile. If the buffer variable f↑ is full, the result is the index of the component in the buffer.

f can't be associated with a physical file which is a tape.

PROMPT

Usage

prompt (f)
prompt (f,e)
prompt (f,e1,...,en)
prompt (e)
prompt (e1,...,en)
prompt

Parameters

f the name of a logical textfile. If f is omitted, the system uses the standard file *output*.

e An expression of any simple, *string*, or PAC type, or a string literal. The system writes the value of e on f according to the formatting conventions described for the procedure *write*. Several expressions may appear separated by commas.

Description

The procedure *prompt* (f) causes the system to write any buffers associated with textfile f to the device. *Prompt* does not write a line marker on f. The current position is not advanced and the buffer variable f^ becomes undefined.

The programmer will normally use *prompt* when directing I/O to and from a terminal. *Prompt* causes the cursor to remain on the same line after output to the screen is complete. The user may then respond with input on the same line.

The expression parameter e behaves exactly like the equivalent parameters in the procedure *write* (see below).

Usage

put (f)
put

Parameter

f The name of a logical file opened in the write-only or read-write state. If f is omitted, the system uses the standard file *output*.

Description

The procedure *put* (f) assigns the value of the buffer variable f^ to the current component and advances the current position. Following the call, f^ is undefined.

An error occurs if f is open in the read-only state.

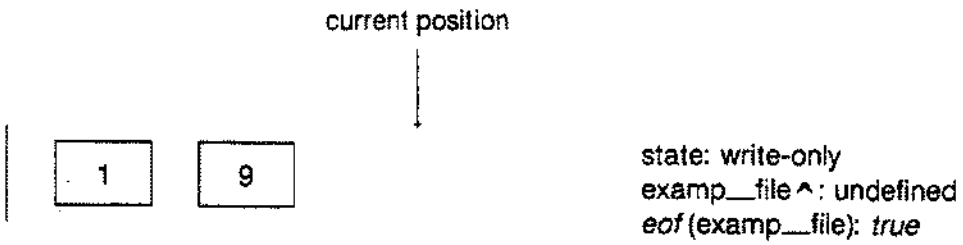
Illustration

Suppose examp__file is a file of *integer* with a single component opened in the write-only state by *append*. Furthermore, we have assigned 9 to the buffer variable examp__file^. To place this value in the second component, we call *put*:

```
append (examp__file);
examp__file^ := 9;
```

PUT

put (examp__file);



state: write-only
examp__file^: undefined
eof (examp__file): *true*

Usage

read (f,v)
read (f,v1,...,vn)
read (v)
read (v1,...,vn)

Parameters

f The name of a logical file opened in the read-only or read-write state. If f is omitted, the system uses the standard file *input*.

v The name of a simple, string, or PAC variable when f is a textfile. If f is not a textfile, its components must be assignment compatible with v. Any number of v parameters may appear separated by commas.

Description

The procedure *read* (f,v) assigns the value of the current component of f to the variable v, advances the current position, and causes any subsequent reference to the buffer variable f^ to actually load the buffer with the new current component.

The parameter v may be a component of a packed structure.

If f is a textfile, an implicit data conversion may precede the *read* operation (see below).

The call

read (f,v1,...,vn);

is equivalent to

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

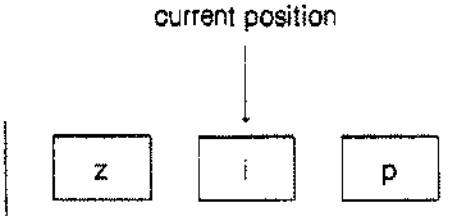
where fr is the reference established to file variable parameter f at the call to *read*(f,v1...,vn).

READ

Illustration

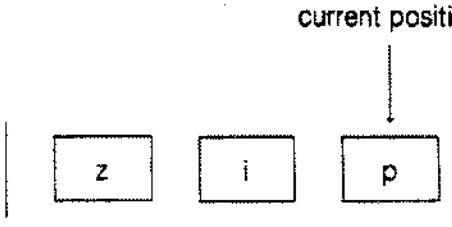
Suppose examp__file is a file of *char* opened in the read-only state. The current position is at the second component. To read the value of this component into char__var, we call *read:*

|initial condition|



state: read-only
examp__file^: i or undefined
eof (examp__file): *false*
char__var: old value, if any

read (examp__file,char__var)



state: read-only
examp__file^ (deferred): p
eof (examp__file): *false*
char__var: i

Implicit Data Conversion

If f is a textfile, its components are type *char.* The parameter v, however, need not be type *char.* It may be any simple, *string,* or PAC type. The *read* procedure performs an implicit conversion from the ASCII form which appears in the textfile f to the actual form stored in the variable v.

If v is type *real, longreal, integer,* or an integer subrange, the *read* operation searches f for a sequence of characters which satisfies the syntax for these types. The search skips preceding blanks or end-of-line markers. If v is *longreal,* the result is independent of the letter preceding the scale factor.

An error occurs if the *read* operation finds no non-blank characters or a faulty sequence of characters, or if an integer value is outside the range of v. After *read,* a subsequent reference to the buffer variable f^ will actually load the buffer with the character immediately following the number read. The programmer should also note that *eof* will be *false* if a file has more blanks or line markers, even though it contains no more numeric values.

If v is a variable of type *string* or PAC, then *read*(f,v) will fill v with characters from f. When v is type PAC and *eoln*(f) becomes *true* before v is filled, the operation puts blanks in the rest of v. If v is type *string* and *eoln*(f) becomes *true* before v is filled to its maximum length, no blank padding occurs. *Strlen*(v) then returns the actual number of characters in v. The programmer may wish to use this fact to determine the actual length of a line in a textfile.

If v is a variable of an enumerated type, *read* (f,v) searches f for a sequence of characters satisfying the syntax of a Pascal/3000 identifier. The search skips preceding blanks and line markers. Then the operation compares the identifier from f with the identifiers which are values of the type of v, ignoring upper and lower case distinctions. Finally, it assigns an appropriate value to v. An error occurs if the search finds no non-blank characters, if the string from f is not a valid Pascal/3000 identifier, or if the identifier doesn't match one of the identifiers of the type of v.

Table 6-2 shows the results of calls to *read* with various sequences of characters for different types of v.

If v is a variable of type *string* or PAC, then *read*(f,v) will fill v with characters from f. When v is type PAC and *eoln*(f) becomes *true* before v is filled, the operation puts blanks in the rest of v. If v is type *string* and *eoln*(f) becomes *true* before v is filled to its maximum length, no blank padding occurs. *Strlen*(v) then returns the actual number of characters in v. The programmer may wish to use this fact to determine the actual length of a line in a textfile.

If v is a variable of type *string* or PAC, and *eoln* is *true* when *read* is called, a string of length 0 is returned or the PAC is blankfilled. The user must use *readln* to advance beyond the current line when v is of type *string*.

Table 6-2. IMPLICIT DATA CONVERSION

Sequence of characters in f following current position	Type of v	Result stored in v
(space) (space) 1.850	*real*	1.850
(space) (end-of-line) (space) 1.850L	*longreal*	1.850
10000 (space) 10	*integer*	10000
8135 (end-of-line)	*integer*	8135
54 (end-of-line) 36	*integer*	54
1.583E7	*real*	1.583×10^7
1.583L+7	*longreal*	1.583×10^7
(space) Pascal/3000	*string*[5]	' Pasc'
(space) Pas (end-of-line) cal/3000	*string*[9]	' Pas'
(space) Pas (end-of-line) cal/3000	PAC {length 9}	' Pas
	PAC {length 9}	
(end-of-line) Pascal/3000	*string*[5]	'---------'
	enumerated	
(space) Monday (space)	enumerated	Monday

Usage

readdir (f,k,v)
readdir (f,k,v1,...,vn)

Parameters

- f The name of a logical file which is not a textfile.
- k The index of a component in f.
- v The name of a variable. The components of f must be assignment compatible with v. Any number of v parameters may appear separated by commas.

Description

The procedure *readdir* (f,k,v) places the current position at component k and then reads the value of that component into v. Formally, this is equivalent to

seek(f,k);
read(fr,v); *read* v from the file reference established by the seek.

The call *get* (f) is not required between *seek* and *read* because of the definition of *read* (see Appendix I).

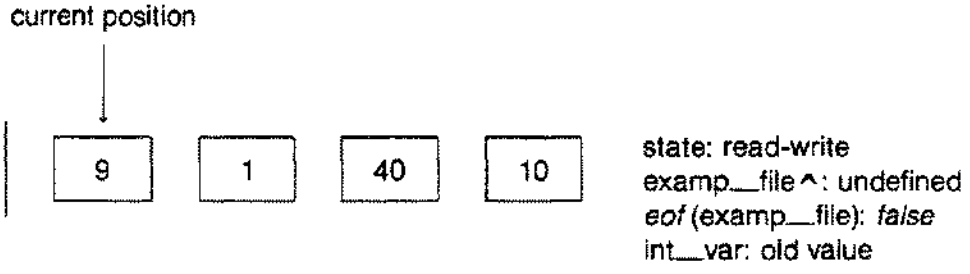
The programmer may use the procedure *readdir* only with files opened for direct access. Thus, a textfile cannot appear as a parameter for *readdir*.

READDIR

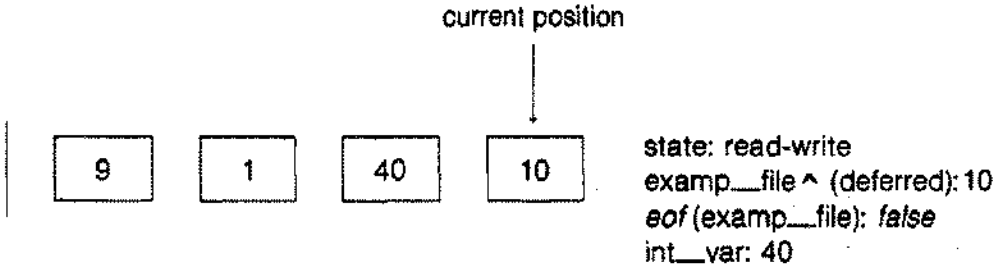
Illustration

Suppose examp__file is a file of *integer* with four components opened in the read-write state. The current position is the first component. To read the third component into int__var, we call *readdir.* After *readdir* executes, the current position is the fourth component.

|initial condition|



readdir (examp__file,3,int__var);



Usage

readln (f)
readln (f,v)
readln (f,v1,...,vn)
readln (v)
readln (v1,...vn)
readln

Parameters

f The name of a logical textfile. If f is omitted, the system uses the standard file *input*.

v The name of a variable. The type of v may be any simple type, a *string* type, or a PAC. Any number of variables may appear separated by commas.

Description

The procedure *readln* (f,v) reads a value from the textfile f into the variable v and then advances the current position to the beginning of the next line, i.e. the first character after the next end-of-line marker. The operation performs implicit data conversion if v is not type *char* (see discussion of *read* above).

The call *readln* (f,v1,...,vn) is equivalent to

```
read (f,v1,...,vn);
readln (f);
```

If the parameter v is omitted, *readln* simply advances the current position to the beginning of the next line.

RESET

Usage

reset (f)
reset (f,s)
reset (f,s,t)

Parameters

f The name of a logical file f may not be omitted.

s The name of an MPE physical file which the system will associate with f. s may be a string expression or PAC variable.

t Parameters specifying carriage control and file access. The possibilities are:

CCTL-specifies that textfile f has carriage control.

NOCCTL-specifies that textfile f has no carriage control.

SHARED-specifies that f may be open to more than one process.

EXCLUS-specifies that f may be open to only one process at a time.

t may be a string or PAC variable, or a string literal. Two parameters may appear separated by comma. The compiler ignores leading and trailing blanks and considers upper and lower case equivalent.

The default file access for all files is EXCLUS; the default carriage control for text files is CCTL.

Description

The procedure *reset* (f) opens the file f in the read-only state and places the current position at the first component. The contents of f, if any, are undisturbed. The programmer may then read from f sequentially.

If f is not empty, *eof* (f) is *false* and a subsequent reference to the buffer variable f^ will actually load the buffer with the first component. The components of f may now be read in sequence. If f is empty, however, *eof* (f) is *true* and f^ is undefined. A subsequent call to *read* produces an error.

If f is already open at the time *reset* is called, the system automatically closes and then reopens it. If the parameter s is specified, the system closes any physical file previously associated with f.

When the identifier for f appears as a program parameter and s is not specified in the call to *reset*, the system uses an MPE file with a name made up of the first 8 characters of f's identifier. This MPE file is associated with f. An error occurs if it doesn't exist.

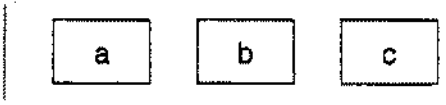
When f does not appear as a program parameter and s is not specified, the system maintains any previous association of an MPE file with f. If there is no such association, it uses a temporary nameless MPE file opened in the read-only state.

RESET

Illustration

Suppose examp__file is a closed file of *char* with three components. To read sequentially from examp__file, we call *reset*:

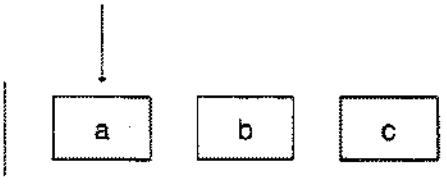
initial condition



state: closed

reset (examp__file);

current position



state: read-only
examp__file^ (deferred): a
eof (examp__file): *false*

Usage

rewrite (f)
rewrite (f,s)
rewrite (f,s,t)

Parameters

f The name of a logical file. f may not be omitted.

s The name of an MPE physical file the system will associate with f. s may be a string expression or PAC variable.

t Parameters specifying carriage control and file access. These are:

CCTL-specifies that textfile f has carriage control.

NOCCTL-specifies that textfile f has no carriage control.

SHARED-specifies that f may be open to more than one process.

EXCLUS-specifies that f may be open to only one process at a time.

t may be a string or PAC variable, or a string literal. Two parameters may appear separated by a comma. The compiler ignores leading and trailing blanks and considers upper and lower case equivalent.

The default file access for all files is EXCLUS; the default carriage control for textfiles is CCTL.

REWRITE

Description

The procedure *rewrite* (f) opens the file f in the write-only state and places the current position at the beginning of the file. The system discards any previously existing components of f. The function *eof* (f) returns *true* and the buffer variable f^ is undefined. The programmer may now write on f sequentially.

If f is already open at the time *rewrite* is called, the system closes it automatically and then reopens it. If s is specified, the system closes any physical file previously associated with f.

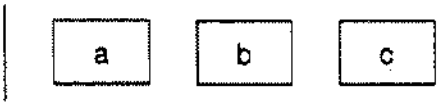
When the identifier for f appears as a program parameter and s is not specified, the system uses an MPE file with a name made up of the first 8 characters of f's identifier. This file is associated with f. If an MPE file with the default name doesn't exist, the system creates a temporary one. The programmer can save this file using the procedure *close* with the SAVE parameter.

When f does not appear as a program parameter and s is not specified, the system maintains any previous association of an MPE file with f. If there is no such association, it creates a temporary nameless MPE file opened in the write-only state. The programmer cannot save this file. It becomes inaccessible after the process terminates or the physical-to-logical file association changes.

Illustration

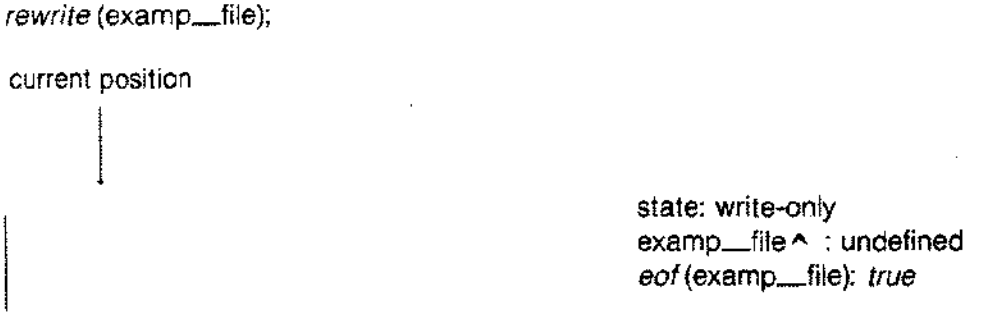
Suppose examp__file is a closed file of *char* with three components. To discard these components and write sequentially to examp__file, we call *rewrite*:

|initial condition|



state: closed

rewrite (examp__file);



SEEK

Usage

seek (f,k)

Parameters

f The name of a logical direct access file.

k The integer index of a component of f. This may be an integer expression.

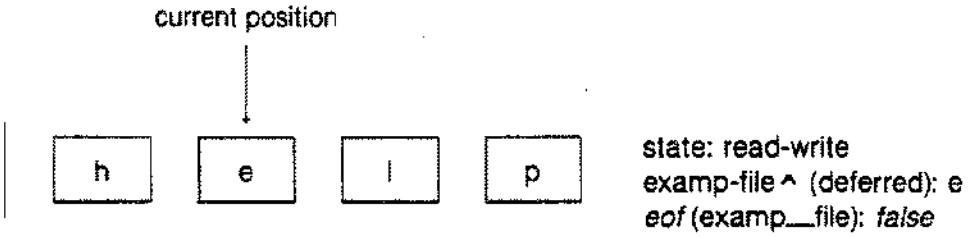
Description

The procedure *seek* (f,k) places the current position of f at component k. If k is greater than the index of the highest-indexed component ever written to f, the function *eof* (f) returns *true*, otherwise *false*. The buffer variable f^ is undefined following the call to *seek*. An error occurs if f is not open in the read-write state.

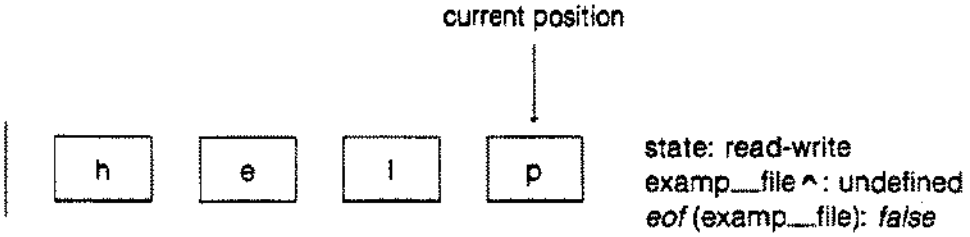
Illustration

Suppose examp__file is a file of *char* with four components opened for direct access. The current position is the second component. To change it to the fourth component, we call *seek*.

|initial condition|



seek (examp__file,4);



Usage

write (f,e)
write (f,e1,...,en)
write (e)
write (e1,...,en)

Parameters

f The name of a logical file. If f is omitted, the system uses the standard file *output*.

e If f is not a textfile, an expression whose result type is assignment compatible with the components of f. If f is a textfile, e may be an expression whose result type is any simple or *string* type, a variable of type *string* or PAC, or a string literal. Also, the programmer may format the value of e as it is written to a textfile (see below).

Description

The procedure *write* (f,e) assigns the value of e to the current component of f and then advances the current position. After the call to *write*, the buffer variable f^ is undefined. An error occurs if f is not open in the write-only or read-write state. An error also occurs if the current position of a direct access file is greater than *maxpos* (f).

The call *write* (f,e1,...en)
is equivalent to

```
write(fr,e1);
write(fr,e2);
.
.
.
write(fr,en);
```

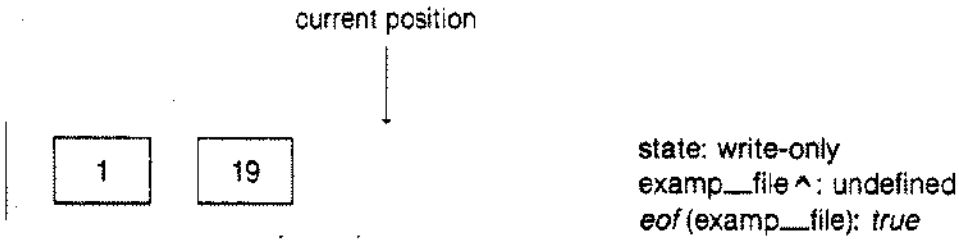
where fr is the reference established to file variable parameter f at the call to *write*(f,e1...,en).

Illustration

Suppose examp__file is a file of *integer* opened in the write-only state and that we have written one number to it. To write another number, we call *write* again:

|initial condition|

write (examp__file, 19);



Formatting of Output to Textfiles

When f is a textfile, the result type of e need not be *char*. It may be any simple, *string*, or PAC type, or a string literal. The programmer may format the value of e as it is written to f using the integer field-width parameters m and, for real or longreal values, n. If m and n are omitted, the system uses default formatting values. Thus, three forms of e are possible in source code:

e	|default formatting|
e:m	|when e is any type|
e:m:n	|when e is real or longreal|

Table 6-3 shows the system default values for m.

Table 6-3. DEFAULT FIELD WIDTHS

TYPE of e	DEFAULT FIELD WIDTH (m)
char	1
integer	12
real	12
longreal	20
boolean	length of identifier
enumerated	length of identifier
string	current length of string
PAC	length of PAC
string literal	length of string literal

If e is *boolean* or an enumerated type, the operation writes the value of e on f in upper case.

When m is specified and the value of e requires less than m characters for its representation, the operation writes e on f preceded by an appropriate number of blanks. If the value of e is longer than m, it is written on f without loss of significance, i.e. m is defeated, provided that e is a numeric type. Otherwise, the operation writes only the leftmost m characters. M may be 0 if e is not a numeric type.

When e is type *real* or *longreal*, the programmer may specify n as well as m. In this case, the operation writes e in fixed-point format with n digits after the decimal point. If n is 0, the decimal point and subsequent digits are omitted. If the programmer doesn't specify n, the operation writes e in floating-point format consisting of a coefficient and a scale factor. In no case is it possible to write more significant digits than the internal representation contains. This means *write* may change a fixed-point format to a floating-point format in certain circumstances.

Examples

```
PROGRAM show_formats (output);
VAR
  x: real;
  lr: longreal;
  george: boolean;
  list: (yes, no, maybe);
BEGIN
  writeln(999);                {default formatting}
  writeln(999:1);              {format defeated}
  writeln('abc');
  writeln('abc':2);            {string literal truncated}
  x:= 10.999;
  writeln(x);                  {default formatting}
  writeln(x:25);
  writeln(x:25:5);
  writeln(x:25:1);
  writeln(x:25:0);
  lr:= 19.1111;
  writeln(lr);
  george:= true;
  writeln(george);             {default format}
  writeln(george:2);
  list:= maybe;
  write(list);                 {default formatting}
END.
```

The output of this program is:

```
          999
999
abc
ab
1.099900E+01
                  1.0999001E+01
                        10.99900
                            11.0
                             11
 1.91110992431640625L+01
TRUE
TR
MAYBE
```

Usage

writedir (f,k,e)
writedir (f,k,e1,...,en)

Parameters

f The name of a logical file opened for direct access.

k The integer index of a component of f.

e An expression. Its result type must be assignment compatible with the components of f.

Description

The procedure *writedir* (f,k,e) places the current position at the component of f specified by k and then writes the value of e to that component. It is equivalent to

```
seek(f,k);
write(fr,e); Write e on the file reference established by the seek.
```

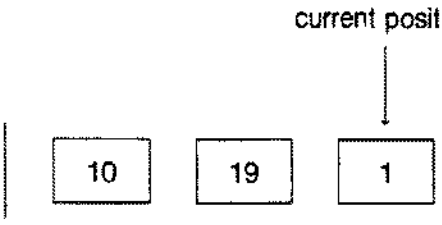
An error occurs if f has not been opened in the read-write state or if k is greater than *maxpos* (f). After *writedir* executes, the buffer variable f^ is undefined and the current position is k+1.

WRITEDIR

Illustration

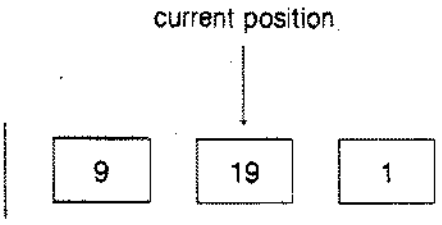
Suppose file examp__file is a file of *integer* opened for direct access. The current position is the third component. To write a number to the first component, we call *writedir*.

|initial condition|



state: read-write
examp__file^ (deferred): 1
eof (examp__file): *false*

writedir (examp__file, 1,4 + 5);



state: read-write
examp__file^: undefined
eof (examp__file): *false*

Usage

writeln (f)
writeln (f,e)
writeln (f,e1,...,en)
writeln (e)
writeln (e1,...,en)
writeln

Parameters

f The name of a logical textfile open in the write-only state. If f is omitted, the system uses the standard file *output*.

e An expression with a simple, *string*, or PAC result type, or a string literal.

Description

The procedure *writeln* (f,e) writes the value of the expression e to the textfile f, appends an end-of-line marker, and places the current position immediately after this marker. After execution, the file buffer f^ is undefined and *eof* (f) is *true*. The programmer may write the value of e with the formatting conventions described for the procedure *write* (see above).

The call *writeln*(f,e1,...,en) is equivalent to

```
write(fr,e1);
write(fr,e2);
.
.
.
write(fr,en);
writeln(fr);
```

where fr is the reference established to file variable parameter f at the call to *writeln*(fr).

The call *writeln* without the file or expression parameters effectively inserts an empty line in the standard file *output*.

LOGICAL FILES

Any file declared in the declaration part of a Pascal/3000 block is a logical file. Within a program, the scope of a file name is the scope of any other Pascal/3000 identifier. The programmer, however, may associate the logical file with a physical MPE file that exists outside the program (see below). Then operations performed on the logical file are performed on the physical file.

A logical file consists of a sequence of components of the same type. This type may be any type, except the type file or a structured type with a file type component. Every logical file has a buffer variable and a current position pointer.

The buffer variable is the same type as the type of the file's components. It is denoted:

f^

where f is the designator of the logical file. The programmer may use the buffer variable to preview the value of the current component.

The current position pointer is an integer index, starting from 1. It indicates the component that the next input or output operation will reference. The function *position* returns the value of this index, except in the case of textfiles.

After certain file operations, such as *write* with direct access files, the buffer variable is undefined. The programmer must call *get* before f^ will access the value of the current position. After other operations, such as *read*, a subsequent reference to f^ will successfully access the current component. No *get* is necessary (see Appendix I).

The programmer may assign the contents of f^ to a declared variable of the appropriate type. Alternatively, the value of an expression with an appropriate result type may be assigned to f^.

Textfiles are a special class of logical files substructured into lines (see below). *Input* and *output* are standard textfiles (see below).

The programmer must explicitly open any logical file before performing a file operation, except for *input* and *output* when they appear as program parameters (see below). The four file opening procedures are *reset, rewrite, append,* and *open* (see below). The manner of opening a logical file determines its 'state'. For example, a file opened with *append* is in the write-only state. No input operation is possible.

The programmer may use the procedures *read, write, get,* and *put,* and the functions *eof,* and *lnum* with any appropriately opened logical file, regardless of its type.

Example

```
PROGRAM show_logfile (input,output,bfile);
TYPE
  book_info = RECORD
    title  : PACKED ARRAY [1..50] OF char;
    author : PACKED ARRAY [1..50] OF char;
    number : 1..32000;
    status : (on_shelf,checked_out,lost,ordered)
  END;
VAR
  old_book: book_info;
  bfile   : FILE OF book_info;   {Declaring a logical file. }
  posnum  : integer;
BEGIN
  .
  .
  reset(bfile);   {Opening logical file which is associated }
  .               {by default with MPE file named 'BFILE'.  }
  .
  .
  old_book:= bfile^;          {Assigning buffer variable to }
  .                           {declared variable.           }
  .
  posnum:= position(bfile);       {Using index of current }
  .                               {component.             }
  .
END.
```

TEXTFILES

Textfiles are a special class of logical files which are substructured into lines by end-of-line markers. The programmer may declare textfiles with the standard identifier *text* (see Section 3). The components of a textfile are type *char*.

If the current position in a textfile advances to a line marker (i.e. beyond the last character of a line), the function *eoln* returns *true* and the buffer variable is assigned a blank. When the current position advances once more, a reference to the buffer variable will access the first character of the next line and *eoln* returns *false*, unless the next line has no characters. An end-of-line marker is not an element of type *char*. Only the procedure *writeln* places it in a textfile. A line marker always precedes an eof condition, whether the programmer terminated the last line with *writeln* or not.

The procedures *readln*, *writeln*, *page*, *prompt*, and *overprint*, and the functions *eoln* and *linepos* are available exclusively for textfiles.

Reading from a textfile may entail implicit data conversion. In certain cases, the operation searches the textfile for a sequence of characters which satisfies the syntax for a *string*, PAC, or simple type other than *char*.

Writing to a textfile may entail formatting of the output value. The programmer can specify a field-width parameter or allow the system to use various default field-width values.

The programmer cannot open textfiles for direct access. Their format is incompatible with certain direct access operations.

The system defines two standard textfiles, *input* and *output* (see below).

STANDARD FILES INPUT AND OUTPUT

The standard textfiles *input* and *output* often appear as program parameters. When they do, there are several important consequences:

(1) The programmer may not declare *input* and *output* in the source code.

(2) The system automatically resets *input* and rewrites *output*.

(3) The system automatically associates *input* and *output* with the MPE files $STDIN and $STDLIST. These files usually represent the terminal. Thus, *input* will read from the terminal and *output* will write to it. At the terminal, a colon (:) in the left-most column indicates end-of-file. The programmer may change these default associations with a file equation.

(4) If certain file operations omit the logical file name parameter, *input* or *output* is the default file. For example, the call *read* (x), where x is some variable, reads a value from *input* into x. Or consider:

```
PROGRAM sample (output);
BEGIN
  writeln('I like Pascal!');
END.
```

The program displays the string literal on the terminal screen. *Output* must appear as a program parameter; *input* need not appear, however, since the program doesn't use it.

OPENING AND CLOSING FILES

A program must open a logical file before any input, output, or other file operation is legal. Four file opening procedures are available: *reset, rewrite, append,* or *open.* When they appear as program parameters, the standard textfiles *input* and *output* are exceptions to this rule. The system automatically resets *input* and rewrites *output.*

The procedure *reset* opens a file in the read-only state without disturbing its contents. After *reset*, the current position is the first component and the program can read data sequentially from the file. No output operation is possible.

The procedure *rewrite* opens a file in the write-only state and discards any previous contents. After *rewrite*, the current position is the beginning of the file. The program can then write data sequentially to the file. No input operation is possible.

The procedure *append* is identical to the procedure *rewrite* except that the current position is placed after the last component and the file contents are undisturbed. The program can then append data to the file.

The procedure *open* opens a file in the read-write state. The contents of the file, if any, are undisturbed and the current position is the beginning of the file. The program may then read or write data.

A file opened in the read-write state is a direct access file. Using the procedure *seek*, the programmer can place the current position anywhere in the file. Furthermore, direct access files permit calls to the standard procedures *readdir* or *writedir*, which are combinations of *seek* and the procedures *read* or *write.* Direct access files have a maximum number of components. The function *maxpos* returns this number.

In contrast, files opened in the read-only or write-only states are sequential files; the current position only advances one component at a time and the maximum number of components cannot be determined by a Pascal function.

The procedure *close* explicitly closes any logical file and its associated physical file. The programmer need not use this procedure, however, before opening a file in a new state. For example, suppose file f is in the write-only state and the program calls *reset* (f). The system first closes f and its associated physical file and then reopens f in the read-only state. This is default closing.

The system also closes a file not on the heap by default when the program exits from the scope in which the file was declared. It closes a heap file by default when the procedure *dispose* uses the pointer to the file as a parameter, when the procedure *release* specifies the heap region in which the file variable is allocated, or when the program terminates.

When using *close*, the programmer may specify the disposition of a file. When the system closes a file, on the other hand, the disposition is the same as the disposition of the file when it was opened.

PHYSICAL FILES

The MPE operating system controls physical files which exist independently of a Pascal/3000 program. These files may be permanent files on disc or other media, or interactive files created at a terminal.

The programmer may associate a particular MPE file with a logical file declared in a Pascal/3000 program. The type of the logical file determines the characteristics of the MPE file. For example, the system associates a logical file of *integer* with an MPE file which is a fixed length binary file with 2 word records. File output operations create a MPE file with these characteristics; input operations require a file with these characteristics.

Except for textfiles, all MPE files associated with Pascal logical files are fixed length binary files. The system associates textfiles with variable length ASCII files with carriage control. The record length of non-textfiles depends on the type of the component; files of *integer* have 2 word records; files of *char* 1 word records; files of *longreal* 4 word records; etc. In contrast, the maximum record length of a textfile is one line of 254 bytes.

ASSOCIATING LOGICAL AND PHYSICAL FILES

A program doesn't affect the outside world unless its logical files are associated with physical files at run time. Then file operations work concurrently on logical and physical files.

In Pascal/3000, there are several ways this physical-to-logical file association can occur:

(1) The name of a logical file appears as a program parameter.

(2) The second parameter of one of the file-opening procedures specifies a physical file.

(3) The INFO parameter of the RUN command passes names of physical files to the program, and these names then appear as the second parameters of file-opening procedures.

(4) An MPE file equation specifies a physical-to-logical file association.

We consider each case in detail.

(1) A logical file name may appear as a program parameter. When this name is the first parameter for one of the file-opening procedures and there is no second parameter, the system uses a default physical file name consisting of the first 8 characters of the logical file name. This name must be an acceptable MPE filename, e.g. it cannot contain the underscore (_) character. For example, consider this source code:

```
PROGRAM case_one (input,output,file1);
VAR
  file1: FILE OF integer;
    .
    .
BEGIN
  rewrite(file1);
    .
    .
END.
```

The system associates an MPE file FILE1 with the logical file. If none exists, it creates a temporary file with this default name.

The standard files *input* and *output* are exceptions to this scheme. When they appear as program parameters, the system automatically associates them with the MPE files $STDIN and $STDLIST.

If the name of the logical file doesn't appear in the program parameter list and if a file-opening procedure doesn't have a second parameter, the system associates a temporary nameless MPE file with the logical file, provided there is no previously physical-to-logical file association. The programmer cannot save this file. After the process terminates or after the physical-to-logical file association changes, it is inaccessible.

(2) The second parameter of a file-opening procedure specifies a physical file to be associated with the logical file. For example:

```
PROGRAM case_two (input,output);
VAR
  file1 : FILE OF integer;
    .
    .
BEGIN
  rewrite(file1,'numfile');
    .
    .
END.
```

The logical file file1 is associated with the MPE physical file NUMFILE.

This association holds, even if the name of the logical file appears as a program parameter. For example:

```
PROGRAM case_three (input,output,file1);
VAR
  file1 : FILE OF integer;
    .
BEGIN
  rewrite(file1,'numfile');
    .
END.
```

The system still associates the MPE file NUMFILE, not FILE1, with file1.

The second parameter of a file-opening procedure may be a string or PAC variable. It need not be a string literal.

(3) The INFO parameter of the RUN command can pass a string literal of up to 255 characters to a Pascal variable. This variable may then appear as the string parameter of a file-opening procedure. The variable must be a program parameter. It is declared as a *string* or PAC type in the usual manner. For example:

```
:RUN PRG4; INFO="intfile"                    {MPE command. PRG4 is program}
                                             {file for case four source.   }

PROGRAM case_four (input,output,fname);
VAR
  fname : string[15];
  file1 : FILE OF integer;
    .
    .
BEGIN
  rewrite(file1,fname);
    .
    .
END.
```

The system associates the MPE file INTFILE with the logical file file1.

The programmer may list the names of several physical files in the INFO parameter. In source code, the programmer can extract the individual file names using indexing for a PAC variable or the standard function *str* for a string variable.

(4) Finally, an MPE FILE command may associate a logical file and a physical file, or change a physical-to-logical file association. The FILE command's 'formaldesignator' may be the name of a logical file and the 'filereference' the name of the physical file (see MPE Commands Reference Manual). In this case, the logical file must appear as a program parameter. For example:

```
:FILE file1=numfile
:RUN PRG5                                   {PRG5 is case_five.}
```

```
PROGRAM case_five (input,output,file1);
VAR
  file1 : FILE OF integer;
    .
BEGIN
  rewrite(file1);
    .
END.
```

The MPE file NUMFILE is associated with the logical file file1.

Alternatively, the 'formaldesignator' may be the name of a physical file specified by the string parameter of a file opening procedure. Suppose:

```
:FILE intfile=numfile
:RUN PRG6                                   {PRG6 is case_six.}
```

```
PROGRAM case_six (input,output);
VAR
  file1 : FILE OF integer;
    .
BEGIN
  rewrite(file1,'intfile');
    .
END.
```

The system associates NUMFILE, not INTFILE, with file1.

The procedures *read* and *write* perform the fundamental input and output operations. *Read* (f,x) copies the contents of the current component into x and advances the current position. *Write* (f,x) copies x into the current component and advances the current position.

The original Pascal standard describes *read* and *write* in terms of the buffer variable f^ and the procedures *get* and *put*. The procedure *put* writes the contents of the buffer variable to the current component and then advances the position. *Write* (f,x) is thus equivalent to

```
f^:= x;
put (f);
```

Read (f,x) is equivalent to

```
x:=f^;
get (f);
```

In the Jensen and Wirth Pascal Report, the procedure *get* copies the current component to the buffer variable and advances the position.

These definitions of *get* and *read*, however, have certain unfortunate consequences when I/O operations occur with interactive devices such as terminals, which were not available at the time Pascal was designed. In particular, at the initiation of a program or following a call to *readln*, the system reads the next line before the user can write a prompt.

HP Standard Pascal addresses this issue by defining a 'deferred' *get* which postpones the actual loading of a component into the buffer variable. Appendix I offers a formal description of deferred *get* and other HP Standard Pascal I/O operations. The programmer should keep these practical implications in mind:

I/O CONSIDERATIONS

(1) Suppose *read* (f,x) has just placed the value of component n in x. Then a reference to f^ copies the value of component n+1 into the buffer variable. It isn't necessary to call *get* explicitly. If *get* is called, however, a reference to f^ copies the value of component n+2 into the buffer. Component n+1 is skipped.

(2) The buffer variable is undefined after calls to *put, write, seek, writedir, writeln, open, rewrite,* and *append.* Before inspecting the current component, the programmer must call *get* explicitly.

(3) It is best not to use the buffer variable with direct access files. After *read,* for example, a reference to f^ places the next component in the buffer even if f^ appears on the left side of an assignment statement.

(4) When reading a file sequentially, there may come a time when no component is available for assignment to x. Calling *read* in this case will cause a run-time error. The programmer should use *eof* to determine if another component exists. On some files, notably terminals, this may require that a device read be performed to request another component. The component is held in the files's buffer variable and will be produced as the next result of a call to *read.*

(5) If f is a direct access file, *eof* (f) is distinct from *maxpos* (f). In particular *eof* is determined by the highest-indexed component ever written to f. *Maxpos,* on the other hand, is a limit on the size of the associated physical file. An error occurs if a progam attempts to read a component beyond the current *eof.* It is always possible, however, to write to a component with an index no greater than *maxpos* (f). This will create a new *eof* condition if the index of the component written is greater than the index of any previously written component. It is never possible to write beyond *maxpos* (f).

(6) When writing to a direct access file, the programmer may skip certain components. If the file is later read sequentially, these components will have unpredictable values.

(7) In a direct access file, the system doesn't allocate components preceding n until n is written. If n is very large and preceded by many unused components, this allocation may take a significant amount of time. The programmer should write to lower-indexed components in preference to higher-indexed components.

(8) Under the MPEIII operating system, a call *append* (f) when f has variable length records may force a system read of the entire file. Under MPEIV, a similar problem may arise when f is a nameless file with variable length records.

STANDARD PROCEDURES AND FUNCTIONS

SECTION VII

ARITHMETIC FUNCTIONS

The eight standard arithmetic functions are *abs, arctan, cos, exp, ln, sin, sqr,* and *sqrt.*

The type of the value returned depends on the type of the argument. The functions *abs* and *sqr* return integer values if integer arguments are used. The other arithmetic functions return real values if passed integer arguments. All functions return a real or longreal value when passed a real or longreal argument.

ABS

Usage

abs (x)

Argument

x A numeric expression.

Description

The function *abs* (x) computes the absolute value of the numeric expression x. If x is an integer value, the result will also be an integer. Taking the absolute value of *minint* causes a warning message at compile time and an integer overflow at run time.

Examples

Call	Return
`abs(-13)`	`13 {integer result}`
`abs(-7.11)`	`7.110000E+00.`

ARCTAN

Usage

arctan (x)

Argument

x A numeric expression.

Description

The function *arctan* (x) computes the arctangent of x. The result is in radians within the range -pi/2..pi/2.

Examples

Call	Return
`arctan(2)`	`1.107149E+00`
`arctan(-4.002)`	`-1.325944E+00`

Usage

cos (x)

Argument

x A numeric expression.

Description

The function *cos* (x) computes the cosine of x, where x is interpreted to be in radians.

Example

Call	Return
`cos(1.62)`	`-4.91836E+00`

EXP

Usage

exp (x)

Argument

x A numeric expression. Integer expressions must be in the range −176.. 176; real expressions −176.7525.. 176.7525; longreal expressions −176.75253104.. 176.75253104.

Description

The function *exp* (x) computes e to the power of x, where e is the base of the natural logarithm. If x is less than the lower bound of its permissible subrange, an underflow occurs and the value 0 is returned without an error message. If x is greater than the upper bound, a run-time error occurs.

Examples

Call	Return
exp(3)	2.008554E+01
exp(8.8E-3)	1.008839E+00

Usage

ln (x)

Argument

x Any positive numeric expression, excluding 0.

Description

The function *ln* (x) computes the natural logarithm of x. If x is 0 or less than 0, a run-time error occurs.

Examples

Call	Return
`ln(43)`	`3.761200E+00`
`ln(2.121)`	`7.518874E-01`
`ln(0)`	`{error}`

SIN

Usage

sin (x)

Argument

x A numeric expression.

Description

The function *sin* (x) computes the sine of x, where x is interpreted to be in radians. X can be any numeric value.

Example

Call	Return
`sin(0.024)`	`2.399770E-02.`

Usage

sqr (x)

Argument

x Any numeric expression.

Description

The function *sqr* (x) computes the value of x squared. If x is an integer value, the result is also an integer. If the value to be returned is greater than the maximum value for a particular type, a run-time error occurs.

Examples

Call	Return
sqr(3)	9
sqr(1.198E3)	1.435204E+06.
sqr(*maxint*)	{error}

SQRT

Usage

sqrt (x)

Argument

x Any positive numeric expression.

Description

The function *sqrt* (x) computes the square root of x. If x is less than 0, a run-time error occurs.

Examples

Call	Return
`sqrt(64)`	`8.000000E+00`
`sqrt(13.5E12)`	`3.674235E+06`
`sqrt(0)`	`0.000000E+00`
`sqrt(-5)`	`(error)`

The three predicate functions *eof*, *eoln*, and *odd* return boolean values. Section 6 describes *eof* and *eoln*.

ODD

Usage

odd (x)

Argument

x Any integer expression.

Description

The function *odd* (x) returns *true* if x is odd, and *false* otherwise.

Examples

Call	Return
odd(2 + 4)	*false*
odd(-32767)	*true*
odd(32768)	*false*
odd(0)	*false*

The two transfer functions are *trunc* and *round*.

TRUNC

Usage

trunc (x)

Argument

x Any real or longreal expression.

Description

The function *trunc* (x) returns an integer result which is the integral part of x. The absolute value of the result is not greater than the absolute value of x An integer overflow occurs if the result is not in the range *minint..maxint*.

Examples

Call	Return
trunc(5.61)	5
trunc(-3.38)	-3
trunc(18.999)	18

Usage

round (x)

Argument

x Any real or longreal expression.

Description

The function *round* (x) returns the integer value of x rounded to the nearest integer. If x is positive or zero, then *round* (x) is equivalent to *trunc* (x + 0.5); otherwise, *round* (x) is equivalent to *trunc* (x - 0.5). An integer overflow occurs if the result is not in the range *minint..maxint*.

Examples

Call	Return
round(3.1)	3
round(-6.4)	-6
round(-4.6)	-5
round(1.5)	2

ORDINAL FUNCTIONS

The ordinal functions are *chr, ord, pred,* and *succ.*

CHR

Usage

chr (x)

Argument

x An integer expression in the range 0..255.

Description

The function *chr* (x) returns the character value, if any, whose ordinal number is equal to the value of x. An error occurs if x is not within the range 0..255.

Examples

Call	Return
chr(63)	'?'
chr(82)	'R'
chr(13)	(carriage return)

Usage

ord (x)

Argument

x Any ordinal expression.

Description

The function *ord* (x) returns the integer representing the ordinal associated with the value of x. If x is an integer, x itself is returned. If x is type *char*, the result is an integer value between 0 and 255 determined by the ASCII order sequence. If x is any other ordinal type (i.e., a predefined or user-defined enumerated type), then the result is the ordinal number determined by mapping the values of the type onto consecutive non-negative integers starting at zero. For example, since the standard type *boolean* is predefined as:

TYPE boolean = (false,true)

the call *ord* (*false*) returns 0, and the call *ord* (*true*) returns 1.

For any character ch, the following is true:

chr (*ord* (ch)) = ch

Examples

Call	Return
ord('a')	97
ord('A')	65
ord(-1)	-1
ord(yellow)	2 {TYPE color=(red,blue,yellow)}
ord(red)	0

PRED

Usage

pred (x)

Argument

x Any ordinal expression.

Description

The function *pred* (x) returns the value, if any, whose ordinal number is one less than the ordinal number of x. The type of the result is identical with the type of x. A run-time error occurs if *pred* (x) does not exist. For example, suppose:

```
TYPE day = (monday,tuesday,wednesday)
```

Then,

pred (tuesday) = monday

but *pred* (monday) is undefined.

Examples

Call	Return
pred(1)	0
pred(-5)	-6
pred('B')	'A'
pred(*true*)	*false*

Usage

succ (x)

Argument

x Any ordinal expression.

Description

The function *succ* (x) returns the value, if any, whose ordinal number is one greater than the ordinal number of x. The type of the result is identical with the type of x. A run-time error occurs if *succ* (x) does not exist. For example, suppose:

TYPE color = (red, blue, yellow)

Then,

succ (red) = blue

but *succ* (yellow) is undefined.

Examples

Call	Return
succ(1)	2
succ(-5)	-4
succ('a')	'b'
succ(*false*)	*true*
succ(*true*)	(error)

NUMERIC CONVERSION FUNCTIONS

The three numeric conversion functions are *binary*, *hex*, and *octal*. All three accept arguments which are string or PAC variables, or string literals. The compiler ignores leading and trailing blanks in the argument. All other characters must be legal digits in the indicated base.

Since *binary*, *hex*, and *octal* return an integer value, which is represented as a 32 bit quantity on the HP3000, the programmer must specify all 32 bits if a negative result is desired. Alternatively, the programmer may negate the positive representation.

BINARY

Usage

binary (s)

Argument

s Any string or PAC variable, or a string literal.

Description

The function *binary* (s) converts s to an integer. S is interpreted as a binary value.

Examples

Call	Return
`binary('10011')`	19
`-binary('10011')`	-19
`binary('11111111111111111111111111101101')`	-19

Usage

hex (s)

Argument

s Any string or PAC variable, or a string literal.

Description

The function *hex* (s) converts s to an integer. S is interpreted as a hexadecimal value.

Examples

Call	Return
hex('FF')	255
hex('FFFFFF01')	-255

OCTAL

Usage

octal (s)

Argument

s Any string or PAC variable, or a string literal.

Description

The function *octal* (s) converts s to an integer. S is interpreted as an octal value.

Examples

Call	Return
`octal('77')`	63
`octal('37777777701')`	-63

Pascal/3000 supports a number of standard functions and procedures which manipulate string expressions, variables, and literals. The standard string functions include *str, strlen, strltrim, strmax, strpos, strrpt,* and *strrtrim.* The string procedures are *setstrlen, strappend, strdelete, strinsert, strmove, strread, strwrite.*

A string expression may consist of a string literal, a string variable, a string constant, a function result which is a string, or an expression formed with the concatenation operator.

Note: Strings must be initialized by the user just like any other variable. The strings functions and procedures assume that its string parameters contain valid information.

SETSTRLEN

Usage

setstrlen (s,e)

Parameters

s A string variable.

e An integer expression. The value of e cannot be greater than the maximum length of s.

Description

The procedure *setstrlen* (s,e) sets the current length of s to e without modifying the contents of s.

If the new length of s is greater than the previous length of s, the extra components will be undefined. No blank filling occurs. If the new length of s is less than the previous length of s, previously defined components beyond the new length will no longer be accessible.

SETSTRLEN

Examples

```
VAR
  alpha: string[80];
BEGIN
    .
  alpha:= 'abcdef';        {strlen(alpha) = 6}
    .
  setstrlen(alpha,2*strlen(alpha));  {Doubles current length }
    .                                {of alpha. Alpha[7]     }
    .                                {through alpha[12] not  }
    .                                {defined.               }
    .
  setstrlen(alpha,2)                 {Alpha[3] through       }
    .                                {alpha[80] unavailable. }
END.
```

Usage

str (s,b,e)

Arguments

- s A string expression.
- b An integer expression indicating the index of the starting character.
- e An integer expression indicating the length of the substring.

Description

The function *str* (s,b,e) returns the portion of s which starts at s [b] and is of length e. The result is type *string* and may be used as a string expression.

An error occurs if the *strlen*(s) is less than the sum of b and e minus 1, or b.

Example

```
VAR
  i: integer;
  wish_list: string[132];
  granted: string[5];
BEGIN
   .
   .
  i:= 13;
  wish_list:= 'wish1 wish2 wish3 wish4 wish5';
  granted:= str(wish_list,i,5);        {Selects the 3rd wish.}
   .                                   {Granted is 'wish3'.  }
   .
END.
```

STRAPPEND

Usage

strappend(s1,s2)

Parameters

s1 A string variable.

s2 A string expression.

Description

The procedure *strappend*(s1,s2) appends string s2 to s1. The call passes s1 as an actual variable parameter to the procedure. *Strlen*(s2) must be less than or equal to *strmax*(s1) -*strlen*(s1). That is, it cannot exceed the number of characters left to fill in s1. The current length of s1 is updated to *strlen*(s1) + *strlen*(s2).

Example

```
VAR
  message: string[132]
BEGIN
    .
  message:= 'Now hear ';
  strappend(message,'this!');
    .
END.
```

Usage

strdelete (s,p,n)

Parameters

s A string variable.

p An integer expression representing the starting index of the deletion.

n An integer expression representing the number of characters to be deleted.

Description

The procedure *strdelete*(s,p,n) deletes n characters from s starting at component s[p], and the current length of s is updated to current length of s-n.

Example

```
VAR
  uncensored, censored: string[80];
BEGIN
    .
  uncensored:= 'Attack at 6 a.m.!';
  strdelete(uncensored,7,strlen(uncensored)-7);
  censored:= uncensored;          {Censored is 'Attack!'.}
    .
END.
```

STRINSERT

Usage

strinsert (s1,s2,n)

Parameters

s1 A string expression.

s2 A string variable.

n An integer or an integer expression representing the offset in s2 where insertion will begin.

Description

The procedure *strinsert*(s1,s2,n) inserts string s1 into s2 starting at s2[n]. The resulting string may not exceed *strmax*(s2). The current length of s1 is updated to *strlen*(s1) + *strlen*(s2).

Examples

```
VAR
  remark: string[80];
BEGIN
   .
  remark:= 'There is' missing!';
  strinsert(' something ,remark,9);
   .
END.
```

Usage

strlen (s)

Argument

s A string expression.

Description

The function *strlen* (s) returns the current length of the string expression s.

If s is not initialized, *strlen* (s) is undefined.

Example

```
VAR
  ars, vita: string[132];
  b: boolean;
BEGIN
  .
  IF strlen(ars) > strlen(vita) THEN
    b:= true
  ELSE
    halt;
  .
END.
```

STRLTRIM

Usage

strltrim (s)

Argument

s A string expression.

Description

The function *strltrim* (s) returns a string consisting of s trimmed of all leading blanks. The function *strrtrim* trims trailing blanks (see below).

Example

```
VAR
  s: string[80];
BEGIN
   .
  s:= '          abc';
  s:=strltrim(s);       {s is now 'abc'}
   .                    {strlen(s)=3}
END.
```

Usage

strmax (s)

Argument

s A string variable.

Description

The function *strmax* (s) returns the maximum length of s.

Example

```
VAR
  s: string[132];
BEGIN
    .
  IF strlen(s) = strmax(s) THEN
    BEGIN
      s:= strltrim(s);
      s:= strrtrim(s);
    END;
    .
END.
```

STRMOVE

Usage

strmove (n,s1,p1,s2,p2)

Parameters

n An integer expression indicating the number of characters to be copied.

s1 A string expression or PAC variable.

p1 An integer expression indicating the offset in s1 from which copying will start.

s2 A string or PAC variable.

p2 An integer expression indicating the offset in s2 where copying will start.

Description

The procedure *strmove*(n,s1,p1,s2,p2) copies n characters from s1, starting at s1[p1], to s2, starting at s2[p2]. String length is updated, if needed, to p2 + (n-1) if p2 + (n-1) > *strlen*(s2).

If p2 equals *strlen*(s2) + 1, *strmove* is equivalent to appending a subset of s1 to s2.

The programmer may use *strmove* to convert PAC's to strings and vice versa. It is also an efficient way of manipulating subsets of PAC's.

Note: You should not *strmove* into an uninitialized variable regardless of its type.

Example

```
VAR
  pac: PACKED ARRAY[1..15] OF char;
    s: string[80];
BEGIN
  .
  pac:= 'Hewlett-Packard';
  .
  strmove(15,pac,1,s,1);   {Converts a PAC to a string.}
  .
END.
```

Usage

strpos (s1,s2)

Arguments

s1 A string expression.

s2 A string expression.

Description

The function *strpos* (s1,s2) returns the integer index of the position of the first occurrence of s2 in s1. If s2 is not found, zero is returned.

Example

```
CONST
  separator = ' ';
VAR
  i: integer;
  names: string[80];
BEGIN
   .
  names:= 'Jon Jill Ruth Marnie Bob Joan Wendy';
  i:= strpos (names,separator);
  IF i <> 0 THEN
    strdelete(names,1,i);              {deletes first name}
   .
END
```

STRREAD

Usage

strrread (s,p,t,v)
strread (s,p,t,v1,...,vn)

Parameters

- s A string expression.
- p An integer expression.
- t An integer or integer subrange variable.
- v A simple, string, or PAC variable. Any number of v parameters may appear separated by commas.

Description

The procedure *strread* (s,p,t,v) reads a value from s, starting at s [p] , into the variable v. After the operation, the value of the variable appearing as the t parameter will be the index of s immediately after the index of the last component read into v.

S is treated as a single-line textfile. *Strread* (s,p,t,v) is analogous to *read* (f,v) when f is a textfile of one line (see Section 6). Like *read, strread* implicitly converts a sequence of characters from s into the types *integer, real, longreal, boolean, enumerated,* PAC, or *string* (see Section 6).

An error occurs if *strread* attempts to read beyond the current length of s.

The call

strread (s,p,t,v1,...vn);

is equivalent to

strread (s,p,t,v1);
strread (s,t,t,v2);
.
.
strread (s,t,t,vn);

Example

```
VAR
  s: string[80];
  p,t: 1..80;
  m,n: integer;
BEGIN
   .
  s:= '   12  564   ';
   .
  p:= 1;
  strread(s,p,t,m);       {The value of m will be 12; }
   .                      {t will be 6.               }
   .
  strread(s,t,t,n);       {The value of n will be 564;}
   .                      {t will be 11.              }
END.
```

STRRPT

Usage

strrpt (s,n)

Arguments:

- s A string expression.
- n An integer expression indicating the number of repetitions.

Description

The function *strrpt* (s,n) returns a string composed of s repeated n times.

Example

```
CONST
  one = '1';
VAR
  b_num: string[32];
BEGIN
  .
 b_num:= strrpt(one,strmax(b_num));
  .
END.
```

Usage

strrtrim (s)

Argument

s A string expression.

Description

The function *strrtrim* (s) returns a string consisting of s trimmed of trailing blanks. Leading blanks are stripped by the function *strltrim* (see above).

Example

```
VAR
  s: string[80]
BEGIN
   .
  s:= 'abc          ';
   .
  s:= strrtrim(s);       {s is now 'abc'}
   .                     {strlen(s)=3}
END.
```

STRWRITE

Usage

strwrite (s,p,t,e)
strwrite (s,p,t,e1,...en)

Parameters

- s A string variable.
- p An integer expression.
- t An integer or integer subrange variable.
- e A simple or string expression, or a PAC variable. Any number of e parameters may appear separated by commas.

Description

The procedure *strwrite* (s,p.t,e) writes the value of e on s starting at s [p] . After the operation, the value of the variable appearing as the t parameter will be the index of the component of s immediately after the last component of s that *strwrite* has accessed.

S is treated as a single-line textfile. *Strwrite* (s,p,t,e) is analogous to write (f,e) when f is a one-line textfile (see Section 6). As with *write, strwrite* also permits the programmer to format the value of e as it is written to s using the formatting conventions described in Section 6. The same default formatting values hold for *strwrite* (see Table 6-3).

An error occurs if *strwrite* attempts to write beyond the maximum length of s, or if p is greater than *strlen* (s) + 1.

The call

strwrite (s,p,t,e1,...en);

is equivalent to

strwrite (s,p,t,e1);
strwrite (s,t,t,e2);
.
.
strwrite (s,t,t,en);

Examples

```
VAR
  s: string[80]
  p,t: 1..80;
  f,g: integer;
BEGIN
   .
  f:= 100;
  g:= 99;
   .
  p:=1;
  strwrite(s,p,t,f:1);        {S is now '100'; t is 4     }
   .
  strwrite(s,t,t,' ',g:1);    {S is now '100 99'; t is 7. }
   .
END.
```

HEAP PROCEDURES

Pascal/3000 distinguishes two classes of variables: static and dynamic.

The programmer explicitly declares a static variable in the declaration part of a block and may then refer to it by name in the body. The compiler allocates storage for this variable on the stack. The system does not deallocate this space until the process closes the scope of the variable.

On the other hand, the programmer does not declare a dynamic variable and cannot refer to it by name. Instead, a declared pointer references this variable (see Section 3). The system allocates and deallocates storage for a dynamic variable during program execution as a result of calls to the standard procedures *new* and *dispose*. The area of memory reserved for dynamic variables is termed the 'heap'. On the HP3000, this is the DL-DB area of the stack.

Pascal/3000 also supports the standard procedures *mark* and *release*, and the compiler options HEAP_DISPOSE and HEAP_COMPACT. *Mark* records the state of the heap. A subsequent call to *release* returns the heap to the state recorded by *mark*. Effectively, this disposes any variables allocated since the call to *mark*. The compiler option HEAP_DISPOSE permits the reallocation of storage space deallocated by *dispose*. The option HEAP_COMPACT allows the concatenation of available free space in the heap. Section 8 fully describes both compiler options.

When it prepares a program into an executable program file, the MPE Segmenter allocates a few thousand extra words of stack space. If a program uses a large heap, this default extra space may not be sufficient at run time. The programmer may reserve enough space by specifying values for the DL or MAXDATA parameters of the :PREP or :RUN commands.

The Pascal/3000 support library includes the procedures GETHEAP and RTNHEAP. These procedures allocate and deallocate regions of the DL-DB area (see Appendix F). A subsystem such as VPLUS uses these procedures when it is called from a Pascal program (see Appendix H).

Dynamic variables permit the creation of temporary buffer areas in memory. Furthermore, since a pointer may be a component of a structured dynamic variable, it is possible to write programs with dynamic data structures such as linked lists or trees.

Usage

new (p)
new (p,t1,...,tn)

Parameters

p Any pointer variable.

t A case constant. Nested variants may appear separated by commas.

Description

The procedure *new* (p) allocates storage for a dynamic variable on the heap and assigns its address to the pointer variable p. If insufficient heap space is available for the allocation, a run-time error occurs.

If the dynamic variable is a record with variants, then the programmer may use t to specify a case constant. This constant only determines the amount of storage allocated. The procedure call does not actually assign it to the dynamic variable. For nested variants, the programmer must list the values contiguously and in the order of their declaration (see example below).

If the programmer calls *new* for a record with variants and doesn't specify any case constants, the compiler determines storage by the size of the fixed part plus the size of the largest variant.

The programmer should avoid using an entire dynamic record variable allocated with one or more case constants as an operand in an expression, an actual parameter, or on the left side of an assignment statement. The variant may be smaller than the actual size at run time.

P may be a component of a packed structure.

Pointer dereferencing accesses the actual values stored in a dynamic variable on the heap (see Section 4).

NEW

Examples

```
PROGRAM show_new (output);
TYPE
  marital_status = (single, engaged, married, widowed, divorced);
  year = 1900..2100;
  ptr  = ^person_info;
  person_info = RECORD
                  name: string[25];
                  birdate: year;
                  next_person: ptr;
                  CASE status: marital_status OF
                    married..divorced: (when: year;
                                        CASE has_kids: boolean OF
                                          true: (how_many: 1..50)
                                        ;);
                    engaged: (date: year),
                END;single:( );
VAR
  p : ptr;
BEGIN                                  {Various legal calls of new.}
  .
  .
  new(p);
  .
  .
  new(p,engaged);
  .
  .
  new(p,married);
  .
  .
  new(p,widowed,false);
  .
  .
END.
```

DISPOSE

Usage

dispose (p)
dispose (p,t1,...,tn)

Parameters

p A pointer variable.

t A case constant value.

Description

The procedure *dispose* (p) indicates that the storage allocated for the dynamic variable referenced by p is no longer needed. The system will not actually reallocate the space unless the compiler option HEAP_DISPOSE is ON. An error occurs if p is NIL or undefined. After *dispose*, the system has closed any files in the disposed storage and p is undefined.

If the programmer specified case constant values when calling *new*, the identical constants must appear as t parameters in the call to *dispose*. Otherwise, the system may deallocate an incorrect amount of storage.

P must not reference a dynamic variable which is currently an actual variable parameter, an element of the record variable list of a WITH statement, or both.

DISPOSE

Examples

```
PROGRAM show_dispose (output);
TYPE
  marital_status = (single, engaged, married, widowed, divorced);
  year = 1900..2100;
  ptr  = ^person_info;
  person_info = RECORD
                  name: string[25];
                  birdate: year;
                  next_person: ptr;
                  CASE status: marital_status OF
                    married..divorced: (when: year;
                                        CASE has_kids: boolean OF
                                          true: (how_many:1..50);
                                          false: ()
                                        ;);
                    engaged: (date: year);
                END;single:( );
VAR
  p : ptr;
BEGIN
  .
  .
  new(p);
  .
  .
  dispose(p);
  .
  .
  new(p,engaged);
  .
  .
  dispose(p,engaged);
  .
  .
  new(p,married,false);
  .
  .
  dispose(p,married,false);
  .
  .
END.
```

Usage

mark (p)

Parameter:

p A pointer variable.

Description

The procedure *mark* (p) marks the state of the heap and sets the value of p to specify that state. In other words, *mark* saves the state of the heap in p, which the programmer must not subsequently alter by assignment.

The pointer variable appearing as the p parameter must be a dedicated variable. That is, it should not currently point to a dynamic variable when it is used with *mark*.

Mark is used in conjunction with *release*.

Example:

See *release* example below.

RELEASE

Usage

release (p)

Parameter:

p A pointer variable which previously appeared as a parameter in a call to *mark*.

Description

The procedure *release* (p) returns the heap to its state when *mark* was called with p as a parameter. This has the effect of deallocating any heap variables allocated since the program called *mark* (p). The system can then reallocate the released space. The system automatically closes any files in the released area.

An error occurs if the programmer never passed p as a parameter to *mark*, or if it was previously passed to *release* explicitly or implicitly (see example below).

After *release*, p is undefined.

Examples

```
PROGRAM show_markrelease;
VAR
 w,x,y: ^integer;
BEGIN
  .
  mark(w);
  .
  release(w);  (Returns heap to state marked by w.      )
  .
  mark(x);
  .
  mark(y);
  .
  release(x);  {Returns heap to state marked by x. The }
  .            {pointer y no longer marks a heap state.}
END.           {Release(y) is now an error.            }
```

The transfer procedures are *pack* and *unpack*.

PACK

Usage

pack (a,i,z)

Parameters

a Any ARRAY [m..n] OF t.

i An expression which is type compatible with the index of a.

z Any PACKED ARRAY [u..v] OF t.

Description

The procedure *pack* (a,i,z) assigns components of the unpacked array a, starting at component i, to each component of the packed array z. The unpacked array must be as long as or longer than the packed array, i.e. n-m >= v-u. The value of i must be greater than or equal to m, the lower bound of a. Since all the components of z are assigned a value, the normalized value of i must be less than or or equal to the difference between the lengths of a and z plus 1, i.e. i-m+1 <= (n-m) - (v-u) + 1. Otherwise, a range error occurs when *pack* attempts to access a non-existent commponent of a (see example below).

The component types of arrays a and z must be type identical. The index types of a and z, however, may be incompatible.

The call *pack* (a,i,z) is equivalent to:

```
BEGIN
  k:= i;
  FOR j:= u TO v DO
    BEGIN
      z[j]:= a[k];
      IF j <> v THEN k:= succ(k);
    END;
END;
```

where k and j are variables that are type compatible with the index type of a and the index type of z, respectively.

Examples

```
PROGRAM show_pack (input,output);
TYPE
  clothes = (hat, glove, shirt, tie, sock);
VAR
  dis : ARRAY [1..10] OF clothes;
  box : PACKED ARRAY [1..5] of clothes;
  index: integer;
  .
  .
BEGIN
  .
  .
  index:= 1;
  pack(dis,index,box);   {After pack executes, box contains     }
  .                      {the first 5 components of dis.        }
  .
  index:= 8;
  pack(dis,index,box);   {An error results when pack attempts   }
  .                      {to access non-existent 11th component}
  .                      {of dis.                               }
END.
```

Usage

unpack (z,a,i)

Parameters

z Any PACKED ARRAY [u..v] OF t.

a Any ARRAY [m..n] OF t.

i An expression that is type compatible with the index of a.

Description

The procedure *unpack* (z,a,i) successively assigns the components of the packed array z, starting at component u, to the components of the unpacked array a, starting at a [i] .

All the components of z are assigned. Hence, z must be shorter than or as long as a, i.e. (v-u) <= (n-m). Also, the normalized value of i must be less than or equal to the difference between the lengths of a and z plus 1, i.e. i-m+1 <= (n-m) - (v-u) + 1. Otherwise, an out-of-range error occurs when *unpack* attempts to index a beyond its upper bound (see example below).

The index types of a and z need not be compatible. The components of the two arrays, however, must be type identical.

UNPACK

The call *unpack* (z,a,i) is equivalent to:

```
BEGIN
  k:= i;
  FOR j:= u TO v DO
    BEGIN
      a[k]:= z[j];
      IF j <> v THEN k:= succ(k);
    END;
END;
```

where k and j are variables that are type compatible with the indices of a and z respectively.

Examples

```
PROGRAM show_unpack (input,output);
TYPE
  suit_types = (casual, business, leisure, birthday);
VAR
  suit : PACKED ARRAY [1..5] OF suit_types;
  kase : ARRAY [1..10] OF suit_types;
    .
    .
BEGIN
    .
    .
    unpack(suit,kase,1);   {After execution, the first 5      }
    .                      {components of kase contain the    }
    .                      {value of suit.                    }
    .
    unpack(suit,kase,7);   {An error results because unpack   }
    .                      {attempts to assign a component of }
    .                      {suit to a component of kase which }
    .                      {is out of range.                  }
END.
```

Pascal/3000 supports the additional procedures *assert, halt,* and the functions *baddress, ccode, sizeof,* and *waddress.* Except for halt, all these additional operations are Pascal/3000 extensions of HP Standard Pascal.

ASSERT

Usage

assert (b,i,p)
assert (b,i)

Parameters

b A boolean expression.

i A integer expression.

p A procedure identifier. P may be omitted.

Description

The procedure *assert* (b,i,p) evaluates the boolean expression b. If b is *true,* control passes to the statement after the call. If b is *false,* the system calls p using the value of i as its only parameter. After p executes, the program continues provided the compiler option ASSERT_HALT is OFF. If it is ON, the program terminates.

The procedure heading for p must have the form:

PROCEDURE p (i: *integer*);

If the *assert* call omits parameter p, b is evaluated. If it is *false,* the system issues a run-time error message including the value of i. Execution continues if ASSERT_HALT is OFF. If it is ON, the program aborts.

ASSERT

The compiler option ASSERT_HALT determines the effect of the *assert* call on execution when b is *false*. Section 8 describes this option in full. Its default setting is OFF.

The programmer may use *assert* to test assumptions, specify invariant conditions, and check data structure integrity.

Assert is a Pascal/3000 extension of HP Standard Pascal.

Example

```
$ASSERT_HALT ON$
PROGRAM show_assert(input,output);
VAR
  n: integer;
PROCEDURE proc1 (i:integer);
  BEGIN
    write('Assert called this procedure ');
    writeln('and passed it the value ',i:3);
  END;
BEGIN
  write('Please enter an integer: ');
  prompt;
  readln(n);
  assert(n > 100, 99, proc1);
  writeln('The program didn''t abort!');
END.
```

Usage

baddress (v)

Argument

v A variable.

Description

The function *baddress* (v) returns the DB relative byte address of the variable specified by the parameter v. This variable may not be type file or a file type component of a structured variable. Also, it cannot be a component of a packed structure, except if it is a component of a PAC.

Baddress is useful for calling certain intrinsics which require byte addresses for parameters.

Baddress is a Pascal/3000 extension of HP Standard Pascal.

Baddress returns an integer in the range −32768.. 32767.

BADDRESS

Examples

```
TYPE
  rec_type = RECORD
               f1: integer;
               f2: boolean;
               f3: char;
             END;
VAR
  n: integer;
  r: rec_type;
  p: ^rec_type;
  a: ARRAY [1..10] OF 0..255;
  pac: PACKED ARRAY [1..10] OF char;
  pab: PACKED ARRAY [1..10] OF boolean;

                Calls
            ----------------
              baddress(n)
              baddress(r)
              baddress(r.f3)
              baddress(p)
              baddress(p^)
              baddress(p^.f3)
              baddress(a)
              baddress(a[4])
              baddress(pac)
              baddress(pac[2])    {Legal since component type }
                                  {is char.                  }
              baddress(pab)
              baddress(pab[2])    {Error.                    }
```

Usage

ccode

Description

The function *ccode* returns an integer value in the range 0..2. This number indicates the condition code resulting from a call to a procedure or function declared as INTRINSIC or EXTERNAL SPL according to the following scheme:

Number	Condition Code
0	CCG
1	CCL
2	CCE

The MPE Intrinsics Reference Manual gives the meaning of the condition code for each intrinsic.

The value returned by *ccode* is valid any time after return from a call to an intrinsic or external SPL routine and before either the next similar call or an exit from the procedure or function where the call occured. Furthermore, it is not possible to access the value from a procedure or function nested within the procedure or function where the call occured.

Ccode is a Pascal/3000 extension of HP Standard Pascal.

Example:

```
PROGRAM show_ccode;
LABEL 99;
PROCEDURE proc1; INTRINSIC;
BEGIN
   .
  proc1;            {intrinsic call}
  CASE ccode OF
    0: ;
    1: GOTO 99;
    2: BEGIN
         .
       END
  END{CLOSE}
END.
```

HALT

Usage

halt (n)
halt

Parameter:

n An integer expression. N may be omitted.

Description

The procedure *halt* (n) causes execution of a program to abort. The system displays the value of the integer expression n with an error message.

Halt calls the MPE intrinsic QUIT which discards the high order word for n. Thus, the value of n actually displayed will be in the range -32768..32767.

Example:

```
PROGRAM show_halt;
CONST
  div_by_0 = 99;
VAR
  x,y: real;
BEGIN
    .
    .
  IF x <> 0.0 THEN      {If x is 0 when IF executes, the program}
    y:= y/x             {terminates and an error message with   }
  ELSE                  {99 appears.                            }
    halt(div_by_0);
    .
    .
END.
```

Usage

sizeof (v)
sizeof (v,t1,...tn)

Arguments:

- v Any variable, except a file variable or a component of a packed structure.
- t A case constant when v is a record variable. Nested variants may appear separated by commas.

Description

The function *sizeof* (v) returns the number of bytes of storage required for v. If v is a record variable with variants, the programmer may select a variant by specifying a case constant with the t parameter. Otherwise, *sizeof* will return the size of the largest variant.

It is not legal for v to be a component of a packed structure, a file, or a file type component of a structured variable.

For a variable of a simple data type, the number returned by *sizeof* is equivalent to the storage required for the variable in the 'unpacked' context described in Section 9. For example, if v is type *char* or *boolean*, *sizeof* returns 1.

The programmer will find the *sizeof* function useful when calling intrinsics such as FWRITE or FREAD.

Sizeof is a Pascal/3000 extension of HP Standard Pascal.

Sizeof returns an integer in the range 0.. 32767.

SIZEOF

Examples

```
TYPE
  rec_type = RECORD
               f1: integer;
               CASE boolean OF
                 true: (v1: 0..10);
                 false: (v2: longreal);
             END;
VAR
  lr: longreal;
   b: boolean;
  ch: char;
  sr: 0..10;
   a: ARRAY [1..10] OF -32768..32767;
  pa: PACKED ARRAY [1..10] OF char;
   r: rec_type;
   p: ^rec_type;
  pr: PACKED RECORD
        f1: 0..10;
        f2: char;
      END;
```

Call	Return
sizeof(lr)	8 (bytes)
sizeof(b)	1
sizeof(ch)	1
sizeof(sr)	2
sizeof(a)	20
sizeof(a[3])	2
sizeof(pa)	10
sizeof(pa[3])	(error)
sizeof(r)	12
sizeof(r.f1)	4
sizeof(r,true)	6
sizeof(r,false)	12
sizeof(p)	2
sizeof(p^)	12
sizeof(p^.f1)	4
sizeof(p^,true)	6
sizeof(pr)	2
sizeof(pr.f1)	(error)

Usage

waddress (i)

Argument

i The name of a variable, procedure, or function.

Description

The function *waddress* (i) returns the DB relative word address of i when i is a variable name, and the external P label when it is a procedure or function name. An error occurs if the variable is type file or a file type component of a structured variable. Also, it is not legal to select a component of a packed structure as an argument, except when this component is an element of a PAC.

When referencing a component of an array which occupies an odd byte, *waddress* will return the address of the previous component since this component is on the word boundary (see example).

The programmer may use the *waddress* function when calling procedures in other languages such as FORTRAN or COBOL. Also, *waddress* is useful when arming the XLIBTRAP intrinsic (see Section 10).

Waddress is a Pascal/3000 extension of HP Standard Pascal.

Waddress returns an integer in the range −32768.. 32767.

WADDRESS

Examples

```
TYPE
  rec_type = RECORD
               f1: integer;
               f2: boolean;
             END;
VAR
  n   : integer;
  r   : rec_type;
  p   : ^rec_type;
  a   : ARRAY [1..10] OF integer;
  pac: PACKED ARRAY [1..10] OF char;
  pab: PACKED ARRAY [1..10] OF boolean;
PROCEDURE pro;
  BEGIN
  END;
FUNCTION f: integer;
  BEGIN
  END;
```

Calls

```
waddress(n)
waddress(r)
waddress(r.f2)
waddress(p)
waddress(p^)
waddress(p^.f2)
waddress(a)
waddress(a[4])     {Same as waddress(a[3]).          }
waddress(pac)
waddress(pac[3])   {Legal since component type is    }
                   {char.                            }
waddress(pab)
waddress(pab[3])   {Error.                           }
waddress(pro)
waddress(f)
```

COMPILER OPTIONS

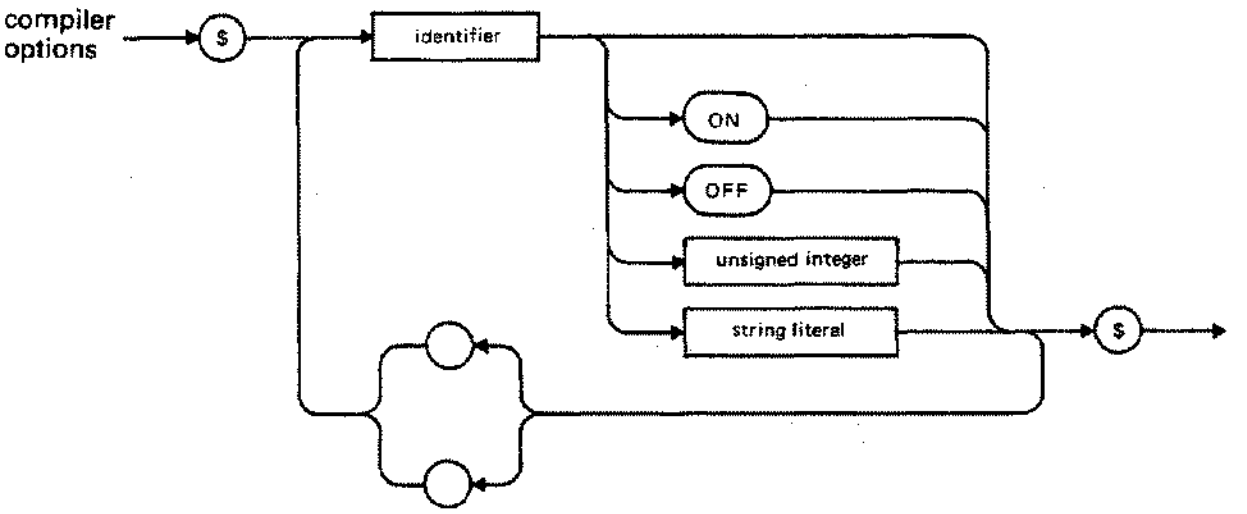
SECTION VIII

INTRODUCTION

Compiler options direct the action of the Pascal/3000 compiler as it processes source code.

Dollar signs ($) bracket a compiler option or series of options. The option name may be followed by the words ON or OFF, an unsigned integer, or a string literal. Commas (,) or semi-colons (;) must separate several options appearing within one pair of dollar signs.

Syntax



The programmer may write the option name or the words ON or OFF in any combination of upper and lower case letters. A string literal may also use upper and lower case indifferently, except in the case of the options TITLE and COPYRIGHT. For example, the options

```
$ANSI ON, INCLUDE 'MYFILE'$
$ansi on; include 'myfile'$
$AnSI oN, INcluDE 'MyFile'$
```

are equivalent.

INTRODUCTION

The programmer must place certain compiler options at particular locations in source code. For example, the option GLOBAL must precede the program heading. Others, such as TITLE, may occur anywhere.

Many options have default settings which remain in effect until the programmer explicitly overrides them. For example, the option LIST is ON by default. This means the compiler always produces a listing of the program it is processing unless the programmer writes $LIST OFF$ somewhere in the source code.

The Pascal/3000 compiler performs three major steps:

(1) It scans source code to produce tokens.

(2) It parses these tokens into intermediate data structures (abstract-syntax trees).

(3) Finally, it generates HP3000 object code from these structures.

The compiler scans, parses, and emits object code for one 'compilation block' at a time in source code. A compilation block is a procedure or function from any level, or the outer block of the source program. A compilation block should be distinguished from a Pascal block, which is a syntactical unit of source code.

Some compiler options are only meaningful for entire compilation blocks. For example, the TABLES option produces an identifier map of an entire compilation block if it is set ON when the compiler finishes parsing the block and is ready to emit object code. In other words, it is not possible to generate a map for part of a procedure, function, or outer block.

Table 8-1 summarizes the various options and the actions they perform.

Table 8-1. COMPILER OPTIONS

Option	Action	Default Setting
ALIAS	Substitutes an alias as external name for a procedure of function.	none
ANSI	Causes compiler to issue a warning when a non-Ansi Pascal feature appears in source code.	OFF
ASSERT_HALT	Causes execution to halt when assert is called and the boolean expression is false.	OFF
CHECK_ACTUAL_ PARM	Specifies level of checking for actual parameters of procedure or function call.	3
CHECK_FORMAL_ PARM	Specifies level of checking for formal parameters of procedure or function.	3
CODE	Causes code to be generated after parsing of a procedure, function, or outer block.	ON
CODE_OFFSETS	Displays a table showing the number of a statement in the listing and its offset from the starting p register.	OFF

Table 8-1. COMPILER OPTIONS ((continued)

COPYRIGHT	Inserts a copyright notice and specified name in the USL and program files.	none
ENDIF	Delimits previous use of a $IF.	
ELSE	Allows you to select alternate code to be compiled if the previous IF expression was false.	
EXTERNAL	Used in conjunction with the GLOBAL option to permit separate compilation of procedures.	PASCAL
GLOBAL	Used in conjunction with the EXTERNAL option to permit separate compilation of procedures.	PASCAL
HEAP__COMPACT	Causes free-space areas in the heap to be combined.	OFF
HEAP__DISPOSE	Permits disposed areas in heap to be reallocated.	OFF
IF	Conditionally compiles blocks of source code. Its identifiers must have been defined by the SET option.	
INCLUDE	Allows specified file to be compiled with source text.	none
LINES	Sets the number of listing lines per page.	59
LIST	Produces listing of source code as it is compiled.	ON

Table 8-1. COMPILER OPTIONS (continued)

LIST__CODE	Produces mnemonic listing of generated object code.	OFF
PAGE	Causes the output listing to start a new page.	none
PARTIAL__EVAL	Permits partial evaluation of boolean expressions.	ON
PRIVATE__PROC	Allows use of normal Pascal scope conventions for names of non-level 1 procedures or functions.	ON
RANGE	Emits range checking code for assignments, array indexing, pointers, and set operations.	ON
SEGMENT	Changes current segment name to specified name.	SEG'
SET	Defines Boolean variables and assigns a Boolean value to the variable for use in a $IF expression.	
SYMDEBUG	Produces symbolic information headers in the USL that is used by the HPToolset product for symbolic debugging of user programs.	
SKIP__TEXT	Causes compiler to skip source code.	OFF
SPLINTR	Specifies name of SPL intrinsic file to be searched when a function or procedure is declared INTRINSIC.	SPLINTR. PUB. SYS

Table 8-1. COMPILER OPTIONS (continued)

STANDARD_LEVEL	Specifies Pascal level which will be compiled	HP
SUBPROGRAM	Permits compilation of a subset of level 1 procedures or functions.	all level 1 procedures, functions
TABLES	Produces an identifier map for a procedure, function or outer block.	OFF
TITLE	Places specified string literal as title on each listing page.	(see below)
USLINIT	Initializes USL file to empty.	none
WIDTH	Sets number of columns compiler will process from each record of source code.	132 characters
XREF	Produces cross reference of a procedure, function, or the outer block.	OFF

Usage

$ALIAS s$

Parameter

s A string literal.

Description

The option ALIAS specifies an external name, s, for a procedure or function. ALIAS must appear in the procedure or function heading after the reserved words PROCEDURE or FUNCTION, and before the body or directive following the heading.

The programmer may use ALIAS to define multiple internal names of an intrinsic, to interface with a library routine containing a single quote (') in its name, or to differentiate internal names which would not be unique in the first 15 characters when they become external names. This last possibility arises, for example, if the option PRIVATE_PROC is OFF and different but synonymous non-level 1 procedures nested in distinct level 1 procedures are compiled into the same USL file (see example below).

ALIAS

Example

```
PROGRAM show_alias;
  .
  .
PROCEDURE A $ALIAS 'intrinname'$; INTRINSIC;   {The intrinsic  }
PROCEDURE B $ALIAS 'intrinname'$; INTRINSIC;   {now has two    }
  .                                            {internal names.}
  .
PROCEDURE xx $ALIAS 'x''x'$; INTRINSIC;  {The intrinsic has a  }
  .                                      {single quote in its  }
  .                                      {name.                }
$PRIVATE_PROC OFF$
PROCEDURE proc1;
  FUNCTION doit (n: integer)    : boolean; $ALIAS 'D1'$
    BEGIN
      .
      .
    END;  {doit}
  BEGIN
    .
    .
  END;  {proc1}
PROCEDURE proc2;
  FUNCTION doit (a,b: integer) : integer; $ALIAS 'D2'$
    BEGIN
      .
      .
    END; {doit}
  BEGIN
    .
  END; {proc2}

BEGIN {show_alias}
  .
END.
```

The compiler processes proc1 and proc2 with PRIVATE_PROC OFF. This means the two non level 1 functions doit, which are distinct but homonymous, will be compiled as separate RBM's, and only one will be active. To avoid this impasse, ALIAS gives the functions the external names D1 and D2. The resulting USL looks like this:

Usage

```
$FONT ' unsigned integer ' unsigned integer '$
```

Parameter

unsigned integer	A number for the Primary and Secondary character sets.

Description

The $FONT option allows you to set primary and secondary character fonts for source listings printed on a 2680 printer. $FONT uses a string parameter consisting of two unsigned integers separated by a comma. The first integer is the set number for the primary font, and the second integer is the set number for the secondary font. The primary font is the default, and can be changed at any time by entering the $FONT option again.

You can shift from the primary font into the secondary font by entering a control N character within a string within the compiler option, or within a comment in your program. To change back to the primary font, enter a control O character.

Currently this option can only be used with the 2680 page printer. See the intrinsic description of FDeviceControl for further information.

$SET

Usage

```
                          TRUE                                  TRUE
$SET 'IDENTIFIER ={                } [ ,  IDENTIFIER ={                }. . .] '$
                          FALSE                                 FALSE
```

Description

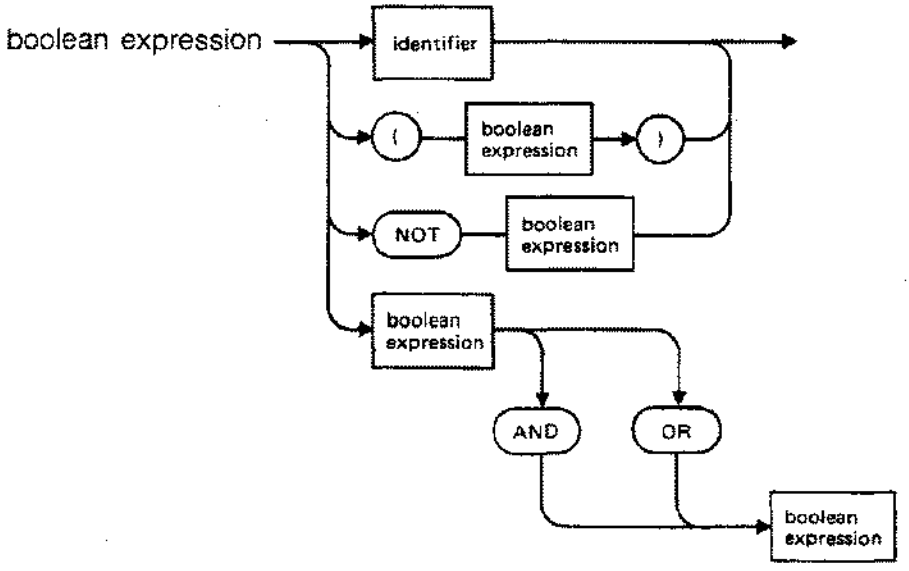
The $SET compiler option enables you to define allowed BOOLEAN variables and assign a BOOLEAN value to the variable. The value assigned by the $SET option is used by the $IF compiler option. The $SET option can use any legitimate Pascal identifier. Note that the identifiers defined by the $SET option are known only to the compiler and not the program. Therefore, for example, the variable 'X' can be used in your source program and can be defined in the $SET compiler option as well. A value of True or False must be assigned to the variable you define. You may define and assign a value to more than one variable by separating each by a comma.

The $SET option is the only way to give an identifier used in the condition of a $IF option a value. For an example of the $SET option, see the $IF compiler option in this section.

Usage

$IF 'boolean Expression'$ [Source code] [$Else] [Source code] $ENDIF$

where boolean expression is



Parameter

Identifier - A Boolean identifier that was defined by the $SET compiler option. All Boolean identifiers used in a Boolean expression are defined in the $SET option.

$Else - Optional compiler option that specified code to compile if the expression was False. There can only be one $ELSE for each $IF.

$ENDIF - Required delimiter for $IF statement.

$IF

Description

The $IF compiler option is used to conditionally compile blocks of source code. This option requires a string parameter representing a Boolean expression that is computed using identifiers defined and assigned a value in the $SET compiler option.

The Boolean operators NOT, AND, and OR are allowed in the $IF expression.

The $IF compiler option works similarly to the Pascal If statement. In its simplest form using just $IF and $ENDIF, the compiler evaluates the string expression, and if the value of the BOOLEAN expression is true, the code between the $IF and $ENDIF is compiled. If the value is false, the compiler skips the code between the $IF and $ENDIF and starts the compilation on the line following the $ENDIF.

$ELSE parallels the Pascal else condition. If $ELSE is used, the source following it is compiled if the previous $IF Boolean expression had a value of False. Only one $ELSE can be used for each $IF. $ENDIF serves as the delimiter for $ELSE as well as for its associated $IF.

$IF can be nested for up to 16 levels. If there are more than 16 levels, you receive an error message and the code within the illegal $IF block is compiled.

Example

```
$SET 'X=TRUE, Y=FALSE' $     {must be declared before Program Header}
  $IF 'X' $   {A}
     .
     .        {code compiled}
     .
  $ENDIF$     {A}

  $IF 'X AND Y' $  {B}
     .
     .             {code not compiled}
     .

  $ELSE$
       $IF 'X' $   {C}
            .
            .      {code compiled}
            .
       $ENDIF$     {C}

       $IF 'Y' $   {D}
            .
            .      {code not compiled}
            .

       $ENDIF$     {D}

  $ENDIF$          {B}
```

$SYMDEBUG

Allows you to symbolically debug your program with the HPToolset utility.

Usage

$SYMDEBUG$

Description

The $SYMDEBUG compiler option allows you to symbolically debug your Pascal program with the HPToolset utility. When this option is specified, the compiler puts symbolic information into the USL file to be used by TOOLSET when you use its Symbolic Debugging feature. The $SYMDEBUG option must appear in your source file before the declaration statements. If you do not enter the option, no symbolic information is passed to the USL file and your program cannot be debugged symbolically.

```
USL FILE <filename>

SEG'
   OB'                35  OB A C N
   PROC2               1  P  A C N R
   D2                  4  P  A C N R
   PROC1               1  P  A C N R
   D1                  3  P  A C N R

FILE SIZE      377600( 1777.  0)
DIR. USED         337(    1.137)   INFO USED        116(    0.116)
DIR. GARB.          0(    0.  0)   INFO GARB.         0(    0.  0)
DIR. AVAIL.     37241(  175. 41)   INFO AVAIL.   337662( 1577. 62)
```

ANSI

Usage

$ANSI ON$
$ANSI OFF$

Default Setting

OFF

Description

When the ANSI option is ON, the compiler issues a warning whenever it encounters a feature in source code that is not legal in ANSI Standard Pascal. The warning appears as part of the listing.

The option $ANSI ON$ is equivalent to the option $STANDARD_LEVEL ANSI$ (see below).

Example

```
$ANSI ON$
$ASSERT_HALT ON$
PROGRAM show_ansi (input,output);
TYPE
  a = ARRAY [1..10] OF integer;
CONST
  count = a[1,2,3,4,5,6,7,8,9,10];
VAR
  i: integer;
BEGIN
  read (i);
  assert (i = count[i],99);
  writeln('Your number is acceptable');
END.
```

This source code produces the following listing:

```
PAGE 1 <Listing title>


    1.000       0  0   $ANSI ON$
    2.000       0  0   $ASSERT_HALT ON$
    3.000       0  0   PROGRAM show_ansi (input,output);
                                  ^

****  WARNING #  1 THIS FEATURE IS HP STANDARD PASCAL (517)
    4.000       0  0   TYPE
    5.000       0  0     a = ARRAY [1..10] OF integer;
    6.000       0  0   CONST
    7.000       0  0     count = a[1,2,3,4,5,6,7,8,9,10];
                           ^

****  WARNING #  2 THIS FEATURE IS HP STANDARD PASCAL (517)
                                                 {For CONST after}
                           ^                     {TYPE section.   }
****  WARNING #  3 THIS FEATURE IS HP STANDARD PASCAL (517)
                                                 {For structured }
    8.000       0  0   VAR                       {constant.      }
    9.000       0  0     i: integer;
   10.000       0  1   BEGIN
   11.000       0  1     read (i);
   12.000       1  1     assert (i = count[i],99);
                             ^

****  WARNING #  4 THIS FEATURE IS HP3000 PASCAL (518)
   13.000       2  1     writeln ('Your number is acceptable');
   14.000       2  1   END.

              NUMBER OF ERRORS =  0     NUMBER OF WARNINGS =  4
```

ASSERT_HALT

Usage

$ASSERT_HALT ON$
$ASSERT_HALT OFF$

Default Setting

OFF

Description

When the ASSERT_HALT option is ON, a program terminates if the boolean parameter of an *assert* call evaluates *false* (see Section 7). If a procedure parameter p appears in the *assert* call, the program halts after p has executed.

If ASSERT_HALT is OFF, the program will not terminate, regardless of the value of the boolean parameter in the *assert* call.

ASSERT_HALT may appear anywhere in source code.

Example

```
$ASSERT_HALT ON$
PROGRAM show_asserthalt (input,output);
VAR
  i: integer;
BEGIN
  write('Please enter an integer: ');
  prompt;
  read(i);
  assert(i<10,99);
  writeln('Good show! You didn''t abort the program.');
END.
```

CHECK_ACTUAL_PARM

Usage

$CHECK_ACTUAL_PARM n$

Parameter

n An integer in the range 0..3.

Default Setting

3

Description

The CHECK_ACTUAL_PARM option specifies the level of checking the MPE Segmenter will perform when a program calls a procedure or function. The level specified, n, determines the amount of information placed in the USL file. The Segmenter uses this information to check the actual parameters against the formal parameters of the function or procedure. The levels are:

0 - No checking.
1 - Check function type.
2 - Check function type and the number of procedure or function parameters.
3 - Check function type, the number of procedure or function parameters, and the type of each parameter.

Level 3 is the default setting.

If the procedure or function has a lower checking level, the Segmenter ignores the level indicated by CHECK_ACTUAL_PARM and uses the lower level.

The compiler generates no parameter checking information for procedures or functions declared INTRINSIC. When a language specification appears with the EXTERNAL directive (see example), the checking code will be compatible with the external language.

CHECK_ACTUAL_PARM may appear anywhere in source code.

CHECK_ACTUAL_PARM

Example

```
PROGRAM show_actparmcheck;
TYPE
  a = PACKED ARRAY [1..32] OF boolean;
VAR
  v : a;
PROCEDURE fortproc(VAR p : a); EXTERNAL FORTRAN;
BEGIN
  .
  $CHECK_ACTUAL_PARM 0$
  fortproc(v);
  .
END.
```

CHECK_FORMAL_PARM

Usage

$CHECK_FORMAL_PARM n$

Parameter

n An integer in the range 0..3.

Default Setting

3

Description

The CHECK_FORMAL_PARM option specifies the level of checking the MPE Segmenter will perform when a procedure or function is called. The level specified, n, determines the amount of information placed in the USL file. The Segmenter uses this information to check the formal parameters of the declared procedure or function against the actual parameters in the calling program, procedure, or function. The possible levels are:

0 - No checking.
1 - Check function type.
2 - Check function type and the number of procedure or function parameters.
3 - Check function type, the number of procedure or function parameters, and the type of each parameter.

Level 3 is the default setting.

If the checking level of the procedure or function call is lower, the Segmenter ignores the checking level specified by CHECK_FORMAL_PARM and uses the lower value.

CHECK_FORMAL_PARM may appear anywhere in source code.

CHECK_FORMAL_PARM

Example

```
PROGRAM show_chkformparm;

$CHECK_FORMAL_PARM 1$
PROCEDURE proc1;
  BEGIN
    .
  END;

$CHECK_FORMAL_PARM 3$              {Restores default setting  }
FUNCTION func1: integer;
  BEGIN
    .
  END;
BEGIN
  .
END.
```

Usage

$CODE ON$
$CODE OFF$

Default Setting

ON

Description

If the CODE option is ON, the compiler generates object code when it finishes parsing a compilation block. CODE may appear anywhere in source code, but it only affects the object code for an entire procedure, function, or outer block. To suppress code emission for smaller portions of source, the programmer may use the SKIP_TEXT option or the Pascal comment symbols.

Example

```
PROGRAM show_code;
PROCEDURE proc1;
  BEGIN
    .
    .
  END;
PROCEDURE proc2;
  BEGIN
    .
    .
  $CODE OFF$              {Compiler generates no object code for}
    .                     {any part of proc2, even though CODE  }
    .                     {OFF is in the middle of proc2.       }
  END;  {proc2}
$CODE ON$
BEGIN  {show_code}
  .
  .
END.
```

CODE_OFFSETS

Usage

$CODE_OFFSETS ON$
$CODE_OFFSETS OFF$

Default Setting

OFF

Description

The option CODE_OFFSETS causes the compiler to list the number of each executable statement in a compilation block, starting from 0, and its p register offset in octal from the starting value of p for that block. The information appears as part of the listing. If no code is generated for a particular statement, '******' appears instead of a p register offset.

If the option PRIVATE_PROC is OFF, the p register is reset for for each compilation block. When PRIVATE_PROC is ON (the default setting), the p register offset accumulates as the the compiler encounters executable statements from any nested compilation blocks, e.g. level 2 procedures.

The compiler inserts certain information whenever the p register is reset. In particular, it uses 11 words to record the version of the compiler, the date, and the time.

CODE_OFFSETS has no effect if the LIST option is OFF.

CODE_OFFSETS may occur anywhere in source code but it only acts on an entire compilation block. In other words, it is not possible to list statement offsets for part of a procedure, function, or outer block.

The programmer may use CODE_OFFSETS in conjunction with the TABLES option, a PMAP from the MPE Segmenter, and the MPE Debug utility to determine break points in a program (see Section 10).

Example

```
$PRIVATE_PROC OFF$
$CODE_OFFSETS ON$
PROGRAM show_offsets (output);
PROCEDURE proc1;
  PROCEDURE subproc1;
    BEGIN
      writeln('This is subproc1');
      writeln;
      writeln
    END;
  BEGIN
    writeln('This is proc1');
    subproc1
  END;
BEGIN
  writeln('This is the main program');
  proc1
END.
```

This source code results in the following listing:

CODE_OFFSETS

```
PAGE 1 <Listing title>

  1.000     0  0     $PRIVATE_PROC OFF$
  2.000     0  0     $CODE_OFFSETS ON$
  3.000     0  0     PROGRAM show_offsets (output);
  4.000     0  0     PROCEDURE proc1;
  5.000     0  0       PROCEDURE subproc1;
  6.000     0  1         BEGIN
  7.000     0  1           writeln('This is subproc1');
  8.000     1  1           writeln;
  9.000     2  1           writeln
 10.000     2  1         END;

                          C O D E   O F F S E T S

                    STMT  P LOC     STMT  P LOC     STMT  P LOC
                      0  000013       1  000031       2  000033

 11.000     0  1       BEGIN
 12.000     0  1         writeln('This is proc1');
 13.000     1  1         subproc1
 14.000     1  1       END;

                          C O D E   O F F S E T S

                    STMT  P LOC     STMT  P LOC
                      0  000013       1  000031

 15.000     0  1     BEGIN
 16.000     0  1       writeln('This is the main program');
 17.000     1  1       proc1;
 18.000     2  1       writeln
 19.000     2  1     END.

                          C O D E   O F F S E T S

                    STMT  P LOC     STMT  P LOC     STMT  P LOC
                      0  000035       1  000053       2  000054
```

COPYRIGHT

Usage

$COPYRIGHT s$

Parameter

s A string literal.

Description

The COPYRIGHT option places a copyright notice in the USL file. The notice will also appear in the program file. The s parameter specifies a name which will be part of the notice.

COPYRIGHT may only appear before a program heading.

The text of the notice is:

> (C) Copyright <current year> by <s>. All rights reserved. No part of this program may be photocopied, reproduced, or transmitted without prior written consent of <s>.

The compiler respects distinctions between upper and lower case letters in the s parameter.

Example

```
$COPYRIGHT 'Blaise Pascal'$
PROGRAM show_copyright;
BEGIN
  writeln('Got any dice?')
END.
```

EXTERNAL

Usage

$EXTERNAL$
$EXTERNAL 'PASCAL'$
$EXTERNAL 'SPL'$
$EXTERNAL 'NONE'$

Default setting

PASCAL

Description

The option EXTERNAL, used in conjunction with the option GLOBAL, permits the separate compilation of procedures and functions. When EXTERNAL appears in source code, the compiler generates information about the variables declared in the outer block that will allow them to be matched up with variables of the same name and type in an outer block compiled with the GLOBAL option. The compiler doesn't generate object code for the statement part of the outer block, only for the procedures and functions.

The optional string parameter determines the type of checking information the compiler places in the USL file. 'PASCAL' is used to match a Pascal outer block compiled with $GLOBAL$ or $GLOBAL 'PASCAL'$; 'SPL' to match an SPL outer block or a Pascal outer block compiled with $GLOBAL 'SPL'$; 'NONE' to relax all type checking. $EXTERNAL$ is equivalent to $EXTERNAL 'PASCAL'$.

Because of requirements of the MPE Segmenter, global variables in a program compiled with the EXTERNAL option must be unique within 15 characters.

EXTERNAL must appear before the program heading. The body of the outer block should be empty, i.e. there should be no statements between BEGIN and END. EXTERNAL and GLOBAL may not appear in the same source.

An outer block compiled with the GLOBAL option may declare several variables. The outer block in the code compiled with EXTERNAL need not mention all of these. Only the variables referenced in its procedures or functions must appear.

Example

See the GLOBAL example below.

Usage

```
$GLOBAL$
$GLOBAL 'PASCAL'$
$GLOBAL 'SPL'$
$GLOBAL 'NONE'$
```

Description

The option GLOBAL, used in conjunction with the option EXTERNAL, permits separate compilation of procedures and functions. When GLOBAL is specified, the compiler prepares information about the variables declared in the outer block which will allow them to be matched with variables of the same name and type used in a procedure or function compiled with EXTERNAL.

The optional string parameter determines the type of checking information the compiler will place in the USL file. 'PASCAL' is used to match a Pascal procedure of function compiler with $EXTERNAL$ or $EXTERNAL 'PASCAL'$; 'SPL' to match a SPL routine, or a Pascal procedure or function compiled with $EXTERNAL 'SPL'$; 'NONE' to relax all checking. $GLOBAL$ is equivalent to $GLOBAL 'PASCAL'$.

The compiler processes all of the GLOBAL source code and emits object code for the outer block as well as all the functions and procedures.

Because of requirements of the MPE Segmenter, global variables in a program compiled with the GLOBAL option must be unique within 15 characters.

GLOBAL must appear before the program heading. GLOBAL and EXTERNAL may not occur in the same source.

Source code compiled with GLOBAL and source code compiled with EXTERNAL are placed in the same USL file. At prep time, the MPE Segmenter is able to determine the addresses of the global variables used in the code compiled with EXTERNAL.

GLOBAL

Example

```
$GLOBAL$
PROGRAM show_global (input,output);
VAR
  a,b,c,d: integer;
  state: boolean;
PROCEDURE proc1; EXTERNAL;
BEGIN
  .
  .
  IF a > b THEN state:= true;
  proc1;
  .
  .
END.


$EXTERNAL$
PROGRAM show_external (input, output);
VAR
 state: boolean;    {This will be matched with the variable      }
PROCEDURE proc1;    {declared in the outer block of show_global.}
  BEGIN
   .
  IF state THEN...  {Reference to variable declared in outer    }
   .                {block of show_global.                       }
   .
  END;
BEGIN               {The body of this outer block is empty.     }
END.
```

HEAP__COMPACT

Usage

$HEAP__COMPACT ON$
$HEAP__COMPACT OFF$

Default Setting

OFF

Description

The option HEAP__COMPACT works in conjunction with the option HEAP__DISPOSE to permit the concatenation of free space in the heap. HEAP__COMPACT has no effect if HEAP__DISPOSE is OFF.

HEAP__COMPACT must appear before the program heading.

HEAP__COMPACT is useful when a program manipulates many dynamic record variables of various sizes (see example below).

HEAP__COMPACT takes effect when specified in the main program.

Example

```
$HEAP_COMPACT ON; HEAP_DISPOSE ON$
PROGRAM show_compact;
TYPE
  name = PACKED ARRAY [1..25] OF char;
  big_rec = RECORD
              f1: ARRAY [1..100] OF name;
              f2: FILE OF integer;
            END;
  small_rec = PACKED RECORD
                f1: (Ives, Carter, Thompson, Copeland);
                f2: boolean;
              END;
VAR
  p1: ^big_rec;
  p2: ^small_rec;
BEGIN
  .
  .
END.
```

HEAP__DISPOSE

Usage

$HEAP__DISPOSE ON$
$HEAP__DISPOSE OFF$

Default Setting

OFF

Description

When the option HEAP__DISPOSE is ON, a call to *dispose* (see Section 6) creates free space in the heap. A subsequent call to *new* can then reuse this storage. If HEAP__DISPOSE is OFF, on the other hand, the system will not reallocate the disposed storage.

HEAP__DISPOSE must be ON in order for the option HEAP__COMPACT (see above) to have any meaning.

HEAP__DISPOSE takes effect when specified in the main program.

Example

```
$HEAP_DISPOSE ON$
PROGRAM show_heap;
TYPE
  big_array = ARRAY [1..1000] OF longreal;
VAR
  ptr: ^big_array;
  i,j: integer;
BEGIN
  FOR i:= 1 TO 500 DO          {If HEAP_DISPOSE is OFF, an error }
    BEGIN                      {results when the heap overflows. }
      new(ptr);
      FOR j:=1 TO 1000 DO
        ptr^[j]:= j;
      dispose(ptr);
    END;
END.
```

Usage

$INCLUDE s$

Parameter

s A string literal.

Description

The INCLUDE option permits inclusion of another file which the compiler will process as source code. The parameter s represents the name of the included file, which may be fully qualified by group and account names, and a lockword. Upper and lower case letters are equivalent in s. The compiler reads the designated file until it encounters an EOF marker. Then it resumes processing from the source line after the INCLUDE option. This means the compiler ignores any options listed immediately after INCLUDE or any subsequent source code on the same line as INCLUDE.

INCLUDE may appear anywhere in source code.

INCLUDE options may be nested. That is, the included code may itself contain INCLUDE options.

Example

```
PROGRAM show_include;
VAR
  $INCLUDE 'globvars'$          {GLOBVARS file is:     }
BEGIN                           {                      }
  i:= 3;                        {  i: integer;         }
  x:= 1.55;                     {  x: real;            }
    .
    .
END.
```

LINES

Usage

$LINES n$

Parameter

n An integer not less than 20.

Default Setting

59

Description

The option LINES specifies the number (n) of lines that will appear on a single page of the listing. The parameter n may not be less than 20.

LINES may appear anywhere in a source program.

Example

```
$LINES 20$
PROGRAM show_lines;
  .
  .
BEGIN
  writeln('The listing has 20 lines per page.');
  .
  .
END.
```

Usage

$LIST ON$
$LIST OFF$

Default Setting

ON

Description

When the option LIST is ON, the compiler produces a listing of the source code it is processing. LIST may appear anywhere in source code.

The first column of the listing displays the editor line number of the source code. If the source file is unnumbered, the compiler supplies a sequence of numbers starting with 1 in increments of 1. The second column shows the number associated with a Pascal statement in the code location table. If a '***' appears in the second column, then the line is within a Pascal comment or the SKIP_TEXT option. The third column exhibits the BEGIN-END level number in each procedure.

When compilation is complete, the system displays information about the number of errors and warnings. It also indicates the processor time, elapsed time, number of lines compiled, and the number of lines processed per minute (see example below). These times and rates depend on the actual processor and the version of the MPE Operating System in use. Unless it is relevant to the example, this information does not appear with sample listings elsewhere in this manual.

If the programmer is entering source code interactively and the listing file is also the terminal ($STDLIST), then the LIST option does not redisplay the source code on the screen. Any error messages, however, will appear.

When LIST is ON, the programmer may invoke other options which produce extra information or control the listing. These options are ANSI, CODE_OFFSETS, LINES, LIST_CODE, PAGE, STANDARD_LEVEL, TABLES, TITLE, and XREF. If LIST is OFF, setting any of these options ON has no effect until LIST is turned ON.

LIST

If a warning or error occurs during compiliation, a message appears on the listing with the following format:

**** WARNING n <message> or **** ERROR n <message>

with, in most cases, a caret (^) above pointing to the feature or problem. N is an integer which indicates the error or warning is the n'th error or warning in the current compilation. If the error message catalog for the compiler is not available, or if the error or warning occurs when the compiler's stack is very large, e.g. in a level 4 procedure, the message consists of the Pascal error number only. Appendix C lists the compile-time errors by number.

Errors and warnings on listings of more than one page are 'chained'. That is, the first error or warning on a page will include a message indicating the page where the last previous error or warning occurred. This message also appears on the last page when it doesn't have an error or warning.

Example

```
$LIST ON$      {default setting}
PROGRAM show_list (input,output);
{Shows typical listing. This comment
 spans across
 three lines.}
VAR
  a,b: integer;
PROCEDURE check (VAR n: integer);
  EXTERNAL FORTRAN;
BEGIN
  read(a,b);
  IF a > b THEN
    BEGIN
      c:= a + b;   {An intentional error.}
      WHILE a <> b DO
        BEGIN
          a:= a - 1;
          writeln(a);
        END;
    END
  ELSE check (a);
END.
```

```
PAGE   1  <Listing title>


    1.000       0  0   $LIST ON$      {default setting}
    2.000       0  0   PROGRAM show_list (input,output);
    3.000       0  0   {Shows typical listing. This comment
    4.000      **  0    spans across
    5.000       0  0    three lines.}
    6.000       0  0   VAR
    7.000       0  0     a,b: integer;
    8.000       0  0   PROCEDURE check (VAR n: integer);
    9.000       0  0     EXTERNAL FORTRAN;
                                    ^
**** WARNING #  1       THIS FEATURE IS HP3000 PASCAL (518)
   10.000       0  1   BEGIN
   11.000       0  1     read(a,b);
   12.000       1  1     IF a > b THEN
   13.000       2  2       BEGIN
   14.000       2  2         c:= a + b;    {An intentional error.}
                             ^
**** ERROR #  1    IDENTIFIER NOT DEFINED (014)
   15.000       3  2         WHILE a <> b DO
   16.000       4  3           BEGIN
   17.000       4  3             a:= a - 1;
   18.000       5  3             writeln(a);
   19.000       5  3           END;
   20.000       5  2       END
   21.000       6  2     ELSE check (a);
   22.000       6  1   END.

                NUMBER OF ERRORS =  1      NUMBER OF WARNINGS =  1
                PROCESSOR TIME 0: 0: 2     ELAPSED TIME 0: 0:12
                NUMBER OF LINES =   22     LINES/MINUTE = 660.0
```

LIST_CODE

Usage

$LIST_CODE ON$
$LIST_CODE OFF$

Default Setting

OFF

Description

If the option LIST_CODE is ON, the compiler produces a HP 3000 mnemonic listing of the object code it generates for a compilation block. LIST_CODE has no effect if the LIST option is OFF.

LIST_CODE may appear anywhere in source code, but it affects only an entire procedure, function, or outer block. It is not possible to list object code for part of a compilation block.

The first column of the object code listing indicates the P location offset from the beginning of the procedure, function, or outer block; the second column the object code in octal; the third column the code in ASCII with non-printable characacters displayed as periods (.); and the fourth column the mnemonic for the instruction.

PCAL and LLBL instructions include the name of the procedure or function called. Also, if the EXTERNAL option is used, the first 15 characters of the name of a global variable appear instead of the DB offset.

The programmer will usually use the option CODE_OFFSETS in combination with LIST_CODE.

Example

```
$LIST_CODE ON$
$CODE_OFFSETS ON$
PROGRAM show_listcode;
PROCEDURE proc1;
  VAR
    m,n: integer;
  BEGIN
    m:= 1;
    n:= 9;
    m:= m + n;
  END;
BEGIN
  proc1;
END.
```

This source code produces the following listing:

```
  1.000     0  0   $LIST_CODE ON$
  2.000     0  0   $CODE_OFFSETS ON$
  3.000     0  0   PROGRAM show_listcode;
  4.000     0  0   PROCEDURE proc1;
  5.000     0  0     VAR
  6.000     0  0       m,n: integer;
  7.000     0  1     BEGIN
  8.000     0  1       m:= 1;
  9.000     1  1       n:= 9;
 10.000     2  1       m:= m + n;
 11.000     2  1     END;

                                  C O D E   O F F S E T S

                STMT  P LOC      STMT  P LOC      STMT  P LOC
                  0  000014        1  000016        2  000021

                                  C O D E   L I S T I N G

                P OFFSET   DATA   ASCII INSTRUCTION            P OFFSET   DATA   ASCII INSTRUCTION
                 000013   000707   ..    DZRO,DZRO     <---    000021   151403   ..    LDD   Q+3
                 000014   000733   ..    DZRO,INCA             000022   151401   ..    LDD   Q+1
                 000015   151403   ..    STD   Q+3             000023   001100   .@    DADD,NOP
                 000016   000600   ..    ZERO,NOP              000024   161403   ..    STD   Q+3
                 000017   021011   *.    LDI   11              000025   031400   3.    EXIT 0
                 000020   161401   ..    STD   Q+1

 12.000     0  1   BEGIN
 13.000     0  1     proc1;
 14.000     0  1   END.

                                  C O D E   O F F S E T S

                STMT  P LOC
                  0  000017

                                  C O D E   L I S T I N G

                P OFFSET   DATA   ASCII INSTRUCTION            P OFFSET   DATA   ASCII INSTRUCTION
                 000013   040006   @.    LDRD P+6      <---    000017   000000   ..    NOP, NOP
                 000014   025013   *.    LDNI 13               000020   000000   ..    PCAL  TERMINATE'
                 000015   000500   ..    ZERO,NOP              000021   001000   ..    DCMP,NOP
                 000016   000000   ..    PCAL  P'INITHEAP'3000

                NUMBER OF ERRORS =  0      NUMBER OF WARNINGS =  0
                PROCESSOR TIME 0: 0: 3     ELAPSED TIME 0: 0:28
                NUMBER OF LINES =  14      LINES/MINUTE = 280.0
```

PAGE

Usage

$PAGE$

Description

The PAGE option causes the compiler listing to a line printer to peform a page eject and start a new page.

PAGE may appear anywhere in source code.

Example

```
PROGRAM show_page (output);
BEGIN
  writeln('This appears on the 1st page of the listing');
  $PAGE$
  writeln('This appears on the second');
END.
```

PARTIAL_EVAL

Usage

$PARTIAL_EVAL ON$
$PARTIAL_EVAL OFF$

Default Setting

ON

Description

When the PARTIAL_EVAL option is ON, the compiler processes source code so that the system will determine the value of a boolean expression by evaluating the minimum number of operands. If PARTIAL_EVAL is OFF, on the other hand, the system evaluates all the operands in a boolean expression at run-time.

Partial evaluation usually permits more readable source code and results in more efficient object code. With PARTIAL_EVAL OFF, for example, the programmer may have to write a series of nested IF statements to prevent run-time errors:

```
IF index IN [lower..upper] THEN
  IF ptr_array [index] <> NIL THEN
    IF ptr_array [index] ^ = 5 THEN
      found_it := true;
```

If index is out of range, then the reference to ptr_array[index] fails.
If index is valid, but ptr_array[index] is NIL, then ptr_array[index] ^ fails.

With PARTIAL_EVAL turned ON, however, the programmer may rewrite this code as follows:

```
found_it:= (index IN [lower..upper] ) AND (ptr_array [index] <>NIL)
           AND (ptr_array [index] ^= 5);
```

Evaluation of the boolean expression stops when the result is known. Thus, if index is invalid, the system never evaluates the the expression (ptr_array [index] <> NIL), preventing a range violation. Likewise, if ptr_array [index] is NIL, the system never evaluates the expression (ptr_array [index] ^ = 5).

Not all Pascal compilers permit partial evaluation. Programs relying on this feature may not work when compiled elsewhere.

PRIVATE__PROC

Usage

$PRIVATE__PROC ON$
$PRIVATE__PROC OFF$

Default Setting

ON

Description

When the option PRIVATE__PROC is ON, the compiler puts the object code for non-level 1 procedures or functions and their containing level 1 procedures or functions into the same Relocatable Binary Module (RBM). This means the names of the non-level 1 procedures or functions do not appear in the USL file. Instead, they are maintained as unnamed private entry points. Only the names of level 1 procedures or functions are in the USL directory. (Because of the requirements of the MPE segmenter, these level 1 names must be unique within 15 characters.)

With PRIVATE__PROC ON, therefore, the programmer can observe the usual conventions of Pascal scope. In particular, two different level 1 procedures or functions may contain non-level 1 procedures or functions with the same name.

If PRIVATE__PROC is OFF, however, the compiler compiles the non- level 1 procedures and functions into separate RBM's. This means all procedure or function names from any level must be unique within 15 characters. The Pascal scope convention for non-level 1 procedure or function names does not hold.

The programmer can set PRIVATE__PROC OFF if a level 1 procedure and the procedures and functions nested within it would produce more object code with PRIVATE__PROC ON than the maximum permitted in a single RBM.

PRIVATE__PROC may appear between the declaration of level 1 procedures or functions, or in the declaration part of the outer block. That is, the programmer cannot use it within the block of a level 1 procedure or function.

Example

```
$PRIVATE_PROC ON$  (default setting)
PROGRAM show_privateproc;
  PROCEDURE proc1;
    FUNCTION check (n: integer) : boolean;
      BEGIN
        .
        .
      END;  (check)
  BEGIN
    .
    .
  END;  (proc1)

  PROCEDURE proc2;
    FUNCTION check (a,b: integer) : integer; (synonymous with  )
      BEGIN                                  (function in proc1)
        .
        .
      END; (check)
  BEGIN
    .
    .
  END; (proc2)
BEGIN (show_privateproc)
  .
  .
END.  (show_privateproc)
```

This source code produces the following USL directory:

```
USL FILE <filename>

SEG'
   OB'                35  OB A C N
   PROC2               1  P  A C N R
   PROC1               1  P  A C N R

FILE SIZE      377600( 1777.  0)
DIR. USED         337(    1.137)    INFO USED         116(    0.116)
DIR. GARB.          0(    0.  0)    INFO GARB.          0(    0.  0)
DIR. AVAIL.     37241(  175. 41)    INFO AVAIL.    337662( 1577. 62)
```

RANGE

Usage

$RANGE ON$
$RANGE OFF$

Default Setting

ON

Description

When the option RANGE is ON, the compiler generates range checking code for assignments, array indexing, parameter passing, pointers, CASE statements, and set operations. This code causes a program to terminate and an error message to appear if a value is out of range. If RANGE is OFF, the compiler does not generate checking code.

The compiler minimizes the amount of range checking code produced when RANGE is ON. If it is able to determine at compile time that at a value can never be out of range, it does not issue checking code.

RANGE may appear anywhere in source code.

Example

```
$RANGE ON$    {default setting}
PROGRAM show_range;
TYPE
  index = 1..25;
VAR
  samp_array: ARRAY [index] OF integer;
  m,n: index;
  i: integer;
BEGIN
  .
  FOR i:= m TO n DO          {The compiler doesn't generate   }
    samp_array[i]:= i;       {range checking code for this FOR}
  .                          {statement since i can never be  }
  .                          {out of bounds.                  }
END.
```

Usage

$SEGMENT s$

Parameter

s A string literal.

Default Setting

SEG'

Description

The SEGMENT option specifies a name, s, for the current segment. If a segment with the specified name exists, the compiler places the generated object code in it. Otherwise, it creates a new segment with the name indicated in the s parameter.

The compiler continues to place object code in the designated segment until it encounters another SEGMENT option.

When SEGMENT doesn't appear, the compiler uses the name SEG' as the default name of the current segment.

The compiler ignores distinctions between upper and lower case letters in the s parameter.

SEGMENT may appear anywhere in source code, but the compiler puts the object code for an entire compilation block in the last named segment. It is not possible to place part of a compilation block in a particular segment.

SEGMENT

Example

```
$SEGMENT 'Sample'$
PROGRAM show_segment (output);
PROCEDURE proc1;
  BEGIN
    writeln;
  END;
BEGIN
  proc1;
  writeln
END.
```

This source code produces the following USL directory:

```
USL FILE <filename>

SAMPLE
   OB'                40  OB A C N
   PROC1               3  P  A C N R

FILE SIZE      377600( 1777.  0)
DIR. USED         266(    1. 66)    INFO USED        141(    0.141)
DIR. GARB.          0(    0.  0)    INFO GARB.         0(    0.  0)
DIR. AVAIL.     37312(  175.112)    INFO AVAIL.   337637( 1577. 37)
```

Usage

$SKIP__TEXT ON$
$SKIP__TEXT OFF$

Default Setting

OFF

Description

When the option SKIP__TEXT is ON, the compiler ignores all subsequent source code, including any compiler options, until SKIP__TEXT is turned OFF.

SKIP__TEXT may appear anywhere in source code.

Example

```
PROGRAM show_skiptext (output);
BEGIN
  writeln('This will print.');
  $SKIP_TEXT ON$
  writeln('This won''t.');
  $SKIP_TEXT OFF$
END.
```

SPLINTR

Usage

$SPLINTR s$
$SPLINTR$

Parameter

s A string literal. If omitted, the compiler restores the default setting.

Default Setting

SPLINTR.PUB.SYS

Description

The SPLINTR option permits the programmer to specify an SPL intrinsic file which the system will search for a procedure or function declared with the INTRINSIC directive (see Section 2). The programmer may fully qualify this file name, s, with with group and account names.

The default value of the SPLINTR option is the MPE file SPLINTR.PUB.SYS. Unless the programmer specifies another file, the system searches this default file for an intrinsic.

A file specified in a SPLINTR option remains in effect until the programmer uses SPLINTR again. To restore the file SPLINTR.PUB.SYS as the designated file, the programmer can omit the s parameter (see example below).

Example

```
PROGRAM show_splinter;
PROCEDURE proc1; INTRINSIC; {System searches SPLINTR.PUB.SYS    }
$SPLINTR 'myfile'$          {for proc1.                         }
PROCEDURE proc2; INTRINSIC; {System searches MYFILE for proc2.  }
$SPLINTR$
PROCEDURE proc3; INTRINSIC; {System searches SPLINTR.PUB.SYS    }
BEGIN                       {for proc3.                         }
  .
END.
```

Usage

$STANDARD_LEVEL 'ANSI'$
$STANDARD_LEVEL 'HP'$
$STANDARD_LEVEL 'HP3000'$

Default Setting

HP

Description

The STANDARD_LEVEL option sets the level of syntax which the compiler will process routinely. If it encounters a Pascal language feature which is not legal at the specified level, the compiler issues a warning message on the listing and then compiles the feature normally.

In order of additional language features, the three levels are ANSI, HP, and HP3000. The ANSI level refers to the proposed (May 20, 1981) Pascal standard from the American National Standards Institute; HP, the default level, indicates Hewlett Packard Standard Pascal; HP3000 is Pascal/3000, the language described in this manual. The level must appear between single quote marks. The compiler ignores distinctions between upper and lower case letters.

STANDARD_LEVEL may occur anywhere in source code.

Section 1 outlines the salient features of HP Standard Pascal and Pascal/3000.

STANDARD_LEVEL

Example

```
$STANDARD_LEVEL 'ANSI'$  (equivalent to ANSI ON)
PROGRAM show_level (output);
PROCEDURE proc1;
  VAR i: integer;
      b: boolean;
  BEGIN
    assert(b,i);
  END;
BEGIN
END.
```

This source code produces the following listing:

```
PAGE  1 <Listing title>


    1.000      0  0  $STANDARD_LEVEL 'ANSI'$
    2.000      0  0  PROGRAM show_level (output);
                                    ^
**** WARNING  #  1  THIS FEATURE IS HP STANDARD PASCAL (517)
    3.000      0  0  PROCEDURE proc1;
    4.000      0  0    VAR i: integer;
    5.000      0  0    b: boolean;
    6.000      0  1    BEGIN
    7.000      0  1    assert (b,i);
                           ^
**** WARNING  #  2  THIS FEATURE IS HP3000 PASCAL (518)
    8.000      0  1    END;
    9.000      0  1    BEGIN
   10.000      0  1    END.

                  NUMBER OF ERRORS = 0      NUMBER OF WARNINGS = 2
```

SUBPROGRAM

Usage

$SUBPROGRAM s$
$SUBPROGRAM$

Parameter

s A string literal. S may be omitted.

Description

The option SUBPROGRAM causes the compiler to emit code only for the level 1 procedures or functions specified in the parameter s. The compiler also processes procedures or functions nested within the specified level 1 procedures and functions. It does not, however, compile the outer block.

SUBPROGRAM must appear before the program heading.

If s is omitted or if it is entirely blanks, the compiler processes all level 1 procedures or functions. S may contain the names of any number of level 1 procedures or functions separated by commas. If there are too many to fit on one line, the programmer may write another SUBPROGRAM option. The s parameters are concatenated.

An asterisk (*) may follow the name of a procedure or function in s. The compiler then processes the compilation block with the LIST, CODE, and TABLES options ON. Subsequent use of LIST, CODE, or TABLES in the source code of designated procedures or functions, however, will override the asterisk mechanism.

The programmer can use the SUBPROGRAM option to select parts of a large program for compilation. This minimizes the number of entries in the directory of the USL file. The compiler scans the entire source program and performs syntax and semantic checking, but it only generates object code for the specified level 1 procedures and functions.

SUBPROGRAM

Example

```
$SUBPROGRAM 'proc2*'$           (Asterisk turns ON options    )
PROGRAM show_subprg (output);   (LIST, CODE, and TABLES.      )
PROCEDURE proc1;
  BEGIN
    writeln('This won''t be compiled');
  END;
PROCEDURE proc2;
  BEGIN
    writeln('This will be compiled');
  END;
BEGIN
  writeln('The outer block isn''t compiled')
END.
```

This source code results in the following USL file directory:

```
USL FILE <filename>

SEG'
   PROC2              33  P  A C N R

FILE SIZE      377600( 1777.  0)
DIR. USED         234(    1. 34)    INFO USED          54(    0. 54)
DIR. GARB.          0(    0.  0)    INFO GARB.          0(    0.  0)
DIR. AVAIL.     37344(  175.144)    INFO AVAIL.    337724( 1577.124)
```

Usage

$TABLES ON$
$TABLES OFF$

Default Setting

OFF

Description

When the option TABLES is ON, the compiler produces an identifer map for a compilation block. The map appears as part of the listing. Thus, TABLES has no effect if the LIST option is OFF.

TABLES may appear anywhere in source code, but the compiler only issues a map if the option is ON when it completes parsing of a procedure, function, or outer block.

The map shows the declared identifiers, their class, type, and address or constant value. This information is important when the programmer uses the MPE Debug utility.

The first column lists in alphabetical order the initial 20 characters of all the identifiers declared at the current level. Field names of record types appear indented under the record name. Variables which are neither local nor global also appear in this column since the compiler allocates storage for them on the current scope.

The second column displays the class of each identifier. The compiler distinguishes the following classes: USER DEFINED, CONSTANT, VARIABLE, NON LOC VAR, FIELD, FUNCTION, TAG FIELD, PARAMETER, NON LOC PARM, and PROCEDURE.

The third column shows the type of the identifier. The types include: INTEGER, SHORT INTEGER, REAL, BOOLEAN, SUBRANGE, ENUMERATED, CHAR VALUE, CHAR ARRAY, STRING LITERAL, ARRAY, RECORD, SET, FILE, and POINTER.

Note: The $SUBPROGRAM$ option disables printing of global types and constants when $TABLES ON$.

The fourth column indicates the register-relative location of an identifier in octal or, if it is a constant, its value in decimal or characters. For a record type, the maximum word size in octal appears instead of an address. Fields of a record type are in the form W@B, where W is the word offset and B is the bit offset within the word, both in octal. Finally, the octal size of the field in bits, bytes, or words appears.

Under these four columns, the identifier map shows the amount of primary and secondary storage the compilation block requires and the number of non-local, non-global variables referenced within it. All these values are in octal.

Example

```
$TABLES ON$
PROGRAM show_map (input,output);
CONST
  realnum = 19.9;
  maxsize = 100;
  title = 'Customer List';
TYPE
  answer = (yes, no);
  rec = RECORD
          ch: char;
        CASE tag : answer OF
          yes : (messsage: PACKED ARRAY[1..20] OF char);
          no  : (i: integer);
        END;
VAR
  customer: rec;
PROCEDURE proc1 (VAR num: real);
  VAR
    debt: boolean;
  PROCEDURE subproc1;
    BEGIN
      IF debt THEN writeln
    END;
  BEGIN
  END;
FUNCTION func1: integer; EXTERNAL;
BEGIN
END.
```

This source code produces the following listing from which the editor line numbers have been removed:

```
0  0     $TABLES ON$
0  0     PROGRAM show_tables (input,output);
0  0     CONST
0  0       realnum = 19.9;
0  0       maxsize = 100;
0  0       title = 'Customer List';
0  0     TYPE
0  0       answer = (yes, no);
0  0       rec = RECORD
0  0               ch: char;
0  0             CASE tag : answer OF
0  0               yes : (messsage: PACKED ARRAY[1..20] OF char);
0  0               no  : (i: integer);
0  0             END;
0  0     VAR
0  0       customer: rec;
0  0     PROCEDURE proc1 (VAR num: real);
0  0       VAR
0  0         debt: boolean;
0  0       PROCEDURE subproc1;
0  1         BEGIN
0  1           IF debt THEN writeln
1  1         END;

                I D E N T I F I E R   M A P

 IDENTIFIER      CLASS         TYPE          ADDRESS/VALUE

 DEBT            NON LOC VAR   BOOLEAN       Q +1,I

 PRIMARY Q  STORAGE = 1      SECONDARY Q  STORAGE = 0
 NON LOCAL VARIABLES = 1

0  1       BEGIN
0  1       END;
              I D E N T I F I E R   M A P

 IDENTIFIER      CLASS         TYPE          ADDRESS/VALUE

 DEBT            VARIABLE      BOOLEAN       Q +1
 NUM             PARAMETER     REAL          Q -4,I
 SUBPROC1        PROCEDURE

 PRIMARY Q  STORAGE = 1      SECONDARY Q  STORAGE = 0
 NON LOCAL VARIABLES = 0
```

(continued)

```
0  0     FUNCTION func1: integer; EXTERNAL;
0  1     BEGIN
0  1     END.
```

I D E N T I F I E R M A P

IDENTIFIER	CLASS	TYPE	ADDRESS/VALUE
ANSWER	USER DEFINED	ENUMERATED	
CUSTOMER	VARIABLE	RECORD	DB+2,I
FUNC1	NON LOC FUNC	INTEGER	Q -5
INPUT	PARAMETER	FILE	DB+0,I
MAXSIZE	CONSTANT	SHORT INTEGER	100
NO	CONSTANT	ENUMERATED	1
OUTPUT	PARAMETER	FILE	DB+1,I
PROC1	PROCEDURE		
REALNUM	CONSTANT	REAL	1.990000E+01
REC	USER DEFINED	RECORD	MAX RECORD SIZE = 13
CH	FIELD	CHAR VALUE	0@0 FOR 1 BYTE(S)
TAG	TAG FIELD	ENUMERATED	0@10 FOR 1 BYTE(S)
I	FIELD	INTEGER	1@0 FOR 2 WORD(S)
MESSSAGE	FIELD	ARRAY	1@0 FOR 24 BYTE(S)
TITLE	CONSTANT	STRING LITERAL	Customer List
YES	CONSTANT	ENUMERATED	0

PRIMARY DB STORAGE = 3 SECONDARY DB STORAGE = 443
NON LOCAL VARIABLES = 0

Usage

$TITLE s$

Parameter

s Any string literal.

Default Setting

HEWLETT PACKARD 32106A.00.00 PASCAL/3000 (C) Hewlett Packard Co. 1981 <date><time>

Description

The option TITLE places the specified title, s, next to the page number in the top left corner of subsequent pages of the listing. The default setting is restored when s is ''. The listing has a blank title when s is ' '.

The compiler respects upper and lower case letters in the s parameter. They appear as written in the title.

TITLE may occur anywhere in source code.

Example

```
$TITLE 'My Program'$
$PAGE$
PROGRAM show_title (output);
BEGIN
  writeln('Greetings!')
END.
```

This source code produces the following listing:

TITLE

```
PAGE   1  <Default listing title>

    1.000       0  0   $TITLE 'My Program'$
    2.000       0  0   $PAGE$

PAGE   2  My Program

    3.000       0  0   PROGRAM show_title (output);
    4.000       0  1   BEGIN
    5.000       0  1     writeln('Greetings!')
    6.000       0  1   END.
```

Usage

$USLINIT$

Description

The option USLINIT causes the compiler to initialize the USL file to empty before placing any object code in it. If USLINIT is not used, the compiler appends new object code to any code already in the USL.

If the programmer does not specify a USL file when invoking the Pascal/3000 compiler and if $OLDPASS is not a USL file, or if the contents of a specified USL file are obviously incorrect, the system initializes the USL file to empty whether USLINIT occured in code or not.

USLINIT must appear before the program heading.

Example

```
$USLINIT$
PROGRAM show_uslinit (output);
BEGIN
  write('Object code for this program will be placed in ');
  writeln('an empty USL file.')
END.
```

WIDTH

Usage

$WIDTH n$

Parameter

n An integer in the range 10..132.

Default Setting

Size of record in source file.

Description

The WIDTH option sets the number of columns, n, which the compiler will read from each record of the file containing the source code. N may not be smaller than 10 or greater than 132. The default setting is 132.

WIDTH permits the compiler to ignore non-legal comments at the end of subsequent input lines.

For an INCLUDE file, the WIDTH option is reset to 132 or the specified setting within the included file. It returns to the previous setting at the end of the included file.

WIDTH may appear anywhere in source code.

Example

```
$WIDTH 30$
PROGRAM show_width (output);   The compiler ignores this text
BEGIN                          since it is beyond column 30.
  writeln('The width is 30')
END.
```

Usage

$XREF ON$
$XREF OFF$

Default Setting

OFF

Description

When the option XREF is ON, the compiler produces a cross reference for each compilation block. The cross reference is part of the listing, so XREF has no effect if LIST is OFF.

XREF may occur anywhere in source code. However, if it is placed in the middle of a procedure or function, only subsequent source code will appear in the cross reference.

The cross reference lists the first 15 characters of each identifier and its occurrences within the source code. If an identifier is declared in a block which contains the block where it occurs, the cross reference indicates its declaration level.

The cross reference shows the occurrence of an identifier by listing the number of the editor line of the source code where it appears. A symbol may prefix this number:

- @ means the identifier was declared on that line.
- * means the identifer was modified or could be modified on that line.

An editor line number appears one time for each time an identifier occurs in source code. If a source file is unnumbered, the cross reference will use the compiler-assigned sequence number.

If source code is from an include file, the include file number and a slash (/) appear before the editor line number. The compiler prints the name and number of the include file at the end of every cross reference page.

XREF

Example

```
$XREF ON$
PROGRAM show_xref (input,output);
$INCLUDE 'const'$                    {see below}
VAR
  n: integer;
  t: boolean;
PROCEDURE check (VAR b: boolean);
  BEGIN
    IF  n > k THEN b:= true
      ELSE b:= false;
  END;
BEGIN
  readln(n);
  check(t);
  IF t THEN writeln('Too big!')
    ELSE writeln('No problem');
END.
```

The INCLUDE file is:

```
CONST
  k = 100;
```

When the compiler processes show_xref and its included file, the following listing results:

```
PAGE 1  <Listing title>

   1.000      0  0   $XREF ON$
   2.000      0  0   PROGRAM show_xref (input,output);
   3.000      0  0   $INCLUDE 'const'$
   1.000      0  0   CONST
   2.000      0  0     k = 100;
   4.000      0  0   VAR
   5.000      0  0     n: integer;
   6.000      0  0     t: boolean;
   7.000      0  0   PROCEDURE check (VAR b: boolean);
   8.000      0  1     BEGIN
   9.000      0  1       IF  n > k THEN b:= true
  10.000      2  1         ELSE b:= false;
  11.000      2  1     END;

                    C R O S S   R E F E R E N C E
                    ---------   -----------------

    B                     @ 00007.000    * 00009.000  * 00010.000
    K                   0   00009.000
    N                   0   00009.000

  12.000      0  1   BEGIN
  13.000      0  1     readln(n);
  14.000      1  1     check(t);
  15.000      2  1     IF t THEN writeln('Too big!')
  16.000      4  1       ELSE writeln('No problem');
  17.000      4  1   END.


                    C R O S S   R E F E R E N C E
                    ---------   -----------------

    CHECK                 @ 00007.000     00014.000
    K                       00009.000  @ 1/00002.000
    N                     @ 00005.000     00009.000  * 00013.000
    READLN                  00013.000
    SHOW_XREF               00002.000
    T                     @ 00006.000   * 00014.000    00015.000
    WRITELN                 00015.000     00016.000

  INCLUDE FILE #  NAME
                1 const
```

STORAGE AND EXECUTION EFFICIENCY

SECTION IX

INTRODUCTION

The Pascal/3000 compiler converts source code into machine language instructions and data definitions. Data definitions allocate space on the stack for variables. The compiler does not allocate space for type definitions, but the type of a declared constant or variable determines the amount of space allocated.

There are three distinct contexts which may affect the storage for a declared variable: (1) it is independent, i.e. not a component of another structure, or (2) it is a component of an unpacked structure, or (3) it is a component of a packed structure. For example, the simple type *boolean* uses 1 word of storage in the first case, 1 byte in the second, and 1 bit in the third. On the other hand, the type *integer* requires 2 words of storage in all three cases.

If the reserved word PACKED precedes the declaration of an array or record, the compiler optimizes storage for certain simple data types within the structured type. This reduces the amount of space required by the program, but increases the time necessary to access data.

While it is syntactically legal to declare a packed file or set, this doesn't change the size of storage allocated by the compiler. Only packing an array or record actually alters the amount of space reserved on the data stack.

The standard function *sizeof* returns the amount of storage in bytes for a variable (see Section 7).

The following pages describe the storage allocation for each data type and explain how the programmer may use this information to write faster or more compact Pascal/3000 programs.

BOOLEAN STORAGE

Independent: 1 word

Unpacked: 1 byte

Packed: 1 bit

Notes

False is represented by 0, *true* by 1.

When it is independent, a boolean variable requires 1 word of storage. The left byte contains the boolean value and the right byte is undefined.

When a boolean variable is a component of an unpacked array or record, the compiler may use the right byte of the 1 word allocation for the next component. This means the boolean variable or declared constant will effectively occupy 1 byte of storage.

In a packed array or record, boolean variables require 1 bit of storage aligned by bit boundary.

INTEGER STORAGE

Independent: 2 words

Unpacked: 2 words

Packed: 2 words

Notes

For the simple type *integer*, storage allocation is identical in the three contexts. Bit 0 of the first word is the sign bit.

The compiler aligns integer storage on word boundaries.

For an integer field of a record, slightly better machine code results when the field has an even word offset. This permits the compiler to issue double word machine instructions.

INTEGER SUBRANGE STORAGE

Independent: 1 or 2 words

Unpacked: 1 or 2 words

Packed: Minimum number of bits.

Notes

As an independent variable or in an unpacked structure, an integer subrange requires 1 word of storage when it is contained in the range -32768..32767. Otherwise, it takes 2 words. Consider these examples:

As an independent variable or in an unpacked structure, an integer subrange requires 1 word of storage when it is contained in the range -32768..32767. Otherwise, it takes 2 words. Consider these examples:

Subrange	Allocation
0..8	1 word
-32768..32767	1 word
10..40000	2 words
-70000..1	2 words

In a packed array or record, an integer subrange requires the minimum number of bits necessary to represent each value of the subrange, if the subrange is in the range −32768.. 32767, otherwise it takes two words.

Subrange	Packed Allocation
0..3	2 bits
-3..0	3 bits {1 bit for the sign}
0..4	3 bits
1..7	3 bits
1..8	4 bits
0..255	8 bits
400..401	9 bits {no bias is used}
0..65000	32 bits

The compiler aligns the field representing the subrange by bit boundary and never permits the field to cross a word boundary. In a packed array, a field of 6, 7, or 8 bits takes an entire byte of storage; a field of 9 to 16 bits takes a word (see Array Storage below).

ENUMERATED STORAGE

Independent: 1 word

Unpacked: 1 byte or 1 word

Packed: Minimum number of bits

Notes

When it is independent, an enumerated type variable requires 1 word of storage. If the number of its elements is less than or equal to 256, the left byte of the word represents the value and the right byte is undefined.

If the enumerated variable is a component of an unpacked array or record and if the number of its elements is less than or equal to 256, the compiler may use the right byte of the 1 word allocation for the next component. In this case, the enumerated variable effectively requires 1 byte of storage.

In a packed array or record, an enumerated variable requires the minimum number of bits necessary to represent its values. For example:

Enumerated Type	Allocation
(east,west,north,south)	2 bits
(one, two, three, four, five)	3 bits

The compiler aligns the bit field by bit boundary and never permits the field to cross a word boundary. In a packed array, a type requiring 6, 7, or 8 bits takes 1 byte of storage; a type needing 9 or more bits requires a full word (see Array Storage below).

SUBRANGE OF ENUMERATED STORAGE

Independent: 1 word

Unpacked: Same as host type

Packed: Minimum number of bits

Notes

A subrange of an enumerated type requires the same storage as its host type, except in packed structures.

An independent variable which is a subrange of some enumerated type requires 1 word of storage. If the number of elements of the host type is less than 257, the left byte of the word represents the subrange values and the right byte is undefined.

In an unpacked record or array, a subrange enumerated variable occupies the same storage as its host type. That is, if the number of elements in the host type is less than 257, the compiler may use the right byte of the 1 word allocation for the next component.

In a packed array or record, the system determines the storage for a subrange enumerated variable according to the upper bound of the subrange. For example, suppose:

```
TYPE
  e_type = (Lee, Ron, Dave, Steve, Chris, Jon, Jean);
VAR
  sub_e_type: PACKED RECORD
                f1: Dave..Chris;

              END;
```

Field f1 of the variable sub_e_type has three elements, but the compiler calculates its storage from the first element of the host type, e_type, to the upper bound of the subrange. In other words, f1 requires 3 bits.

In a packed array, a subrange of 6, 7, or 8 bits takes an entire byte of storage; a subrange of 9 or more bits an entire word (see Array Storage below).

Independent: 2 words

Unpacked: 2 words

Packed: 2 words

Notes

The storage requirement for a variable or declared constant of type *real* is always 2 words, regardless of the context.

The system stores the real value in HP3000 floating point format (see Compiler Library Reference Manual).

For a real variable which is a field of a record, slightly more efficient machine code results if the field has an even word offset. This permits the compiler to generate double word machine instructions.

LONGREAL STORAGE

Independent: 4 words

Unpacked: 4 words

Packed: 4 words

Notes

Longreal variables or declared constants always require 4 words of storage, regardless of the context.

The system stores the longreal value in HP3000 floating point format (see Compiler Library Reference Manual).

In contrast to integer and real variables, there is no gain in machine instruction efficiency if longreal variables which are record fields have even word offsets.

CHAR STORAGE

Independent: 1 word

Unpacked: 1 byte

Packed: 1 byte

Notes

An independent char variable or declared constant requires 1 word of storage. The left byte represents the value and the right byte is undefined.

In an unpacked array or record, the compiler allocates the *char* type component 1 word, but may use the right byte for the next component. This means the component effectively takes 1 byte of storage.

In a packed record, a *char* type component takes 8 bits of storage The compiler aligns this field by bit boundary and never permits it to cross a word boundary. In a packed array, the same component takes 1 byte of space and is aligned by byte boundary.

POINTER STORAGE

Independent: 1 word

Unpacked: 1 word

Packed: 1 word

Notes

A pointer type variable requires 1 word of storage regardless of the context in which it appears.

The pointer value NIL is represented in storage by the positive integer 32767.

ARRAY STORAGE

In general, the size of an array allocation is the sum of the allocation of its components. The compiler determines this sum by the formula:

(product of cardinalities of index types) * (allocation of one component)

The compiler stores the components in row major order.

In an unpacked array, components of certain types require less storage than independent variables or declared constants of the same type. Consider this example:

```
VAR
  beauty: boolean;
  truth:  ARRAY [1..4] OF boolean;
```

The variable beauty takes 1 word of storage. The left byte contains the value and the right byte is undefined.

left byte	right byte
boolean value	not defined

The unpacked array truth, on the other hand, only takes 2 words of storage, not 4. The compiler uses the undefined right byte for the subsequent component.

first word		second word	
truth [1]	truth [2]	truth [3]	truth [4]

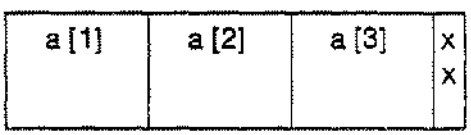
ARRAY STORAGE

The same sort of default storage optimization occurs when the component type of an unpacked array is an enumerated type less than 257 elements, a subrange of such an enumerated type, or a *char* type. There is no storage difference between a packed or unpacked array of *char*.

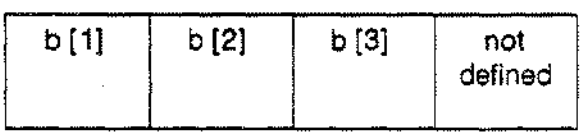
In a packed array, bit fields represent certain types of components. Types with this representation include *boolean*, subrange of integer, enumerated, and subrange of enumerated types. The number of bits required, however, does not strictly determine the amount of storage. In particular, a field of 6, 7, or 8 bits requires 1 byte of storage, and fields of 9 or more bits take 1 word. Consider this example:

```
VAR
  a: PACKED ARRAY [1..3] OF 0..31;
  b: PACKED ARRAY [1..3] OF 0..32;
```

The component type of a is the subrange 0..31 which requires a minimum of 5 bits to represent its values. The index of a has 3 elements. This means the compiler can store the entire array in 1 word. The components occupy successive fields of 5 bits and the last bit is undefined:



On the other hand, the component type of b requires 6 bits to represent all its values. Since no more than two 6-bit fields can fit into a single word in any case, the compiler assigns each field a single byte. The storage required for b, then, is two words. The right byte of the second word is unused.



The packed attribute of an array does not distribute to components of type array or record. For example, in the declaration

```
TYPE
  upa = ARRAY [1..4] OF boolean;
VAR
  pa: PACKED ARRAY [1..10] OF upa;
```

the array upa remains unpacked even when it is the component of the packed array pa.

RECORD STORAGE

The size of a record allocation is the sum of the allocation of the fixed part and, if any, the tag field and the largest variant.

In an unpacked record, fields of certain types may require less storage than independent variables of the same type. Consider this example:

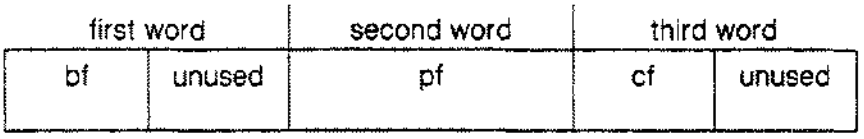
```
VAR
  bv : boolean;
  cv : char;
  upr: RECORD
         bf: boolean;
         cf: char;
       END;
```

The independent variables bv and cv each require 1 word of storage. The left byte contains the value and the right byte is undefined. The unpacked record upr also requires only 1 word of storage, not 2. The right byte of the bf allocation is used for the next field of the record, cf.

This default optimization of storage in unpacked records implies that the programmer may control the total storage requirement simply by controlling the order in which fields appear in source code. Consider this declaration:

```
VAR
  upr: RECORD
         bf: boolean;
         pf: ^upr;
         cf: char;
       END;
```

This structure requires 3 words of storage. The value of bf appears in the left byte of the first word and the right byte of this word is unused. The value of pf requires 1 word and cannot be put in the unused byte in the first word since no field may cross a word boundary. The value of cf occurs in the left byte of the third word and the right byte of this word is undefined:



The programmer can reduce the storage for upr to 2 words, however, simply by changing the order in which the fields are listed. Consider:

```
VAR
  upr: RECORD
         bf: boolean;
         cf: char;
         pf: ^upr;
       END;
```

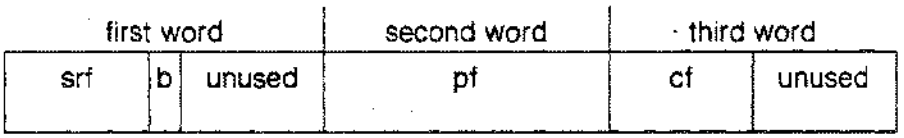
Now the value of bf occupies the left byte of the first word and the value of cf the right. The second and last word stores the value of pf:

In a packed record, the progammer can also control the total amount of storage allocated by the order of the fields. Suppose:

```
VAR
  pr: PACKED RECORD
        srf: 0..32;
          b: boolean;
         pf: ^pr;
         cf: char;
      END;
```

RECORD STORAGE

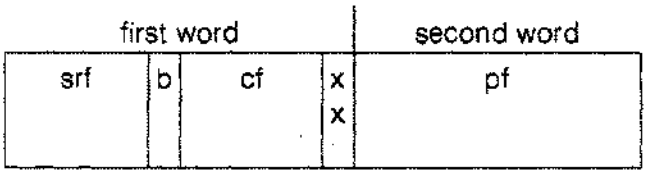
With the fields in this order, pr requires 3 words of storage. The field srf takes the first 6 bits of the first word. The field b occupies the next immediate bit. Bits 8-15 of the first word are unused. The field pf requires all of the second word. The field cf takes the first byte of the third word:



Again, the programmer can reduce the total storage by reordering the fields. Suppose:

```
VAR
  pr: PACKED RECORD
        srf: 0..32;
          b: boolean;
         cf: char;
         pf: ^pr
       END;
```

Now the first word contains the bit fields for srf, b, and cf, and only the last bit is unused. The total storage for pr is 2 words:



In contrast to packed arrays, bit fields in packed records always occupy exactly the minimum number of bits. In the example above, for instance, srf takes exactly 6 bits, the minimum required to represent all its values. If srf were a component of a packed array, however, it would occupy 8 bits.

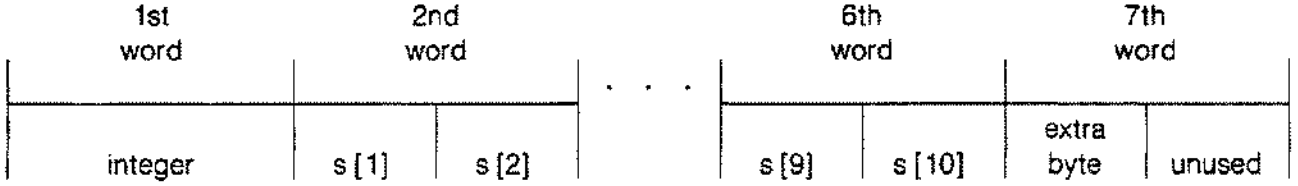
The packed attribute does not distribute to fields which are records or arrays. In other words, an array or record which is a field of a packed record is unpacked unless the progammer explicitly packs it in source code.

STRING STORAGE

The compiler allocates storage for a string according to the declared maximum length of the string. Each character takes a single byte. As well, the system requires 1 word of storage for the integer indicating the current length of the string and 1 extra byte for the implementation of certain standard string functions, i.e. *strpos*. This final byte is not accessible to the programmer. Thus, the variable s, when declared as

```
VAR
  s: string[10];
```

will take 7 words of storage: 1 word for the integer indicating the current length; 5 words for the 10 characters; and 1 word for the 'housekeeping' byte. The right byte of this final word is unused:



The call *sizeof* (s) returns 14.

If the maximum length of s is odd, the compiler uses the right byte of the last word for the extra byte. It does not have to allocate an extra word. For example, if s has a maximum length of 9, the compiler allocates 6 words of storage and *sizeof* (s) returns 12.

SET STORAGE

The compiler allocates storage for a set in minimum units of single words according to the ordinal base type of the set. For certain base types, the cardinality of this type directly determines the number of bits and, hence, the number of words needed to represent the set. This is the case, for example, with enumerated base types. Suppose:

```
VAR
  s: SET OF (fire, air, earth, water);
```

The cardinality of the base type of s is 4 and the compiler allocates 1 word of storage. Bits 0 through 3 of this word will represent the members of s.

The standard type *boolean* has a cardinality of 2 and is represented by the subrange 0..1. It requires 1 word of storage. The type *char* has 256 elements and implies the subrange 0..255. It requires 16 words of storage. When a set declaration has the base type *integer*, the compiler defaults the cardinality to 256 and assigns 16 words of storage. Values outside the range 0..255 cannot be members of this set.

When a subrange is specified as the base type, the compiler allocates storage in a different manner. It determines the positions of the upper and lower bounds on a logical word axis and then assigns storage according to the number of words 'occupied' by the subrange. This means more words than the actual number of subrange bits required may be allocated. This scheme, however, permits the machine code for set operations to avoid shift operations. The compiler treats any sort of subrange base type in this manner — subranges of *integer, char,* or enumerated types.

Starting at the origin and going right on the logical word axis, the subrange 0..15 occupies logical word 0; the subrange 16..31 logical word 1; the subrange 32..47 logical word 2; etc. Going to the left, the subrange -16..-1 is in logicial word -1; the subrange -32..-17 logical word -2; etc. (see Fig. 9-1a).

Then to allocate storage, the compiler subtracts the lower bound word position from the upper bound word position and adds 1. The result is the number of words required for storage. Suppose:

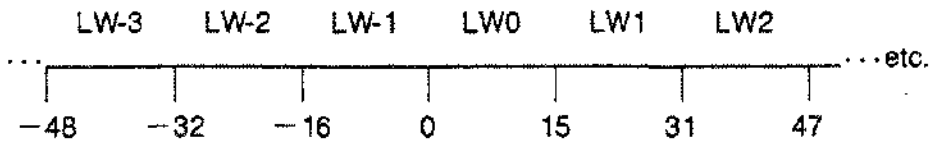
```
VAR
  s: SET OF -7..18;
```

The upper bound of the subrange representing the base type of s falls in logical word 1. The lower bound is in logical word -1 (see Fig. 9-1b). Subtracting the latter from the former and adding 1 results in 3, and this is the number of words the compiler will allocate for the storage of s. Bit 8 of the first word represents the value -7, bit 9 the value -6, and so on. Bit 2 of the third word represents the upper bound, 18. This means bits 0-7 of the first word and bits 3-15 of the third are unused (see Fig. 9-1c).

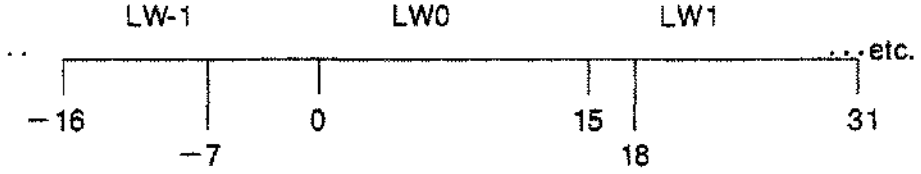
The cardinality of the subrange -7..18 is 26. Without reference to the logical word axis, i.e. if the base type of s were an enumerated type, this would require 2 words of storage. In fact, s requires 3 words of storage.

Thus, in order to optimize storage, the programmer should avoid small subranges which overlap logical word boundaries. For example, if the base type of a set is the subrange 15.. 16, the compiler will allocate 2 words of storage, even though only two bits represent the set.

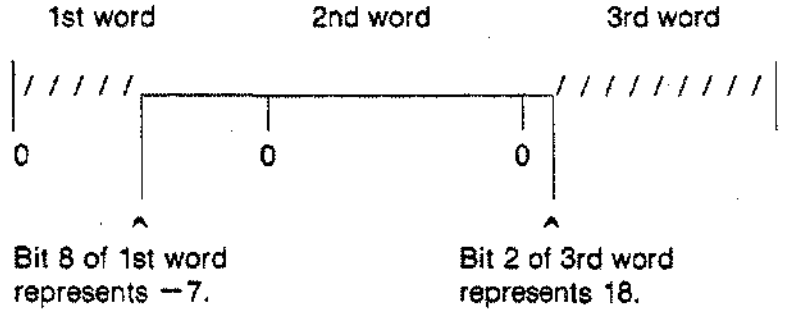
SET STORAGE



(a) The logical word axis. LW = logical word.



(b) Position of subrange −7 . . 18 on logical word axis.



(c) Actual storage of the type SET OF −7..18. / = unused.

Figure 9-1. SET STORAGE

The declaration of a logical file causes the compiler to allocate space on the stack for the file control block and the file buffer variable.

The size of the file control block varies from 8 to 13 words, depending on the type of the file. The programmer has no control over the size of this allocation.

The size of the buffer variable storage, on the other hand, depends on the type of the file component. For example, a file of *integer* requires 2 words of storage for the buffer variable; a file of *longreal* 4 words; a file of *char* 1 word; etc. Textfiles, however, are buffered one line at a time, not one character. The storage for a text file buffer variable, then, is 128 words.

In certain cases, the judicious programmer may optimize file operation speed or file buffer size. Suppose:

```
VAR
  f: FILE OF integer;
```

The buffer variable of f requires only 2 words of storage. Alternatively, the programmer may declare:

```
VAR
  f: FILE OF ARRAY[1..100] OF integer;
```

The buffer variable for f now takes 200 words of space on the stack. However, it is now possible to perform certain file operations more efficiently. For example, the programmer can assign values to components in the buffer and then write the buffer to the file:

```
FOR i:= 1 TO 100 DO f^[i]:= 10;
put(f);
```

STORAGE OPTIMIZATION

A SUMMARY

The previous pages of this chapter describe the storage requirements of the various Pascal/3000 data types in detail. Here is a summary of the ways the programmer can optimize storage:

(1) In an unpacked array, components of type *boolean, char,* or enumerated types with less than 257 elements occupy one byte of storage. Thus, two components of such an array require 1 word of storage. In particular, this means an unpacked array of *char* takes the same storage as a packed array of *char* (PAC).

(2) In a packed array, certain data types appear as bit storage fields aligned by bit boundary. A bit storage field never crosses a word boundary. These types include *boolean, char,* integer subrange, enumerated, and subrange of enumerated types. If a bit field requires 6, or 7 bits, the compiler assigns a byte storage field since no more than two 6 or 7 bit fields could fit in a single word in any case. Thus, the programmer may wish to tailor the data types so that only a 5 bit storage field is needed. Three 5 bit fields would occupy a single word.

(3) In a packed or unpacked record, the programmer may optimize storage by deliberately listing the record fields in a particular order in source code.

(4) The compiler determines set storage by the logical word requirement of the base type of the set. This means that a small subrange which overlaps a logical word boundary may require more storage than a similar subrange which doesn't. The programmer can optimize set storage by positioning a subrange on the logical word axis so that it crosses the minimum number of logical word boundaries.

(5) The size of the file buffer variable on the stack or the heap depends on the type of the file component. The programmer may choose to minimize this storage or maximize it in order to avoid the high overhead of frequent input or output operations between the data stack and a file on external disc.

Additionally, the programmer should note that the system makes a single copy of each value parameter when a program calls a procedure or function. This may use up a critical amount of storage if, for example, a value parameter is a large array.

EXECUTION EFFICIENCY

As well as storage, the programmer must often consider the interrelated issue of execution efficiency, especially when alternative versions of source code are otherwise equivalent.

Addressing Modes

The compiler generates object code which accesses program data with a variety of machine instructions. Depending on the type of the data, these instructions may use direct or indirect addressing. Direct addressing is more efficient in terms of space and time.

The compiler issues direct address instructions for global variables and local variables which require 3 or fewer words of storage. It emits indirect address instructions for variables requiring more than 3 words of storage, or for variables which are non-local and non-global. Also, it generates indirect address instructions for all dynamic variables on the heap and for reference parameters.

For value parameters, on the other hand, the compiler produces indirect address instructions if the parameter is very large or if the actual parameter list is very long. It may emit direct address instructions when one or both these conditions are false. The TABLES option (see Section 8) will indicate the actual case.

Table 9-1 summarizes these observations.

Table 9-1. DATA ACCESS

DATA CLASS	ADDRESSING MODE
Static variables	
Global <= 3 words	Direct
Local <= 3 words	Direct
Non-local, non-global	Indirect
Dynamic variables	Indirect
Parameters	
Reference	Indirect
Value	Direct or Indirect

EXECUTION EFFICIENCY

Indexing Arrays and Records

The access time for an element of a packed structure can be significantly greater if the programmer uses a variable expression rather than a constant to determine the offset at run time. For example, it is faster to select a component of a packed array with a constant or constant expression index rather than with a variable.

On the other hand, there is not much time difference when a constant index selects a component of a packed or an equivalent unpacked structure.

In sum, to maximize storage and speed, the programmer should prefer packed structures indexed with constants or constant expressions.

Partial Evaluation

The system evaluates boolean expressions compiled with PARTIAL_EVAL ON more quickly than boolean expressions compiled with PARTIAL_EVAL OFF.

Common Subexpressions

The compiler does not eliminate common subexpressions, but the programmer may often do so by using temporary variables to save intermediate results. The programmer may eliminate subexpressions involving the re-calculation of record addresses by employing the WITH statement (see below).

Constant Folding

If an expression contains more than one literal, declared constant, or constant expression, the programmer may optimize performance by grouping these elements next to one another in source code. For example, the expression:

```
2 + A + B + maxsize
```

where A and B are variables and maxsize is a declared constant, results in less efficient object code than the expression:

2 + maxsize + A + B

In the second case, the compiler is able to evaluate the first three tokens, i.e. 2, +, and maxsize, and replace them with a new constant without generating any object code. This is termed 'constant folding'.

Since evaluation proceeds from left to right when operator precedence is equal, however, the expression

2 + A + maxsize

is not optimized by

A + 2 + maxsize

which is equivalent to (A + 2) + maxsize. Instead, the programmer should write

2 + maxsize + A or A + (2 + maxsize)

Numeric Data Types

Using exact subranges may save time and the programmer should prefer real variables to longreal variables. Operations with reals are generally faster; longreals offer more precision at the expense of execution speed.

An integer subrange in the range -32768..32767 requires one word of storage. It may not be the case, however, that two integer operands typed within this range will necessarily result in single word arithmetic. For example, suppose:

```
VAR
 m: 0..32767;
 n: -100..100;
BEGIN
  .
 m:= (n * m) MOD 32767;
  .
END
```

EXECUTION EFFICIENCY

Regardless of the 1-word typing of m and n, the compiler emits double word instructions to evaluate m * n since the result is potentially greater than 32767 or less than -32768. Thus, the programmer should use subranges as close to zero as possible.

Range Checking

Range checking is extremely useful in debugging a program. It also can add a significant amount of overhead which the programmer may wish to eliminate from fully tested, frequently executed portions of a program by recompiling with the RANGE option OFF. However, the system performs certain checking, e.g. for division by zero, regardless of the RANGE setting.

Sets

The system performs set operations most efficiently on 1-word sets with identical base types. Also, it handles 2-word sets more efficiently than larger sets.

WITH Statement

The programmer may use the WITH statement to avoid the repeated calculation of a record address when referencing more than one field in the record (see Section 3). For example, these WITH statements provide greater efficiency:

```
WITH p^.r DO <statement>
WITH p^.a[i] DO <statement>
```

If there is no address recalculation, WITH provides no gain in efficiency. These statements, for example, will save typing but not execution time:

```
WITH r DO <statement>
WITH p^ DO <statement>
```

Structured Constants

Structured constants are declared constants defined with record, array, string, or set constructors (see Section 2). A structured constant requires the same amount of storage as an equivalent structured variable (except in the case of a record with variants), but the compiler stores the structured constant in the program code segment and not in the data stack. Furthermore, the programmer must initialize a structured variable at run time. In versions of Pascal without structured constants, this must be done component by component.

The system does not copy structured constants to and from the data stack for each activation of a procedure or function, which saves stack space as well as execution time.

Structured constants can improve on CASE statements which map one data type onto another. For example, the following functions are equivalent:

```
TYPE
  color = (red,blue);
  hue   = (red,blue,purple);

FUNCTION shade
  (color1,
   color2: color): hue;
BEGIN
  CASE color1 OF
    red:
      CASE color2 OF
        red:  shade := red;
        blue: shade := purple;
      END;

    blue:
      CASE color2 OF
        red:  shade := purple;
        blue: shade := blue;
      END;
  END;
END;
```

```
FUNCTION shade
  (color1,
   color2: color): hue;

TYPE
  row = ARRAY [color] OF hue;

  trans_table =
    ARRAY [color] OF row;


CONST
  table = trans_table
    [row [red, purple],
     row [purple, blue]];

BEGIN
  shade := table[color1,color2];
END;
```

EXECUTION EFFICIENCY

FOR Statement

FOR loops with 1-word integer control variables are much faster than FOR loops with 2-word integer or byte control variables.

CASE Statement

More efficient object code results when the programmer specifies case constants in a CASE statement by subrange rather than by iteration. That is,

```
CASE speed OF
  1..4: <statement>
END;
```

is better than

```
CASE speed OF
  1,2,3,4: <statement>
END;
```

The programmer cannot rely on the system to consider the case constants of a CASE statement in any particular order. The system treats all CASE constant values as being equally likely.

USING PASCAL/3000

SECTION X

Before a Pascal/3000 source program becomes a valid HP3000 process, three steps must occur:

(1) The Pascal/3000 compiler must translate the source code into binary form and store it as one or more relocatable binary modules (RBM) in a specially-formatted disc file called a user subprogram library (USL). In USL form, however, the system cannot execute a program.

(2) The MPE Segmenter must prepare the USL for execution by binding the RBM's from the USL into linked, re-entrant code segments organized in a program file. During preparation, the Segmenter also defines the initial requirements of the user data stack.

(3) The MPE Operating System must allocate and initiate execution of the program. In allocation, a process binds the segments from the program file to referenced external segments from a segmented library (SL). Then the process moves the first code segment and the associated data stack into main memory and initiates execution.

The programmer can advance through each of these steps independently, controlling the specifics of each process along the way. In particular, it is possible to use the MPE commands :PASCAL, :PREP, and :RUN for steps 1, 2, and 3, respectively.

Alternatively, the programmer may combine steps with a single MPE command. For example, the MPE command :PASCALPREP performs steps 1 and 2; the MPE command :PASCALGO steps 1, 2, and 3; the MPE command :PREPRUN steps 2 and 3.

Subsequent pages discuss the MPE commands :PASCAL, :PASCALPREP, and :PASCALGO in detail. They also explain how the programmer may invoke the Pascal/3000 compiler with the :RUN command. In the discussion of these commands, optional parameters appear in square brackets.

This section also outlines techniques for debugging Pascal/3000 programs and trapping run-time errors.

:PASCAL

Format

:PASCAL [textfile] [, [uslfile] [, [listfile]]] [;INFO = "text"]

Parameters

textfile — The name of the input file which the Pascal/3000 compiler will read. This may be any ASCII-coded file. If omitted, the file $STDIN, the current input device, is the default file. If the input file is an MPE disc file, it must be stored in a group with LOCK access.

uslfile — The name of the USL file on which the compiler will write the object code. This may be any binary file. If omitted, the file $OLDPASS is the default file. If no file is in the passed state, the system uses $NEWPASS, which is closed subsequently as $OLDPASS.

The programmer may create a new USL file in one of four ways:

(1) By specifying a non-existent USL file in the parameter. This creates a permanent USL file of the correct size and type.

(2) By saving a default $OLDPASS USL file with the :SAVE command.

(3) By building a USL file with the MPE Segmenter command -BUILDUSL.

(4) By building a new file of USL type with the :BUILD command. The filecode parameter must be 1024 or USL.

listfile — The name of the file on which the compiler will write the program listing. This can be any ASCII file. If omitted, the system assigns the file $STDLIST as the default file. Typically, this is the terminal in a session or the printer in a batch job.

text The text field of the INFO parameter permits the programmer to specify initial compiler options. Pascal/3000 brackets this field with dollar signs and places it before the first line of source code in the textfile.

Description

The MPE command :PASCAL invokes the Pascal/3000 compiler and causes it to process the specified source program and generate object code to a USL file. All of the parameters of the :PASCAL command are optional with the resulting default values indicated above.

When the textfile parameter is omitted, the default textfile is $STDIN. In a session, this will be the terminal and the programmer may enter source code interactively. A special prompt (>) appears on screen. The programmer signals the end of source code by entering the colon (:) immediately after the prompt. If the listfile is $STDLIST, the listing is not echoed back to the terminal. If the list file is $NULL or a file other then $STDLIST, the compiler displays lines with errors on $STDLIST.

Examples

```
:PASCAL mypro,myusl;INFO = "USLINIT"
{Complies souce file mypro into USL file myusl, which is     }
{initialized to empty.                                       }

:PASCAL mypro,,*LP;INFO = "PAGE (Final Version)"
{Compiles source file mypro into $OLDPASS, prints listing    }
{on line printer with initial page eject, and inserts        }
{leading comment in source code.                             }

:PASCAL ,myusl
{Invokes compiler for interactive entry of source code. The  }
{compiler will place the object code in the USL file myusl.  }
```

:PASCALPREP

Format

:PASCALPREP [textfile] [, [progfile] [, [listfile]]] [;INFO = "text"]

Parameters

textfile — The name of an input file from which the compiler will read the source program. This can be any ASCII file. If omitted, the compiler uses $STDIN as the default file, which permits the programmer to enter source code interactively. If the input file is an MPE disc file, it must be stored in a group with LOCK access.

textfile — The name of an input file from which the compiler will read the source program. This can be any ASCII file. If omitted, the compiler uses $STDIN as the default file, which permits the programmer to enter source code interactively.

progfile — The name of the program file on which the Segmenter will write the prepared program segments. This can be any binary file. If omitted, the compiler uses the file $NEWPASS as the default file.

The programmer may create a program file in two ways:

(1) By specifying a non-existent program file with the progfile parameter. The system creates a temporary file of the correct size and type.

(2) By building a new program file with the :BUILD command. The filecode parameter must be 1029.

If the programmer specifies an existing program file, the system reuses this file. An error occurs if this file is too small or if its file code is not PROG.

listfile — The name of a file on which the compiler will write the program listing. This can be any ASCII file. If omitted, the compiler uses the system file $STDLIST as the default file.

text — The text field of the INFO parameter permits the programmer to specify initial compiler options. Pascal/3000 brackets this field with dollar signs and places it before the first line of source code in the textfile.

Description

The MPE command :PASCALPREP compiles a Pascal/3000 program into a USL file and then prepares this USL file into a specified program file. All of the parameters of the command are optional.

If the programmer omits the textfile parameter, the system defaults to $STDIN as the source file. During a session, this will be the terminal. The programmer may then enter source code interactively. A special prompt (>) appears on screen. To terminate the source code, the programmer must enter a colon (:) immediately after the prompt.

The MPE Segmenter assigns a few thousand extra words of heap and stack space to programs compiled and prepared with the :PASCALPREP command. If a program requires a large heap, or if it is deeply recursive, however, the programmer may have to increase the available space by using the DL or MAXDATA parameters with the :RUN command.

Examples

```
:FILE LP;DEV=LP
:PASCALPREP test, testprog,*lp;INFO = "TABLES ON"
{Compiles source file test, prints listing on line printer with}
{TABLES option ON, and prepares resulting USL file into program}
{file testprog.                                                 }

:PASCALPREP test; INFO = "LIST OFF"
{Compiles source file test, suppressing the listing. Prepares   }
{resulting USL file into the default program file $NEWPASS,     }
{which may be run as the program file $OLDPASS                  }


:PASCALPREP ,myprog
{Permits interactive entry of source code at terminal. This     }
{code is compiled and the resulting USL file is prepared into   }
{the program file myprog.                                       }
```

:PASCALGO

Format

:PASCALGO [textfile] [, [listfile]] [;INFO = "text"]

Parameters

textfile	The name of an input file from which the compiler will read source code. This can be any ASCII-coded file. If omitted, the compiler uses $STDIN as the default file, which permits the programmer to create source code interactively at the terminal. If an input file is an MPE disc file, it must be stored in a group with LOCK access.
textfile	The name of an input file from which the compiler will read source code. This can be any ASCII-coded file. If omitted, the compiler uses $STDIN as the default file, which permits the programmer to create source code interactively at the terminal.
listfile	The name of a file to which the compiler will transmit the program listing. If omitted, the default file is $STDLIST.
text	Pascal 3000 inserts the text field of the INFO parameter before the first line of source code in the textfile and brackets it with dollar signs ($). Thus, the programmer may use the INFO parameter to specify initial compiler options.

Description

The MPE command :PASCALGO compiles, prepares, and executes a Pascal/3000 program. All of the parameters are optional. After successful completion of :PASCALGO, the program file is the temporary file $OLDPASS, which the programmer may save using the MPE :SAVE command.

If the textfile parameter is omitted, the system permits the interactive creation of source code at the terminal. A special prompt (>) appears. The programmer signals the end of source code by entering a colon (:) immediately after the prompt.

The MPE Segmenter allocates a few thousand extra words of stack space for a program compiled, prepared, and executed by the :PASCALGO command. If a program uses a large heap, or if it is deeply recursive, this default extra space may not be sufficient. The program will not execute successfully and the programmer will have to use an alternative to :PASCALGO.

Examples

```
:PASCALGO test;INFO = "CODE_OFFSETS ON;TABLES ON"
{Compiles, prepares, and then executes the source file test. }
{The listing appears on $STDLIST with two compiler options   }
{turned on by the INFO parameter.                            }


:PASCALGO universe,$NULL
{Compiles the source text universe, discarding the listing,  }
{and then prepares and executes the program.                 }
```

:RUN PASCAL.PUB.SYS

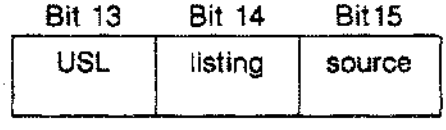
The Pascal/3000 compiler is a program file named PASCAL in the PUB group of the SYS account. The programmer may use the MPE command :RUN to execute PASCAL.PUB.SYS, i.e. invoke the Pascal/3000 compiler.

The default source, USL, and listing files for the compiler are $STDIN, $OLDPASS, and $STDLIST, respectively. To override these default values, the programmer must perform two steps: (1) equate the non-default file with its formal designator using an MPE :FILE command; (2) select an appropriate value for the PARM parameter of the :RUN command. This value indicates which files are not defaulted.

The compiler recognizes these formal file designators:

Formal Designator	File
PASTEXT	source file
PASUSL	USL file
PASLIST	listing file

The PARM parameter of the :RUN command indicates which files have appeared in file equations. The compiler opens these files instead of the default files. For the Pascal/3000 compiler, the PARM parameter accepts an integer value in the range 0..7. The low order three bits of the PARM field represent the three files:



DEBUGGING PASCAL/3000 PROGRAMS SYMBOLICALLY

Pascal/3000 programs can be debugged symbolically with the HPToolset utility by entering the $SYMDEBUG compiler option in your source file before any declaration statements.

The Symbolic Debug feature of HPToolset allows you to debug your program by referencing a procedure name or the compiler generated line numbers of your listing instead of having to know memory locations.

Refer to the HPToolset Reference Manual for information on how to run TOOLSET and use its debugging facility.

The integer value of PARM sets these bits as follows:

Value	Files present in FILE commands
0	none
1	source
2	listing
3	listing, source
4	USL
5	USL, source
6	USL, listing
7	USL, listing, source

An error occurs if the PARM value sets a bit for one of the three files and if no file equation for that file exists. On the other hand, if a file equation exists and the PARM value doesn't set the bit, the compiler will use the default file.

Setting PARM to 0 is equivalent to the command :PASCAL without parameters.

The :RUN command also has an optional INFO parameter. Pascal/3000 inserts the text field of this parameter before the first line of source code and brackets it with dollar signs ($). Thus, as with the commands :PASCAL, :PASCALPREP, and :PASCALGO, the programmer may use INFO to specify initial compiler options.

Examples

```
:FILE PASTEXT=MYSOURCE
:FILE PASLIST; DEV=LP
:RUN PASCAL.PUB.SYS;PARM=3; INFO="TABLES ON"
{This sequence of MPE commands will compile the file MYSOURCE }
{into the default USL file $NEWPASS. The listing will appear  }
{on the line printer and will include an identifier table.    }


:FILE PASUSL=TESTUSL
:RUN PASCAL.PUB.SYS;PARM=4; INFO="USLINIT"
{This sequence compiles source code entered interactively at  }
{the terminal, i.e. $STDIN, into the USL file TESTUSL, which  }
{the compiler initializes to empty.                           }
```

RUNNING PASCAL/3000 PROGRAMS

The MPE :RUN command has two optional parameters, PARM and INFO, whose values the programmer may pass to any Pascal/3000 program. The PARM field is a 16 bit signed integer. The INFO field is a string of up to 255 characters, including the double or single quote delimiters at the beginning and end.

The programmer may obtain the values of PARM or INFO in a Pascal/3000 program by specifying appropriate parameters in the program heading. These parameters, which normally contain the names of logical files, may specify a variable for PARM, a variable for INFO, or both.

After placing them in the program parameter list, the programmer must declare the identifiers as global variables in the declaration part of the outer block.

The variable for PARM must be type *integer* or an integer subrange. The system will convert the PARM value to this type and perform range checking if necessary.

The variable for INFO must be type *string* or PAC. The system will range check the length of the INFO field with this variable if needed. If the INFO field is shorter than a packed array of char, the array will be blank filled.

The system performs range checking on the PARM value and the length of the INFO field depending on the setting of the RANGE option when the compiler encounters the first line of the executable part of the outer block.

Section 6 discusses ways the programmer may use the INFO parameter to associate physical and logical files.

Example

```
PROGRAM example_1 (parm,info);
VAR
  parm: integer;
  info: PACKED ARRAY [1..255] OF char;
BEGIN
END.

PROGRAM example_2 (i, input, output, p);
VAR
  p: 1..10;
  i: string[10];
BEGIN
END.

PROGRAM example_3 (j);
VAR
  j: -1..1;
BEGIN
END.

PROGRAM example_4 (a);
VAR
  a: PACKED ARRAY [1..132] OF char;
BEGIN
END;
```

DEBUGGING PASCAL/3000 PROGRAMS

To debug a Pascal/3000 program, the programmer may use the TABLES and CODE_OFFSETS compiler options in conjunction with a PMAP and the MPE Debug facility.

The TABLES option lists each declared identifier and its stack location (see Section 8). The CODE_OFFSETS option shows the P register offset for each statement in a compilation block (see Section 8). The PMAP indicates the procedure location within a code segment (see MPE Reference Manual) and is available through the :PREP command. The Debug facility is documented in the Debug/Stack Dump Reference Manual.

In general, the programmer may follow these steps:

(1) Compile the program into a USL file with the TABLES and CODE_OFFSETS options ON and direct the listing to the line printer.

(2) Prepare the USL file into a program file invoking the PMAP option and directing the map to the line printer.

(3) Run the program using the DEBUG facility. Set appropriate break points by using the segment number and the code location from the PMAP combined with the statement offset from the listing.

(4) Resume program execution and, when the breakpoint occurs, use the variable locations on the listing to display or otherwise manipulate the current variable values.

To illustrate these steps, we consider a sample program. Show_Debugging has a level 1 'blackbox' function and a level 1 procedure which calls the function. When calling the function, the procedure passes a global variable, a local variable, and an actual parameter from the main program as parameters. The procedure stores the result of the function call in another local variable and then writes this variable on the standard file *output*.

The source code for Show_Debugging is:

```
$TABLES ON;CODE_OFFSETS ON$
PROGRAM Show_Debugging (output);
TYPE
  smallint = -32768..32767;  {Takes 1 word of storage. }
VAR
  global_var: smallint;      {A global variable.       }

FUNCTION blackbox            {This blackbox function is}
    (parm1: smallint;        {invisible to the ordinary}
     parm2: smallint;        {reader.                  }
     parm3: smallint
    ): smallint;
  VAR temp_result: smallint;
  BEGIN
    { find max of parm1 and parm2 }
    IF parm1 > parm2
      THEN  temp_result := parm1
      ELSE  temp_result := parm2;
    { find min of temp_result and parm3 }
    IF temp_result < parm3
      THEN  blackbox := temp_result
      ELSE  blackbox := parm3;
  END;

PROCEDURE a_level_1_proc (parm_val: smallint);

  VAR local_var: smallint;   {A local variable.       }
      result : smallint;
  BEGIN
    result := 0;
    local_var := -32768;
    result := blackbox (global_var,local_var,parm_val);
    writeln('Blackbox returns', result:6);
  END;

BEGIN (Show_Debugging)
  global_var := 7;
  a_level_1_proc(3);
END.  (Show_Debugging)
```

When compiled, this source code produces the following listing, which we have annotated with numbers bracketed by asterisks, e.g. *1*, to aid subsequent discussion.

```
 1.000  0  0   $TABLES ON;CODE_OFFSETS ON$
 2.000  0  0   PROGRAM Show_Debugging (output);
 3.000  0  0   TYPE
 4.000  0  0     smallint = -32768..32767;
 5.000  0  0   VAR
 6.000  0  0     global_var: smallint;
 7.000  0  0
 8.000  0  0   FUNCTION blackbox
 9.000  0  0       (parm1: smallint;
10.000  0  0        parm2: smallint;
11.000  0  0        parm3: smallint
12.000  0  0       ): smallint;
13.000  0  0     VAR temp_result: smallint;
14.000  0  1     BEGIN
15.000  0  1       IF parm1 > parm2
16.000  1  1         THEN  temp_result := parm1
17.000  2  1         ELSE  temp_result := parm2;
18.000  3  1       IF temp_result < parm3
19.000  4  1         THEN  blackbox := temp_result
20.000  5  1         ELSE  blackbox := parm3;
21.000  5  1     END;
22.000  6  0
```

C O D E O F F S E T S

STMT	P LOC	STMT	P LOC	STMT	P LOC
0	000014	2	000022	4	000027
1	000017	3	000024	5	000032

I D E N T I F I E R M A P

IDENTIFIER	CLASS	TYPE	ADDRESS/VALUE
BLACKBOX	FUNCTION	SUBRANGE	Q -7
PARM1	PARAMETER	SUBRANGE	Q -6
PARM2	PARAMETER	SUBRANGE	Q -5
PARM3	PARAMETER	SUBRANGE	Q -4
TEMP_RESULT	VARIABLE	SUBRANGE	Q +1

PRIMARY Q STORAGE = 1 SECONDARY Q STORAGE = 0
NON LOCAL VARIABLES = 0

```
23.000  6  0  PROCEDURE a_level_1_proc (parm_val: smallint);
24.000  0  0    VAR local_var: smallint;
25.000  0  0        result: smallint;
26.000  0  1    BEGIN
27.000  0  1      result := 0;
28.000  1  1      local_var := -32768;
29.000  2  1      result:=blackbox(global_var,local_var,parm_val);
30.000  3  1      writeln ('Blackbox returns', result:6);
31.000  3  1    END;
32.000  4  0
```

C O D E O F F S E T S

	1	*6*
STMT P LOC	STMT P LOC	STMT P LOC
0 000014	1 000016	2 000020	3 000026

I D E N T I F I E R M A P

IDENTIFIER	CLASS	TYPE	ADDRESS/VALUE
*3*LOCAL_VAR	VARIABLE	SUBRANGE	Q +1
*4*PARM_VAL	PARAMETER	SUBRANGE	Q -4
*5*RESULT	VARIABLE	SUBRANGE	Q +2

PRIMARY Q STORAGE = 5 SECONDARY Q STORAGE = 0
NON LOCAL VARIABLES = 0

```
33.000  0  1  BEGIN (main)
34.000  0  1    global_var := 7;
35.000  1  1    a_level_1_proc (3);
36.000  1  1  END.
```

C O D E O F F S E T S

STMT P LOC	STMT P LOC
0 000035	1 000037

I D E N T I F I E R M A P

IDENTIFIER	CLASS	TYPE	ADDRESS/VALUE
A_LEVEL_1_PROC	PROCEDURE		
BLACKBOX	NON LOC FUNC	SUBRANGE	Q -7
*2*GLOBAL_VAR	VARIABLE	SUBRANGE	DB+1
OUTPUT	PARAMETER	FILE	DB+0,I
SMALLINT	USER DEFINED	SUBRANGE	

PRIMARY DB STORAGE = 2 SECONDARY DB STORAGE = 214
NON LOCAL VARIABLES = 0

We then prepare the USL file to a program file with the :PREP command and the PMAP option. The resulting PMAP looks like this:

```
PROGRAM FILE <progam filename>

SEG'              0
   NAME           STT  CODE ENTRY SEG
   OB'              1     0    13
   TERMINATE'       4                ?
   P'REWRITE        5                ?
   P'CLOSEIO        6                ?
   P'INITHEAP'3000  7                ?
   A_LEVEL_1_PROC   2    51    64
   P'WRITELN       10                ?
   P'WRITESTR      11                ?
   P'WRITESINT     12                ?
   BLACKBOX         3   134   147
   SEGMENT LENGTH       204

PRIMARY DB       3    INITIAL STACK   10240    CAPABILITY      600
SECONDARY DB   214    INITIAL DL          0    TOTAL CODE      204
TOTAL DB       216    MAXIMUM DATA        ?    TOTAL RECORDS     7
ELAPSED TIME   00:00:00.837              PROCESSOR TIME   00:00.277
```

The necessary information to use the Debug facility successfully is now on hand. We will set a break point at the place in the program where the level 1 procedure calls blackbox; examine the values of the three parameters and the value of the local variable which will store the function return; and then set a second break point immediately after the call to blackbox and again look at the local variable storing the returned value.

We begin by executing the program with the DEBUG option specified. To set the first break point at the call to blackbox, we first use the PMAP to find the segment number, 0, and the code location, 51, for a_level_1_proc. Then we turn to the listing which shows that the call to blackbox occurs in statement 2 of this procedure. The offset of this statement is 20 (see *1* above).

Thus, the initial Debug prompt and our response will be

```
?b  0.51+20
```

followed by

```
?r
```

to 'resume' execution.

When the process reaches the break point, Debug again prompts us for a command. To display the value of the global_var parameter, we look at the listing to find the variable's location on the stack. It is DB+1 (see *2* above). We respond to the Debug prompt accordingly, and the current value appears in decimal:

```
?d  db+1,i
DB+1        +00007
```

The value of global_var at the time of the call to blackbox is 7.

In analogous fashion, we now display the current values of local_var, parm_val, and result, using the locations Q+1, Q-4, and Q+2, respectively (see *3*, *4*, and *5* above):

```
?d q+1,i
Q+1         -32768
?d q-4,i
Q-4         +00003
?d q+2,i
Q+2         +00000
```

The value of local_var is -32768; parm_val 3; result 0.

We now set the second break point immediately after the function call, i.e. statement 3 of the procedure, and then resume execution. The code offset of statement 3 is 26 (see *6* above). The segment number and procedure entry point are unchanged.

```
?b  0.51+26
?r
```

At the break, we again display the value of the variable containing the function return. Its location is still Q+2 (see *5* above).

```
?d  q+2,i
Q+2         +00003
```

The new value of result is 3.

Now, by resuming, we allow execution to finish:

```
?r

Blackbox returns      3

END OF PROGRAM
```

TRAPPING RUN-TIME ERRORS

An error in Pascal/3000 may be a compile-time error, a run-time error, or an undetected error. These three types of error occur when:

(1) The compiler detects and reports the error at compile time. The error message appears on the listing with a caret (^) pointing to the location of the problem. Appendix C discusses the compile-time errors.

(2) The system detects an error at run time. The system will report the error and abort the program unless the programmer has created and armed a trap procedure. Appendix D lists the run-time errors.

(3) Neither the compiler nor the system detect the error and no message appears. Appendix E discusses currently undetected errors for Pascal/3000. In any future release, an undetected error may become a compile-time or run-time error.

Pascal/3000 permits the programmer to use the XLIBTRAP intrinsic to trap any software-related run-time error. Also, the XARITRAP intrinsic can trap hardware-related run-time errors such as integer overflow or division by 0.

To use the XLIBTRAP intrinsic for software-related run-time errors, the programmer must follow these steps:

(A) Declare the XLIBTRAP intrinsic in Pascal source code with the INTRINSIC directive (see Section 2 and the MPE Intrinsics Reference Manual).

(B) Declare the trap procedure using the appropriate formal parameters. In particular, the first formal parameter must be a VAR parameter. It will return the stack marker created when the error occurred. (In the example below, only the 1st word is returned.) The second and third parameters must be 1-word VAR parameters. The second returns the number of the error. The third is a flag which the programmer can set with an integer value within the trap procedure.

According to the setting of this flag on exit from the trap procedure, the system will abort the program, continue execution, print the system error message, or suppress this message. The following table indicates how various types of flag values determine the permutations of the possible actions.

Flag	Action
0, or <0 and even	Continue execution; suppress message
>0 and even	Continue execution: print message
>0 and odd	Stop execution; print message
<0 and odd	Stop execution; suppress message

If the flag is not set anywhere in the trap, the system uses 1 as the default flag value on exiting the procedure.

The Compiler Library Reference Manual, Section IV, examines these parameters in detail.

(C) Arm the trap by calling XLIBTRAP in the executable part of the program. The first actual parameter must be the external label of the trap procedure. In Pascal/3000, this is available from the *waddress* function when the name of the trap procedure is the argument (see Section 7). The second actual parameter must be a reference parameter, i.e. a variable. The intrinsic SPL parameter type is INTEGER, so a suitable type is the integer subrange -32768..32767. This second parameter returns the previous external label to the program, or 0 if no label existed.

A trap is disarmed when the first actual parameter of the XLIBTRAP call is 0.

After a XLIBTRAP call, the condition code returned by the procedure *ccode* indicates the success of the operation (see MPE Intrinsics Reference Manual).

The example below illustrates these three steps.

To use the XARITRAP intrinsic, the programmer must follow a series of analogous steps. The only possible parameter for the trap procedure, however, is a 1-word VAR parameter which returns a bit pattern indicating which hardware error occured. The parameters of the XARITRAP intrinsic are described in detail in the MPE Intrinsics Reference Manual. XCONTRAP, XSYSTRAP and XARITRAP handlers cannot be totally written in Pascal.

Example:

```
PROGRAM FileErrorTrap(input,output);

{The main program requests the name of a file for processing}
{from the user. If this name causes a file system error at  }
{the call to reset, the trap procedure prints the error    }
{message and lets the user re-enter a file name. Otherwise, }
{the program aborts with a message from the trap.          }

  TYPE
    ShortInteger = -32768..32767;
  VAR
    Try_Again  : boolean;
    Old_P_Label : ShortInteger;
    File1 : FILE OF integer;
    File_Name : PACKED ARRAY[1..40] OF char;

  PROCEDURE XLibTrap; INTRINSIC;              {step A}
```

Since level 2,3... procedures are only known to Pascal all TRAP HANDLERS should be level 1 procedures.

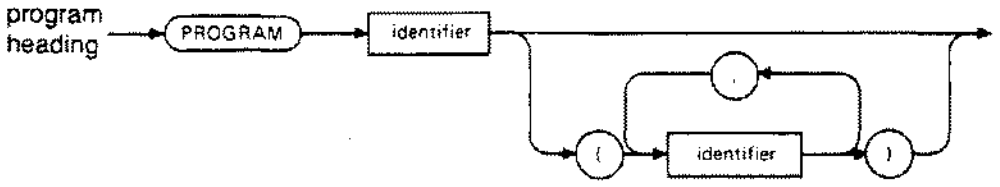
TRAPPING RUN-TIME ERRORS

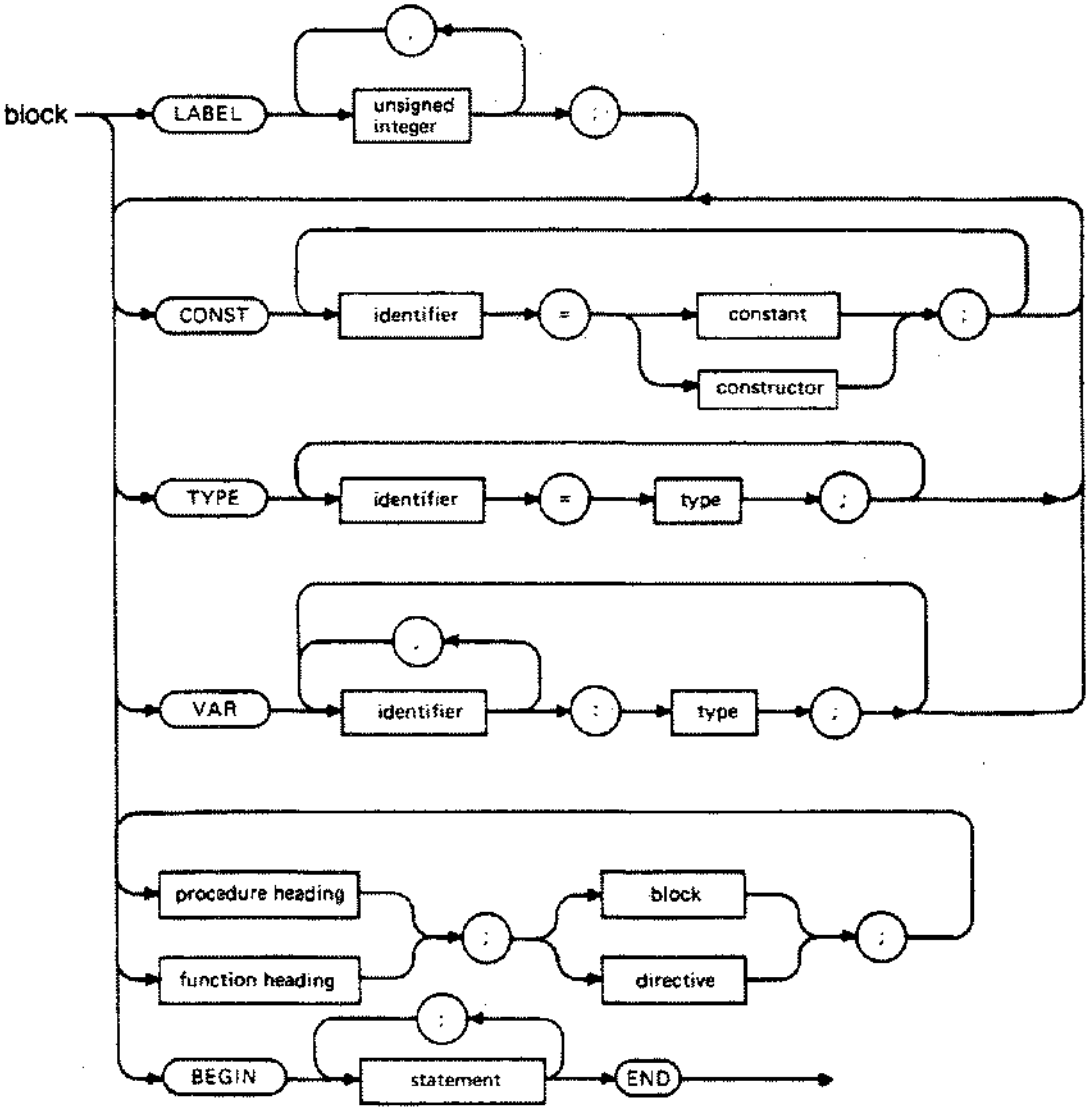
```
PROCEDURE Lib_Traps(                          {step B}
                    VAR StkMrk,
                        ErrorNum,
                        AbortFlag : ShortInteger
                   );  {must be a level 1 procedure}
  TYPE
    MsgLen = 1..72;
  VAR
    Message_Buffer : PACKED ARRAY[MsgLen] OF char;
    Message_Length : MsgLen;
    FS_Error : ShortInteger;

  PROCEDURE FErrMsg; INTRINSIC;

  PROCEDURE FCheck;  INTRINSIC;

  BEGIN  {Lib_Traps}

    CASE ErrorNum OF
    692:
      BEGIN  {File open error}
        FCheck(0,FS_Error);
        FErrMsg(FS_Error,Message_Buffer,Message_Length);
        writeln(Message_Buffer : Message_Length);
        Try_Again := true;
        AbortFlag:= 0;  {permits return to main program}
      END;   {File open error}
    OTHERWISE
      Try_Again := false;
    END; {CASE ErrorNum}

    IF NOT Try_Again THEN
      BEGIN
        writeln('*** Error detected during execution ***');
        writeln('*** Library Error No. ',ErrorNum:8,' ***');
        AbortFlag:= -1;  {causes abort without message}
      END;

  END;   {Lib_Traps}
```

```
BEGIN  {FileErrorTrap}

  XLibTrap(waddress(Lib_Traps), Old_P_Label);     {step C}

  Try_Again := true;

  WHILE Try_Again DO
    BEGIN
      prompt('Type file name for input: ');
      readln(File_Name);
      Try_Again := false;   {Only try again if this one fails.}
      reset(File1,File_Name);
    END;

  WHILE NOT eof(File1) DO
    BEGIN
    {Process File1}
    END;

END.   {FileErrorTrap}
```

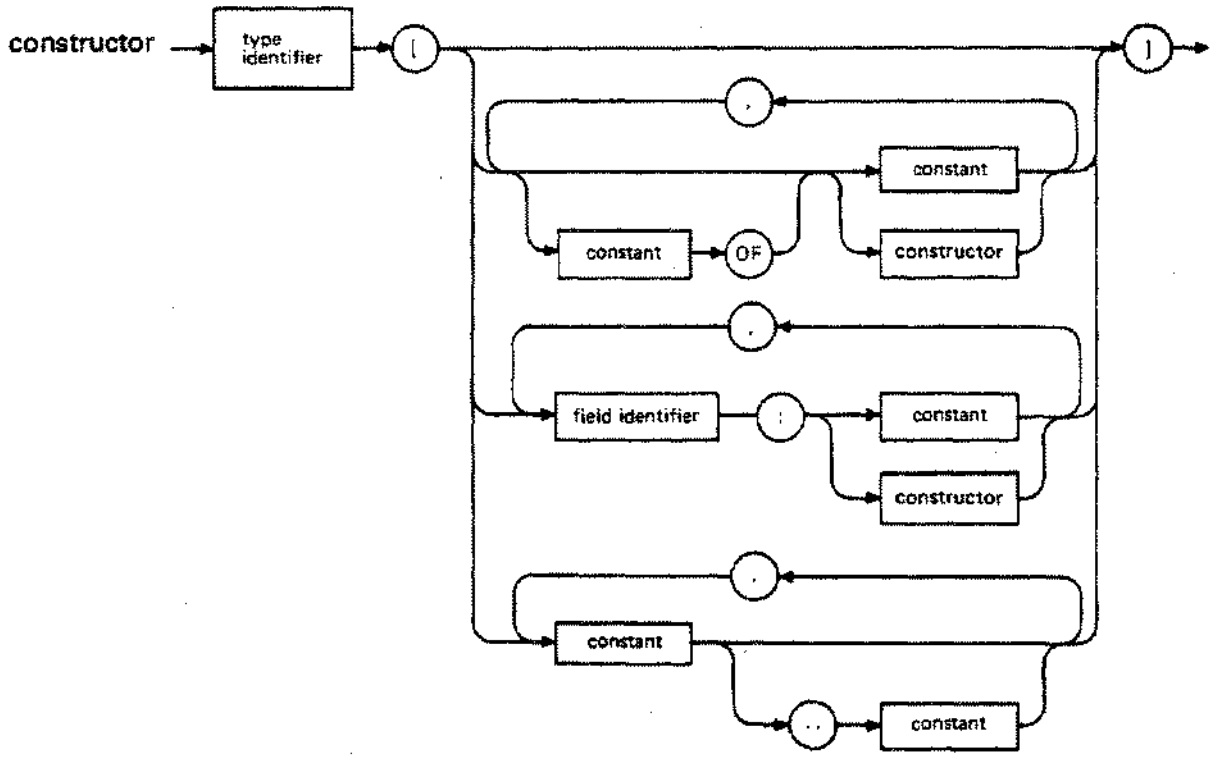
PASCAL/3000 SYNTAX DIAGRAMS

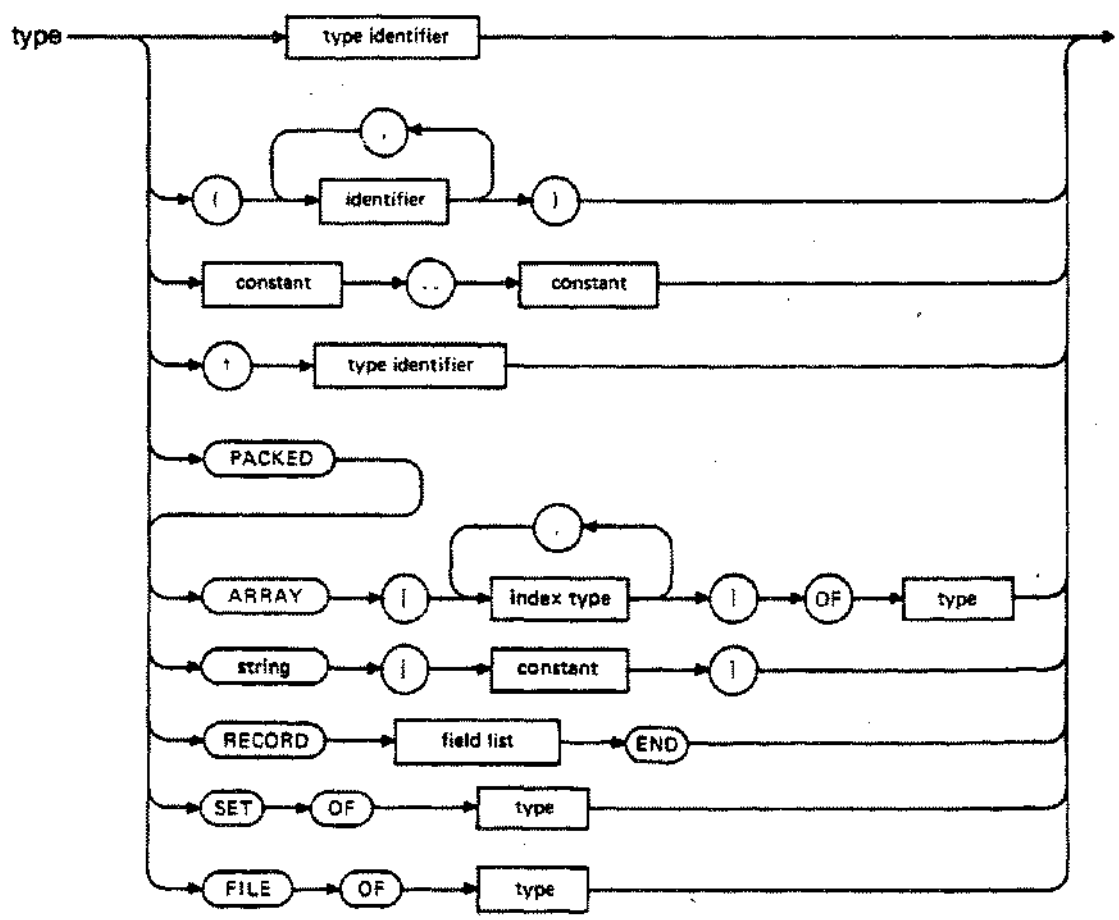
APPENDIX
A

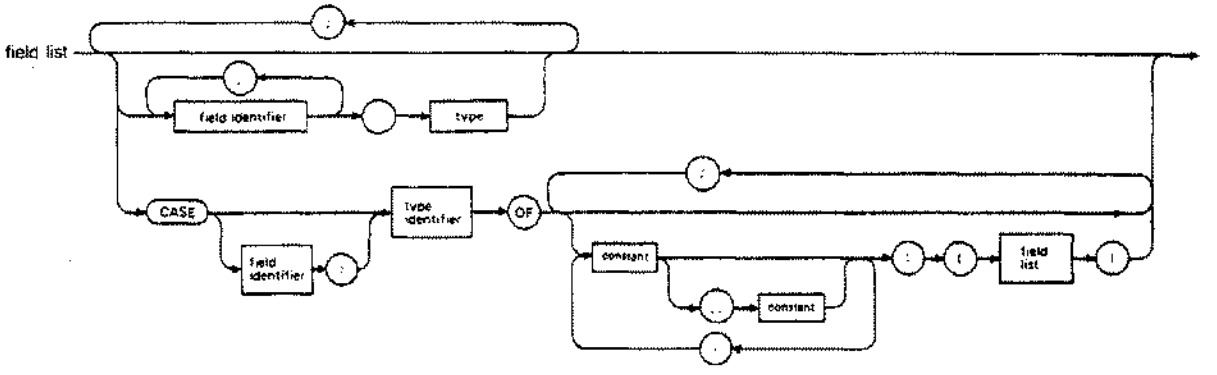


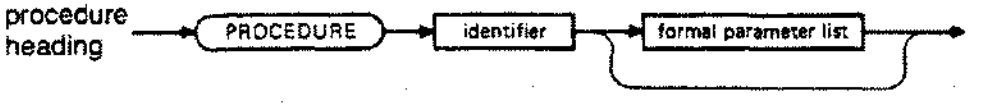


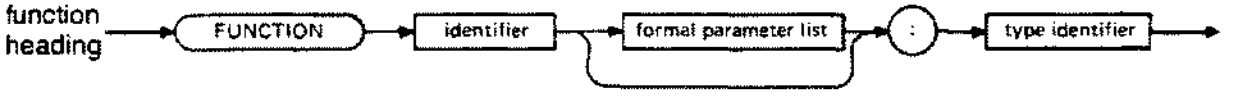
PASCAL/3000 SYNTAX DIAGRAMS



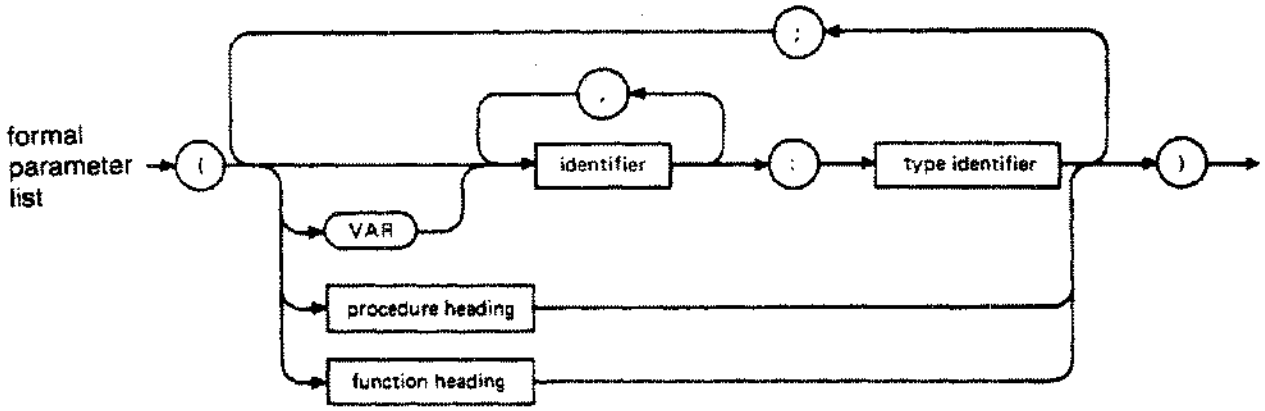


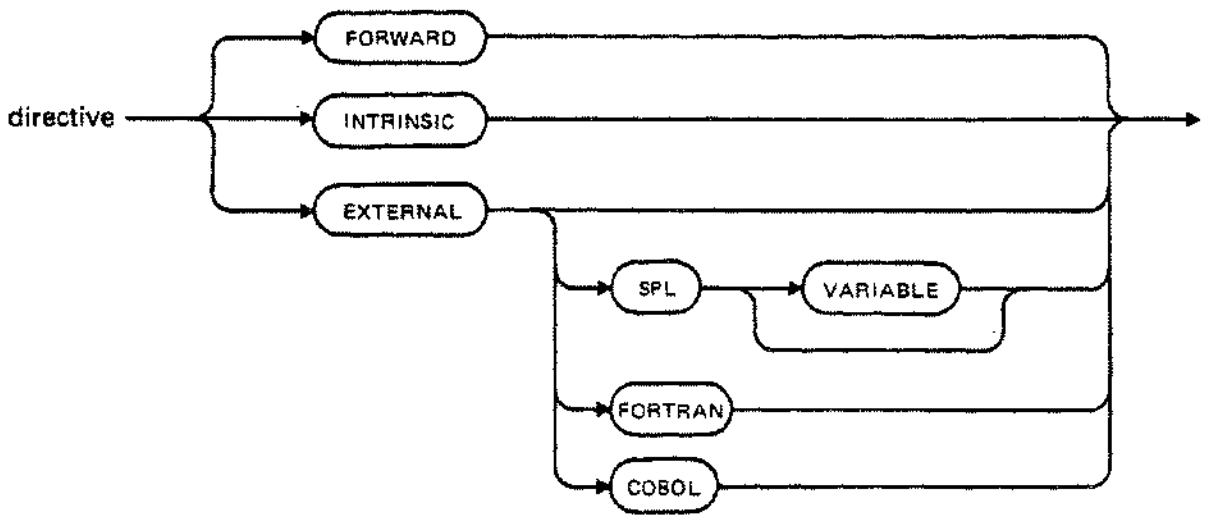

field list
field identifier
type
CASE
field identifier
type identifier
OF
constant
constant
field list

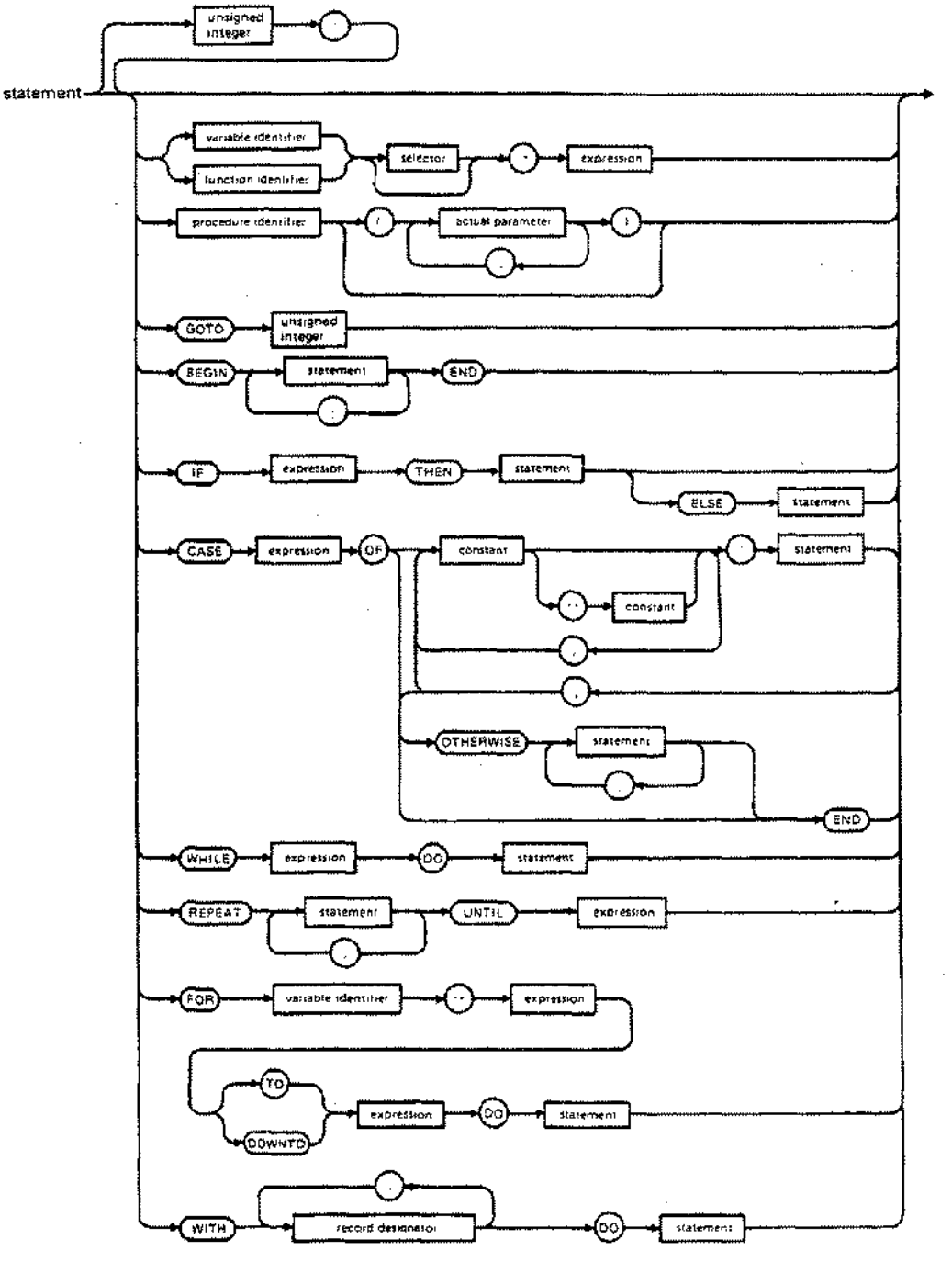

procedure heading
PROCEDURE
identifier
formal parameter list

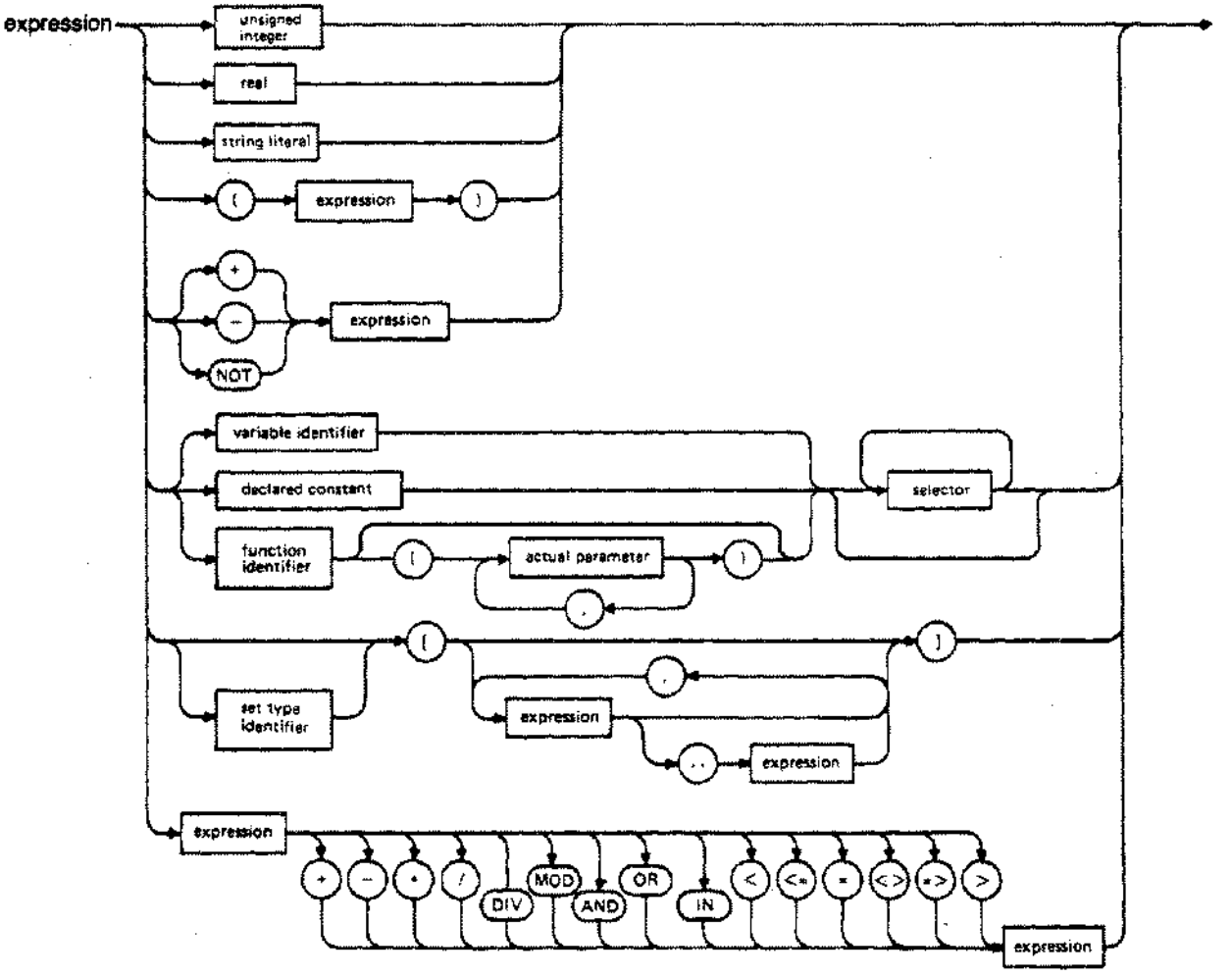

function heading
FUNCTION
identifier
formal parameter list
type identifier

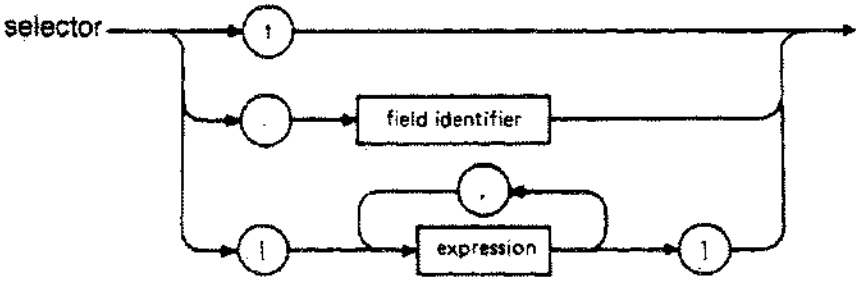
PASCAL/3000 SYNTAX DIAGRAMS

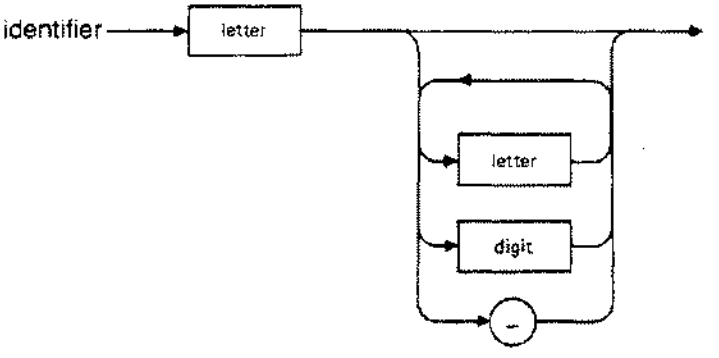


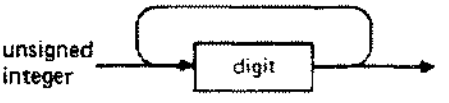


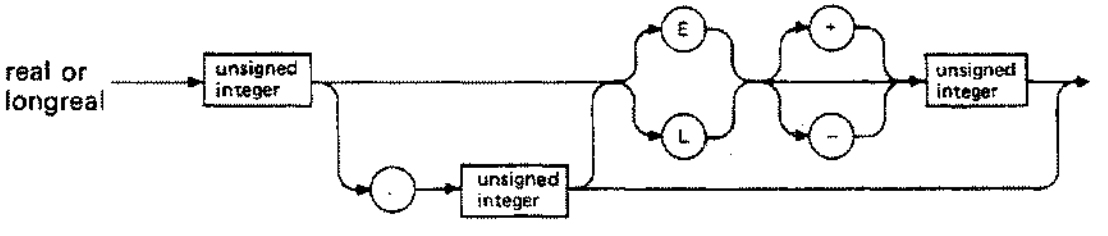

statement
unsigned integer
variable identifier
function identifier
selector
expression
procedure identifier
actual parameter
GOTO
unsigned integer
BEGIN
statement
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
statement
constant
OTHERWISE
statement
END
WHILE
expression
DO
statement
REPEAT
statement
UNTIL
expression
FOR
variable identifier
expression
TO
DOWNTO
expression
DO
statement
WITH
record designator
DO
statement


expression
unsigned integer
real
string literal
(
expression
)
+
-
NOT
expression
variable identifier
declared constant
function identifier
(
actual parameter
,
)
selector
[
set type identifier
expression
,
..
expression
]
expression
+
-
*
/
DIV
MOD
AND
OR
IN
<
<=
=
<>
#>
>
expression

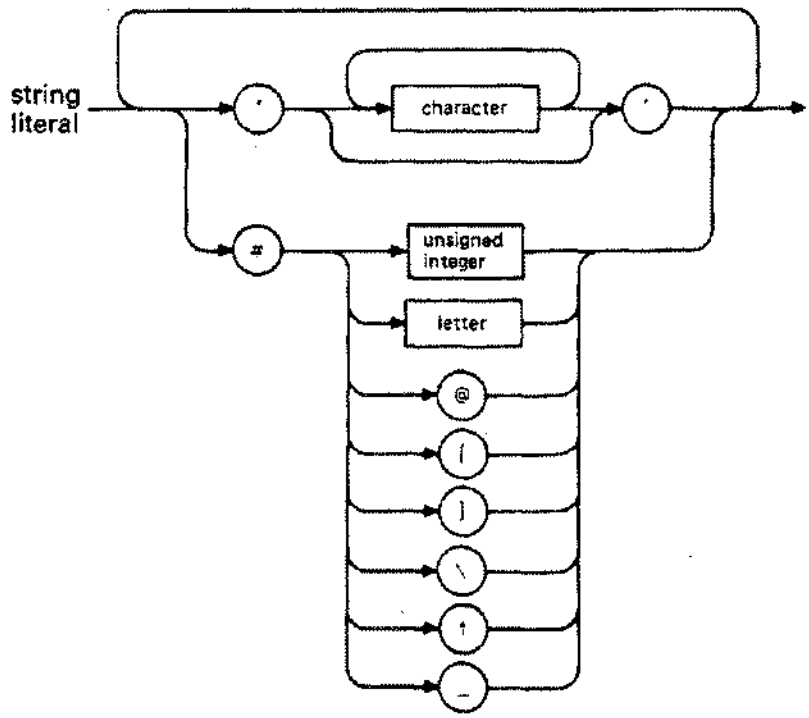

selector
↑
.
field identifier
[
,
expression
]

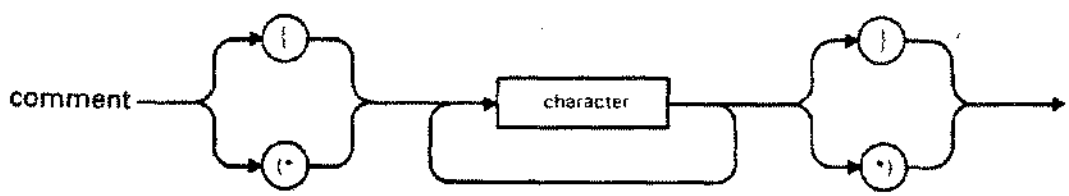

identifier
letter
letter
digit
_

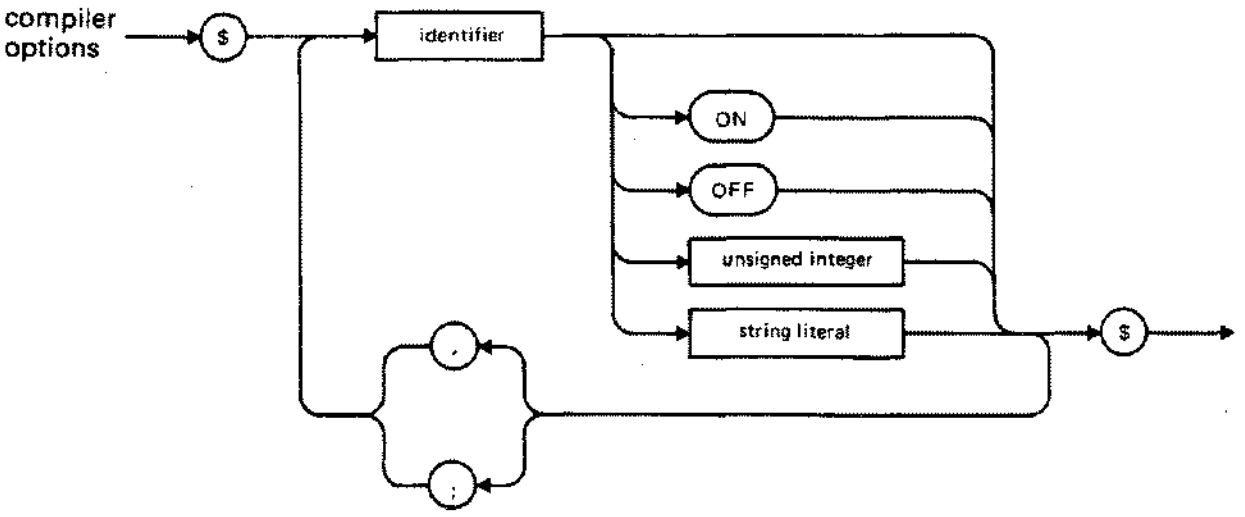

unsigned
integer
digit


real or
longreal
unsigned
integer
E
L
+
-
unsigned
integer
.
unsigned
integer

PASCAL/3000 SYNTAX DIAGRAMS






compiler options
$
identifier
ON
OFF
unsigned integer
string literal
$
,
;

RESERVED WORDS AND STANDARD IDENTIFIERS

APPENDIX B

Reserved Words

Reserved words are indivisible symbols with a fixed meaning. The programmer may not redefine a reserved word, or use it other than in its defined way, except within a string literal or a comment.

Reserved words appear in upper case througout this manual. Within a Pascal/3000 source text, however, they may appear in any combination of upper and lower case. The Pascal/3000 reserved words are:

AND	ELSE	IN	OTHERWISE	THEN
ARRAY	END	LABEL	PACKED	TO
BEGIN	FILE	MOD	PROCEDURE	TYPE
CASE	FOR	NIL	PROGRAM	UNTIL
CONST	FUNCTION	NOT	RECORD	VAR
DIV	GOTO	OF	REPEAT	WHILE
DO	IF	OR	SET	WITH
DOWNTO				

OTHERWISE is a special case. It ceases to be a reserved word when the ANSI compiler option is ON, or when the STANDARD_LEVEL option is set to ANSI (see Section 8). When one of these conditions hold, OTHERWISE may appear as a programmer-defined identifier.

Standard Identifiers

Standard identifiers are predefined identifiers which the compiler will recognize without explicit declaration in source code. The programmer, however, may redefine a standard identifier in source code. In this case, the compiler recognizes the new definition within the scope of the declared identifier.

Standard identifiers appear in italics throughout this manual. In source code, the compiler recognizes standard identifiers in any combination of upper and lower case.

RESERVED WORDS AND STANDARD IDENTIFIERS

The Pascal/3000 standard identifiers are:

Standard Constants

false	*maxint*	*minint*	*true*

Standard Types

boolean	*integer*	*real*	*text*
char	*longreal*	*string*	

Standard Files

input	*output*

Standard Functions

abs	*exp*	*position*	*striltrim*
arctan	**fnum*	*pred*	*strmax*
**baddress*	*hex*	*round*	*strpos*
binary	*linepos*	*sin*	*strrpt*
**ccode*	*ln*	**sizeof*	*strrtrim*
chr	*maxpos*	*sqr*	*succ*
cos	*octal*	*sqrt*	*trunc*
eof	*odd*	*str*	**waddress*
eoln	*ord*	*strlen*	

Standard Procedures

append	*open*	*release*	*strinsert*
**assert*	*overprint*	*readln*	*strmove*
close	*pack*	*reset*	*strread*
dispose	*page*	*rewrite*	*strwrite*
get	*prompt*	*seek*	*unpack*
halt	*put*	*setstrlen*	*write*
mark	*read*	*strappend*	*writedir*
new	*readdir*	*strdelete*	*writeln*

*Pascal/3000 only.

COMPILE-TIME ERRORS AND WARNINGS

APPENDIX C

This appendix lists the annotated compile-time error messages and warnings for the Pascal/3000 compiler. Compile-time errors are numbered in the range 0..499; warnings are in the range 500..599.

The text of the message is followed by notes explaining the situations which cause the error or warning. An exclamation point (!) in the messages reproduced here will be replaced in an actual message with an appropiate token. These error and warning messages together with the notes are available on line in the file PASCAT.PUB.SYS.

COMPILE-TIME ERRORS

001 FLOATING POINT OVERFLOW (001)
1. The absolute value of a real number is greater than 1.15792E77.
2. The absolute value of a longreal number is greater than 1.1579208923731 62E77.

002 FLOATING POINT UNDERFLOW (002)
1. The real number is nonzero and the absolute value is less than 8.63617E-78.
2. The longreal number is nonzero and the absolute value is less than 8.636168555094445E-78.

003 ERROR IN FLOATING POINT NUMBER REPRESENTATION (003)
1. The real or longreal number must have a digit after the decimal point.

004 AN EXPONENT IS REQUIRED HERE (004)
1. The exponent for a real or longreal number is missing. A number is required after the 'E' or 'L'.

005 ILLEGAL CONTROL CHARACTER CONSTANT (005)
1. The value of the constant following the sharp (#) is greater than 255.
2. The only characters that can follow a sharp (#) are a letter, @, [,], \, ^, or _.

006 A QUOTE IS EXPECTED HERE (006)
1. The end of line was found before the terminating quote. Strings literals cannot span source lines.

007 INTEGER OVERFLOW (007)
1. The absolute value of the integer is greater than *maxint*, i.e. 2147483647.

008 END OF FILE FOUND BEFORE EXPECTED (008)
1. The compiler expects more source code. There may be an unmatched BEGIN-END or an unclosed comment.

009 UNRECOGNIZED CHARACTER (009)
1. An illegal character was found in the source.

010 100 ERRORS--PROGRAM TERMINATED (010)
1. Only 100 errors are allowed before the compiler stops.

COMPILE-TIME ERRORS

011 A COMMA IS REQUIRED HERE (011)

1. A comma is needed to separate procedure/function names in the SUBPROGRAM compiler option.

012 VARIABLE SPECIFICATION NOT ALLOWED HERE (012)

1. Only SPL procedures are allowed to have a variable number of parameters.

013 IDENTIFIER DOUBLY DEFINED (013)

1. An identifier in a parameter list is a duplicate of another identifier.
2. The procedure/function name is defined earlier and is not a FORWARD procedure/function.
3. The field name of a record is already declared.
4. The identifier is already declared in the current scope.

014 IDENTIFIER NOT DEFINED (014)

1. The identifier is an undeclared variable, constant, procedure or function.
2. The type identifier is undeclared.

015 INVALID VARIABLE USE (015)

1. The control variable of a FOR loop is being modified in the component statement of the FOR loop, e.g. it is the control variable of a nested FOR loop, the left side of an assignment statement, or an actual reference parameter of a user-defined or standard procedure.
2. The variable appears in the variable list of a WITH statement but is not a record type.
3. The identifer appears with subscripts but it is not an array or string.

016 TYPE IDENTIFIER REQUIRED HERE (016)

1. A constant or variable identifier has been used where a type identifier is required.

017 INVALID TYPE IDENTIFIER USE (017)

1. A type identifier has been used where a constant or variable identifier is required.
2. The construct in which the identifier occurs is not legal in this context. This is often an array, record, or set constructor in executable code.

018 A CONSTANT EXPRESSION IS REQUIRED HERE (018)

1. A variable occurs where a constant is required.
2. An expression with variables occurs where a constant expression is required.
3. The expression contains an operator or a standard procedure or function which is not legal in a constant expression.
4. The expression contains constant operands which are not legal, e.g. real, set, or boolean values.

019 INVALID FORWARD TYPE IDENTIFIER DEFINITION (019)

1. The identifier appeared in a forward type definition and is now being declared as something other than a type.

020 BOOLEAN EXPRESSION IS REQUIRED HERE (020)

1. An expression with a boolean result is required here.

021 AN ORDINAL EXPRESSION IS REQUIRED HERE (021)
1. An expression with an ordinal result is required here.

022 INCOMPATIBLE SUBRANGE BOUNDS (022)
1. The type of the lower bound is not compatible with the type of the upper bound in a subrange.

023 AN INTEGER EXPRESSION IS REQUIRED HERE (023)
1. An expression with an integer result is required for the repeat factor in the 'OF' construct in a constructor.

024 LOWER BOUND OF SUBRANGE IS GREATER THAN UPPER BOUND (024)
1. The lower bound is greater than the upper bound in a subrange type declaration.

025 FOUND UNEXPECTED "!" (025)
1. The compiler was not expecting this token and it has been discarded. The token is illegal here or a previous undetectable error has caused the compiler to issue this message, e.g. a semi-colon (;) before ELSE.

026 MISSING "!" (026)
1. The compiler expected this token but it was omitted or badly misspelled. The correct token was inserted.

027 "!" FOUND BEFORE EXPECTED. SOURCE MISSING. (027)
1. The compiler found this token before it was expected. The compiler was able to accept it by inserting dummy conditions or statements.

028 MISUNDERSTOOD SOURCE BEFORE "!" (028)
1. The compiler has discarded some previously accepted source code preceding this token. Either the token is inappropriate but the compiler has been able to accept it by ignoring previous code, or the token is correct and code must now be discarded.

029 " NOT ALLOWED AS A STRING LITERAL DELIMITER (029)
1. A double quote cannot delimit a string literal.

030 OPEN FAILED ON SOURCE FILE "!" (030)
1. The compiler could not open the source file.
2. The compiler could not open the include file.

031 READ FAILED ON SOURCE FILE (031)
1. The compiler could not read the source file.
2. The compiler could not read the include file.

032 EMPTY SOURCE FILE (032)
1. The source file is empty.

COMPILE-TIME ERRORS

033 MISSPELLED RESERVED WORD: "!" (033)
1. The reserved word is misspelled.

034 FORWARD TYPE "!" NOT FOUND (034)
1. The identifier occurs in a pointer type definition but is not subsequently defined.

035 FORWARD PROCEDURE "!" NOT DECLARED (035)
1. A procedure declared with the FORWARD directive is not subsequently defined. The definition may be missing, or the name appearing in the definition may be misspelled.

036 VIOLATION OF PASCAL SCOPING RULES (036)
1. The scope of a Pascal identifier is the entire block in which it is declared. It is not possible to use an identifier from an enclosing level and then to redefine it at the new level.

037 INVALID USE OF "!" IN POINTER DEFINITION (037)
1. A non-type identifier defined on a previous level was used in a pointer type definition.

038 ILLEGAL PASCAL CONSTRUCT (038)
1. The use of the FOR construct with strings is illegal.

039 "!" ACCESSED, BUT NOT INITIALIZED (039)
1. A simple variable appears in an expression, as a value parameter, or in some other accessing reference and it has never appeared in an assigning reference, i.e. as a reference parameter, on the left side of an assignment statement, etc.
2. Some component of a structured variable appears in an accessing reference but no component of that variable has yet appeared in an assigning reference.

040 INVALID STRING TYPE USE (040)
1. The standard type identifier *string* is not used to define a string type.

041 MISSING SEPARATER BETWEEN NUMBER AND IDENTIFIER (041)
1. A character was detected immediately following a number. Pascal requires that a separater (space, comment, End of Line) be inbetween a number and an identifier or reserved word.

060 OPERAND NOT OF TYPE BOOLEAN (060)
1. A non-boolean operand appears with the operator NOT, OR, or AND.

061 WRONG TYPE OF OPERAND FOR ARITHMETIC OPERATOR (061)
1. A non-numeric operand appears with an arithmetic operator.

062 TYPE OF OPERAND NOT ALLOWED WITH OPERATOR (062)
1. An operand of this type cannot be used with this operator.

063 BASE TYPE OF OPERAND AND SET DO NOT AGREE (063)
1. The operand on the left of an IN operator is not type compatible with the set on the right.

064 TYPES OF OPERANDS DO NOT AGREE (064)

1. The operands can be used separately but not at the same time with the operator. For example, <boolean> = <integer>.

065 ASSIGNMENTS CANNOT BE MADE TO FILES (065)

1. An assignment cannot be made to a file or a structured variable with a file type component.
2. Structured constants cannot contain files. Building a structured constant with a type that contains a file is illegal.
3. Variables which contain files cannot be passed as value parameters.

066 ASSIGNMENT TYPE CONFLICT (066)

1. The expression on the right side of an assignment statement is not assignment compatible with the receiving entity on the left.
2. A constant in a constructor is not assignment compatible with the component to which it is being assigned.
3. The subrange type of the expression being assigned does not intersect the type of the receiving entity.

067 TYPE OF EXPRESSION NOT ALLOWED IN SUBRANGE (067)

1. The expression defining a subrange bound is not an ordinal expression.

068 ILLEGAL ASSIGNMENT TARGET (068)

1. An assignment was made to an identifier which is not a non-file variable or a function result, e.g. a declared constant, a set or string type identifier.

069 INVALID CONSTANT EXPRESSION (069)

1. This expression is not legal in a CONST declaration. It is not a constant expression, or it is a constant expression and the results of the arithmetic would be out of range of minint..maxint.

070 ILLEGAL TO ASSIGN TO (070)

1. The identifier denotes an entity which cannot appear on the right side of an assignment statement, e.g. a set or string type identifier.

080 ARRAY INDEX TYPES NOT COMPATIBLE (080)

1. The subscript in an array reference is not compatible with the type of the index in the array declaration.

081 ARRAY ELEMENT TYPES NOT EQUIVALENT (081)

1. Pack and unpack array parameters must have identical component types.

082 INVALID ARRAY SIZE (082)

1. The size of the array is too big for the compiler.
2. In pack or unpack the destination array is not large enough.

083 WRONG NUMBER OF ELEMENTS FOR ARRAY OR STRING CONSTANT (083)

1. While building an array or string constant, more components were specified than declared.
2. All the components were not specified while building an array constant.

COMPILE-TIME ERRORS

084 INVALID INDEX TYPE (084)
1. Index type is not an ordinal type.

085 REFERENCE TYPE MUST BE STRING OR ARRAY (085)
1. Tried to index structure which is not an array or string.

086 MAXIMUM STRING LENGTH MUST BE BETWEEN 1 AND 32767 (086)
1. Tried to declare string with with a maximum length < 1 or > 32767.

087 EXPRESSION FOR MAXIMUM LENGTH MUST BE TYPE INTEGER (087)
1. Tried to declare a string with a non-integer constant expression for the maximum length.

088 INCORRECT NUMBER OF INDICES FOR STRING DECLARATION (088)
1. A string can only have one index in a declaration.
2. No index was supplied in a string declaration.

089 TOO MANY SUBSCRIPTS IN STRING OR ARRAY REFERENCE (089)
1. The number of subscripts in the reference exceeds the number of subscripts in the declaration of the array or string.

090 ILLEGAL CONSTRUCT FOR AN ARRAY OR STRING INDEX (090)
1. A subrange construct was used as an array or string index.

100 INVALID RECORD REFERENCE (100)
1. Record field referenced without specifying a record variable, constant, or function call which returns a record.

101 INVALID FIELD IDENTIFIER (101)
1. The identifier is not one of the fields of the record used in the reference.

102 INVALID TAG TYPE (102)
1. The tag in a new or *dispose* procedure call is not a tag value of the specified record.

103 POINTER OR FILE REQUIRED FOR DEREFERENCE (103)
1. A pointer or file is required in a dereference.

104 POINTER VARIABLE IS REQUIRED HERE (104)
1. *New*, *dispose*, *mark*, and *release* all require a pointer variable as the first parameter.

105 FILES MAY NOT APPEAR IN THE VARIANT PART OF A RECORD (105)
1. Fields of a file type or a structure containing a file type may not appear in the variant part of a record.

120 INVALID BASE TYPE FOR SET (120)
1. The base type of a set is not an ordinal type.

121 ITEM NOT A LEGAL SET ELEMENT (121)
1. Element of a set is not an ordinal type.

122 OPERAND NOT A SET (122)
1. Right operand for an IN operator is not a set.

123 SET ELEMENTS NOT TYPE COMPATIBLE WITH EACH OTHER (123)
1. In an untyped set constructor, this element is not compatible with the first element in the set.

124 SET ELEMENT NOT COMPATIBLE WITH SET TYPE (124)
1. In a typed set constructor, the set element is not assignment compatible with the base type of the set.

125 SET OF THIS SIZE CANNOT BE CONSTRUCTED (125)
1. To construct this set would require more than 2048 16-bit words.

140 BUILDING OF STRUCTURED CONSTANTS NOT ALLOWED HERE (140)
1. A constructor which is not a set constructor occurs outside of a CONST declaration section.
2. A constructor occurs as an element of a set or string constructor.

141 RECORD CONSTANT HAS MISSING FIELD (S) (141)
1. One or more fields missing in a record constructor. This message will also appear when the name of a field is misspelled.

142 DUPLICATE FIELD NAME (142)
1. This field has already been defined in the constructor.

143 FIELD NAME DESIGNATOR NOT ALLOWED HERE
1. The constructor is not a record constructor.
2. This construction <field name>:<expression> appears outside of a record constructor.

144 MISSING FIELD NAME DESIGNATOR (144)
1. The construction <field name>:<expression> is required in a record constructor.

145 TYPE IDENTIFIER REQUIRED HERE (145)
1. The identifier preceding the left square bracket of a constructor is not a type identifier.

146 CONSTRUCT ONLY ALLOWED FOR ARRAYS AND STRINGS (146)
1. <Count> OF <expression> occurs when the constructor is not an array or string constructor.

147 CONSTRUCT ONLY ALLOWED IN CONSTRUCTORS (147)
1. <Count> OF <expression> is used outside of a constructor.

148 SUBRANGE CONSTRUCT ILLEGAL EXCEPT IN SET CONSTRUCTOR (148)
1. A subrange construct was used in a string declaration or a non set structured constant.

149 TOO BIG STRUCTURED CONSTANT (149)
1. A structured constant has a limit of 16383 words.

160 INVALID BASE TYPE FOR FILE (160)
1. The component type of a file may not be a file or a structure with a file type component.

161 TEXTFILE VARIABLE IS REQUIRED HERE (161)
1. The pre-defined procedure or function in question may only be used with a file of type text.

162 TEXTFILE NOT ALLOWED HERE (162)
1. The standard procedure or function in question may not be used with a file of type text.

163 INVALID TYPE FOR A PROGRAM PARAMETER (163)
1. An identifier in the program parameter list has not been declared as a file variable, or a variable of type PAC, *string*, or *integer*.

164 VARIABLE IS REQUIRED HERE (164)
1. A variable is required as the target for reading from a file or a string.

165 DEFAULT FILE INPUT MUST BE IN PROGRAM PARAMETER LIST (165)
1. The file variable in a standard procedure or function call was defaulted to INPUT, but INPUT was not declared in the program parameter list.

166 DEFAULT FILE OUTPUT MUST BE IN THE PROGRAM PARAMETER LIST (166)
1. The file variable in a standard procedure or function call was defaulted to OUTPUT, but OUTPUT did not appear in the program parameter list.

167 FORMAT EXPRESSION ALLOWED ONLY FOR TEXTFILES (167)
1. A formatted output expression may only occur when writing to a textfile or a string.

168 INTEGER VALUE IS REQUIRED HERE (168)
1. The expressions specifying the field width and the number of decimal digits for an output expression are not type *integer* or an integer subrange.

169 SECOND FORMAT VALUE ALLOWED ONLY FOR REAL OR LONGREAL (169)
1. The format value which specifies the number of decimal digits in an output expression is only legal for output values of type *real* or *longreal*.

190 THIS PROGRAM PARAMETER WAS UNDECLARED: "I" (190)
1. The identifier appeared in the program parameter list but was never declared.

191 DUPLICATE PROGRAM PARAMETER (191)
1. There is more than one PARM parameter or more than one INFO parameter in a program parameter list.

192 PARAMETER "I" DOES NOT MATCH POSSIBLE SPL TYPES (192)
1. The Pascal type of the parameter does not correspond to an acceptable SPL type.

193 PARAMETER "!" DOES NOT MATCH INTRINSIC PARM TYPE (193)

1. The Pascal type of the the parameter does not match the parameter type required by the intrinsic.

194 MISSING FUNCTION RETURN SPECIFICATION (194)

1. The return type is not specified in the function heading.

195 INVALID PARAMETER TO HALT (195)

1. The optional parameter to *halt* is not type *integer* or an integer subrange.

196 THIS INTRINSIC MAY NOT BE USED AS A FUNCTION (196)

1. The specified intrinsic does not return a result and, therefore, cannot be declared as a function.

197 ELEMENTS OF PACKED STRUCTURES CANNOT BE PASSED BY VAR (197)

1. Elements of packed arrays or records may not be passed to a routine expecting a reference parameter.

198 EMPTY PARAMETER MAY NOT BE USED HERE (198)

1. Actual parameters may only be omitted for EXTERNAL SPL VARIABLE procedures or for intrinsics which are option variable intrinsics.

199 PROCEDURE NOT DECLARED (199)

1. The identifier used in the procedure call has not been declared, or it is not a procedure name.

200 PARAMETER "!" MUST BE VAR PARAMETER. (200)

1. The parameter in the intrinsic declaration was specified as a value parameter, but the intrinsic requires a reference parameter.

201 PARAMETER "!" MUST BE VALUE PARAMETER (201)

1. The parameter in the intrinsic declaration was specified as a reference parameter, but the intrinsic requires a value parameter.

202 INVALID USE OF PROCEDURE OR FUNCTION IDENTIFIER (202)

1. A procedure identifier appears as a function call.
2. A function identifier appears as a procedure call.
3. A valid identifier mistakenly appears as a function or procedure identifier.

203 INCONSISTENT DEFINITION OF FORWARD PROCEDURE OR FUNCTION (203)

1. The definition of a procedure declared FORWARD is a function. The definition of a function declared FORWARD is a procedure.
2. The ALIAS in the definition differs from the ALIAS in the FORWARD declaration of a procedure or function.
3. A FORWARD declaration is already provided for a function or procedure now declared FORWARD, EXTERNAL, or INTRINSIC.

204 INVALID DIRECTIVE (204)

1. EXTERNAL, EXTERNAL SPL, EXTERNAL SPL VARIABLE, EXTERNAL FORTRAN, EXTERNAL COBOL, FORWARD, and INTRINSIC are the only legal directives.

205 INVALID LANGUAGE SPECIFICATION (205)

1. The language specified was not FORTRAN, SPL, or COBOL.
2. A language cannot be specified with the FORWARD or INTRINSIC directives.

206 INCORRECT NUMBER OF PARAMETERS (206)

1. The number of actual parameters given is too few or too many for the procedure or function.

207 UNMATCHED PARAMETERS IN FORWARD (207)

1. Parameters in the definition of a procedure or function declared FORWARD do not match the parameters of the original heading.

208 ACTUAL PARAMETER NOT COMPATIBLE WITH FORMAL PARAMETER (208)

1. This actual reference parameter is not type identical with the formal reference parameter in a user-defined function or procedure.
2. This actual value parameter is not assignment compatible with the formal value parameter in a user-defined function or procedure.
3. The type of this actual reference parameter is not convertible to the SPL type of the formal reference parameter of the intrinsic.
4. The type of this actual value parameter is not convertible to the SPL type of the formal value parameter of the intrinsic.
5. This actual reference parameter to a standard function or procedure is not type identical with the formal reference parameter.
6. This actual value parameter to a standard function or procedure is not assignment compatible to the required type.
7. The parameter of the standard *sqr* function is an integer subrange type with a lower bound greater than the square root of *maxint*, or an upper bound less than the negation of the square root of *maxint*. In either case, an integer overflow is possible at run time.

209 NO FURTHER CASE CONSTANT PARAMETERS ALLOWED TO NEW (209)

1. The pointer parameter to *new* points to a record that has no additional nested variant parts.
2. The pointer parameter to *new* points to a record that does not have a variant part.
3. The pointer parameter to *new* points to a structure which is not a record.

210 NO FURTHER CASE CONSTANT PARAMETERS ALLOWED TO DISPOSE (210)

1. The pointer parameter to *dispose* points to a record that has no additional nested variant parts.
2. The pointer parameter to *dispose* points to a record that does not have a variant part.
3. The pointer parameter to *dispose* points to a structure which is not a record.

211 NO FURTHER PARAMETERS ALLOWED TO MARK (211)

1. More than one pointer parameter in a call to *mark*.

212 NO FURTHER PARAMETERS ALLOWED TO RELEASE (212)

1. More than one pointer parameter in a call to *release*.

213 VALUE PARAMETER MAY NOT CONTAIN FILE COMPONENT (213)

1. This value formal parameter is a file or a structured type with a file type component. This is equivalent to assigning to a file.

214 FUNCTION TYPE MAY NOT CONTAIN FILE COMPONENT (214)

1. This function return type is a file or a structured type that contains a file type component. This is equivalent to assigning to a file.

215 COMPILER LEVEL WRONG -- PROBABLY UNMATCHED "END" (215)

1. This occurrence of END cannot match a BEGIN because all compound statements have been terminated. The compiler disregards the extraneous END.

216 BAD CONSTANT PARAMETER (216)

1. This parameter to *succ* is a constant value equal to the maximum value of an ordinal type.
2. This parameter to *pred* is a constant value equal to the minimum value of an ordinal type.
3. This string constant parameter to *binary, octal,* or *hex* contains an invalid character, or represents a value outside the range *minint..maxint.*
4. This parameter to *abs* is a constant value equal to *minint* but *abs* (*minint*) > *maxint.*
5. This parameter to *chr* is a constant value outside the range 0..255.

217 PROCEDURE OR FUNCTION NOT IN INTRINSIC FILE (217)

1. An incorrect SPLINTR file was specified prior to the declaration of the procedure or function.
2. The intrinsic name differs slightly from the procedure or function name declared INTRINSIC. The ALIAS option should be used, or the spelling of the ALIAS parameter corrected.
3. The procedure has never been put into the SPLINTR file.

218 SPLINTR FILE NOT CHECKED (218)

1. Due to a prior error, the SPLINTR file was never opened. Thus, no attempt was made to look up this procedure or function.

219 "STRING" IS NOT ALLOWED AS A VALUE PARAMETER (219)

1. A string formal value parameter must have a specified maximum length.

220 FUNCTION "!" NOT ASSIGNED TO (220)

1. A function of a simple type has no assigning reference to the result in the function body.
2. A function of a structured type has no assigning reference to any component of the result in the function body.

221 DECLARED FUNCTION TYPE DOES NOT MATCH INTRINSIC TYPE (221)

1. The Pascal type of the return of a function declared INTRINSIC does not match the SPL type of the value returned by the intrinsic.

COMPILE-TIME ERRORS

222 VARIABLE PARAMETER REQUIRED HERE (222)

1. An expression appears as an actual reference parameter instead of a variable.
2. A constant appears as an actual reference parameter instead of a variable.
3. A component of a structured constant appears as an actual reference parameter instead of a variable.

223 ILLEGAL PARAMETER FORM (223)

1. The integer parameter to a string procedure/function is not compatible with a 16 bit integer.
2. The actual parameter is a procedure or function identifier, but the corresponding formal parameter is not a procedure or function heading.
3. The parameters of the actual procedural or functional parameter are not congruent with the parameters of the formal procedural or functional parameter.
4. The parameter of a call to *waddress* or *sizeof* is a component of a packed structure.
5. The parameter of a call to *baddress* is a component of a packed structure other than a PAC.
6. The third parameter of a call to *assert* is not a procedure identifier, or the parameter of such a procedure is not an integer value parameter.

224 SYSTEM ADDRESSING LIMIT EXCEEDED (224)

1. . Q-minus addressing for parameters or function results is exceed. (Q-64)
2. The storage limit for variables at run time is exceeded. (DB+255; Q+127 or storage exceeded.)

230 INVALID CONTROL VARIABLE IN FOR STATEMENT (230)

1. The control variable of the FOR loop is a record field.
2. The control variable of the FOR loop is defined in a scope containing the current scope.
3. The control variable of the FOR loop is a formal parameter of a procedure or function containing the FOR statement.
4. The identifier used as the control variable of the FOR is not a variable.

231 CONTROL VARIABLE NOT AN ORDINAL TYPE (231)

1. The control variable of the FOR loop is not an ordinal type.

232 EXPRESSION NOT COMPATIBLE WITH CONTROL VARIABLE (232)

1. The expressions for the initial and final values are not type compatible with the control variable of a FOR loop.

233 INITIAL AND FINAL EXPRESSIONS NOT COMPATIBLE (233)

1. The types of the expressions for the initial and final values of the FOR loop are not type compatible.

250 DUPLICATE CASE LABEL (250)

1. The case label is the same as a case label that appeared previously.
2. The case label is contained in a previous case label subrange.
3. The case label subrange contains at least one case label that appeared previously.

251 CASE LABEL OF INCORRECT TYPE (251)

1. The type of the case label is not the same as the type of of the tag in the select expression.

252 CASE LABEL TYPE NOT SAME AS PREVIOUS CASE LABEL (252)

1. There was an error in the tag type or select expression and the case labels are checked against each other. The type of the current case label does not match the type of previous case labels.

270 INVALID LABEL - MUST BE AN INTEGER BETWEEN 0 AND 9999 (270)

1. This label is not an integer.
2. A colon (:) appears or was inserted by the compiler where no label was desired.

271 LABEL HAS NOT BEEN DECLARED (271)

1. This label marks a statement but never appeared in a LABEL declaration for this block.

272 LABEL DECLARED MORE THAN ONCE (272)

1. This label already appeared in this LABEL section or in a LABEL section in an enclosing scope.

273 SAME LABEL NOT ALLOWED ON MORE THAN ONE STATEMENT (273)

1. This label has already marked a statement.

274 LABEL "!" NOT USED (274)

1. The label appears in a LABEL declaration but is never used to mark a statement.

275 LABEL REFERENCED BY GOTO OUTSIDE STRUCTURED STATEMENT (275)

1. This label appears in a component statement of a structured statement and was previously referenced by a GOTO statement:
   (a) preceding the structured statement.
   (b) in a preceding component statement of the same structured statement.
   (c) contained in an inner procedure or function.

276 GOTO REFERENCES LABEL INSIDE STRUCTURED STATEMENT (276)

1. The label referenced in a GOTO statement appears in a component statement of a structured statement and the GOTO statement appears:
   (a) after the structured statement.
   (b) in a later component statement of the same structured statement.

400 INVALID FILENAME (400)

1. The filename given in the INCLUDE or SPLINTR option is not a legal MPE filename.

401 ILLEGAL NAME IN ALIAS OR SUBPROGRAM OPTION (401)

1. The procedure or function name in an ALIAS option is not a valid identifier.
2. The procedure or function name in a SUBPROGRAM option is not a valid Pascal identifier.

402 NOT A LEGAL SEGMENT NAME (402)

1. The segment name for a SEGMENT option is not legal.

COMPILE-TIME ERRORS

403 IF EXPRESSION CAN NOT BE EVALUATED (403)

1. The expression in a $IF has a syntax error in it.

404 UNMATCHED ENDIF FOUND (404)

1. An ENDIF compiler option was found without a preceeding IF option. This may happen if the compiler rejects an IF because it was out of place, as well as the user not putting in the IF.

405 A BOOLEAN EXPRESSION IS REQUIRED INSIDE STRING (405)

1. A blank string was found as part of a IF.

406 EXPECTED TRUE/FALSE AFTER '=' (406)

1. Misspelled true/false after '=' in $SET.
2. Missing true/false after '=' in $SET.

408 UNMATCHED $ENDIF OR $ELSE FOUND (408)

1. An ENDIF/ELSE compiler option was found without a preceeding IF option. This may happen if the compiler rejects an IF because it was out of place, as well as the user not putting in the IF.

409 EXCEEDED MAXIMUM NESTING LEVEL FOR $IF (409)

1. The nesting of $IF exceeded the maximum allowable nesting level.

410 ILLEGAL IDENTIFIER IN $SET (410)

1. Identifier is misspelled.
2. Expected an identifier and one was not found.

COMPILE-TIME ERRORS

425 CATASTROPHIC COMPILER ERROR !, COMPILE TERMINATED (425)

(1..999) A run-time error was detected by the run-time support library during compiler execution.

(1000..1015) A run-time error was detected in an arithmetic operation during compiler execution.

(2000..2999) A run-time error was detected by a system intrinsic during compiler execution.

426 SYSTEM RESOURCE EXHAUSTED !, COMPILE TERMINATED (426)

(1) The compiler ran out of space in the heap.

(2) The compiler ran out of space in one of its data segments.

427 PROCEDURE CALL OVERFLOW - COMPILE TERMINATED (427)

1. The maximum number of different procedures/functions and private procedures/functions which may be called from a single RBM is 254. This limit has been exceeded.

428 CODE SEGMENT OVERFLOW - COMPILE TERMINATED (428)

1. The maximum number of words permitted in a single RBM (16838) has been exceeded.

451 UNABLE TO OPEN USL FILE (451)

1. FOPEN error on USL file. Unable to open either an old, temporary, or new USL file.

452 UNABLE TO SAVE USL FILE PERMANENT - SAVED TEMPORARY (452)

1. A new USL file was unable to be saved as a permanent file. An attempt was made to save the file as a temporary file.

453 UNABLE TO CLOSE USL FILE (453)

1. A new USL file, which was unable to be saved as a permanent file, cannot be saved as a temporary file and will be lost when compilation is complete.
2. An error occurred when closing $NEWPASS, $OLDPASS, or an existing temporary or permanent USL file.

454 USL FILE ACCESS ERROR (454)

1. A file system error occured while trying to access the USL file with FCHECK.

455 USL FILE ACCESS ERROR (455)

1. A file system error occured while trying to access the USL file with FGETINFO.

456 USL FILE READ ERROR (456)

1. A file system error occured while trying to access the USL file with FREAD.

457 USL FILE WRITE ERROR (457)

1. A file system error occured while trying to access the USL file with FWRITE.

459 OVERFLOW ON USL FILE (459)

1. The available information length (AIL) of the USL file is less than the size of the header to be placed in the information area of the USL file. The default size of the USL file is 1023 records. This may be increased by pre- building the file or by using a file equation. Also, a USL file may have several inactive procedures which can be removed using the USLINIT compiler option or the CLEANUSL command in the Segmenter.

460 DIRECTORY OVERFLOW ON USL FILE (460)

1. The available directory length (ADL) of the USL file is less than the size of the entry to be placed in the directory.

461 PARSER STACK OVERFLOW - TOO MANY NESTED CONSTRUCTS (462)

1. An internal compiler limit on nested structures has been reached. The most common cause is a long list of ELSE-IF's.

WARNINGS

500 OPTION NOT YET IMPLEMENTED (500)

1. This compiler option is not yet implemented.

501 UNRECOGNIZED COMPILER OPTION (501)

1. A compiler option with this name is not recognized by Pascal/3000.

502 THIS OPTION IS NOT ALLOWED HERE (502)

1. The option appears in an illegal location in source code. For example, the GLOBAL option appears anywhere except before the PROGRAM heading.

503 TEXT AFTER INCLUDE OR SKIPTEXT IGNORED (503)

1. Anything on the source line after INCLUDE was ignored.
2. Anything on the source line after a SKIP—TEXT ON was treated as a comment.

504 INTEGER OUT OF RANGE, VALUE NOT CHANGED (504)

1. LINES requires an integer > 20
2. WIDTH requires an integer in the range 10..132.
3. CHECK—ACTUAL—PARM and CHECK—FORMAL—PARM require an integer in the range 0..3.

505 STRING PARAMETER IS REQUIRED, OPTION IGNORED (505)

1. This option requires information in a string literal parameter.

507 BOTH GLOBAL AND EXTERNAL NOT ALLOWED (507)

1. The option GLOBAL occurred after the option EXTERNAL was specified. Since only one is allowed, GLOBAL was ignored.
2. The option EXTERNAL occurred after the option GLOBAL was specified. Since only one is allowed, EXTERNAL was ignored.

COMPILE-TIME ERRORS

508 A '$' IS REQUIRED HERE - ONE INSERTED (508)

1. Compiler option doesn't end with a $ on the same line.

509 EXPRESSION WILL CAUSE A RUN-TIME OVERFLOW (509)

1. The result of an expression will exceed maxint at run time. This is detected for:
   (a) +, -, * when the types of the operands are such that the expression will go over. For example: VAR A: *maxint*-10..*maxint*; Then the expression A + A would never be less than 2 * *maxint* - 10, which is > *maxint*.
   (b) -*minint*.
   (c) the addition, substraction, or multiplication of of two constants resulting in an overflow.

510 EXPRESSION WILL CAUSE A RUN-TIME UNDERFLOW (510)

1. The result of an expression will be less than *minint* at run time. This is detected for:
   (a) +, -, * when the types of the operands are such that that the expression will go under. For example: VAR A: *maxint* - 10..*maxint*; B: *minint*..*minint* + 10 Then the expression B - A would be less than *minint* + 10 - *maxint*, which is < *minint*.
   (b) the addition, substraction, or multiplication of two constants resulting in an underflow.

511 MOD DIVISOR WILL CAUSE A RUN-TIME ERROR (511)

1. In an expression A MOD B, B will be <= 0 at run time.
2. In a constant expression A MOD B, B is <= 0.

512 RUN-TIME DIVISION BY ZERO (512)

1. In an expression A DIV B, B = 0.
2. In a constant expression A DIV B, B = 0.

513 EMPTY INCLUDE FILE (513)

1. The INCLUDE file had no text in it.

514 $ NOT ALLOWED IN INFO PARAMETER (514)

1. The INFO parameter of a :PASCAL, :PASCALPREP, or :PASCALGO command is interpreted as a compiler option with the $ assumed as the leading and trailing character. The $ cannot appear in the INFO string itself.

515 NO DISC SPACE FOR XREF (515)

1. An MPE file error occured trying to open the file needed to do the cross reference. This could be any MPE file error, with OUT OF DISC SPACE being the most likely. A temporary file with the name PASXRFdd, where d is a digit, is another possible cause.

516 NO VARIANT FOR TAG VALUE (516)

1. A *new* was called specifiying a tag constant which did not appear in the case list in the variant part. The maximum space for the record is allocated.

517 THIS FEATURE IS HP STANDARD PASCAL (517)

1. ANSI is ON or STANDARD—LEVEL is set to ANSI and this feature is an HP Standard Pascal extension of ANSI Pascal.

518 THIS FEATURE IS HP3000 PASCAL (518)

1. ANSI is ON or STANDARD—LEVEL is set to HP or ANSI and this feature is unique to the HP3000 implementation of Pascal.

519 BOOLEAN EXPRESSION FOLDED TO A CONSTANT (519)

1. The compiler has folded an expression with IN, AND, or OR and constant operands or, in the case of IN, with a left operand which is a constant appearing the set list.
2. The compiler has folded an expression with =, <>, <=, >=, or > and operands which are non-set constants.
3. With PARTIAL—EVAL ON, the compiler has folded an expression with OR when TRUE is an operand, or an expression with AND when FALSE is an operand.

520 NON-OVERLAPPING TYPES - EXPRESSION FOLDED (520)

1. Two sets with ranges that don't overlap were intersected. The compiler folded the expression to the empty set.
2. An arithmetic comparison was done with operands of types with ranges that do not overlap. The compiler folded the expression. For example, if A: 0..3 and B: 5..7, then A = B is folded to *false*.

521 BODY OF FOR LOOP WILL NEVER EXECUTE (521)

1. Values of the initial and final expressions will prevent the body of the FOR loop from ever executing.
2. Non-overlapping subranges for the types of the initial and final expressions prevent the body of the FOR loop from ever executing.

522 CASE LABEL NOT WITHIN TAG OR SELECT EXPRESSION RANGE (522)

1. The case label value or subrange is not within the range of the tag type and can never be specified in a call to *new* or assigned to the tag field.
2. The case label value or subrange is not within the range of the select expression and can never be selected.

523 INTEGER CONSTANT IS REQUIRED - OPTION IGNORED (523)

1. This compiler option requires an integer parameter, e.g. WIDTH. The compiler has ignored this option.

524 SUBPROGRAM "!" SPECIFIED, BUT NOT FOUND (524)

1. A procedure or function name specified in the SUBPROGRAM option was not found in this source.

COMPILE-TIME ERRORS

525 ANY EXTERNAL GOTO TO THIS LABEL IS AN ERROR (525)

1. This label marks a component statement of a structured statement. This label cannot be referenced by a GOTO statement contained in an external procedure or function, but that error will not be detected until the program is prepared or executed.

526 EXPRESSION FOLDED TO THE EMPTY SET (526)

1. The compiler has determined that a set expression results in an empty set and folded that expression to empty. This warning appears in case the user expected side effects or made some kind of error which caused the folding. Folding occurs when an intersection is performed with the empty set, the empty set occurs on the left side of the set difference operator, or two empty sets appear in a set operation.

527 'ON' OR 'OFF' IS REQUIRED HERE (527)

1. The word ON or OFF is required after this compiler option name, e.g. LIST.

528 PREVIOUS VERSION OF "I" INACTIVATED (528)

1. A procedure or function by the same name already exists in the USL file and has been inactivated.
2. If PRIVATE_PROC was ON, then two level 1 procedure or function names are not unique within the first 15 characters or a copy from a previous compilation is being replaced.
3. If PRIVATE_PROC was OFF, then either duplicate non-level 1 procedure or function names exist (i.e. they are not unique within 15 characters) or duplicate procedure or function names have been introduced due to separate compilation of procedures or functions with names which are identical within the first 15 characters.

529 USL FILE DIRECTORY NOT VALIE - INITIALIZED (529)

1. If the USL file is a new file, the directory has been initialized.
2. If the USL file is a old file, the directory information was not consistent with the USL file structure and has been initialized. Any information that may have been contained in the USL file is no longer accessible.

530 EXPRESSION WILL CAUSE A RUN-TIME SET RANGE ERROR (530)

1. Evaluation of a set construction in which an element of the set list will necessarily fall outside the bounds of the set construction will cause this error.

531 BYTE TO WORD ADDRESS CONVERSION HERE (531)

1. A byte oriented variable being passed to an intrinsic expecting a word oriented reference parameter will result in this warning. Pascal/3000 will properly convert all even byte addresses to word addresses, but changes an odd byte address to the address of the word containing the odd byte.

532 PASWKSP IS NOT A VALID TSAM ROOT FILE (532)

1. The actual file designator for the formal designator PASWKSP is not a TSAM root file.

533 BAD FONT OPTION GIVEN (533)

1. The call to FDeviceControl returned an error condition.

534 CONTROL VARIABLE HAS BEEN ASSIGNED TO NONLOCALLY (534)

1. The control variable may be modified by a nonlocal reference from a routine invoked in the body of the for loop.

535 "!" ACCESSED, BUT NOT INITIALIZED (535)

1. A simple variable appears in an expression, as a value parameter, or in some other accessing reference and it has never appeared in an assigning reference, i.e. as a reference parameter, on the left side of an assignment statement, etc.
2. Some component of a structured variable appears in an accessing reference but no component of that variable has yet appeared in an assigning reference.

RUN-TIME ERRORS

APPENDIX D

This appendix lists the annotated run-time error messages for Pascal/3000. These messages are numbered 600 and above. The messages together with the notes are available on line in the file PASCAT.PUB.SYS. The programmer may trap any run-time Pascal error using the XLIBTRAP intrinsic (see Section 10).

600 INSUFFICIENT HEAP AREA TO ALLOCATE VARIABLE (PASCERR 600)

1. Heap cannot be expanded to allocate a dynamic variable because the MAXDATA value for the program will be exceeded.
2. Heap cannot be expanded to allocate a dynamic variable because the system configuration MAXDATA value will be exceeded.

601 INVALID DISPOSE PARAMETER (PASCERR 601)

1. The pointer parameter to *dispose* in NIL.
2. The pointer parameter to *dispose* does not identify any area allocated by *new*.
3. The pointer parameter to *dispose* identifies an area previously deallocated by *release*.

602 REPEATED USE OF DISPOSE ON GIVEN PARAMETER (PASCERR 602)

1. The pointer parameter to *dispose* identifies an area previously deallocated by *dispose*.

603 DISPOSE PARAMETER ALLOCATED AS DIFFERENT VARIANT (PASCERR 603)

1. The pointer parameter to *dispose* identifies an area allocated by *new* with a different sequence of case constants.
2. The pointer parameter to *dispose* includes case constants, but it identifies an area allocated by *new* without any case constants.
3. The pointer parameter to *dispose* does not include case constants, but it identifies an area allocated by *new* with case constants.

604 DISPOSE PARAMETER CONTAINS AN OPEN SCOPE (PASCERR 604)

1. The pointer parameter to *dispose* identifies an area containing an actual variable parameter, an element of the record variable list of a WITH statement, or both.

605 INVALID RELEASE PARAMETER (PASCERR 605)

1. The parameter to *release* was not set by a previous call to *mark*.
2. The parameter to *release* was set by a call to *mark*, but a previous call to *release* has been made with this parameter.
3. The parameter to *release* was set by a call to *mark*, but that call to *mark* was preceded by a call to *mark* with a different parameter that has already been used as a parameter to *release*.

606 RELEASE PARAMETER ENCLOSES AN OPEN SCOPE (PASCERR 606)

1. The parameter to *release* identifies an area containing an actual variable parameter, an element of the record variable list of a WITH statement, or both.

607 RELEASE PARAMETER ENCLOSES GETHEAP AREA (S) (PASCERR 607)

1. The parameter to *release* identifies an area containing areas the user allocated with the GETHEAP procedure but has not yet deallocated with the RTNHEAP procedure.
2. The parameter to *release* identifies an area containing areas a subsystem allocated with the GETHEAP procedure but has not yet deallocated with the RTNHEAP procedure.

RUN-TIME ERRORS

608 HEAP INTEGRITY LOST / HEAP DATA LOST (PASCERR 608)

1. The internal data structures of the heap have become inconsistent. The most likely causes are:
   a) A field has been assigned to in a variant different than the one specified in a call to *new*.
   b) A pointer to a disposed area, i.e. a dangling pointer, has been dereferenced in an assignment.
   c) An SPL routine has directly accessed the DL-DB area outside of a region allocated by the GETHEAP procedure.
   d) The DLSIZE intrinsic has been called.
   e) The RTNHEAP procedure was unable to return an area.

620 VALUE NOT WITHIN SUBRANGE (PASCERR 620)

1. The value of an ordinal expression is outside of the subrange of the target of an assignment statement.
2. The value of an ordinal expression appearing as an actual parameter is outside the subrange of the formal value parameter.
3. The value of an ordinal expression appearing in an array selector is outside of the subrange of the index type.

621 NO CASE LABEL FOR SELECTOR VALUE (PASCERR 621)

1. The value of the case select expression does not match an of the specified case constants and no OTHERWISE clause appears.

622 INVALID POINTER (PASCERR 622)

1. A pointer with the value of NIL was dereferenced.
2. A pointer with an undefined value was dereferenced.
3. A pointer set by *mark* was dereferenced.
4. A pointer identifing an area previously deallocated was dereferenced.

623 VALUE OF PRED UNDEFINED (PASCERR 623)

1. The minimum value of an ordinal type or subrange was the parameter to *pred*. The result is undefined.

624 VALUE OF SUCC UNDEFINED (PASCERR 624)

1. The maximum value of an ordinal type or subrange was the parameter to *succ*. The result is undefined.

625 SET RANGE ERROR (PASCERR 625)

1. An attempt was made to assign a set to a set variable when the set contains an element not within the set range of the variable.
2. An attempt was made to pass a set to a formal parameter when the set contains an element not within the set range of the parameter.

626 ATTEMPT TO DO MOD BY NEGATIVE VALUE (PASCERR 626)

1. An attempt was made to perform the MOD operation when the right operand is negative.

640 BAD PROCEDURAL PARAMETER (PASCERR 640)

1. A non-level 1 procedure or function was passed as a procedural or functional parameter to an external, non-Pascal routine.

650 STRING OVERFLOW (PASCERR 650)

1. An attempt was made to index beyond the maximum length of the string.

651 STRING INDEX EXCEEDS CURRENT LENGTH (PASCERR 651)

1. An attempt was made to index beyond the current length of the string.

652 DESIGNATED CHARACTER POSITION (S) OUTSIDE STRING (PASCERR 652)

1. The specified offset is greater than the current length of the string.

653 DESIGNATED CHARACTER POSITION (S) OUTSIDE PAC (PASCERR 653)

1. The specified offset is greater than the upper bound of the PAC.

654 ATTEMPT TO READ PAST END OF STRING (PASCERR 654)

1. Attempt was made to read beyond the maximum length of the string.

655 INVALID NUMBER OF CHARACTERS SPECIFIED (PASCERR 655)

1. The number of characters to be copied in the predefined procedure STRMOVE is less than zero.

670 INVALID CHARACTER FOR HEX DIGIT (PASCERR 670)

1. The character was not in the set 0..9, A..F, or a..f.

671 INVALID CHARACTER FOR OCTAL DIGIT (PASCERR 671)

1. The character was not in the set 0..7.

672 INVALID CHARACTER FOR BINARY DIGIT (PASCERR 672)

1. The character was not in the set 0..1.

673 NUMBER OF SIGNIFICANT DIGITS CAUSED OVERFLOW (PASCERR 673)

1. The number of significant digits was more than 32 for the standard function *binary*, 11 for the function *octal*, or 8 for the function *hex*.

PASCAL FILE ERRORS

Using the XLIBTRAP intrinsic, it is possible to trap run-time file errors. Certain of these errors actually correspond to errors detected by MPE File System intrinsics which Pascal invokes to perform I/O operations. For errors of this type, a call to the intrinsic FCHECK will yield the particular MPE File System error. For other errors, however, FCHECK will not return meaningful results. In the following annotations, the advisability of using FCHECK is indicated.

690 OPEN ERROR: PHYSICAL FILE COULD NOT BE CLOSED (PASCERR 690)

1. An attempt was made to open a file, but the logical file was already associated with a physical file and this physical file could not be closed prior to opening another physical file. FCHECK may be called.

691 OPEN ERROR: MISMATCH OF LOGICAL/PHYSICAL FILES (PASCERR 691)

1. The characteristics of the logical file are not compatible with those of the associated physical file. For example, a physical file with variable length records may not be opened for direct access. FCHECK should not be called.

692 FILE OPEN ERROR (PASCERR 692)

1. An error occurred when FOPEN was called to open the file. FCHECK with 0 as the file number will give the MPE File System error number.

693 ERROR OCCURRED WHILE READING FROM FILE (PASCERR 693)

1. MPE detected an error during a call to FREAD. FCHECK may be called.

694 ATTEMPT TO READ PAST EOF (PASCERR 694)

1. The current position is past the last component of the file. FCHECK should not be called.

695 ERROR OCCURRED WHILE WRITING TO FILE (PASCERR 695)

1. MPE detected an error during a call to FWRITE. FCHECK may be called.

696 WRITE ON READ-ONLY FILE (PASCERR 696)

1. An attempt was made to perform an output operation on a file opened for input access only. FCHECK should not be called.

697 OPEN ERROR: UNABLE TO INITIALIZE POSITION (PASCERR 697)

1. A request was made to open a logical file already associated with the physical file and MPE was unable to reposition the file pointer at the beginning of the physical file. FCHECK may be called.

698 OPEN ERROR: UNABLE TO EMPTY FILE (PASCERR 698)

1. *Rewrite* was unable to empty the file of its previous contents. FCHECK may be called.

699 UNABLE TO CLOSE FILE (PASCERR 699)

1. The file could not be closed as requested. FCHECK may be called. If the file system error returned by FCheck is FSERR 72 (Invalid File Number), then the likely reasons for the error are either 1) FClose was used to close the file at the system level, thereby making the file number invalid when Pascal attempts to close it or 2) all or part of the File Control Block for the file has been overwritten by the user program.

700 ERROR OCCURRED DURING DIRECT ACCESS I/O (PASCERR 700)

1. MPE detected an error during a file operation on a direct access file. FCHECK may be called.

701 ILLEGAL CHARACTER IN NUMBER (PASCERR 701)

1. An attempt was made to read a number from a text file but an illegal character was found before a valid number. FCHECK should not be called.

702 INPUT VALUE OVERFLOW (PASCERR 702)

1. The numeric value read is too large for the type of the variable. FCHECK should not be called.

703 ATTEMPT TO WRITE PAST PHYSICAL BOUNDS OF FILE (PASCERR 703)

1. THe current record position is past the physical limit of the file. FCHECK should not be called.

704 READ ATTEMPTED FROM OUTPUT FILE (PASCERR 704)

1. An attempt was made to perform an input operation on a file opened only for output. FCHECK should not be called.

705 FILE NOT OPENED FOR DIRECT ACCESS (PASCERR 705)

1. An attempt was made to perform a direct access file operation on a file not opened for direct access with the *open* procedure. FCHECK should not be called.

706 FILE NOT OPENED (PASCERR 706)

1. An attempt was made to access an unopened file. FCHECK should not be called.

707 INVALID OPEN OPTION (PASCERR 707)

1. An invalid open option was found in the third parameter to one of the file opening procedures. FCHECK should not be called.

708 COULD NOT OPEN FILE FOR APPEND ACCESS (PASCERR 708)

1. Some physical files cannot have their record pointers automatically positioned at the end of the file when opened for append access. This error indicates that Pascal could not successfully prepare the file for append access. FCHECK may be called.

709 FIELD WIDTH LESS THAN ZERO (PASCERR 709)

1. The field width in a formatted write of a non-numeric expression was less than zero. FCHECK should not be called.

710 FIELD WIDTH LESS THAN 1 (PASCERR 710)

1. The field width in the formatted write of a numeric expression was less than 1. FCHECK should not be called.

711 NO DIGITS AFTER DECIMAL POINT (PASCERR 711)

1. No digits occur after the decimal point in a formatted write of a real or longreal expression. FCHECK should not be called.

712 INPUT VALUE UNDERFLOW (PASCERR 712)

1. The value read is too small to be represented in the variable. FCHECK should not be called.

713 FIELD TOO SMALL TO PRINT NUMBER (PASCERR 713)

1. An internal PASCAL error that should be reported to Hewlett-Packard. FCHECK should not be called.

714 INVALID CLOSE OPTION (PASCERR 714)

1. An invalid disposition option was found in the second parameter to *close*. FCHECK should not be called.

715 INVALID ENUMERATED IDENTIFIER FOR INPUT (PASCERR 715)

1. An attempt was made to read an enumerated identifier from a textfile, but either a valid Pascal identifier was not found or the identifier found was not an identifier of that enumerated type. FCHECK should not be called.

716 CANNOT WRITE ENUMERATED VALUE (PASCERR 716)

1. An attempt was made to write an enumerated variable to a textfile, but the current ordinal value of the variable is not within the range of the enumerated type. FCHECK should not be called.

717 INVALID BOOLEAN READ (PASCERR 717)

1. An attempt was made to read a boolean value from a textfile, but a non-boolean value was found. FCHECK should not be called.

718 INVALID FLOATING POINT NUMBER REPRESENTATION (PASCERR 718)

1. An attempt was made to read a real or longreal number from a textfile, but an invalid floating point number was found. FCHECK should not be called.

UNDETECTED ERRORS

APPENDIX
E

The following errors are currently undetected by the compiler at compile time or by the system at run time. In any future release, an undetected error may become a detected error.

There is no significance to the order in which the errors are listed here.

Errors which are only detected when the ANSI option is ON, or when STANDARD_LEVEL is set to ANSI, do not appear on this list.

1. Each component of a structured function result must be assigned a value in the body of the function.

2. If assignment to a function result is conditional, it must occur at run time.

3. A control variable in a FOR statement cannot be changed in the statement after DO by calling a procedure or function with a non-local reference to the variable.

4. A parameter of a *dispose* call cannot be an actual variable parameter, an element of a record variable list of a WITH statement, or both. Similarly, a dynamic variable in a region of the heap deallocated by a call to *release* cannot fall in one of these categories.

5. When the tag field of a record with variants is changed, all previous variants become undefined.

6. For records with tagless variants, reference to a field for a particular variant means that other previous variants become undefined.

7. All tag values in a record declaration must be specified in the variant part.

UNDETECTED ERRORS

8. All possible record variants must be specified in a record declaration.

9. When a value is established for the tag field of a record with variants, it is illegal to use a field in another variant.

10. The compiler does not guarantee dectection of uninitialized variables especially in the following cases:

a. The path to use of a variable may not include the initializing assignment. Suppose:

```
PROCEDURE Proc_A;
VAR
  x,y: integer;
BEGIN
  .
  IF <condition> THEN x:= 10 ELSE y:= x;
  .
END;
```

The assignment after ELSE will not cause a compile time error, even if x has not been initialized outside of the IF statement. (The compiler counts the assignment after THEN as initialization.)

b. Not all the components of a record or array have been assigned values. (The compiler counts the assignment to a single component as initialization of the entire variable.)

c. A uninitialized global variable appears in a program compiled with the GLOBAL or EXTERNAL options, or in a program which contains procedures or functions declared with the EXTERNAL directive. (The compiler cannot check outside the current source code.)

d. An uninitialized dynamic variable on the heap. (This cannot be detected by the compiler or at run time.)

11. An actual reference parameter may not be an expression consisting of a single variable in parentheses.

12. Case constant labels can't be constant expressions.

13. Range checking code is suppressed when the type of logical file is identical with the type of a variable to which a file component is assigned. A physical file associated with the logical file, however, may have values out of range and the consequent error will be undetected.

14. Not all uninitialized variables are detected.

USING INTRINSICS

APPENDIX F

MATCHING INTRINSIC PARAMETERS

When the compiler encounters a procedure or function declared INTRINSIC in Pascal/3000 source code, it performs the following steps:

(1) It opens the current SPL intrinsic file, if it isn't open. The SPLINTR compiler option allows the programmer to designate the intrinsic file (see Section 8). The default file is SPLINTR.PUB.SYS.

(2) It checks that the specified intrinsic is in the SPL intrinsic file. An error occurs if it isn't.

(3) It collects information about the intrinsic parameters.

If the programmer has declared Pascal formal parameters, the compiler then checks these against the intrinsic parameters. If the intrinsic parameter is a reference parameter, the following conditions apply:

(A) The Pascal/3000 formal parameter may only be a VAR parameter.

(B) Any Pascal/3000 data type is acceptable. (For strings, only the character part is passed. The current length is discarded.)

(C) If the Pascal data type entails an SPL BYTE data object and the intrinsic expects a word address, a warning appears when odd byte addresses might not correctly convert to the word address.

Condition B means that the programmer can use, for example, a VAR parameter of type *char* to correspond to an intrinsic parameter of the SPL type INTEGER.

If the intrinsic parameter is a value parameter, the following conditions apply to the formal parameter:

(D) The Pascal/3000 formal parameter cannot be a VAR parameter.

MATCHING INTRINSIC PARAMETERS

(E) Depending on the SPL type of the intrinsic parameter, only certain Pascal types are acceptable (see below). Furthermore, the system will perform certain checking when the intrinsic call executes.

Condition E means that programmer cannot, for example, declare a formal parameter of type *char* to correspond with an intrinsic parameter of SPL type INTEGER.

When formal parameters are specified, the compiler checks the actual parameters of the intrinsic call for compatibility in the usual manner. In fact, the only reason the programmer will use formal parameters is to provide strong type checking of the actual parameters.

On the other hand, the programmer may choose to omit any or all formal parameters in a procedure or function declared INTRINSIC. In this case, the compiler uses the information from the splinter file to check the actual parameters when the program calls the intrinsic. If the intrinsic parameter is a reference parameter, the following conditions apply to the actual parameter:

(F) It must be a variable. It cannot be a constant, function reference, or procedural parameter.

(G) Any Pascal/3000 data type may appear as an actual parameter. The programmer must be aware of potential misinterpretation.

(H) The compiler converts word addresses to byte addresses and vice versa. It issues a warning if there is the possibility that an odd byte address will be converted to a word address.

Condition G, like condition B, means the programmer could, for example, pass a *char* type variable to an intrinsic parameter of SPL type INTEGER.

If the intrinsic parameter is a value parameter, this condition holds for the actual parameter:

(I) The type of the actual parameter is restricted according to the SPL type of the intrinsic parameter and certain checking may occur (see below).

Conditions E and I are really equivalent. They mean that if an intrinsic parameter is a value parameter, the Pascal type of the formal parameter or the actual parameter when there is no formal parameter is restricted. Also, the system will range check certain actual value parameters when the intrinsic call occurs, provided that the RANGE option was not OFF at compile time.

An intrinsic value parameter may be one of six SPL types: INTEGER, DOUBLE, LOGICAL, BYTE, REAL, or LONG. It is not possible to pass an array by value to an intrinsic. Table F-1 presents each intrinsic value parameter SPL type and the corresponding permissible Pascal types for formal or actual parameters.

No USL file parameter type checking information is generated for intrinsic calls. An intrinsic function return of type DOUBLE, REAL, or LONG must be matched with the Pascal type *integer, real,* or *longreal,* respectively.

Table F-1. INTRINSIC VALUE PARAMETERS AND PASCAL TYPES

Intrinsic Value Parameter SPL type	**Pascal Types for Formal or Actual Parameters**
INTEGER	An integer subrange within the range -32768..32767.
	An enumerated type with more than 256 elements.
	A set requiring 1 word of storage.
	A record requiring 1 word of storage.
	The type *integer*, or an integer subrange outside the range-32768..32767. In both cases, the system checks the actual value passed to the intrinsic, if RANGE is ON. An error occurs if it is not in the range-32768..32767.
DOUBLE	The type *integer.*
	Any integer subrange. If the subrange is within the range -32768..32767, the system converts the 1 word representation to a two word representation at run time.

Table F-1. INTRINSIC VALUE PARAMETERS AND PASCAL TYPES (continued)

LOGICAL	The type *integer*, or an integer subrange outside the range -32768..32767. In both cases, the system will check the actural value when the intrinsic call occurs. If it is outside the range 0..65535, an error results.
	An integer subrange in the range -32768..32767. The system will not check for negative actual values when the intrinsic call occurs.
	An enumerated type with more than 256 elements.
	A set requiring 1 word of storage.
	A record requiring 1 word of storage.
BYTE	The type *char*.
	The type *boolean*.
	An enumerated type with less than 257 elements.
REAL	The type *real*.
	The type *longreal*. The system will truncate the low order two words when the actual value passes to the intrinsic.
	The type *integer* or any integer subrange.
LONG	The type *longreal*.
	The type *real*. The system adds two words of 0 to the real mantissa when the intrincic call occurs.
	The type *integer*, or any integer subrange.

Examples

The reader should consult the MPE Intrinsics Manual for a full description of the intrinsics appearing in these examples.

```
PROGRAM calendar (output);
{Calls the MPE intrinsic CALENDAR, which returns a LOGICAL     }
{value. Bits 0 through 6 of this value represent the year of  }
{the century; bits 7 through 15 the day of the year.          }
TYPE
  date = PACKED RECORD
           year: 1..100;   {requires 7 bits of storage}
           day : 1..365;   {requires 9 bits of storage}
         END;
VAR
  d: date;
FUNCTION calendar: date; INTRINSIC;
BEGIN
  d:= calendar;
  writeln('The year is: ',d.year:1);
  writeln('The day is: ',d.day:1);
END.
```

```
PROGRAM time (output);
{Calls the MPE intrinsic CLOCK, which returns a DOUBLE value. }
{The four bytes represent the hour, minute, second, and tenths}
{of a second, successively.                                   }
TYPE
  t = PACKED RECORD
        CASE boolean OF
          true: (hour, minute, second, tenths: 0..255);
          false: (dblval: integer);
      END;
VAR
  time: t;
FUNCTION clock: integer; INTRINSIC;
BEGIN
  time.dblval:= clock;
  write('The time is ',time.hour:1,':',time.minute:1,':');
  writeln(time.second:1,':',time.tenths:1);
END.
```

MATCHING INTRINSIC PARAMETERS

```
PROGRAM show_fopen (input,output);
{This program uses the FOPEN intrinsic to open a file which  }
{disallows file equations and which is an exclusive access,  }
{multi-record file with no buffering. It then calls the      }
{FCLOSE intrinsic to close the file as a permanent file.     }

TYPE
  small_int= -32768..32767;
VAR
  file_name: PACKED ARRAY[1..8] OF char;
  f_options,
  a_options,
      dispo,
        sec,
      f_num: small_int;
FUNCTION fopen: small_int;INTRINSIC;
PROCEDURE fclose;INTRINSIC;
BEGIN
  file_name:= 'myfile  ';
  f_options:= octal('002000');     {disallow file equations    }
  a_options:= octal('000520'); {exclusive access, multi-record,}
                               {no buffer                      }
  f_num:= fopen(file_name,   {formaldesignator }
                f_options,          {foptions }
                a_options,          {aoptions }
                128,                 {recsize }
                ,                     {device }
                ,                    {formmsg }
                ,                  {userlabels }
                ,                 {blockfactor }
                ,                  {numbuffers }
                20000,               {filesize }
                , {not     }       {numextents }
                , {required}        {initalloc }
                );                   {filecode }
  writeln('The ccode is: ',ccode:1);

  {process file}

  dispo:= octal('000001');
  sec:=0;
  fclose(f_num,dispo,sec);
  writeln('The ccode is: ',ccode:1);
END.
```

PASCAL SUPPORT LIBRARY

The Pascal support library includes three procedures which may be called as external procedures from a HP3000 language or subsystem: GETHEAP, RTNHEAP, and HP32106.

Subsystems such as VPLUS call GETHEAP and RTNHEAP to allocate and deallocate space in the DL-DB area of the stack and avoid possible conflicts with the Pascal heap.

With conventions adopted from the MPE Intrinsics Reference Manual, subsequent pages describe each of these procedures. Data types are SPL data types.

HP32106

Returns the version name for the installed version of the Pascal/3000 support library. The version name is in the form 'HP32106v.uu.ff' where v denotes the major enhancement level, uu the update level, and ff the fix level.

```
BA
HP32106 (versionname);
```

PARAMETER

versionname BYTE ARRAY (required)
The array must contain at least 14 characters.
Input : undefined.
Output: the string 'HP32106v.uu.ff' left justified.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

None.

GETHEAP

Allocates a region of the DL-DB area of the stack of the size requested. The first parameter returns the location of the first word of this region. If the system cannot completely satisfy the request, the third parameter is set to FALSE.

```
                LP      I       L
GETHEAP  (regptr, regsize, regalloc);
```

PARAMETERS

regptr LOGICAL POINTER (required)
Input : undefined.
Output : pointer to region allocated when *regalloc* is TRUE, or undefined when it is FALSE and *regsize* is 0.

regsize INTEGER (required)
Input : number of words required in the region to be allocated.
Output : number of words actually allocated.

regalloc LOGICAL (required)
Input : must be FALSE. TRUE reserved for future internal use.
Output : TRUE if the requested region was completely allocated; FALSE if the allocation was not complete or totally unsuccessful.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

This intrinsic cannot be used with the DLSIZE intrinsic. Nor is it possible to directly manipulate regions of the DL-DB area not allocated by GETHEAP. BASIC/3000 cannot call GETHEAP.

In order for space returned by RTNHEAP to be compacted, the main program must be compiled with $DISPOSE ON$ and $HEAP_COMPACT ON$.

RTNHEAP

Deallocates a region of the DL-DB area of the stack. The pointer and size parameters must accurately match the values returned by a previous call to GETHEAP. If the given area cannot be correctly deallocated, the logical parameter is set to FALSE.

```
                    LP      IV      L
RTNHEAP  (regptr, regsize, regfree);
```

PARAMETERS

regptr LOGICAL POINTER (required)
Input : pointer returned by previous call to GETHEAP.
Output : undefined.

regsize INTEGER by value (required)
Input : size of region corresponding to pointer returned by GETHEAP.

regfree LOGICAL (required)
Input : must be FALSE. TRUE reserved for future internal use.
Output : TRUE if region successfully returned; FALSE if region could not be returned. The heap has been rendered invalid and no subsequent calls to GETHEAP or RTNHEAP will succeed.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

See GETHEAP.

PASCAL/3000 AND OTHER LANGUAGES

APPENDIX G

OVERVIEW

This appendix presents some of the information a programmer needs to call routines coded in SPL, FORTRAN, or COBOL successfully from a Pascal/3000 program, or to call a Pascal/3000 procedure or function from another language.

Calling Other Languages from Pascal

In general, the programmer must declare a procedure or function with the EXTERNAL directive and the name of a language (see Section 2). The formal parameters of the procedure or function must satisfactorily match the formal parameters of the external procedure or function. The CHECK_ACTUAL_PARM compiler option permits the programmer to determine the checking information placed in the USL for use by the Segmenter when performing a PREP or ADDSL (see Section 8).

When the call to the external procedure or function occurs, any actual procedural or functional parameters must be level 1 procedures or functions. The environment of the passed procedure or function must be available to the external routine. It is also inadvisable to pass files by reference to external routines since the conventions for input or output operations differ significantly between languages.

Calling Pascal from Other Languages

The programmer must match the parameters appearing in the non-Pascal program with the formal parameters of the external level 1 Pascal procedure or function. The CHECK_FORMAL_PARM compiler option permits the programmer to determine the checking information placed in the USL file for use by the Segmenter when performing a PREP or ADDSL.

The external Pascal code may be compiled with or without the SUBPROGRAM option. In either case, when the non-Pascal program calls the external Pascal procedure or function, the standard files *input* and *output*, even if they appear as program parameters in the Pascal code, will not have been opened and associated with $STDIN and $STDLIST in the usual way. The programmer must explicitly declare and open a file in the external Pascal procedure or function and, if desired, use a file equation to associate it with $STDIN or $STDLIST. Again, it is generally inadvisable to pass a file by reference to an external Pascal procedure or function since input and output operations may differ radically between languages.

If the non-Pascal program uses subsystems such as VPLUS or DSG and if the external Pascal procedure or functions it calls uses the heap, i.e. the DL-DB area of the stack, the program must declare itself as Pascal to the subsystem by using language code 5, except in the case of BASIC which cannot call a Pascal procedure or function that uses the heap in any case. This permits the subsystem, which also uses the DL-DB area, to call the Pascal procedures GETHEAP and RTNHEAP (see Appendix F).

A Pascal procedure or function with a long parameter list or with a very large value parameter, e.g. a big array, may be processed by the compiler so that a value parameter is pre-evaluated, temporarily stored on TOS (top of stack), and subsequently referenced indirectly. A non-Pascal program cannot call such a Pascal procedure or function because it will not correctly interpret the state of the stack at the start of execution. The programmer may determine if, in fact, a Pascal procedure or function has been compiled in this way by using the TABLES option (see Section 8).

Pascal Strings as Parameters

Pascal string variables are implemented in storage by 1 word and a sequence of bytes. The word holds the integer value for the current length of the string and each byte holds a single ASCII character.

The programmer must take this format into account when passing strings as parameters to other languages or when passing characters from some other language to a Pascal string variable. Here is an example of one way a string parameter could be passed from Pascal to a COBOL, FORTRAN, or SPL routine:

```
PROGRAM pass_string(input,output);
VAR
  test: string[10];
PROCEDURE splt (VAR stest : string);EXTERNAL SPL;
PROCEDURE fort (VAR ftest : string);EXTERNAL FORTRAN;
PROCEDURE cobt (VAR ctest : string);EXTERNAL COBOL;
```

```
BEGIN
  test := 'ABCD';
  writeln(strlen(test), test);
  splt(test); fort(test); cobt(test);
END.
```

SPL procedure:

```
$CONTROL SUBPROGRAM
BEGIN
  PROCEDURE SPLT(STR);
    LOGICAL ARRAY STR;
    BEGIN
      INTRINSIC PRINT;
      PRINT(STR(1),-STR(0),0);
    END;
END.
```

FORTRAN procedure:

```
      SUBROUTINE FORT(ARR)
        LOGICAL ARR(6), LOG(5), STRL
        INTEGER I, LEN
        CHARACTER*10 STR
        EQUIVALENCE (STR, LOG), (LEN, STRL)
        DO 10 I = 1,5
           LOG(I) = ARR(I+1)
10      CONTINUE
        STRL = ARR(1)
        DISPLAY LEN, STR
        RETURN
        END
```

COBOL procedure:

```
$CONTROL SUBPROGRAM
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COBT.
 AUTHOR. ME.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. HP-3000.
 OBJECT-COMPUTER. HP-3000.
 DATA DIVISION.
 LINKAGE SECTION.
 01 ASTRING.
  05 INT PIC 99 COMP.
  05 STR PIC X(10).
 PROCEDURE DIVISION USING ASTRING.
 PARA-1.
   DISPLAY INT, STR.
 PARA-END.
   EXIT PROGRAM.
```

PASCAL AND SPL

Calling SPL from Pascal

To call an external SPL routine from a Pascal program, the programmer must declare the function or procedure with the EXTERNAL SPL or EXTERNAL SPL VARIABLE directive (see Section 2) and match the Pascal types of the formal parameters or result type with the SPL types of the external parameters or result type. Table G-1 lists the corresponding Pascal and SPL types.

SPL cannot accept value parameters of any array type. Thus, the compiler will issue an error message if the Pascal type of a formal value parameter results in an SPL array. Pascal/3000 will generate SPL-compatible type checking information in the USL file for calls to external SPL routines. To have parameter type compatibility checked by the Segmenter, the SPL procedure should be compiled with OPTION CHECK 3.

Calling Pascal from SPL

To call a Pascal procedure or function from an SPL program, the programmer must use an SPL EXTERNAL procedure declaration which provides parameter declarations that are compatible with the Pascal types of the external parameters. Table G-1 shows the Pascal and SPL type correspondences.

An SPL program cannot pass arrays by value to a Pascal procedure or function. Pascal/3000 will generate Pascal type checking information in the USL file for Pascal procedures or functions. SPL procedures which call Pascal should be compiled with OPTION CHECK 0 or the Pascal procedure or function should be compiled with CHECK_FORMAL_PARM set to 0.

Examples:

A Pascal program:

```
PROGRAM Pascal_SPL(input,output);
TYPE
  char_str  = PACKED ARRAY[1..20] OF char;
  small_int = -32768..32767;
VAR
  a_str          : char_str;
  int1,int2,sum : small_int;

PROCEDURE splprc(VAR cstr : char_str;
                     inta : small_int;
                     intb : small_int;
                 VAR total: small_int
                ); EXTERNAL SPL;
BEGIN
  a_str := 'Add these 2 numbers:';
  int1 := 25;
  int2 := 15;
  writeln(a_str,int1,int2);
  splprc(a_str,int1,int2,sum);
  writeln(a_str,sum);
END.
```

An external SPL procedure:

```
$CONTROL SUBPROGRAM
BEGIN
PROCEDURE splprc(cstr,int1,int2,sum);
  VALUE int1,int2;
  INTEGER int1,int2,sum;
  BYTE ARRAY cstr;
  BEGIN
    sum := int1 + int2;
    MOVE cstr := "Sum of two numbers: ";
  END;
END.
```

PASCAL AND SPL

An SPL program:

```
BEGIN
  LOGICAL ARRAY chr(0:9) := "Add these 2 numbers:";
  BYTE ARRAY bchr(*) = chr;
  INTEGER sint:=15,sint2:=25,len;
  INTEGER int, int2, sum;
  BYTE ARRAY csum(*) = sum, cint(*) = int,
             cint2(*) =int2;

  INTRINSIC PRINT,ASCII;
  PROCEDURE pas(chr,sint,sint2,sum);
    VALUE sint,sint2;
    INTEGER sint,sint2,sum;
    BYTE ARRAY chr;
    OPTION EXTERNAL;


  PRINT(chr,10,0);
  len := ASCII(sint,-10,cint(1));
  len := ASCII(sint2,-10,cint2(1));
  PRINT(cint,-2,0);
  PRINT(cint2,-2,0);

  pas(chr,sint,sint2,sum);

  PRINT(chr,10,0);
  len := ASCII(sum,-10,csum(1));
  PRINT(csum,-2,0);
END.
```

A Pascal external procedure:

```
$SUBPROGRAM$
PROGRAM example(input,output);
TYPE
  arr = PACKED ARRAY[1..20] OF char;
  small_int = -32768..32767;

PROCEDURE pas(VAR carr:arr; sint:small_int; sint2:small_int;
              VAR sum:small_int);
  BEGIN
    carr := 'sum of two numbers: ';
    sum := sint + sint2;
  END;
BEGIN
END.
```

Table G-1. PASCAL AND SPL TYPES

Pascal/3000 Type	SPL Type
integer Integer subrange outside the range -32768..32767	DOUBLE
Integer subrange within the range -32768..32767	INTEGER
real	REAL
longreal	LONG
char	BYTE
boolean	BYTE (Pascal *false*=0 Pascal *true* = 1)
Enumerated type with less than 257 elements	BYTE
Enumerated type with more than 256 elements	LOGICAL

Table G-1. PASCAL AND SPL TYPES (continued)

SET (1 word)	LOGICAL
SET (multi word)	LOGICAL ARRAY
RECORD (1 word)	LOGICAL
RECORD (multi word)	LOGICAL ARRAY
^ <Pascal type>	<SPL type> POINTER
string	LOGICAL ARRAY
ARRAY OF <Pascal type>	<SPL type> ARRAY
ARRAY OF <Pascal pointer type>	LOGICAL ARRAY
FILE	LOGICAL ARRAY (The type of the Pascal file is not accounted for.)
Procedural parameters	Procedure Parameter

Calling FORTRAN from Pascal

To call a FORTRAN routine from a Pascal/3000 program, the programmer must declare the procedure or function with the EXTERNAL FORTRAN directive (see Section 2) and match the Pascal types of the formal parameters or result type with the FORTRAN types of the external parameters or result type. Table G-2 lists Pascal types and the corresponding FORTRAN types.

Pascal cannot access a FORTRAN COMMON area. Nor is it possible to pass a file or a label to an external FORTRAN routine. Also, FORTRAN interprets all Pascal array index types as 1..n, regardless of the specified subrange in the Pascal source code.

FORTRAN expects only parameters passed by reference. The Pascal compiler creates a dummy location for a value parameter and passes that address as a reference. This has certain implications when passing dynamic variables. In particular, the programmer must pass a dereferenced pointer as the actual parameter. With a reference formal parameter, the FORTRAN routine will access the heap variable through the address of the object indicated by the pointer (see example below). With a value formal parameter, a copy of the dynamic variable is placed on the stack with a dummy reference; this may be quite expensive if, for example, the variable is a large array.

Pascal/3000 will generate FORTRAN-compatible type checking information in the USL file for calls to external FORTRAN routines. FORTRAN generates parameter type checking information in the USL file by default.

PASCAL AND FORTRAN

Calling Pascal from FORTRAN

To call a Pascal procedure or function from a FORTRAN program, the programmer need not use an EXTERNAL subroutine declaration. However, a Pascal function name must appear in a type statement, e.g. INTEGER PASFUNC, where the FORTRAN type corresponds to the Pascal type of the function result (see Table G-2).

FORTRAN cannot pass arrays by value, so it is not possible to call a Pascal routine with a value parameter of a type corresponding to a FORTRAN array type. For any other type of Pascal value parameter, the programmer must use the backslash (\) notation of FORTRAN/3000.

All parameters in FORTRAN are word addressed, except for character variables and character arrays which are byte addressed.

All data must be passed through the parameter lists between FORTRAN and Pascal since FORTRAN cannot specify global variables and Pascal cannot specify COMMON blocks. The calling FORTRAN program may have a COMMON area, but the external Pascal procedure or function cannot use global variables.

The programmer must set CHECK_FORMAL_PARM to 2 for all Pascal procedures, and 0 for functions to be called by FORTRAN.

Examples

Pascal program:

```
PROGRAM pass_heap_var(input,output);
  TYPE
    ptr = ^arr;
    arr =  PACKED ARRAY [1..80] OF char;
  VAR
    aptr : ptr;

  PROCEDURE fort (VAR arrptr : arr);EXTERNAL FORTRAN;
                 {The use of a reference parameter permits the}
                 {external FORTRAN routine to access the      }
                 {variable through the pointer.               }
BEGIN
  new(aptr);
  aptr^:= 'I am a dynamic variable';
  fort(aptr^);
END.
```

External FORTRAN procedure:

```
SUBROUTINE FORT(PTRARR)
  CHARACTER*80 PTRARR
  DISPLAY PTRARR
  RETURN
  END
```

PASCAL AND FORTRAN

Pascal program:

```
PROGRAM pascal_fort(input,output);
TYPE
  char_str  = PACKED ARRAY[1..20] OF char;
  small_int = -32768..32767;
VAR
  a_str          : char_str;
  int1,int2,sum  : small_int;

PROCEDURE fortprc(VAR cstr : char_str;
                      inta : small_int;
                      intb : small_int;
                  VAR total: small_int
                 ); EXTERNAL FORTRAN;
BEGIN
  a_str := 'Add these 2 numbers: ';
  int1 := 25;
  int2 := 15;
  writeln(a_str,int1,int2);
  fortprc(a_str,int1,int2,sum);
  writeln(a_str,sum);
END.
```

External FORTRAN procedure:

```
SUBROUTINE FORTPRC(CSTR,INT1,INT2,SUM)
    INTEGER INT1, INT2, SUM
    CHARACTER CSTR*20

    SUM = INT1 + INT2
    CSTR = "SUM OF TWO NUMBERS: "

    RETURN
    END
```

FORTRAN program:

```
INTEGER INT1, INT2, ISUM
CHARACTER CSTR*30

CSTR = "Add these 2 numbers"
INT1 = 25
INT2 = 15

DISPLAY CSTR, INT1, INT2
CALL PAS(CSTR,\INT1\,\INT2\,ISUM)
DISPLAY CSTR, ISUM

STOP
END
```

External Pascal procedure:

```
$SUBPROGRAM$

PROGRAM example(input,output);
TYPE
  arr = PACKED ARRAY[1..20] OF char;
  small_int = -32768..32767;

$CHECK_FORMAL_PARM 0$
PROCEDURE pas(VAR carr : arr;
                  sint : small_int;
                 sint2 : small_int;
              VAR sum  : small_int
             );
  BEGIN
    carr := 'Sum of two numbers: ';
    sum := sint + sint2;
  END;

BEGIN
END.
```

Table G-2. PASCAL AND FORTRAN TYPES

Pascal Type	FORTRAN type
integer Integer subrange outside the range -32768..32767	INTEGER*4
Integer subrange inside the range -32768..32767	INTEGER*2
real	REAL
longreal	DOUBLE PRECISION
char	CHARACTER
PACKED ARRAY [1..n] OF *char*	CHARACTER*n
boolean	CHARACTER (1 = *true*, 0 = *false*)
Enumerated type with less than 257 elements	CHARACTER
Enumerated type with more than 256 elements	INTEGER*2

Table G-2. PASCAL AND FORTRAN TYPES (continued)

RECORD real_part : *real*; imag_part : *real*; END;	COMPLEX (parameter type checking must be turned off)
SET (1 word) SET (multi-word)	LOGICAL Array of LOGICAL
string	Array of LOGICAL
ARRAY [] OF <Pascal type> (stored in row-major order)	Array of corresponding FORTRAN type (stored in column-major order)
Procedural parameters	EXTERNAL statement

PASCAL AND COBOL

The data types of Pascal and COBOL differ radically. In general, COBOL data types are either binary or ASCII format (see Table G-3). By taking the size as well as the format into consideration, the programmer can successfully match Pascal and COBOL types.

The following are examples of possible matches between COBOL and Pascal types:

COBOL	Pascal
PIC X (N)	PACKED ARRAY [1..N] OF *char* ARRAY [1..N] OF *char*
PIC S9 (01) -S9 (04) COMP	small_int [-9999..9999]
PIC S9 (05) -S9 (09) COMP	*integer*
PIC S9 (10) -S9 (18) COMP	ARRAY [1..2] OF *integer*
PIC S9 (01) -S9 (18) COMP-3	TYPE nibble = 0..15 PACKED ARRAY [1..28] OF nibble

In the last example, a Pascal record is constructed to hold a COBOL packed decimal number, but a Pascal program cannnot operate on the number.

The COBOL types 01 and 77 always start on word boundaries.

The parameter capabilities of COBOL 68 and COBOL II differ. In particular, COBOL 68 cannot pass by value, but COBOL II can provided the backslash (\) notation is used. COBOL 68 cannot use a parameter on a byte boundary; COBOL II can provided the @ symbol is specified. Finally, COBOL 68 cannot call a Pascal function; COBOL II.can if the GIVING phrase occurs.

Table G-3. COBOL TYPES AND FORMATS

<table>
<tr><th>COBOL Type</th><th>Format</th></tr>
<tr><td>COMP-3</td><td>Packed decimal format with sign in right-most half byte and 2 digits per byte.</td></tr>
<tr><td>COMP</td><td>Binary format. Sign bit 0 is +, 1 is −.

Size — Number of Words
S9 to S9 (4) — 1
S9 (5) to S9 (9) — 2
S9 (10) to S8 (18) — 4</td></tr>
<tr><td>DISPLAY</td><td>Unpacked decimal format (ASCII).

Unsigned- alphanumeric format:
no leading or trailing sign;
1 character per byte.

Sign is leading - alphanumeric format;
sign 'overpunched' in left-most byte.

Sign is trailing - alphanumeric format;
sign 'overpunched' in right-most byte.

Sign is leading, separate - first byte is ASCII '-' for negative,
anything else specifies positive.

Sign is trailing, separate - last byte is ASCII '-' for negative,
anything else specifies positive.</td></tr>
</table>

Examples

```
PROGRAM Pascal_COBOL (input,output);
{Calls a simple COBOL II routine.}
VAR
  int1,
  int2,
  int3 : integer;

{All parameters are passed by reference.}
PROCEDURE subprog1(VAR parm1: integer;
                   VAR parm2: integer;
                   VAR parm3: integer); EXTERNAL COBOL;
BEGIN
  int1 := 25000;
  int2 := 30000;
  subprog1(int1, int2, int3);
  writeln(int3);
END.
```

```
SUBPROGRAM 1:
    $CONTROL SUBPROGRAM
     IDENTIFICATION DIVISION.
     PROGRAM-ID. SUBPROG1.
     AUTHOR. BP.
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     SOURCE-COMPUTER. HP3000.
     OBJECT-COMPUTER. HP3000.
     DATA DIVISION.
     LINKAGE SECTION.
     77 IN1       PIC S9(07) COMP.
     77 IN2       PIC S9(07) COMP.
     77 OUT       PIC S9(07) COMP.
     PROCEDURE DIVISION USING IN1, IN2, OUT.
     PARA-1.
       ADD IN1, IN2, GIVING OUT.
       EXIT PROGRAM.
```

This COBOL 68 program calls a Pascal procedure:

```
$CONTROL USLINIT
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COBOL-TO-PASCAL.
 AUTHOR. BP.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. HP3000.
 OBJECT-COMPUTER. HP3000.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 ASTRING   PIC X(16) VALUE "A COBOL STRING! ".
 77 ANUM      PIC 9(04) USAGE COMP.
 77 RESULT    PIC -ZZZZ.
 PROCEDURE DIVISION.
 FIRST-PARA.
   DISPLAY ASTRING.
   CALL "PASPROG" USING ASTRING, ANUM.
   MOVE ANUM TO RESULT.
   DISPLAY ASTRING, RESULT.
   STOP RUN.
```

```
$SUBPROGRAM$
PROGRAM Pascal_code(input,output);
TYPE
  small_int = -32768..32767;
  charstr = RECORD
              cpart : PACKED ARRAY [1..16] OF char;
            END;
  {Since the COBOL 68 program requires a variable on a word }
  {boundary, this record type disguises the PAC as such a   }
  {variable. For COBOL II, this deception is unnecessary.   }

PROCEDURE pasprog(VAR astr : charstr; VAR num : small_int);
  BEGIN
    astr.cpart := 'A PASCAL STRING!';
    num := 9999;
  END;
BEGIN
END.
```

This COBOL II program calls a Pascal procedure using a byte-addressed parameter, a value parameter, and a reference parameter:

```
$CONTROL USLINIT
 IDENTIFICATION DIVISION.
 PROGRAM-ID. COBOL-TO-PASCAL.
 AUTHOR. BP.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. HP3000.
 OBJECT-COMPUTER. HP3000.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77 ASTRING   PIC X(16) VALUE "A COBOL STRING!".
 77 ANUM      PIC 9(04) USAGE COMP.
 77 ANUM2     PIC 9(04) USAGE COMP.
 77 RESULT    PIC -ZZZZ.
 PROCEDURE DIVISION.
 FIRST-PARA.
   MOVE 9999 TO ANUM.
   DISPLAY ASTRING.
   CALL "PASPROG" USING @ASTRING, \ANUM\, ANUM2.
   MOVE ANUM2 TO RESULT.
   DISPLAY ASTRING, RESULT.
   STOP RUN.
```

```
$SUBPROGRAM$
PROGRAM pas_proc(input,output);
TYPE
  small_int = -32768..32767;
  charstr = PACKED ARRAY [1..16] OF char;
  (COBOL II program will accept a byte-addressed variable. )

PROCEDURE pasprog(VAR astr : charstr;
                      num  : small_int;
                  VAR num2 : small_int);
  BEGIN
    astr := 'A PASCAL STRING!';
    num2 := num;
  END;
BEGIN
END.
```

PASCAL/3000 AND HP3000 SUBSYSTEMS

APPENDIX H

This appendix presents information the programmer needs to know in order to use certain HP3000 subsystems in a Pascal/3000 program. In particular it considers Pascal/3000 in relation to SORT-MERGE/3000, IMAGE/3000, and VPLUS/3000.

PASCAL AND SORT-MERGE

The SORT-MERGE subsystem uses certain addressing modes which can potentially conflict with common addressing modes in the object code generated by the Pascal/3000 compiler. For this reason, the programmer should call the SORTINIT and SORTEND, or the MERGEINIT and MERGEEND intrinsics within the executable portion of a single procedure. Furthermore, the Pascal code occurring between the calls to these intrinsics should only consist of parameterless procedure calls. In outline, this is a possible form of a Pascal procedure using the SORT-MERGE subsystem:

```
PROCEDURE sort;
  PROCEDURE read_file;
    BEGIN
      .
      sortinput (...);          {intrinsic call}
      .
    END;

  PROCEDURE write_file;
    BEGIN
      .
      sortoutput (...);         {intrinsic call}
      .
    END;

  BEGIN {sort}
    .
    sortinit (...);             {intrinsic call}
    read_file;
    write_file;
    sortend (...);              {intrinsic call}
    .
  END; {sort}
```

PASCAL AND SORT-MERGE

The following sample program uses this outline:

```
PROGRAM mailing_list_sort (mailfile);
{Sorts mail file by zip code using SORT intrinsics and reports}
{statistics from sorting procedure.                           }

TYPE
  smallint     = -32768..32767;
  milliseconds = integer;
  sort_key     = RECORD
                   position : smallint;
                   length   : smallint;
                   sequence : (ascending,descending);
                   data_type: (character,
                               twos_complement,
                               floating_point,
                               packed_decimal,
                               display_trailing_sign,
                               packed_decimal_even,
                               display_leading_sign,
                               display_leading_sign_separate,
                               display_trailing_sign_separate
                              );
                 END;

  sort_statistics = RECORD
                      records             : integer;
                      intermediate_passes : smallint;
                      space_available     : smallint;
                      comparisons         : integer;
                      scratch_file_ios    : integer;
                      cpu_time            : milliseconds;
                      elapsed_time        : milliseconds;
                    END;

  mailrec  = RECORD
               name           : PACKED ARRAY[1..23] OF char;
               street_address : PACKED ARRAY[1..23] OF char;
               city           : PACKED ARRAY[1..23] OF char;
               state          : PACKED ARRAY[1..2]  OF char;
               zip            : PACKED ARRAY[1..9]  OF char;
             END;
```

```
VAR
  mailfile : FILE OF mailrec;

PROCEDURE sortinit;   INTRINSIC;
PROCEDURE sortinput;  INTRINSIC;
PROCEDURE sortoutput; INTRINSIC;
PROCEDURE sortend;    INTRINSIC;
PROCEDURE sortstat;   INTRINSIC;

PROCEDURE sort;
  VAR
    statistics : sort_statistics;
    numkeys    : smallint;
    keys       : sort_key;

  PROCEDURE read_unsorted_mailing_list;
    BEGIN
      reset(mailfile);
      WHILE NOT eof(mailfile) DO
        BEGIN
          sortinput(mailfile^,sizeof(mailfile^));
          get(mailfile);
        END;
    END;

  PROCEDURE write_sorted_mailing_list;
    VAR
      length : smallint;
    BEGIN
      rewrite(mailfile);
      sortoutput(mailfile^,length);
      WHILE length > 0 DO
        BEGIN
          put(mailfile);
          sortoutput(mailfile^,length);
        END;
    END;
```

```
  BEGIN { sort }
    numkeys := 1;
    WITH keys DO
      BEGIN
        position := 72;
        length := 9;
        sequence := ascending;
        data_type := character;
      END;
    sortinit(,,,sizeof(mailfile^),,numkeys,keys,,,,statistics);
    read_unsorted_mailing_list;
    write_sorted_mailing_list;
    sortend;
    sortstat(statistics);
  END;  { sort }

BEGIN { mailing_list_sort }
  sort;
END.  { mailing_list_sort }
```

Suppose the unsorted disc file MAILFILE consists of 5 records:

```
Mickey Mouse          Disneyland              Anaheim          CA921010705
Charles Babbage       11 Downing Street       London NW 5      GB000001196
Art Esian             2000 Capitol Avenue     Tumwater         WA995029138
Henrietta T. Moose    19420 Homestead Road    Cupertino        CA950146278
Shamu Whale           Sea World               San Diego        CA921205811
```

Then running this program will sort the contents of MAILFILE by zipcode and produce statistics for the sorting procedure. MAILFILE will now be:

```
Charles Babbage       11 Downing Street       London NW 5      GB000001196
Mickey Mouse          Disneyland              Anaheim          CA921010705
Shamu Whale           Sea World               San Diego        CA921205811
Henrietta T. Moose    19420 Homestead Road    Cupertino        CA950146278
Art Esian             2000 Capitol Avenue     Tumwater         WA995029138
```

This sample program illustrates one way the programmer can use Pascal/3000 data types to define data structures which are suitable for use with the IMAGE subsystem:

```
PROGRAM Pascal_Image(input,output);
TYPE
  pac = PACKED ARRAY [1..20] OF char;
  data_rec =  RECORD          {some detail data-set}
                name: pac;
                position: pac;
                location: pac;
                phone: pac;
                comment: pac;
              END;

  {Set-up of IMAGE parameter data types: }
  single_integer = -32768..32767;
  base_type      = PACKED ARRAY [1..16] OF char;
  password_type  = PACKED ARRAY [1..6] OF char;
  status_type    = ARRAY [1..10] OF single_integer;
  ds_name_type   = PACKED ARRAY [1..12] OF char;
  list_type      = PACKED ARRAY [1..2] OF char;
  buffer_type    = PACKED ARRAY [1..100] OF char;
  key_type       = PACKED ARRAY [1..40] OF char;
  item_type      = PACKED ARRAY [1..8] OF char;
  err_type       = (db_get,db_put,db_find,db_open,db_close);

VAR
  cur_rec          : data_rec;
  cur_get          : data_rec;
  answer           : integer;
  sample_db        : base_type;
  sample_password  : password_type;
  sample_ds        : ds_name_type;
  sample_item      : item_type;
  sample_buff      : buffer_type;
  status           : status_type;
  list             : list_type;
  mode             : single_integer;
  dummy            : single_integer;
```

```
{External declarations of IMAGE procedures: }

PROCEDURE dbopen;INTRINSIC;
PROCEDURE dbput;INTRINSIC;
PROCEDURE dbget;INTRINSIC;
PROCEDURE dbfind;INTRINSIC;
PROCEDURE dbclose;INTRINSIC;

{External error handling routine: }

PROCEDURE fatal_error (stat:status_type;error: err_type);
  EXTERNAL;


{Menu screen: }

PROCEDURE list_menu;
  BEGIN
    writeln('     CONTACT INFORMATION FILE');
    writeln('           1) ADD A RECORD    ');
    writeln('           2) LIST LAST PERSON');
    writeln('           3) FINISHED        ');
    prompt('   PLEASE ENTER DESIRED OPTION #: ');
    readln(answer);
  END;


PROCEDURE get_rec_info;
  BEGIN
    prompt('ENTER CONTACT NAME: ');
    readln(cur_rec.name);
    prompt('ENTER CONTACT POSITION: ');
    readln(cur_rec.position);
    prompt('ENTER CONTACT LOCATION: ');
    readln(cur_rec.location);
    prompt('ENTER TELEPHONE NUMBER: ');
    readln(cur_rec.phone);
    prompt('ENTER COMMENT (PRESS RETURN IF NONE): ');
    readln(cur_rec.comment);
    mode := 1;
    dbput(sample_db, sample_ds, mode, status, list, cur_rec);
    IF status[1] <> 0 THEN
      fatal_error(status, db_put);
  END;
```

```
PROCEDURE finish_up;
  BEGIN
    mode := 3;
    dbclose(sample_db, sample_ds, mode, status);
    IF status[1] <> 0 THEN fatal_error(status,db_close);
    writeln('HAVE A NICE DAY !!!!');
  END;


PROCEDURE print_last_rec;
  VAR
    search_item: item_type;
  BEGIN
    search item := 'name;    ';

    (call dbfind)
    mode := 1;
    dbfind(sample_db, sample_ds, mode, status, search_item,
           cur_rec.name);
    IF status[1] <> 0 THEN fatal_error(status, db_find);

    (call dbget)
    mode := 5;
    dbget(sample_db, sample_ds, mode, status, list,
          cur_rec, dummy);
    IF status[1] <> 0 THEN fatal_error(status, db_get);
    writeln(cur_get.name);
    writeln(cur_get.position);
    writeln(cur_get.location);
    writeln(cur_get.phone);
    writeln(cur_get.comment);
  END;
```

```
BEGIN  (Pascal_Image)
 (set-up data-base information)
  sample_db := '  SAMPLE;       ';
  sample_password := 'EASY; ';
  sample_ds := 'INFO_DETAIL';
  list := '@;';
  mode := 3;
  dbopen(sample_db, sample_password, mode, status);
  IF status[1] <> 0 THEN
    fatal_error(status, db_open);

  answer:= 0;
  WHILE answer <> 3 DO
    BEGIN
      list_menu;
      CASE answer OF
        1 : get_rec_info;
        2 : print_last_rec;
        3 : finish_up;
        OTHERWISE writeln('INVALID _ PLEASE REENTER')
      END
    END
END. (Pascal_Image)
```

VPLUS/3000 uses the DL-DB area of the stack to store screen or form information. Pascal/3000 also uses this area as its 'heap'. To avoid any possible conflict, a Pascal program calling the VPLUS subsystem must use language code 5. This signals VPLUS to call the Pascal library procedure GETHEAP, which allocates a region of the DL-DB area for exclusive use by VPLUS. When the formsfile is closed, VPLUS calls another Pascal library procedure, RTNHEAP, which releases the region previously reserved for the subsystem. (Appendix F describes GETHEAP and RTNHEAP.)

In general, the programmer should define VPLUS common areas and buffers on word boundaries. It will also probably be necessary to specify the MAXDATA parameter of the :PREP or :RUN commands to enlarge the DL-DB area, especially when a Pascal program uses VPLUS and dynamic allocation at the same time.

This sample program illustrates one way the programmer can construct Pascal/3000 data structures suitable for calling VPLUS:

```
PROGRAM Pascal_Vplus(input,output);
TYPE
  word = -32768..32767;
  err_type = (v_openformf, v_openterm, v_closeterm,
              v_closeformf);
  string2  = PACKED ARRAY [1..2] OF char;
  string3  = PACKED ARRAY [1..3] OF char;
  string4  = PACKED ARRAY [1..4] OF char;
  string5  = PACKED ARRAY [1..5] OF char;
  string6  = PACKED ARRAY [1..6] OF char;
  string7  = PACKED ARRAY [1..7] OF char;
  string8  = PACKED ARRAY [1..8] OF char;
  string9  = PACKED ARRAY [1..9] OF char;
  string10 = PACKED ARRAY [1..10] OF char;
  string11 = PACKED ARRAY [1..11] OF char;
  string12 = PACKED ARRAY [1..12] OF char;
  string13 = PACKED ARRAY [1..13] OF char;
  string14 = PACKED ARRAY [1..14] OF char;
  string15 = PACKED ARRAY [1..15] OF char;
  string16 = PACKED ARRAY [1..16] OF char;
  string30 = PACKED ARRAY [1..30] OF char;
  string30 = PACKED ARRAY [1..30] OF char;
  word-2   = ARRAY [1..2] OF word;
  word-5   = ARRAY [1..5] OF word;
```

```
vplus_comarea = RECORD
                  cstatus       : word;
                  language      : word;
                  comarealen    : word;
                  usrbuflen     : word;
                  cmode         : word;
                  lastkey       : word;
                  numerrs       : word;
                  windowenh     : word;
                  multiusage    : word;
                  labeloptions  : word;
                  cfname        : string16;
                  nfname        : string16;
                  repeatapp     : word;
                  freezapp      : word;
                  cfnumlines    : word;
                  dbuflen       : word;
                  skip2         : word;
                  lookahead     : word;
                  deleteflag    : word;
                  showcontrol   : word;
                  skip4         : word;
                  printfilnum   : word;
                  filerrnum     : word;
                  errfilenum    : word;
                  formstrsize   : word;
                  skip6         : word;
                  skip7         : word;
                  skip8         : word;
                  numrecs       : integer;
                  recnum        : integer;
                  skip9         : string4;
                  term_filen    : word;
                  skip10        : string10;
                  retries       : word;
                  term_options  : word;
                  environ       : word;
                  usertime      : word;
                  identifier    : word;
                  labelinfo     : word;
                END;
```

```
CONST
  com_area_init = vplus_comarea
                    [cstatus : 0,
                     language : 5,    { Pascal code number }
                     comarealen : 60,
                     usrbuflen : 0,
                     cmode : 0,
                     lastkey : 0,
                     numerrs : 0,
                     windowenh :  0,
                     multiusage : 0,
                     labeloptions : 0,
                     cfname : '                    ',
                     nfname : '                    ',
                     repeatapp : 0,
                     freezapp : 0,
                     cfnumlines : 0,
                     dbuflen : 0,
                     skip2 : 0,
                     lookahead : 0,
                     deleteflag : 0,
                     showcontrol : 0,
                     skip4 : 0,
                     printfilnum : 0,
                     filerrnum : 0,
                     errfilenum : 0,
                     formstrsize : 0,
                     skip6 : 0,
                     skip7 : 0,
                     skip8 : 0,
                     numrecs : 0,
                     recnum : 0,
                     word-2[2 of 0]
                     skip9 :       ,
                     term_filen : 0,
                     word-5[5 of 0]
                     skip9 :      ',
                     term_filen : 0,
                     skip10 :            ',
                     retries : 0,
                     term_options : 0,
                     environ : 0,
                     usetime : 0,
                     identifier : 0,
                     labelinfo : 0
                    ];
VAR
  terminal : string8;
  formfile : string9;
  term_id  : word;
  com_area : vplus_comarea;
```

```
{ VPLUS/3000 Intrinsic Procedure Declarations }

PROCEDURE vopenterm ; INTRINSIC;
PROCEDURE vopenformf ; INTRINSIC;
PROCEDURE vcloseterm ; INTRINSIC;
PROCEDURE vcloseformf ; INTRINSIC;
PROCEDURE vgetnextform ; INTRINSIC;
PROCEDURE vshowform ; INTRINSIC;
PROCEDURE vreadfields ; INTRINSIC;
PROCEDURE vgetbuffer ; INTRINSIC;
PROCEDURE vputbuffer ; INTRINSIC;
PROCEDURE vinitform ; INTRINSIC;
PROCEDURE vputfield ; INTRINSIC;

PROCEDURE fatal_error (err : err_type; stat:word); EXTERNAL;
PROCEDURE main_menu  ; EXTERNAL;

BEGIN   {Pascal_Vplus}
  terminal := 'X       ';
  formfile := 'FORMFILE ';

  { Initialize comarea }
  com_area := com_area_init;

  vopenterm(com_area, terminal);
  IF com_area.cstatus <> 0 THEN
    fatal_error(v_openterm, com_area.cstatus);

  vopenformf(com_area, formfile);
  IF com_area.cstatus <> 0 THEN
    fatal_error(v_openformf, com_area.cstatus);

  main_menu;

  vcloseterm(com_area);
  IF com_area.cstatus <> 0 THEN
    fatal_error(v_closeterm, com_area.cstatus);

  vcloseformf(com_area);
  IF com_area.cstatus <> 0 THEN
    fatal_error(v_closeformf, com_area.cstatus);

END.  {Pascal_Vplus}
```

I/O DEFINITIONS

APPENDIX

I

This appendix provides formal definitions of certain I/O procedures in HP Standard Pascal.

The tests for existence of a component fail on attempting to access a component which doesn't exist, on receiving an EOF condition from a device, or on attempting to access beyond the last component of a direct access file.

```
TYPE
  file_block = RECORD          {A data structure associated with  }
                               {file with components of type T.   }
  bound     :   integer;       {Maximum number of components.     }
  component:    ARRAY [1..bound] OF T;        {File components.  }
  pos       :   integer;       {Next component to be read index.  }
  buffer    :   T;             {Space for pre-fetched component.  }
  lookahead,                   {Buffer variable pre-fetched.      }
  getok,                       {f^ ref buffers next component.    }
  endoffile,                   {End of file was found.            }
  readable,                    {Read operations are legal.        }
  writeable:    boolean;       {Write operations are legal.       }
                END;

PROCEDURE Setup (f: file; s: string);    {An internal procedure}
  BEGIN
    IF s exists THEN
      BEGIN
        close previously associated file, if any;
        associate file specified by s;
      END
    ELSE IF previous f not open THEN
      associate file specified by file's variable name;
    f.bound := system_established_value;
    f.buffer := undefined;
  END {Setup};

 PROCEDURE Open(f,s);
   BEGIN
     Setup (f,s);
     f.readable := true; f.writeable := true;
     f.lookahead := false; f.getok := false;
     f.endoffile := false;
     f.pos := 1;
   END {Open};
```

```
PROCEDURE Reset(f,s);
  BEGIN
    Setup (f,s);
    f.readable := true;  f.writeable := false;
    f.lookahead := false;   f.getok := true;
    f.endoffile := false;
    f.pos := 1;
  END {Reset};

PROCEDURE Rewrite(f,s);
  BEGIN
    Setup (f,s);
    f.readable := false;  f.writeable := true;
    f.lookahead := false;  f.getok := false;
    f.endoffile := true;
    f.pos := 1;
    destroy any existing components of f;
  END {Rewrite};

PROCEDURE Append(f,s);
  BEGIN
    Setup (f,s);
    f.readable := false;  f.writeable := true;
    f.lookahead := false;  f.getok := false;
    f.endoffile := true;
    f.pos := last component of f + 1;
  END {Append};

PROCEDURE Read(f,x);
  BEGIN
    IF NOT f.readable OR Eof(f) THEN error
    ELSE IF f.lookahead THEN
      BEGIN
        x := f.buffer;
        f.lookahead := false;
      END
    ELSE
      BEGIN
        IF f.component[f.pos] doesn't exist THEN error
        ELSE x := f.component [f.pos];
        f.pos := f.pos+1;
      END;
    f.getok := true;
  END {Read};
```

```
PROCEDURE Write(f,x);
  BEGIN
    IF NOT f.writeable OR (f.pos > f.bound) THEN error
    ELSE
      BEGIN
        f.component [f.pos] := x;
        f.pos := f.pos + 1;
      END;
    f.lookahead := false;  f.getok := false;
    f.buffer := undefined;
  END {Write};

FUNCTION Position (f): integer;
  BEGIN
    IF f.lookahead THEN Position := f.pos - 1
    ELSE Position := f.pos;
  END {Position};

PROCEDURE Seek(f,k);
  BEGIN
    IF NOT (f.readable AND f.writeable) THEN error
    ELSE
      f.pos := k;
    f.lookahead := false;  f.getok := false;
    f.buffer := undefined;
  END {Seek};

PROCEDURE Look (f); {Local procedure to file buffer variable}
  BEGIN
    IF f.getok THEN
      IF f.endoffile THEN error
      ELSE
        BEGIN
          IF component, f.pos, doesn't exist THEN
            f.endoffile := true
          ELSE
            BEGIN
              f.buffer;=f.component [f.pos];
              f.pos := f.pos + 1;
              f.lookahead := true;
            END;
          f.getok := false;
        END;
  END {Look};
```

I/O DEFINITIONS

```
FUNCTION Eof(f): boolean;
  BEGIN
    IF f.readable AND f.writeable THEN
      f.endoffile := f.pos > f.bound
    ELSE IF NOT f.endoffile THEN Look (f);
    Eof := f.endoffile;
  END (Eof);

FUNCTION Maxpos(f): integer;
  BEGIN
    IF NOT (f.readable AND f.writeable) THEN error;
    maxpos := f.bound;
  END (Maxpos);

f^
  BEGIN
    Look (f);
    f^ := f.buffer;
  END (f^);

PROCEDURE Get(f);
  BEGIN
    IF f.endoffile OR NOT f.readable THEN error;
    IF f.getok THEN Look (f);
    f.getok := true;  f.lookahead := false;
  END (Get);

PROCEDURE Put(f);
  BEGIN
    Write(f, f.buffer);
  END (Put);

PROCEDURE Close(f,s);
  BEGIN
    If s is given then perform a system dependent action;
    Return the file contents to the system;
    f.readable := false;  f.writeable := false;
    f.getok := false;  f.endoffile := true;
  END (Close);
```

INDEX

A page reference marked with an asterisk (*) indicates the definition of a term or feature.

INDEX

INDEX

INDEX