RDOK 16750

```
PPPPP   AAAAA   SSSSS   CCCCC   AAAAA   L
P   P   A   A   S       C       A   A   L
PPPPP   AAAAA   SSSSS   C       AAAAA   L
P       A   A       S   C       A   A   L
P       A   A   SSSSS   CCCCC   A   A   LLLLL
```

REFERENCE MANUAL

(Mark 3.4.1 Field Test Version)

Burroughs Corporation

Mission Viejo Plant

December 15, 1983

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a Field Communication Form (FCF) with the Class specified as "2" (System Software), the Type specified as "1" (F.T.R.), and the Product specified as the seven-digit form number of the manual (for example, "5011760"). The FCF should be sent to the following address:

Burroughs Corporation
PA&S/Mission Viejo
25725 Jeronimo Road
Mission Viejo, CA 92691

TABLE OF CONTENTS

1	INTRODUCTION	1
	COMPLIANCE STATEMENT	2
	STRUCTURE OF THIS MANUAL	3
	RELATED DOCUMENTS	6
2	PROGRAM STRUCTURE	7
2.1	PROGRAMS	8
	PROGRAM PARAMETERS	10
	BLOCKS	12
	Scope	12
	Activations	16
2.2	LIBRARIES	18
	USAGE CLAUSE	20
	Sharing Option	20
	Duration Option	22
	INTERFACE PART	24
	Referencing a Library Entry Point	25
3	DECLARATIONS AND DEFINITIONS	29
3.1	LABEL DECLARATIONS	30
3.2	CONSTANT DEFINITIONS	31
3.3	TYPE DEFINITIONS	34
3.3.1	TYPE CONCEPTS	36
	SIMPLE, STRUCTURED, AND POINTER TYPES	36
	ORDINAL TYPES	37
	TYPE IDENTIFIERS	38
	SAME TYPES	39
	COMPATIBLE TYPES	40
	ASSIGNMENT COMPATIBILITY	41
3.3.2	TYPE DESCRIPTIONS	44
	ARRAY TYPES	44
	String Types	45
	BOOLEAN TYPES	47
	CHAR TYPES	48
	ENUMERATED TYPES	49
	FILE TYPES	51
	INTEGER TYPES	52
	POINTER TYPES	53
	REAL TYPES	54
	RECORD TYPES	55
	SET TYPES	59
	SUBRANGE TYPES	61
	TEXTFILE TYPES	63
	VARIABLE-LENGTH STRING (VLSTRING) TYPES	64
3.4	VARIABLE DECLARATIONS	65
3.5	LIBRARY DECLARATIONS	68
3.6	PROCEDURE AND FUNCTION DECLARATIONS	70
	PROCEDURE DECLARATION	71
	FUNCTION DECLARATION	75

FORMAL PARAMETER LISTS 78
ACTUAL PARAMETER LISTS AND PARAMETER MATCHING. 80
Parameter List Congruity 81

4 STATEMENTS 83
ASSIGNMENT STATEMENTS. 85
CASE STATEMENTS. 86
COMPOUND STATEMENTS. 88
FOR STATEMENTS 89
GOTO STATEMENTS. 92
IF STATEMENTS. 95
PROCEDURE INVOCATION STATEMENTS. 97
REPEAT STATEMENTS. 99
WHILE STATEMENTS 100
WITH STATEMENTS. 101

5 EXPRESSIONS. 103
5.1 EXPRESSION CONCEPTS. 105
ARITHMETIC EXPRESSIONS 105
ORDINAL EXPRESSIONS. 105
PRECEDENCE OF OPERATORS. 106
FUNCTION DESIGNATORS 109
5.2 EXPRESSIONS BY TYPE. 111
BOOLEAN EXPRESSIONS. 111
Relational Expressions 113
CHAR EXPRESSIONS 118
ENUMERATED EXPRESSIONS 119
INTEGER EXPRESSIONS. 120
POINTER EXPRESSIONS. 122
REAL EXPRESSIONS 124
SET EXPRESSIONS. 126
STRING EXPRESSIONS 128
VLSTRING EXPRESSIONS 129

6 PREDEFINED PROCEDURES AND FUNCTIONS. 131
6.1 FILE HANDLING PROCEDURES AND FUNCTIONS 132
6.1.1 I/O CONCEPTS 134
TERMINOLOGY. 135
Standard Files and Textfiles 135
Inspection Mode and Generation Mode. 135
Buffer Variables 135
File Attributes. 136
Logical and Physical Files 136
Permanent and Temporary Files. 137
STANDARD FILES 139
Inspection Mode Operations on Standard Files 139
Generation Mode Operations on Standard Files 141
Inspection/Generation Operations on Standard Files . . . 143
Representation of Standard Files 144
TEXTFILES. 145
Inspection Mode Operations on Textfiles. 145
Generation Mode Operations on Textfiles. 148
Representation of Textfiles. 149
The Textfiles "Input" and "Output" 151
USE OF FILE ATTRIBUTES 152

I/O EXCEPTION HANDLING 155
6.1.2 PROCEDURE AND FUNCTION DESCRIPTIONS. 160
ADDSTATION PROCEDURE 160
CLOSE PROCEDURE. 161
DELETESTATION PROCEDURE. 164
EOF FUNCTION 165
EOLN FUNCTION. 166
FILEVALUE FUNCTION 167
GET PROCEDURE. 168
IORES FUNCTION 170
OPEN PROCEDURE 171
PAGE PROCEDURE 174
PUT PROCEDURE. 175
READ PROCEDURE 177
READ TEXTFILE PROCEDURE. 178
READLN PROCEDURE 184
RESET PROCEDURE. 185
REWRITE PROCEDURE. 186
SEEK PROCEDURE 187
SKIPTOCHANNEL PROCEDURE. 188
WRITE PROCEDURE. 189
WRITE TEXTFILE PROCEDURE 190
WRITELN PROCEDURE. 195
6.2 TYPE TRANSFER PROCEDURES AND FUNCTIONS 196
CHR FUNCTION 197
ORD FUNCTION 198
ORDINAL TYPE TRANSFER FUNCTION 199
PACK PROCEDURE 200
UNPACK PROCEDURE 202
6.3 DYNAMIC ALLOCATION PROCEDURES. 204
DISPOSE PROCEDURE. 208
MARK PROCEDURE 209
NEW PROCEDURE. 210
RELEASE PROCEDURE. 211
6.4 LIBRARY HANDLING PROCEDURES AND FUNCTIONS. 212
CANCEL PROCEDURE 213
FREEZE PROCEDURE 214
LIBRARYVALUE FUNCTION. 215
6.5 STRING HANDLING PROCEDURES AND FUNCTIONS 216
CONCAT FUNCTION. 217
DELETE PROCEDURE 218
FILLCHAR PROCEDURE 219
INSERT PROCEDURE 220
LENGTH FUNCTION. 221
POS FUNCTION 222
SCAN FUNCTION. 223
STRING FUNCTION. 225
6.6 ARITHMETIC FUNCTIONS 226
ABS FUNCTION 227
ARCCOS FUNCTION. 227
ARCSIN FUNCTION. 228
ARCTAN FUNCTION. 228
ARCTANH FUNCTION 228
COS FUNCTION 229
COSH FUNCTION. 229

COTAN FUNCTION 229
ERF FUNCTION 230
EXP FUNCTION 230
GAMMA FUNCTION 230
LN FUNCTION. 231
LNGAMMA FUNCTION 231
LOG FUNCTION 231
MAX FUNCTION 232
MIN FUNCTION 233
RANDOM FUNCTION. 234
ROUND FUNCTION 235
SIN FUNCTION 235
SINH FUNCTION. 236
SQR FUNCTION 236
SQRT FUNCTION. 236
TAN FUNCTION 237
TANH FUNCTION. 237
TRUNC FUNCTION 237

6.7 GENERAL PROCEDURES AND FUNCTIONS 238
ABORT PROCEDURE. 239
ACCEPT PROCEDURE 240
DATE PROCEDURE 241
DISPLAY PROCEDURE. 243
ELAPSEDTIME FUNCTION 244
GETATTRIBUTE PROCEDURE 245
IOTIME FUNCTION. 247
ODD FUNCTION 248
PRED FUNCTION. 249
RUNTIME FUNCTION 250
SETATTRIBUTE PROCEDURE 251
SUCC FUNCTION. 253
TIME PROCEDURE 254
WAIT PROCEDURE 256

7 VARIABLES. 257
7.1 VARIABLES BY ACCESS. 257
ENTIRE VARIABLES 258
INDEXED VARIABLES. 259
FIELD DESIGNATORS. 261
DYNAMIC VARIABLES. 262
BUFFER VARIABLES 264
7.2 VARIABLES BY TYPE. 265
ARRAY VARIABLE 265
BOOLEAN VARIABLE 265
CHAR VARIABLE. 265
ENUMERATED VARIABLE. 265
FILE VARIABLE. 265
INTEGER VARIABLE 266
POINTER VARIABLE 266
REAL VARIABLE. 266
RECORD VARIABLE. 266
SET VARIABLE 266
STRING VARIABLE. 266
STATION VARIABLE 267
SUBFILE VARIABLE 267

TEXTFILE VARIABLE. 268
VLSTRING VARIABLE. 268
7.3 UNDEFINED VARIABLES. 269

8 BASIC COMPONENTS 271
CHARACTERS AND CHARACTER STRINGS 272
IDENTIFIERS. 273
NUMBERS. 274
FILE ATTRIBUTES AND MNEMONIC VALUES. 276

9 INTERPRETATION OF PROGRAM TEXT 277
PROGRAM TEXT 278
TOKEN. 278
Reserved Word. 279
Predefined Identifier. 279
Context-Sensitive Identifier 280
Special Token. 281
TOKEN SEPARATOR. 282
Blank. 282
Comment. 282
Record Boundary. 283

A COMPILER CONTROL RECORDS 285
A.1 OPTION PHRASE. 287
A.2 OPTION EXPRESSIONS 288
A.3 OPTIONS. 289
ANSI OPTION. 290
CLEAR OPTION 291
CODE OPTION. 291
DELETE OPTION. 292
ERRORLIMIT OPTION. 293
ERRORLIST OPTION 293
INCLNEW OPTION 294
INCLUDE OPTION 294
LINEINFO OPTION. 296
LIST OPTION. 296
LISTDOLLAR OPTION. 297
LISTINCL OPTION. 297
MAP OPTION 297
MERGE OPTION 298
NEW OPTION 298
NOBOUNDS OPTION. 299
OMIT OPTION. 300
PAGE OPTION. 300
SEQUENCE (SEQ) OPTION. 301
SEQUENCE BASE OPTION 302
SEQUENCE INCREMENT OPTION. 302
STATISTICS OPTION. 303
STRINGS OPTION 303
USER OPTIONS 304
VOID OPTION. 305
WARNSUPR OPTION. 306
XREF OPTION. 306
XREFFILES OPTION 307

B COMPILER FILES 309
INTERACTIVE (CANDE) COMPILATION. 311
BATCH (WFL) COMPILATION. 312

C DATA REPRESENTATION. 313
C.1 SIMPLE TYPES 315
BOOLEANS 315
CHARACTERS 316
ENUMERATIONS 317
INTEGERS 317
REALS. 318
SUBRANGES. 319
C.2 STRUCTURED TYPES 320
ARRAYS 320
Unpacked Arrays. 320
Packed Arrays. 321
FILES. 321
RECORDS. 322
Unpacked Records 322
Packed Records 322
SETS 324
Unpacked Sets. 324
Packed Sets. 324
TEXTFILES. 324
VLSTRINGS. 325
C.3 POINTERS 326
C.4 UNDEFINED OPERANDS 326

D EBCDIC AND ASCII CHARACTER SETS. 327

E COMPARISON WITH ANSI PASCAL. 339
IMPLEMENTATION-DEFINED FEATURES. 340
IMPLEMENTATION-DEPENDENT FEATURES. 343
EXTENSIONS TO STANDARD PASCAL. 345

F ERROR DETECTION AND REPORTING. 347

G RAILROAD DIAGRAMS. 351

H GLOSSARY 355

1 INTRODUCTION

Pascal is a high-level programming language developed by Niklaus Wirth, based on the block-structured nature of ALGOL-60 and the data structuring innovations of C. A. R. Hoare. Because Pascal is an easy-to-learn, general-purpose language, its popularity has increased dramatically in the last several years, particularly in the university and personal computer markets.

The American National Standards Institute (ANSI) has adopted the International Standards Organization (ISO) standard 7185 Level 0 as their standard definition of Pascal. The purpose of the ANSI standard is to increase the portability of Pascal programs from one system to another. The Burroughs B 5000/B 6000/B 7000 Pascal Compiler complies with this standard with the restrictions described later in this section (refer to Compliance Statement, below). Throughout the remainder of this manual, the Burroughs B 5000/B 6000/B 7000 Pascal Compiler is referred to as "Burroughs Pascal", and the Pascal described by the ANSI Standard is referred to as "ANSI Pascal".

This manual is intended as a reference manual for Burroughs Pascal. As such, its purpose is to be a complete description of the syntax and semantics of Burroughs Pascal, within a framework that is designed for quick access of information. The reader is assumed to be familiar with programming language concepts and with the Burroughs B 5000/ B 6000/B 7000 family of systems. Some advance knowledge of the Pascal language is helpful.

The notation used in this manual to represent the syntax of Pascal is the "railroad" syntax diagram, which is frequently used in Burroughs manuals. For those unfamiliar with this notation, a complete description is provided in Appendix G, Railroad Diagrams.

The remainder of this chapter describes the compiler's compliance with the ANSI standard for Pascal, the structure of this manual, and the documents that relate to this description of Burroughs Pascal.

COMPLIANCE STATEMENT

The Burroughs Pascal compiler for B 5000/B 6000/B 7000 systems ("Burroughs Pascal") complies with the requirements of ANSI/IEEE 770X3.97-1982 and with Level 0 ISO 7185 with the following exceptions:

1) The following violations by a program of the requirements of ANSI Pascal are not currently detected:

   a) Illegal branches into <compound statement>s.

   b) Incomplete specification of <case constant>s in the <variant part> of a record declaration.

2) The following features are not currently implemented:

   a) The declaration of files and textfiles as components of structured types.

   b) Dynamic variables of type file or type textfile.

Further information on the relationship of Burroughs Pascal to ANSI Pascal is contained in Appendix E, Comparison with ANSI Pascal, and Appendix F, Error Detection and Reporting.

See also
Comparison with ANSI Pascal 340
Error Detection and Reporting 348

STRUCTURE OF THIS MANUAL

This manual is organized in a "top-down" structure. That is, larger and/or higher-level syntactic components, such as programs, declarations, and statements, are described first. Smaller, lower-level syntactic components, such as variables and identifiers, are described last. The manual includes the chapters and appendices listed below.

1. INTRODUCTION

   This chapter introduces the language and the manual; it also includes the statement of compliance with the ANSI standard for Pascal.

2. PROGRAM STRUCTURE

   The Program Structure chapter describes Pascal programs, libraries, program parameters, and blocks. This chapter also includes discussions of the scope of identifiers and of activations.

3. DECLARATIONS AND DEFINITIONS

   The Declarations and Definitions chapter contains a description of the declaration part of a block, including type definitions and variable declarations. The type definitions section includes discussions of concepts relating to data types in Pascal.

4. STATEMENTS

   The Statements chapter describes the statement constructs available in Pascal.

5. EXPRESSIONS

   The Expressions chapter describes all expression types; it also contains a discussion of the precedence of operators within expressions.

6. PREDEFINED PROCEDURES AND FUNCTIONS

   The Predefined Procedures and Functions chapter describes the various procedures and functions that are available in the language without the programmer's having to define them. These procedures and functions provide facilities in the areas of file handling, type transfer, dynamic variable allocation, library handling, string handling, arithmetic functions, and other general features. The

File Handling Procedures Section contains a detailed description of Pascal input/output concepts and how they relate to the Burroughs B 5000/B 6000/B 7000 I/O subsystem.

7. VARIABLES

The Variables chapter describes variables of different types and how they are referenced within the program.

8. BASIC COMPONENTS

The Basic Components chapter defines some of the small and frequently used components of the syntax of Pascal, such as identifiers and numbers.

9. INTERPRETATION OF PROGRAM TEXT

The Interpretation of Program Text chapter describes how the Burroughs Pascal compiler interprets the program information it reads from its input files. This chapter includes lists of reserved words, predefined identifiers, and context-sensitive identifiers. It also describes the use of comments within the program text.

APPENDIX A. COMPILER CONTROL RECORDS

The Compiler Control Records appendix defines the syntax and semantics of the options that can be used to direct certain aspects of the compilation process.

APPENDIX B. COMPILER FILES

The Compiler Files appendix describes the compiler's input and output files. It also includes a description of how to initiate a compile through the Command AND Edit language (CANDE) and through the Work Flow Language (WFL).

APPENDIX C. DATA REPRESENTATION

The Data Representation appendix describes the format the compiler uses to represent data of the various types.

APPENDIX D. EBCDIC AND ASCII CHARACTER SETS

This appendix contains a table of the ordinal numbers and hexadecimal codes for all EBCDIC and ASCII characters.

APPENDIX E. COMPARISON WITH ANSI PASCAL

This appendix compares Burroughs Pascal to ANSI Pascal in the areas of implementation-defined features, implementation-dependent features, and Burroughs extensions.

APPENDIX F. ERROR DETECTION AND REPORTING

The Error Detection and Reporting appendix describes compile-time and run-time detection of errors; it also describes the errors defined in ANSI Pascal that are not detected in Burroughs Pascal.

APPENDIX G. RAILROAD DIAGRAMS

The Railroad Diagrams appendix describes the notation used throughout this manual to represent the syntax of Pascal.

APPENDIX H. GLOSSARY

The Glossary contains definitions for many of the terms used to describe the semantics of Pascal.

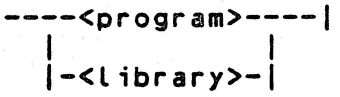
RELATED DOCUMENTS

The following documents contain information that may be of interest to the reader of this manual:

Document	Form No.
CANDE Reference Manual	5014541
I/O Subsystem Reference Manual	5014483
Operator Display Terminal (ODT) Reference Manual	5014491
System Software Utilities Reference Manual	5014426
Work Flow Language (WFL) Reference Manual	5011794
Burroughs Network Architecture (BNA) Architectural Description Reference Manual, Volume 1	1132172
Proposed American National Standard Programming Language Pascal (X3J9/81-093)	
Pascal User Manual and Report, by K. Jensen and N. Wirth, Springer-Verlag, New York, 1978.	

2 PROGRAM STRUCTURE

<program unit>



A <program unit> is the most global Pascal construct, encompassing all data definitions and algorithm descriptions that are to be compiled as a unit. There are two types of program units: programs and libraries. "Program" is used here in its familiar meaning -- a Pascal representation of an algorithm that performs a complete task. A "library" does not usually perform a complete task, but instead provides procedures and functions that can be called by a program (or another library) to perform a part of its task.

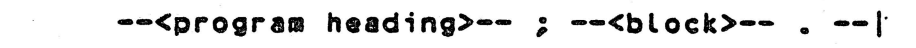
The structure of a library is very similar to the structure of a program, the major differences being that 1) the library's "entry points" (the procedures and functions that are to be available to external programs) must be defined and 2) the outer block of a library is used for initialization only and is run once before the library "freezes" (stops running), at which time the entry points are available to external programs. These concepts are discussed in detail in the Libraries section.

Because of the overall similarity between programs and libraries, the term "program" is used throughout this manual to refer to both programs and libraries. The term "library" is used to specifically refer to libraries.

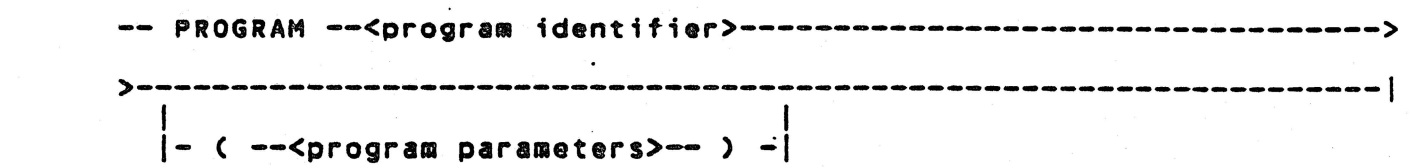
See also
Libraries 18

2.1 PROGRAMS

<program>



<program heading>



<program identifier>



A <program> is similar in format to a procedure or function in that it includes a heading and a <block>. The heading includes the <program identifier>, which is not used for any subsequent purpose, and, optionally, the <program parameters>, which describe permanent files that the program is to read and/or write. The <block> contains the data definitions (declarations) and algorithm descriptions (statements) of the program.

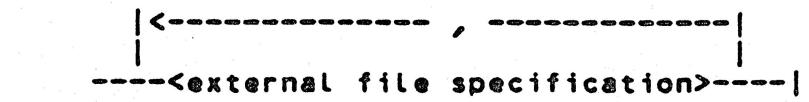
Example

```
program example(input, out_file : file <kind = printer>);

var out_file : text;
    answer : integer;
    val : integer;

function fact (n : integer) : integer;
   begin
   if n > 1 then
      fact := n * fact(n - 1)
   else
      fact := 1;
   end;

begin
rewrite(out_file);
read(input,val);
answer := fact(val);
writeln(out_file, 'The factorial of ',val,' equals ',answer);
end.
```

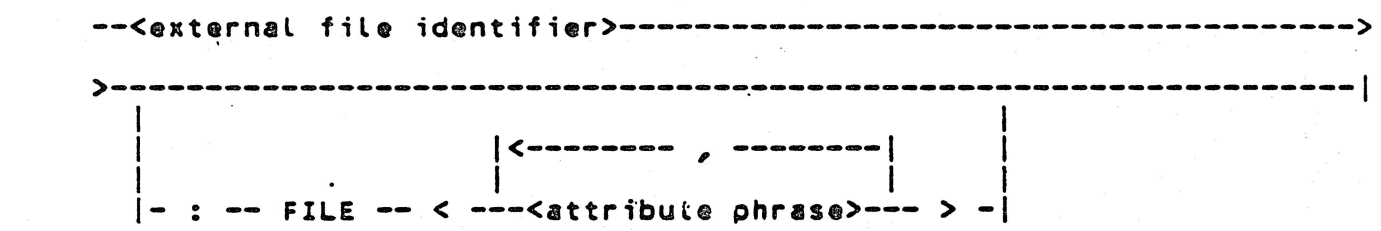
This program will compute the factorial of a number entered through the file "input". The factorial is computed by recursively calling the procedure "fact". The answer is written to file "out_file", which is a printer file.

PROGRAM PARAMETERS

<program parameters>



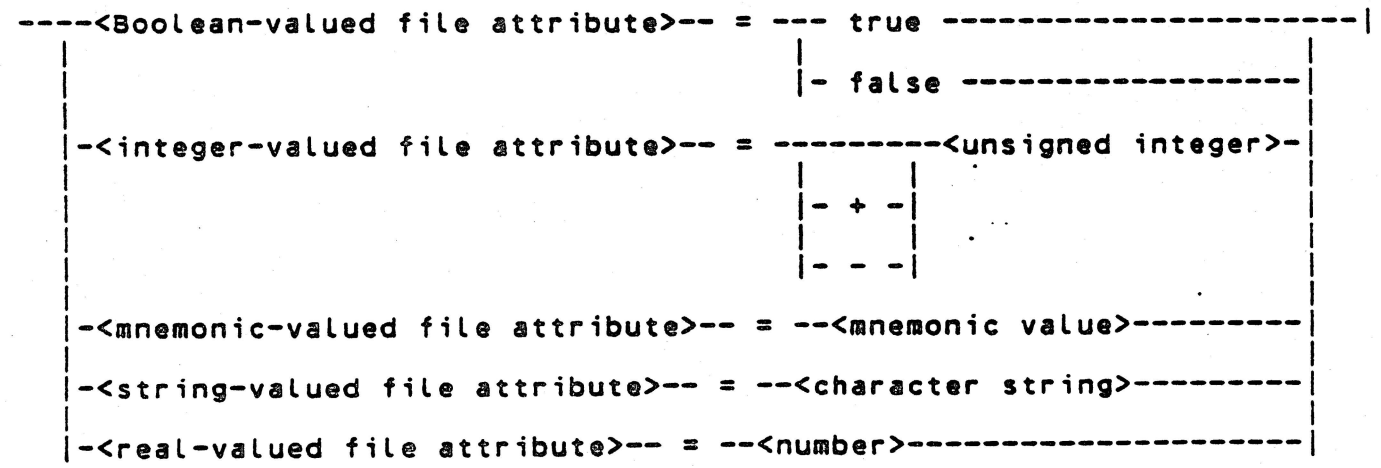
<external file specification>



<external file identifier>



<attribute phrase>



The <program parameters> specify files that the program is to read or write. Optionally, various file attributes of the named files can be assigned values.

An <external file identifier> specified in the program parameters must later appear in the <variable declarations> part of the program <block>, where it must be assigned a <file type> or <textfile type>. The predefined files "input" and "output" are exceptions to this rule; because their appearance in the <program parameters> is equivalent to declaring them in the outer block of the program, they must not appear in the program's <variable declarations>.

When a file appears in the list of <program parameters>, the PROTECTION file attribute for that file is automatically set to SAVE. Thus, if the file is created by the program, it will become a permanent file.

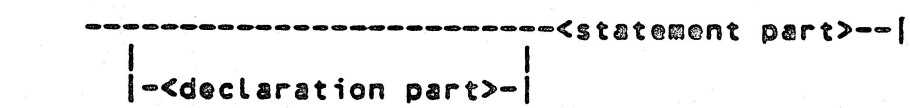
For further information on files, textfiles, and file attributes, please refer to the I/O Concepts section.

The "FILE < <attribute phrase> >" construct (that is, the ability to specify file attributes for program parameters) is a Burroughs extension to ANSI Pascal.

See also
I/O Concepts. 134

BLOCKS

<block>



A <block> is a set of related declarations, which describe data, and statements, which describe actions. The <declaration part> and the <statement part> of blocks are described in the subsequent two chapters.

Pascal is a "block-structured" language derived from the ALGOL family of languages. The Pascal <program> is basically a <block>, and that <block> may contain nested <block>s in the form of procedures and functions. Two related properties of <block>s are fundamental to understanding the structure of a Pascal program: scope and activation.

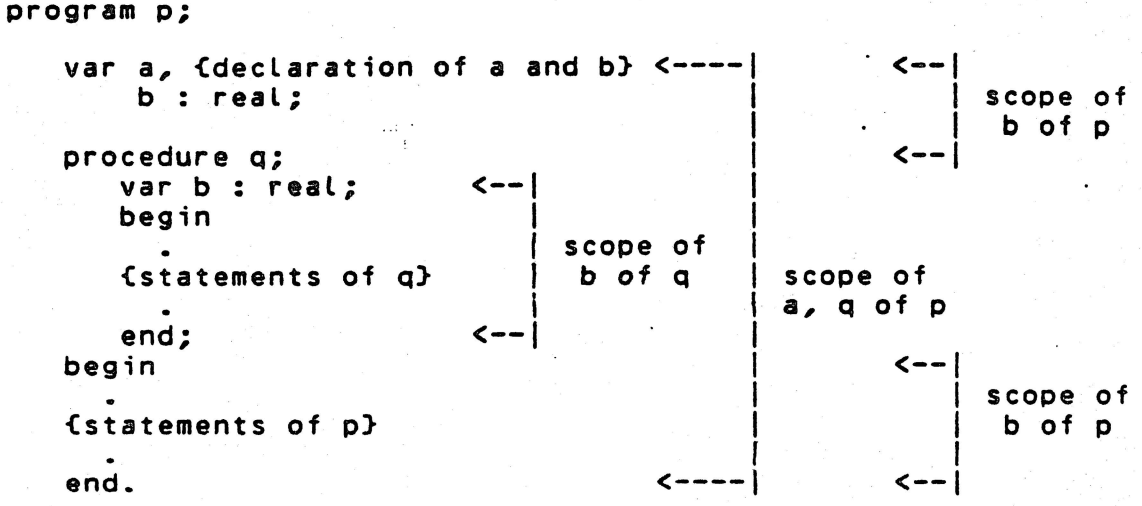
Scope

"Scope" is a property possessed by al. identifiers and labels in a Pascal program. The scope of an identifier or label refers to the region of the program text within which that identifier or label has a specified meaning. The text of a program is divided into regions of scope by the occurrences of blocks, record definitions, WITH statements, and record variable qualifications.

A <block> defines a scope for all identifiers and labels declared in the <declaration part> or <formal parameter list> of that block. If an identifier is declared in block "x", that identifier can be referenced with the defined meaning in all of block "x" and in all procedures, functions, and record definitions within block "x", with the following exception:

> If the same identifier is redefined in the region of a nested procedure, function, or record definition, then the new definition applies in that region and the former definition is unavailable.

Example



A reference to an identifier or label always refers to its closest, or "most local", definition.

The region of a <record type> definition defines a scope for all field identifiers defined in that record. The same nesting rules apply to records as apply to blocks: field identifiers may be redefined in embedded records.

In general, if the occurrence of the definition of an identifier or label is in region "x", that definition does not apply to a region enclosing "x". However, there is one exception: the appearance of an <enumerated constant> in an <enumerated type> definition defines that constant identifier for the closest block containing the definition. Thus, if such a definition occurs within a record, the enumerated constant identifiers can be referenced outside of the record. In the following example, the <enumerated constant>s "red", "green", and "blue" can be referenced within the block in which type "r" is defined:

Example

```
program p;
                                                    <----|
   type r = record                                       |
            f1 : real;                  <--| scope of   |
            f2 : (red, green, blue);    |  f1, f2    |  scope of
            end;                        <--|           |   r, red,
   begin                                              | green, blue
     .                                                |
   {statements of p}                                  |
     .                                                |
   end.                                             <----|
```

Every Pascal program has an implied enclosing region in which all predefined identifiers are automatically declared. Because this region encloses the program, these identifiers can be redefined at any point.

The following rules must be observed when defining identifiers and labels:

a) Any identifier or label that is referenced either must be explicitly defined or must be one of the set of predefined identifiers.

b) With one exception, any reference to an identifier or label must textually follow its definition. The exception is an identifier used to denote the <domain type> of a <pointer type>. In this case, the identifier need only be defined before the end of the <type definitions> in which it appears.

c) An identifier or label cannot be defined more than once in the same procedure, function, or record.

The definition of an identifier or label applies from the beginning to the end of the region, and not from the point of its definition to the end. Thus, a use of an identifier in a region before it is defined is an illegal forward reference even if the same identifier is defined in an enclosing scope.

Example

```
program p;

   type t = real;

   procedure q;
      type x = t;  {invalid}   <--|
           t = integer;           |
      begin                       | scope of
        .                         |  t of q
      {statements of q}           |
        .                         |
      end;                     <--|

   begin
     .
   {statements of p}
     .
   end.
```

The declaration "type x = t" is invalid because it refers to the "t" defined to be "integer" and not the "t" defined to be "real", which appears in the enclosing block.

A WITH statement or record variable qualification defines a new scope for the field identifiers of a referenced record variable.

In a WITH statement, the occurrence of a <record variable> defines a scope for each <field identifier> within that record. The scope extends from the occurrence of the record variable to the end of the WITH statement. WITH statements have the same nesting properties as blocks and records. That is, if a WITH statement causes a field identifier to be defined that has the same spelling as an identifier in an enclosing region, the local (that is, the record) definition applies within the WITH statement.

Record variables may be "qualified" using the syntax "<record variable>.<field designator>". In effect, this syntax establishes a scope for all the field identifiers of the record; the scope extends from the "." to the end of the <field designator>.

See also

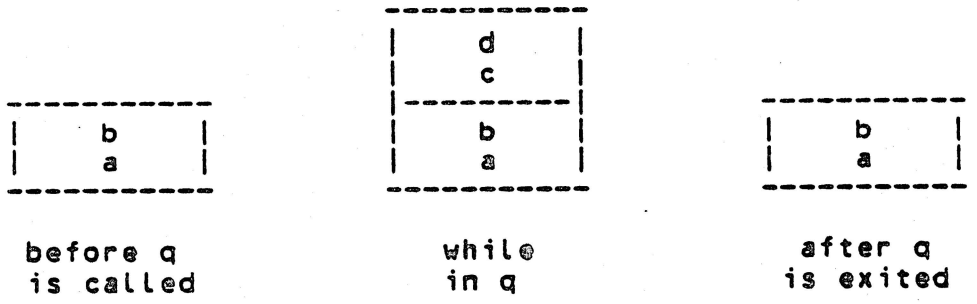
Record Types. 55
With Statements 101

Activations

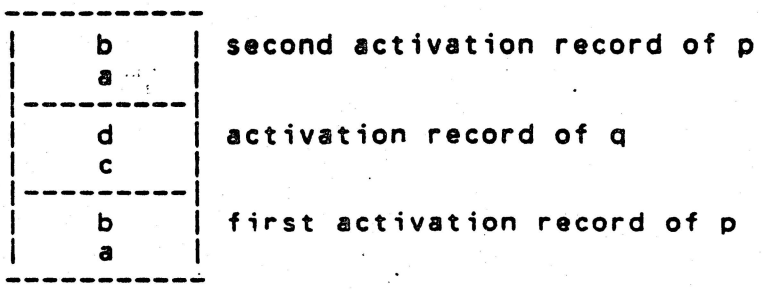
When a <block> is entered, the appropriate local variables must be allocated. These include variables that appear in the <variable declarations> for that <block>, <value parameter>s from the <formal parameter list>, and the function result (if the <block> is a function). These local variables are allocated in an area of storage referred to as an "activation record". Each invocation of a procedure or function has its own activation record, as does the program block.

Storage for an activation record is allocated on entry to the block and deallocated when the block is exited. Thus, on entry, all variables declared within a block are undefined for that invocation. (Pascal local variables differ from FORTRAN local variables and ALGOL "OWN" variables, which retain their previous values when the block is re-entered.) There is one exception to this rule in Pascal. If a file (which is a variable in Pascal) is physically assigned to a permanent file, the "value" of the file can be retained before or after invoking the block in which it is defined.

When a procedure or function is called, the activation record for the current block is saved before the new one is allocated. The processes of allocating and deallocating activation records can be viewed as operations on a stack. Thus, if procedure "p" with local variables "a" and "b" calls procedure "q" with local variables "c" and "d", the storage allocation can be viewed as follows:



A procedure or function can call itself, either directly or indirectly. If, in the previous example, "q" calls "p", the stack will contain the activation records shown below:



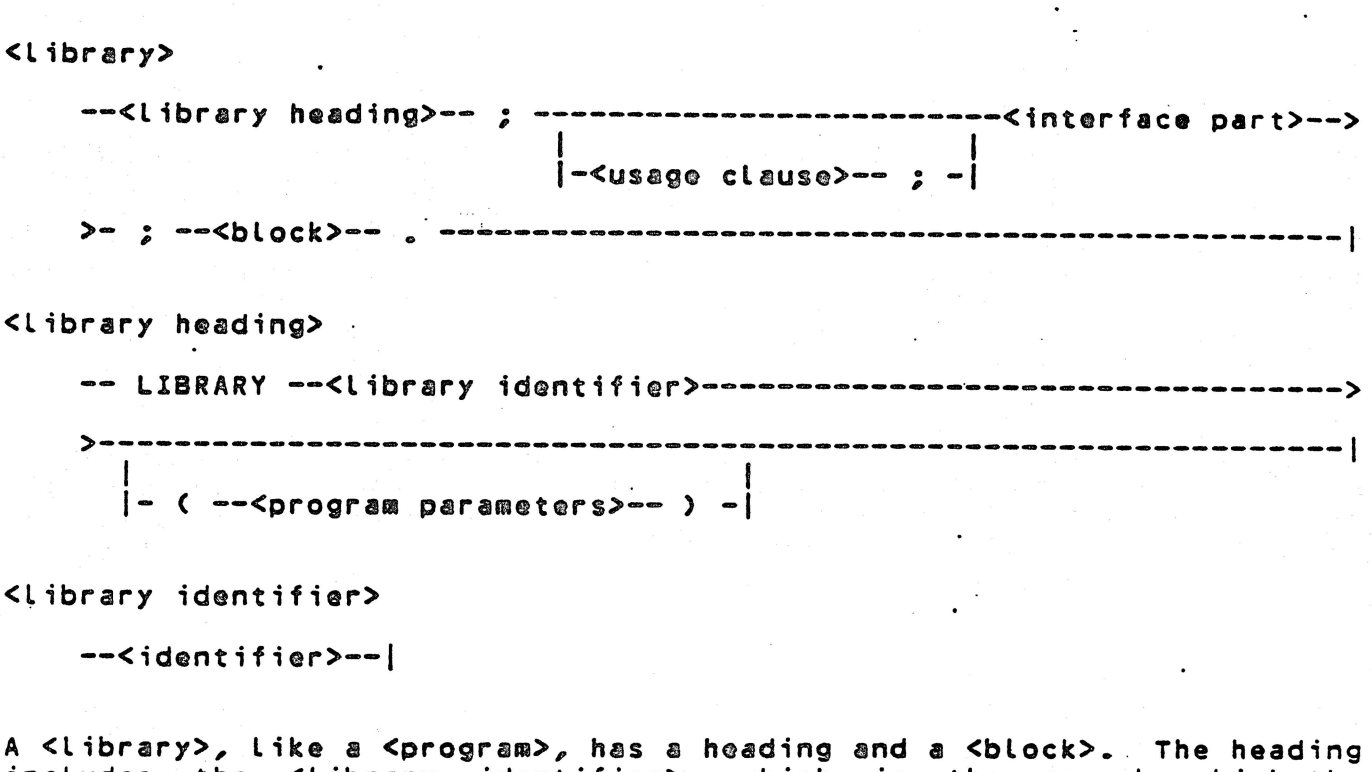
Logically, this process could continue indefinitely; however, the system will eventually run out of storage space.

References to variables in a block refer to the most recently allocated activation record for that block in the stack.

Note that these rules apply to variables. Most are explicitly declared in a block. Variables can also be allocated "dynamically" through the use of the procedure "new". For a discussion of the dynamic allocation of variables, refer to the Dynamic Allocation Procedures section.

See also
Dynamic Allocation Procedures 204

2.2 LIBRARIES



A <library>, like a <program>, has a heading and a <block>. The heading includes the <library identifier>, which is the name by which the library is referenced in programs that call its entry points, and, optionally, the <program parameters>, which are identical in syntax and semantics to those appearing in <program heading>s. The <block> contains the data definitions (declarations) and algorithm descriptions (statements) of the library and is, for the most part, identical in syntax and semantics to the <block> appearing in a <program>.

The <usage clause> defines the duration of the library's existence and the degree to which its global data can be shared among callers. The <interface part> declares the library's "entry points", the procedures and functions that the library makes available to external callers. These items are described in more detail in later subsections.

A library is initiated automatically the first time one of its entry points is invoked by a calling program. When a library is initiated, the code in the outermost block of the the library, referred to as "initialization code", is executed. If a call on the <freeze procedure> appears in the <library>, the library will "freeze" (stop running) at that point, making its entry points available to external programs. If, for some reason, the call on the <freeze procedure> appears in the <library> but is not executed (for example, it appears within a conditional statement and the condition does not allow it to be executed), the library will run to completion and will not freeze. If

there is no call on the <freeze procedure> in the <library>, the library will automatically freeze after the execution of the code in its outermost block.

A library can "thaw" (continue running) when there are no referencing programs, depending on the <duration option> (described later), or when it is referenced in a "THAW" ODT command. When the library "thaws", it resumes execution at the point immediately following its freeze point and executes to termination.

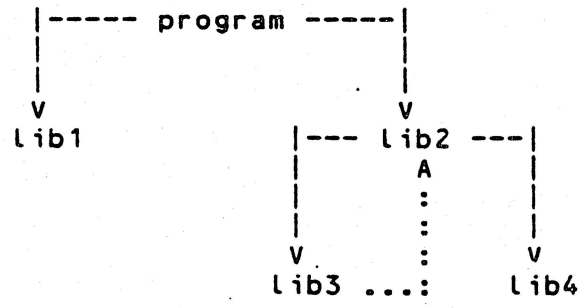
A Pascal program or library that wishes to call one or more library entry points must explicitly specify the library and the entry points it intends to use. The libraries to be used are specified in <library declarations> and their entry poin.s in the <procedure and function declarations>.

See also
Library Declarations. 68
Procedure and Function Declarations 70

Restrictions

The following restrictions apply to the use of libraries:

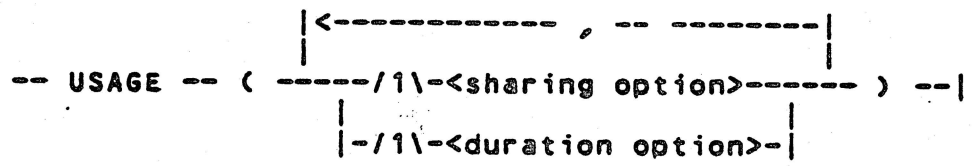
1) Library references cannot be circular. For example, in the following diagram, which shows a program, the libraries it uses, and the libraries that those libraries in turn use, it would be invalid for "lib3" to call "lib2", because such a call creates a circular reference chain:



2) Pointers may not be used as parameters to library entry points. Therefore, no structure containing a pointer can be passed as a parameter.

USAGE CLAUSE

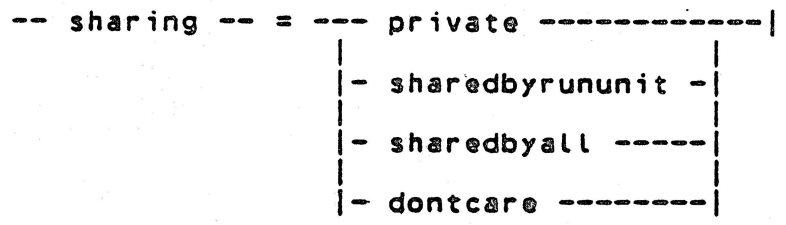
<usage clause>



The <usage clause> specifies the lifetime of the library and the way in which the library is shared by its users.

Sharing Option

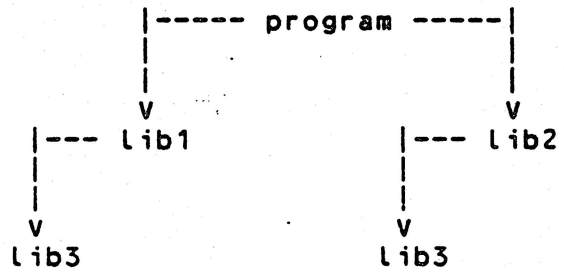
<sharing option>



When a block is entered, an "activation record" is built, containing the variables appearing in the block's <variable declarations> and <formal parameter list>, if any (please refer to the "Activations" section). When a library is initiated, an activation record for its outermost block is built. This activation record contains the allocation of all the global variables of the library. All entry points of a library that address global variables of the library will access this activation record.

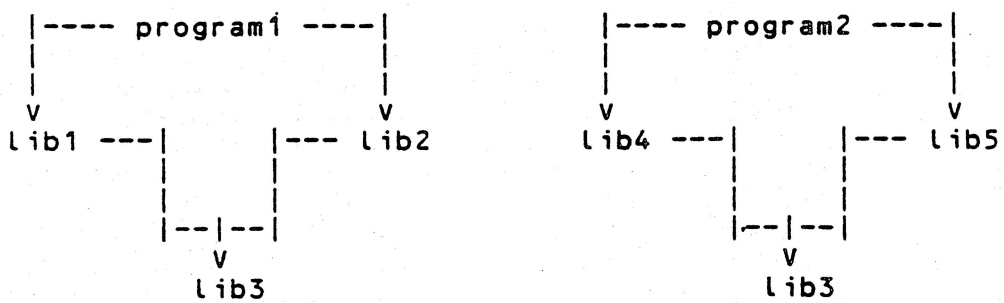
The <sharing option> controls whether or not separate activation records of the outermost block of a library are built when more than one program references it. If this option is not specified, its default value is "sharedbyrununit".

"Private" specifies that a global activation record is created for each declaration of the library. In the following diagram, lib3 is "private" and, therefore, lib1 and lib2 each get a copy of it:

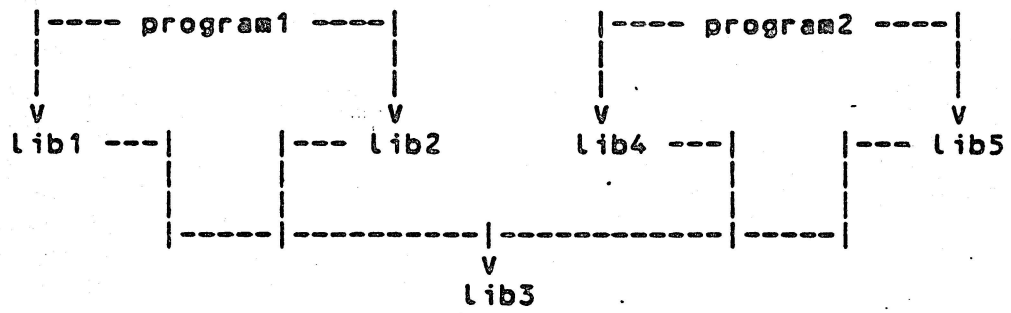


"Sharedbyrununit" specifies that the global activation record is to be shared by a program and all "private" and "sharedbyrununit" libraries that are in the program's run unit. A "run unit" consists of a program and all libraries that are initiated either directly or indirectly by that program. A "program", in this context, does not include either a library that is not frozen or any tasks that are processed off by the program (that is, a process family is not a run unit). A library is its own run unit until after it freezes. For example, if program "p" initiates library "a" and, before it freezes, library "a" initiates library "b", library "b" is in library "a"'s run unit, not in program "p"'s. If library "a" had initiated library "b" after freezing, both library "a" and library "b" would have been in program "p"'s run unit.

All invocations of the library within a run unit share the same instance of the library. For example, in the following diagram, lib3 is "sharedbyrununit":



"Sharedbyall" specifies that the global activation record is to be shared by all simultaneous users of that library. For example, in the following diagram, lib3 is "sharedbyall":



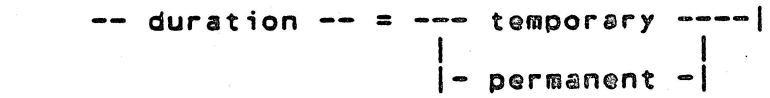
"Dontcare" specifies that any combination of sharing may be used; the sharing option chosen by the MCP remains unknown to the creator and users of the library.

When the global activation record is shared, care must be taken to ensure that one user does not accidentally overwrite information required by another user. Sharing can cause particularly severe problems when multiple users call the dynamic allocation procedures ("new", "dispose", "mark", and "release"). These procedures operate on the "heap", which is a data structure allocated in the activation record of the outermost block of a <program unit>. Because sharing users of the library share the same global activation record, they all access the same heap through the entry points of the library. This situation may result in dangling pointers and/or data corruption.

See also
Dynamic Allocation Procedures 204

Duration Option

<duration option>

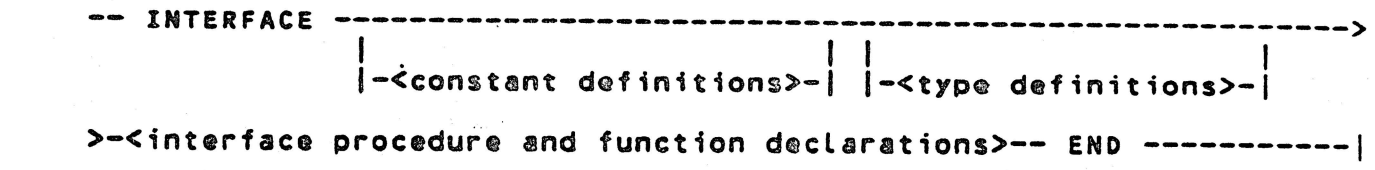


The <duration option> specifies the lifetime of the library, that is, whether the library terminates or remains when no users of the library remain. If this option is not specified, its default value is "temporary".

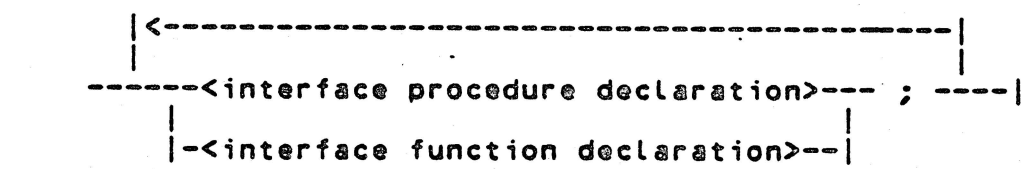
If "temporary" is specified, the library exists only as long as there is at least one reference to it. If the library is not referenced by any user, it "thaws"; that is, it resumes execution at the point immediately following its freeze point and executes to termination. If "permanent" is specified, the created library remains, even if there is no reference to it, until it is explicitly "thawed" through the "THAW" ODT command or is DSed. Please refer to the Operator Display Terminal (ODT) Reference Manual for further information on the "THAW" command.

INTERFACE PART

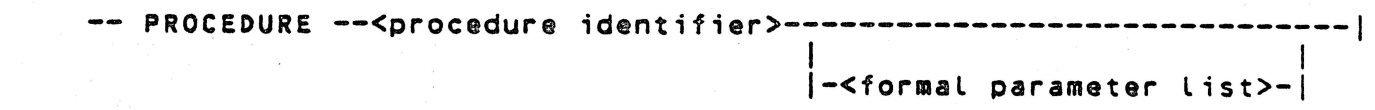
<interface part>



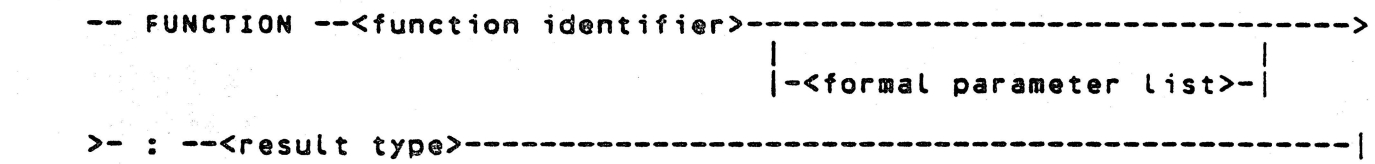
<interface procedure and function declarations>



<interface procedure declaration>



<interface function declaration>



The <interface part> of a <library> defines the set of procedures and/or functions that are provided by the <library> to its users. Only the procedures and functions declared in the <interface part> are available for external use (i.e., are "exported").

The constants and types defined in the <interface part> are not available to the users of the <library>. They are for use in the <interface procedure and function declarations> and may also be used within the block associated with the <library>.

The procedure and function declarations in the <interface part> are incomplete, in that they specify only the procedure or function name, its parameters, and its result type (for functions). The declaration of the procedure or function must be completed in the library <block> by

associating the procedure or function with either a <block> or an <external directive> (please refer to the "Procedure and Function Declarations" section).

See also
Procedure and Function Declarations 70

Referencing a Library Entry Point

To allow a program to invoke a library entry point, the library must be declared in the program's <library declarations> and the entry point must be declared in the program's <procedure and function declarations>, using the <external directive> syntax. Once declared, the entry point can be invoked in the program like any other procedure or function.

During compilation of the program, the procedure or function invocation is verified against the procedure or function declaration in the program, in the same way that any other procedure or function invocation is verified, using the "parameter matching" mechanism described in the "Actual Parameter Lists and Parameter Matching" section. However, during compilation, there is no way to verify that the procedure or function declaration in the program matches the entry point as declared in the library. This verification is performed at run time.

If the description of the entry point in the referencing program does not match its description in the library program, the linkage is not made and the referencing program is terminated. There are several aspects to matching: the entry point type, the number of parameters, the type of parameters, the kind of parameters, and the result type of the entry point if the entry point is a function.

All kinds of parameters (value, variable, procedural, and functional) are allowed. If the library specifies that a parameter is a <variable parameter>, the referencing program can specify that parameter as either a <value parameter> or a <variable parameter>. (Note that a by-name ALGOL parameter cannot be passed to a Pascal library entry point.) If the library specifies that a parameter is a <value parameter>, the referencing program must specify that parameter as a <value parameter>.

Type matching of parameters at run time is complicated by the fact that it is possible that either the library or the referencing program is not written in Pascal and thus may have a different set of data types than are available in Pascal. Even if the library and the referencing program are both written in Pascal, they may have different user-defined types. Type matching between the library entry point declaration and the referencing program's declaration is therefore done on the basis of

general type structure rather than specific type names. Parameters match if they map to the same "generic parameter type", as described below:

Pascal parameter type	Generic parameter type
integer	integer
real	real
Boolean	Boolean
char	integer
enumerated type	integer
short set (1 to 48 elements)	real
long set (more than 48 elements)	array(real) with lower bound
record	array(real) with lower bound
variable length strings	array(real) with lower bound
packed array	See NOTE
array [index-type {,index-type}] of integer	array(integer) with lower bound
array [index-type {,index-type}] of real	array(real) with lower bound
array [index-type {,index-type}] of Boolean	array(Boolean) with lower bound
array [index-type {,index-type}] of char	array(integer) with lower bound
array [index-type {,index-type}] of enumeration	array(integer) with lower bound
array [index-type {,index-type}] of record	array(real) with lower bound
array [index-type {,index-type}] of set (long, short)	array(real) with lower bound
array [index-type {,index-type}] of variable length strings	array(real) with lower bound
array [index-type {,index-type}] of packed array	array(real) with lower bound
textfile	textfile
file	pascalfile

Subranges of types integer, Boolean, char, or enumeration are mapped as their host type, except when they are components of packed arrays, which are described below.

Each user-defined type identifier is resolved to one of the Pascal parameter types shown above according to its general type characteristics. For example, type "color", as it is usually defined,

would be considered an "enumerated type" and would be mapped to the generic type "integer". "Array [index-type1] of array [index-type2] of ..." is equivalent to "array [index-type1, index-type2] of ...", where index-type is one of the allowed types for indices.

NOTE: Packed arrays of integers, reals, sets, records, variable length strings, or other arrays are mapped as unpacked arrays of the same type. For packed arrays of Boolean, char, subrange, or enumeration, the mapping depends on the number of bits it takes to represent the range of the type. If 4 bits or fewer are required, the mapping is to array(hexadecimal character) with lower bound. If 5 to 8 bits are required, the mapping is to array(ebcdic character) with lower bound. If 9 bits or more are required, the mapping is to array(integer) with lower bound. A further description appears in Appendix C, "Data Representation".

See also
Data Representation 314

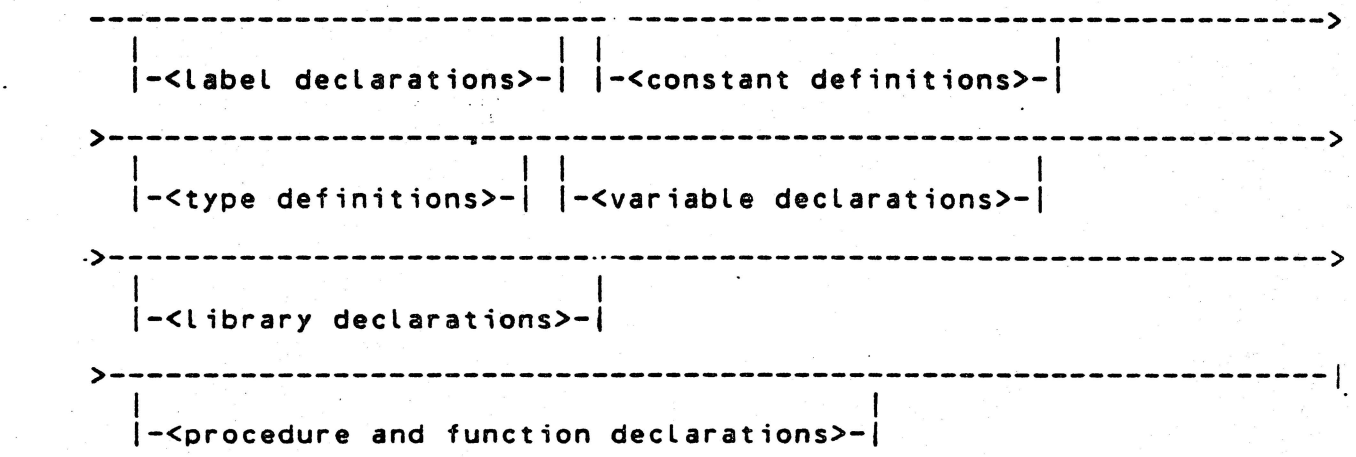
Example

```
library lib; usage(sharing = sharedbyall);
   interface
      type vect = array [1..30] of integer;
      procedure sum (vector1, vector2 : vect);
      function fact (n : integer) : integer;
      function sin (r : real) : real;
   end;

   library mylib (title = 'OBJECT/ARITHLIB');
   procedure sum;
      var i : integer;
      begin
      for i := 1 to 30 do
         vector1[i] := vector1[i] + vector2[i];
      end;
   function sin; mylib;
   function fact;
      begin
      if n < 1 then
         fact := 1
      else
         fact := n * fact(n-1);
      end;

   begin
   freeze;
   end.
```

This library exports the procedure "sum" and the functions "fact" and "sin", all of which appear in the interface part. "Sum" and "fact" are completely declared in the library block. "Sin" is imported from another library.

3 DECLARATIONS AND DEFINITIONS

<declaration part>

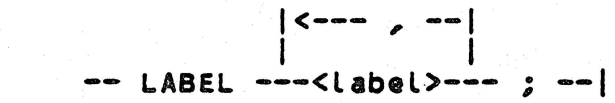


The <declaration part> of a <block> contains a list of declarations and definitions, all of which are optional, but which must appear in the designated order when they appear.

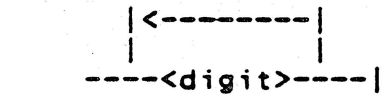
<Constant definitions>, <type definitions>, and <variable declarations> are primarily used to describe the data on which the program is to act. <Label declarations> and <procedure and function declarations> are tools used in describing the program algorithm. <Library declarations> describes other programs from which the program may "import" entry points (i.e., procedures and functions). These components are described in the following sections, in the order in which the components appear in the <declaration part>.

3.1 LABEL DECLARATIONS

<label declarations>



<label>



The <label declarations> identify the <label>s that will be used within the <block>. <Label>s are used to indicate statement locations to which program control can be transferred using the <goto statement>. Any <label> used within a <block> must be declared in the <declaration part> of that <block>.

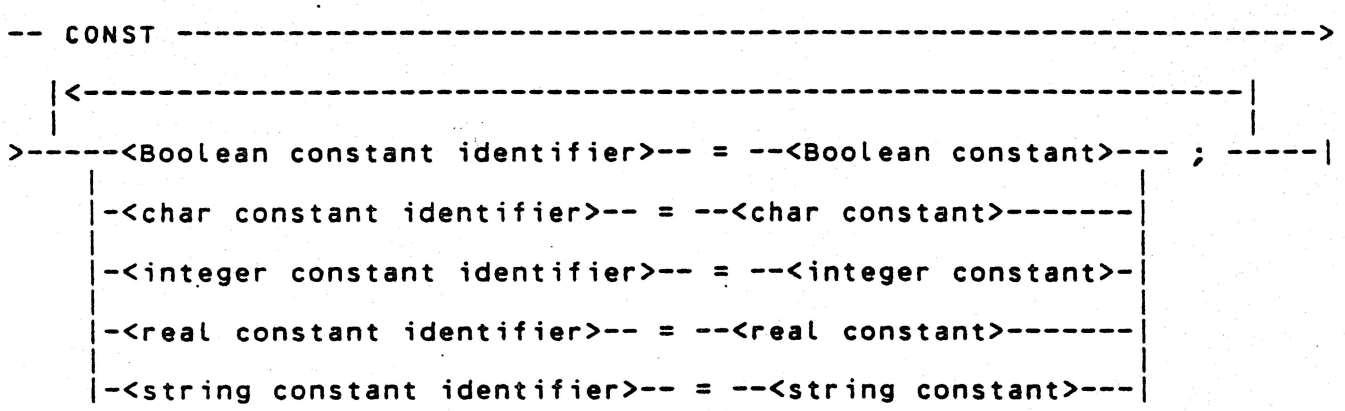
A <label> cannot have more than four significant digits, after leading zeros have been discarded. That is, the smallest <label> number is 0, and the largest <label> number is 9999.

See also

Blocks. 12
Goto Statements 92

3.2 CONSTANT DEFINITIONS

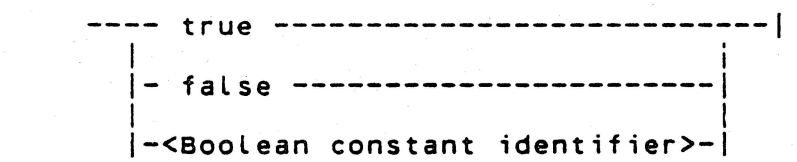
<constant definitions>



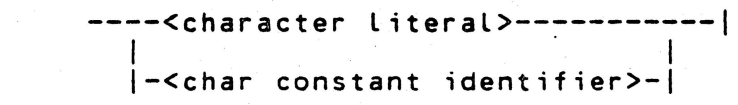
<Boolean constant identifier>
<char constant identifier>
<integer constant identifier>
<real constant identifier>
<string constant identifier>



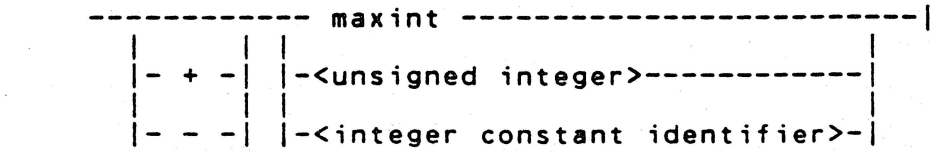
<Boolean constant>



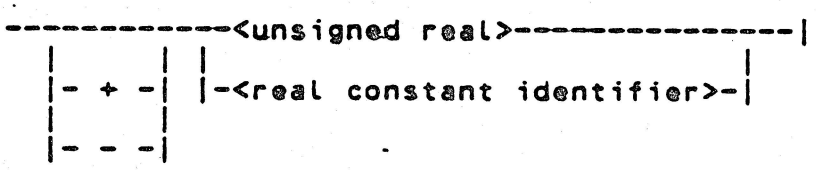
<char constant>



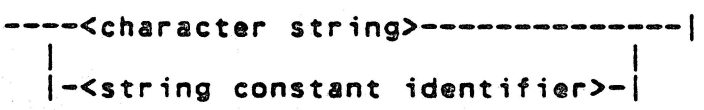
<integer constant>



<real constant>



<string constant>



The <constant definitions> associate <identifier>s with constant values, allowing those values to be referenced by name rather than by specifying the actual values throughout the program. The type of each constant being declared is determined by the type of the constant on the right-hand side of the equal sign, which may be a literal value of a predefined type or a previously declared constant identifier.

"Maxint" is a predefined <integer constant identifier> that has the value 549,755,813,887. "True" and "false" are predefined values of the <Boolean type>. <Identifier>, <character literal>, <unsigned integer>, <unsigned real>, and <character string> are defined in the Basic Components chapter.

See also

Boolean Types 47
Basic Components. 271

Examples

```
const
  always   = true;
  a        = 'a';
  maxbits  = 48;
  minvalue = -4.5;
  greeting = 'Hello';
  intro    = greeting;
  warning  = 'Don''t do it';
```

"Always" is a <Boolean constant identifier> with the value "true"; it may be used wherever a <Boolean constant> is valid.

"A" is a <char constant identifier> that has the value "a".

"Maxbits" is an <integer constant identifier> with the value 48.

"Minvalue" is a <real constant identifier> that has the value -4.5.

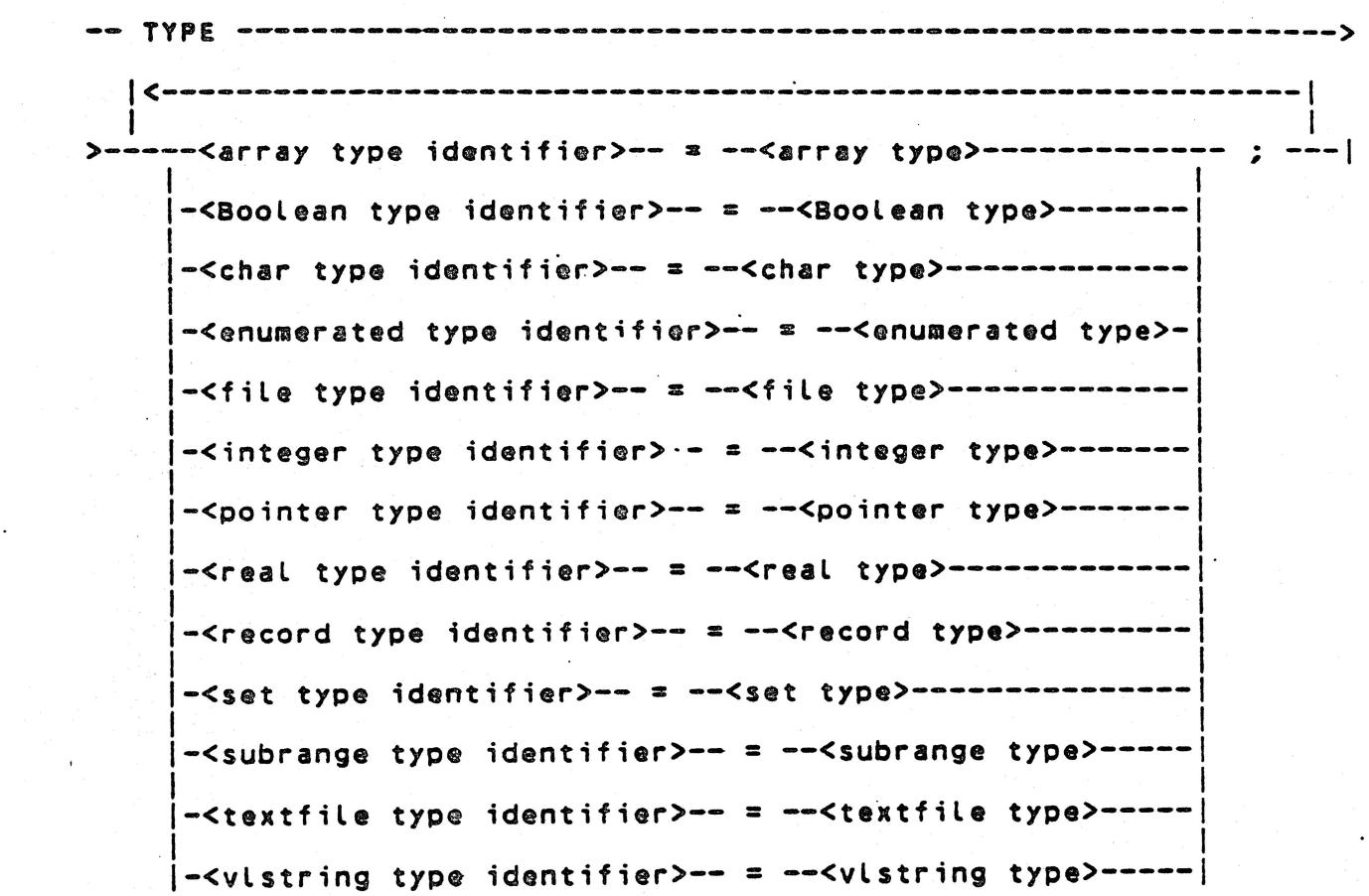
"Greeting" is a <string constant identifier> with the value "Hello".

"Intro" is a <string constant identifier> that has the same value as "greeting".

"Warning" is a <string constant identifier> that has the value "Don't do it".

3.3 TYPE DEFINITIONS

<type definitions>



<array type identifier>
<Boolean type identifier>
<char type identifier>
<enumerated type identifier>
<file type identifier>
<integer type identifier>
<pointer type identifier>
<real type identifier>
<record type identifier>
<set type identifier>
<subrange type identifier>
<textfile type identifier>
<vlstring type identifier>



Every variable, constant, and function has an associated "type", which defines its range of valid values, its internal and external representation, and the operations that may be performed on it. The <type definitions> allow user-defined types to be named and their characteristics specified.

The details of each type's actual representation are described in Appendix C, Data Representation.

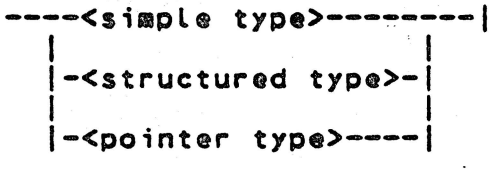
The next subsection describes some general concepts that apply to types, such as type categories, type compatibility, and type identifiers. Subsequent subsections describe the types themselves, in alphabetical order.

See also
Data Representation 314

3.3.1 TYPE CONCEPTS

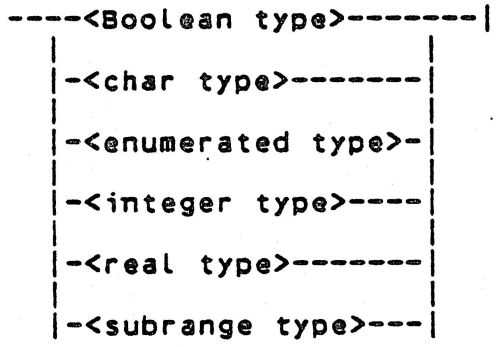
SIMPLE, STRUCTURED, AND POINTER TYPES

<type>



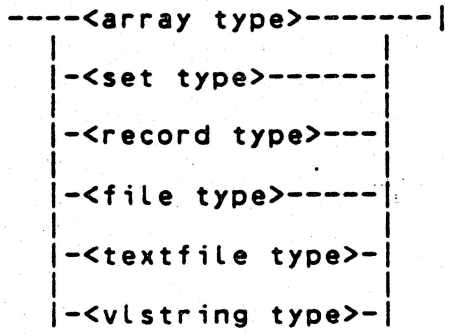
Types can be classified into three categories that relate to their structure: simple types, structured types, and pointer types.

<simple type>

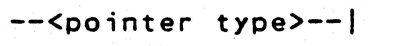


Variables of simple types have only one component. The predefined types Boolean, char, integer, and real are simple types. User-defined derivatives of these predefined types, as well as enumerated types and subrange types, are also simple types.

<structured type>



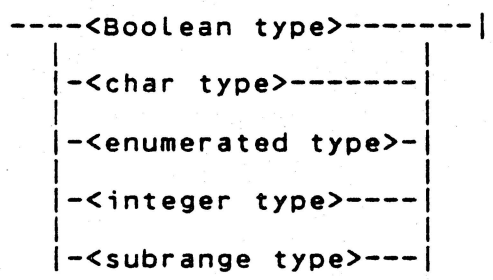
Variables of structured types are composed of multiple components, which may be of one or more simple types, or may be structured themselves. Arrays, sets, records, files, textfiles, and vlstrings are structured types.



Variables of pointer types contain values that are references to variables of simple or structured types.

ORDINAL TYPES

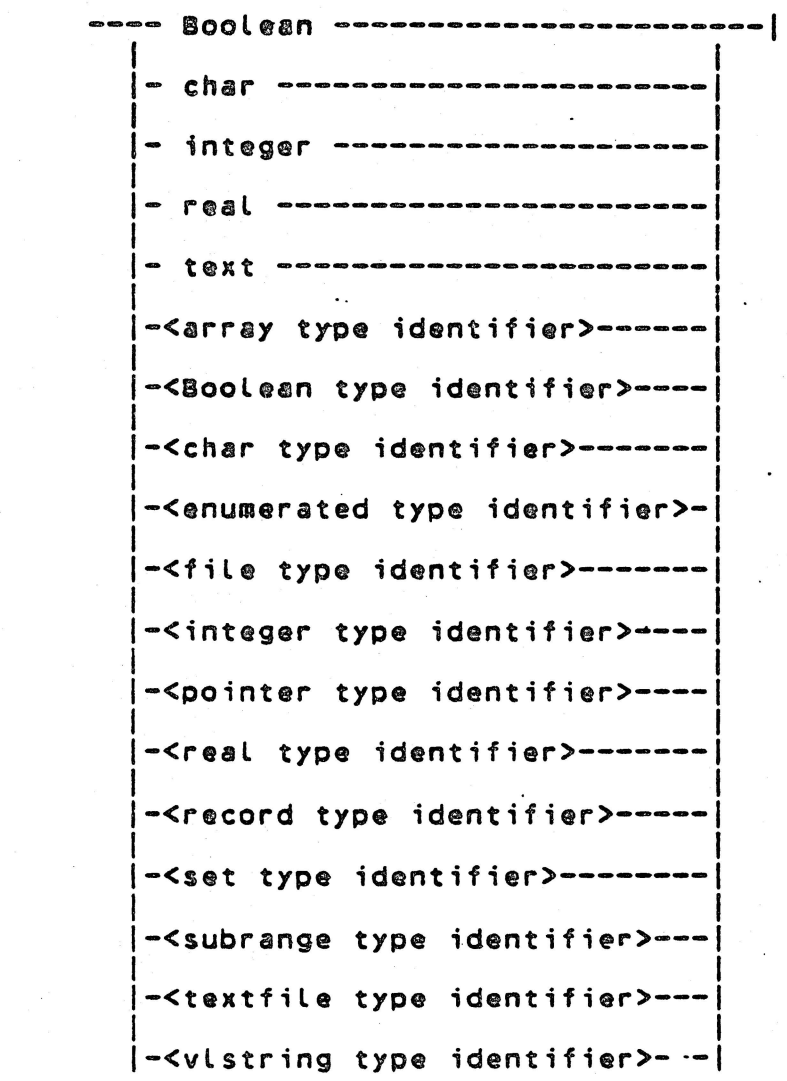
<ordinal type>



Most simple types are also "ordinal types". In an ordinal type, the values have a well-defined sequential relationship to each other. Each value is assigned an "ordinal number" that uniquely identifies its position in the sequence. Thus, a value of an ordinal type can have a "successor" and a "predecessor" in the sequence. Values can also be compared to each other (e.g. greater than, less than) based on their ordinal numbers. The only simple type that is not an ordinal type is the <real type>.

TYPE IDENTIFIERS

<type identifier>



In <type definitions> and <variable declarations>, a type can usually be defined in one of two ways:

1) as a new type (that is, by using the <new array type>, <new enumerated type>, <new file type>, <new pointer type>, <new record type>, <new set type>, <new subrange type>, or <new vlstring type> syntax) or

2) as a derived type, where an <identifier> that has already been defined (or was predefined) as a type identifier is specified.

In other contexts requiring a type specification, new types are not allowed, and previously defined <type identifier>s must be used.

SAME TYPES

Because types can be defined in different ways, it is not always clear when two types are actually the same type. The concept of "same type" is used when describing how <variable parameter>s are matched in procedure and function invocations. More important, the definition of "same type" is used to define compatible types and assignment compatibility, which are described later in this subsection.

The <type identifier>s T1 and T2 are tne "same type" if one of the following conditions is true:

a) one type is defined to be equal to the other or

b) both types are the "same type" as a third type.

The simplest case of "same type" is when T1 is defined to be equal to T2, as shown in the example below:

```
TYPE T1 = T2;   { same type, by rule "a" }
```

Rule "b" describes the situation in which T1 and T2 have a common ancestor, the simplest case of which is the following:

```
TYPE T3 = INTEGER;
     T1 = T3;   { same type, by rule "a" }
     T2 = T3;   { same type, by rule "a" }
{ T1 is the same type as T2, by rule "b" }
```

In the example above, T1 is the "same type" as T3, by the "defined to be equal" rule. Similarly, T2 is also the "same type" as T3. Thus, T1 and T2 are the same type because they are both the same type as T3. By a somewhat longer chain, T1 and T2 can be recognized as the same type in the following example:

```
TYPE T5 = INTEGER;
     T4 = T5;
     T3 = INTEGER;
     T2 = T4;
     T1 = T3;
```

In this example, T2 equals T4, T4 equals T5, and T5 equals INTEGER. T1 equals T3, and T3 equals INTEGER. Therefore, T1 and T2 are the same type, namely INTEGER.

In order to apply the same-type rules, all types must have associated <type identifier>s. For example, even though types T6 and T7, defined below, have exactly the same characteristics and structure, they are NOT the same type:

```
TYPE T6 = ARRAY [1..5] OF INTEGER;
     T7 = ARRAY [1..5] OF INTEGER;
```

However, T6 and T7 would be the same type if declared as follows:

```
TYPE T6 = ARRAY [1..5] OF INTEGER;
     T7 = T6;
```

COMPATIBLE TYPES

In some cases, it is not necessary for types to be the "same type", but they must be "compatible types" for a particular construct to be valid. In particular, the operands in most relational expressions must be of compatible types. Also, the <case constant>s in the <variant> part of a <record type> must be type compatible with the type of the <variant selector>.

Two types, T1 and T2, are compatible if any of the following statements are true:

a) T1 and T2 are the "same type".

b) One type is a subrange of the other, or both types are subranges of the same type.

c) T1 and T2 are <set type>s with compatible <base type>s and either T1 and T2 are both packed or T1 and T2 are both not packed.

d) T1 and T2 are <string type>s.

e) Both types are <vlstring type>s, or one type is a <vlstring type> and the other type is a <string type>.

Allowing <string type>s of different lengths to be compatible is a Burroughs extension to ANSI Pascal.

Examples

```
type t1 = real;
     t2 = t1;
     { t1 and t2 are compatible by rule "a" }

     t3 = 1..10;
     t4 = 5..7;
     t5 = 20..30;
     { t3, t4, and t5 are compatible by rule "b" }

     t6 = set of char;
     t7 = set of 'a'..'z';
     { t6 and t7 are compatible by rule "c" }

     t8 = packed array [1..10] of char;
     t9 = packed array [1..7]  of char;
     t10= string(20);
     { t8 and t9 are compatible by rule "d" }
     { t10 is compatible with t8 and t9 by rule "e" }
```

ASSIGNMENT COMPATIBILITY

Assignment compatibility refers to the validity of assigning a particular value to a variable of a certain type. The rules of assignment compatibility are applied under the following circumstances:

- In an assignment statement, the value of the <expression> must be assignment compatible with the type of the variable or function result being assigned.
- An expression used as an array index must be assignment compatible with the index type in the array declaration.
- The initial value and final value in a <for statement> must be assignment compatible with the type of the control variable.
- An actual parameter must be assignment compatible with the type of the formal value parameter it is to match.
- The values returned by the "read", "time", and "date" procedures must be assignment compatible with the parameters passed to those procedures.

In the definition of assignment compatibility below, V1 and V2 represent two variables, and T1 and T2 are the types of V1 and V2, respectively. As an illustration, consider the assignment statement "V2 := V1". V1 is "assignment compatible" with V2 (or any variable of type T2) if any of the following statements are true:

a) T1 and T2 are the same type and that type is not a <file type> or <textfile type>.

b) V1 and V2 were declared in the same <variable identifier list> in a variable declaration (this rule allows two variables of the same unnamed type to be assignment compatible).

c) T2 is the <real type> and T1 is the <integer type>.

d) T1 and T2 are compatible ordinal types and the value of V1 is valid for type T2.

e) T1 and T2 are compatible set types and all members of the set of V1 are valid for type T2.

f) T1 and T2 are <string type>s and T2 has at least as many components as T1, or T1 is the <char type> and T2 is a <string type>.

g) T1 is a <string type>, <vlstring type>, or the <char type>, T2 is a <vlstring type>, and the number of components in T1 is less than or equal to the <maximum length> of T2.

h) T1 is a <vlstring type>, T2 is a <string type>, and the length of V1 is less than or equal to the number of components in T2.

Allowing string types of different lengths, and character types and string types, to be assignment compatible (rule "f" above) is a Burroughs extension to ANSI Pascal.

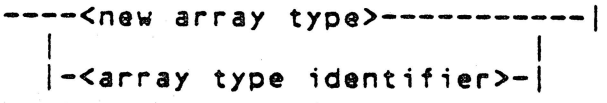
Examples

```
type t1 = real;
     t2 = t1;
     { All values of types t1 and t2 are assignment compatible
       with all variables of types t1 and t2, by rule "a". }

var  v1,
     v2 : array [1..10] of Boolean;
     { All values of v1 are assignment compatible with v2, and vice
       versa, by rule "b". }

     v3 : real;
     v4 : integer;
     { All values of v4 are assignment compatible with v3 (by rule
       "c").  V3 is not assignment compatible with the type of v4.
       That is, "v3 := v4" is allowed, but "v4 := v3" is not. }

     v5 : 7..10;
     v6 : 1..20;
     { All values of v5 are assignment compatible with v6, by rule
       "d", but only some values of v6 are assignment compatible
       with v5. }

     v7 : set of 'a'..'z';
     v8 : set of char;
     { All values of v7 are assignment compatible with v8, by rule
       "e", but only some values of v8 are assignment compatible
       with v7, namely those set values that contain only characters
       between 'a' and 'z', inclusive. }

     v9 : packed array [1..10] of char;
     v10: packed array [1..10] of char;
     { All values of v9 are assignment compatible with v10, and
       vice versa, by rule "f". }

     v11 : string(10);
     v12 : string(20);
     { By rule "g", all values of v11 are assignment compatible with
       v12, but only some values of v12 are assignment compatible
       with v11, namely those strings that have a length of 10
       or less. }

     v13 : packed array [1..5] of char;
     v14 : string(5);
     { All values of v13 are assignment compatible with v14 by
       rule "g", and all values of v14 are assignment compatible
       with v13 by rule "h". }
```

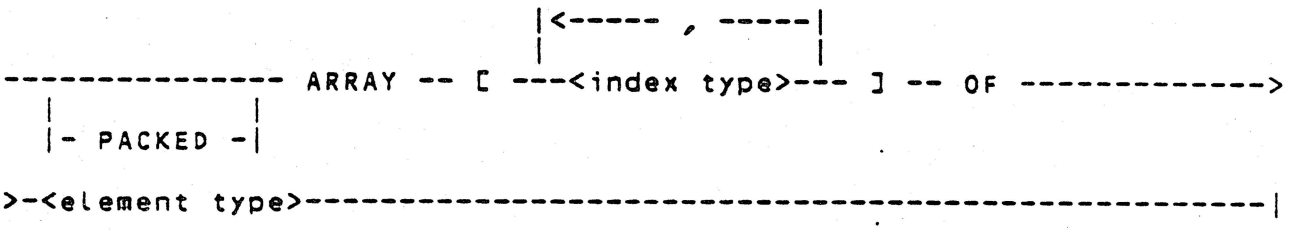
3.3.2 TYPE DESCRIPTIONS

ARRAY TYPES

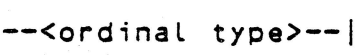
<array type>



<new array type>



<index type>



<element type>

Any <type> that is not a <file type>, a <textfile type>, or a <structured type> containing a <file type> or a <textfile type> as a component.

An array is a structured type containing identical components of a specified <element type>. The array is indexed by the values of a given <index type>. The number of components in the array is determined by the number of values in the <index type>. The <index type> cannot be the <integer type>, but it can be a <subrange type> whose host type is the <integer type>.

If multiple <index type>s are specified, the array is multi-dimensional, each dimension being indexed by one <index type>. An array with N dimensions is synonymous with an array of arrays with N-1 dimensions.

An <array type> that includes the designation "PACKED" will be stored in as economical an amount of space as practical, possibly at the expense of speed in accessing the components. When a multi-dimensional array is

declared using a list of <index type>s and the array is designated PACKED, all component arrays of that array will also be "PACKED" (that is, all dimensions of the array are considered PACKED). (Please refer to Appendix C, Data Representation.)

Examples

```
type t1 = array [Boolean] of array [1..10] of array [size] of real;
     t2 = array [Boolean] of array [1..10, size] of real;
     t3 = array [Boolean, 1..10, size] of real;
     t4 = array [Boolean, 1..10] of array [size] of real;
```

Types "t1", "t2", "t3", and "t4" are equivalent ways of expressing a three-dimensional array with a <component type> of type "real" and with Boolean as its first dimension, the subrange 1..10 as its second dimension, and the <ordinal type identifier> "size" as its third dimension.

```
type p1 = packed array [1..10, 1..8] of Boolean;
     p2 = packed array [1..10] of packed array [1..8] of Boolean;
```

Types "p1" and "p2" are equivalent ways of declaring a packed array with "packed array [1..8] of Boolean" as its component type.

String Types

<string type>

An array that is defined as PACKED ARRAY [1..n] OF CHAR, where n is greater than or equal to 1.

Strings are a special class of arrays that can be used in ways that arrays normally cannot. For example, a variable of a <string type> can be assigned a <character string> value of the same or shorter length; individual characters in the <character string> are assigned to successive components of the array.

When a variable of a <string type> is assigned a value of shorter length or when a formal value parameter of a <string type> is passed an actual parameter of shorter length, the variable or formal parameter is blank filled on the right.

Allowing <string type>s of length 1 and allowing string variables and parameters to be assigned values of shorter length (with blank fill) are Burroughs extensions to ANSI Pascal.

See also
Data Representation 314

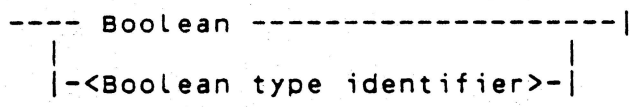
Example

```
type str = packed array [1..10] of char;
```

Type "str" is a <string type> that contains ten characters.

BOOLEAN TYPES

<Boolean type>



"Boolean" is a predefined ordinal type that comprises the values "true" and "false". "False" has the ordinal number 0, and "true" has the ordinal number 1.

All <Boolean type>s are the same type.

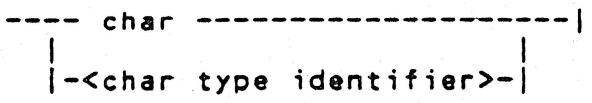
Example

```
type b = Boolean;
```

Type "b" is a <Boolean type identifier>.

CHAR TYPES

<char type>



"Char" is a predefined ordinal type that comprises the standard character set, which is EBCDIC unless changed to ASCII using the STRINGS compiler control option. The mapping of characters to ordinal numbers is defined in Appendix D, EBCDIC and ASCII Character Sets.

All <char type>s are the same type.

See also
STRINGS Option. 304
EBCDIC and ASCII Character Sets 328

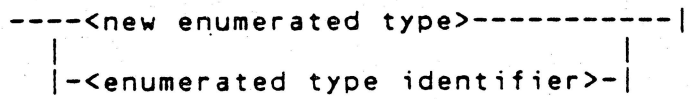
Examples

```
type ch = char;
     c  = ch;
```

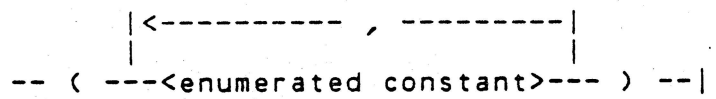
Types "ch" and "c" are both <char type identifier>s.

ENUMERATED TYPES

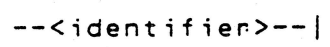
<enumerated type>



<new enumerated type>



<enumerated constant>



An <enumerated type> is a simple, ordinal type that comprises the values specified in the associated list of <enumerated constant>s. The order in which the <enumerated constant>s appear determines their ordinal numbers: the first <enumerated constant> is assigned the ordinal number 0, and each subsequent <enumerated constant> is assigned an ordinal number that is one higher than its predecessor.

The appearance of an <identifier> as an <enumerated constant> in an <enumerated type> definition defines that <identifier> for the block. Because the <identifier> cannot be redefined in the same block, the same <identifier> cannot be used in two <enumerated type> definitions in the same block.

Examples

```
type color = (red, yellow, blue, green, tartan);
     card_suit = (club, diamond, heart, spade);
     tool = (rake, hoe, spade);  { error }
```

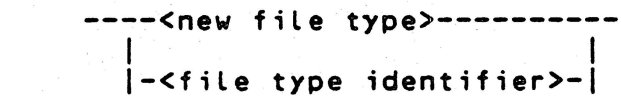
Type "color" is an <enumerated type identifier>. The <enumerated constant> "red" has the ordinal number 0, "yellow" the number 1, "blue" the number 2, "green" the number 3, and "tartan" the number 4.

Type "card_suit" is an <enumerated type identifier>. The <enumerated constant> "club" has the ordinal number of 0, "diamond" the number 1, "heart" the number 2, and "spade" the number 3.

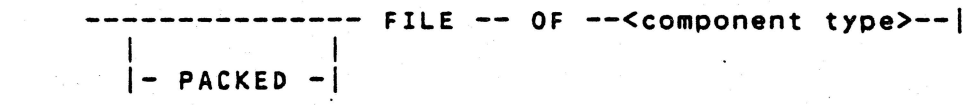
Type "tool" is in error because the identifier "spade" has already been declared (as a value of type "card_suit") in this block.

FILE TYPES

<file type>



<new file type>



<component type>

Any <type> that is not a <file type>, a <textfile type>, or a <structured type> containing a <file type> or a <textfile type> as a component.

A <file type> is a structured type of identical components. It differs from an array in that it is not indexed and has no specified upper bound. Instead, components are accessed through predefined procedures. For additional information on files, please refer to the I/O Concepts section.

The designation "PACKED" has no effect for file types.

See also
I/O Concepts. 134

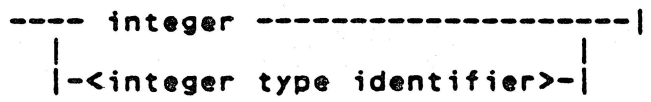
Example

```
type employee = record
                  name, firstname : packed array [1..20] of char;
                  department_code : 0..99;
                  employee_no : 0..9999;
                  end;
     employee_file = file of employee;
```

"Employee_file" is a <file type identifier>; each component of the file is an "employee" record containing the fields "name", "firstname", "department_code", and "employee_no".

INTEGER TYPES

<integer type>



"Integer" is a predefined ordinal type that comprises the integer values from "-maxint" to "maxint", inclusive. The ordinal number of a value of type integer is the value itself.

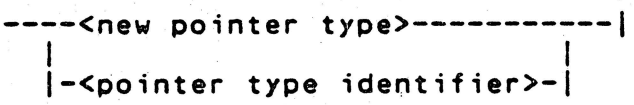
Example

```
type int = integer;
```

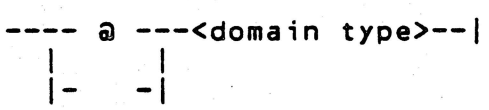
Type "int" is an <integer type identifier>.

POINTER TYPES

<pointer type>



<new pointer type>



<domain type>

Any <type identifier> except a <file type identifier>, a <textfile type identifier>, or a <type identifier> of a <structured type> containing a <file type> or <textfile type> as a component.

A <pointer type> is a special type that is used to reference dynamically allocated variables. A variable of a <pointer type> may reference a variable of its declared <domain type> or may be NIL (that is, not currently referencing a variable). Please refer to the Dynamic Allocation Procedures section for details on dynamic variables.

See also
Dynamic Allocation Procedures 204

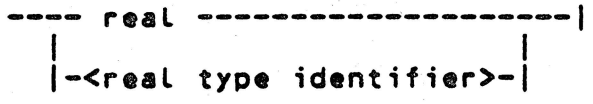
Example

```
type ptr_to_client = @client;
     client = record
              name : packed array [1..20] of char;
              son, daughter : ptr_to_client;
              end;
```

The type "ptr_to_client" is a pointer to a record of type "client".

REAL TYPES

<real type>



"Real" is a predefined simple type that comprises the range of floating-point approximations of values in the range -4.31359146673E68 to +4.31359146673E68. The smallest positive real value is 8.75811540203E-47.

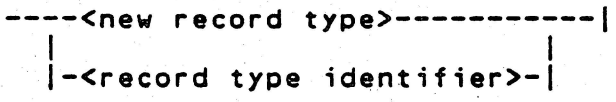
Example

```
type r = real;
```

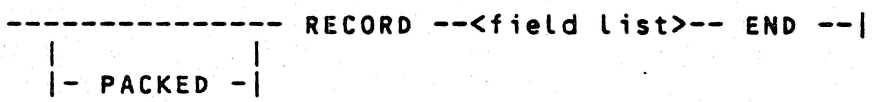
Type "r" is a <real type identifier>.

RECORD TYPES

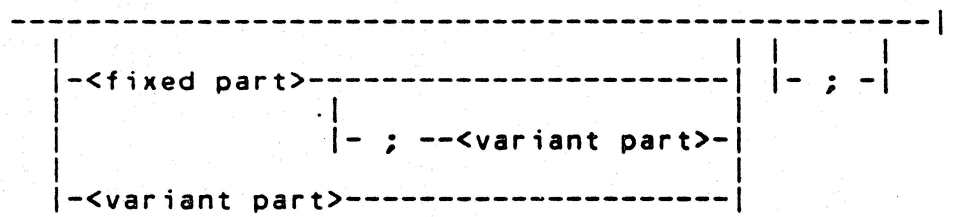
<record type>



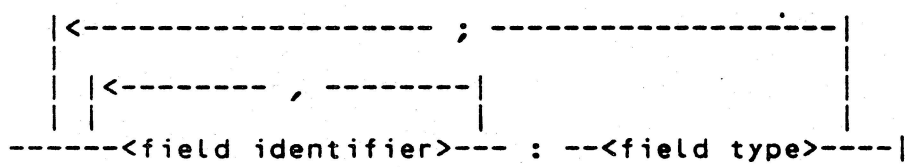
<new record type>



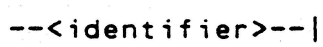
<field list>



<fixed part>



<field identifier>



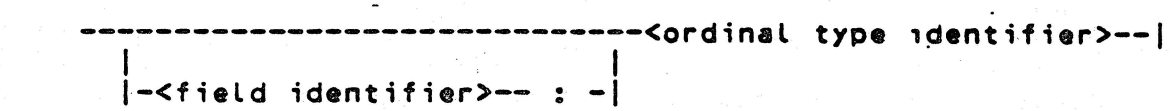
<field type>

Any <type> that is not a <file type>, a <textfile type>, or a <structured type> containing a <file type> or a <textfile type> as a component.

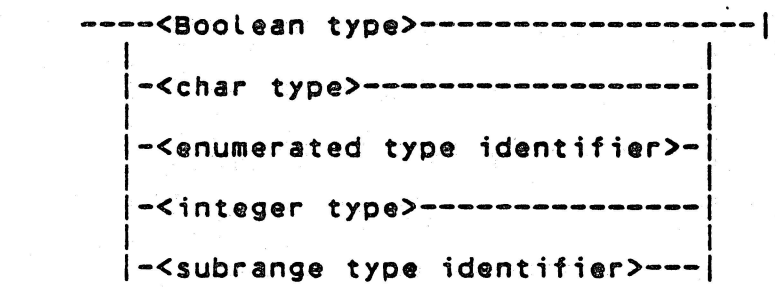
<variant part>



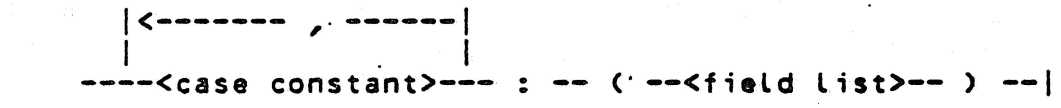
<variant selector>



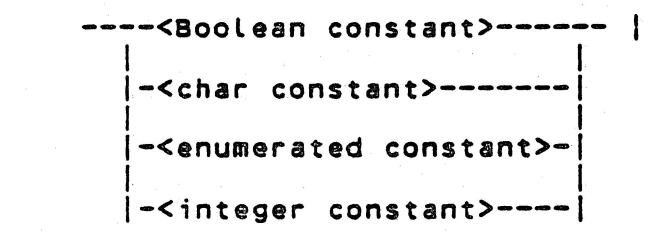
<ordinal type identifier>



<variant>



<case constant>



A <record type> is a structured type that can contain components of different types. These components, called "fields", are referenced by name, not by index (as with arrays) or by current position (as with files).

A record may include a <fixed part>, a <variant part>, both parts, or neither part. A record that includes neither a fixed nor a variant part contains no components and is said to be empty.

The <fixed part> of a record consists of a group of fields that apply to all variables of the <record type>. Each field has a <field identifier> by which it is referenced and an associated <field type>.

The <variant part> of a record is a collection of field definitions, called "variants". The <variant part> allows different variables of the same record type to have different (or partly different) formats, depending on the run-time value of the <variant selector>. Because the format is chosen at run time, there must be one (and only one) variant defined for every possible value of the type specified by the <ordinal type identifier> in the <variant selector>.

The interpretation of the variants at run time depends on whether or not the <variant selector> includes the optional <field identifier>. This <field identifier> is called the "tag field" and is allocated as a field within the record. If a tag field is defined and a variable of that record type is allocated, only fields in the <fixed part> and in the <variant> that includes the value of the tag field as a <case constant> are valid; any attempt to reference a field in another variant is an error. When the value of the tag field for a particular variable is changed, the old variant becomes inactive and all fields in that variant become inaccessible. The new variant becomes active and all fields within the newly active variant are undefined, regardless of any prior state.

If the <field identifier> is omitted (i.e. there is no tag field) and a variable of that record type is allocated, the active variant is selected by assigning a valid value to a field within that variant. At that point, all other variants theoretically become inactive, similar to the state described above for inactive tagged variants. However, in this implementation, the restrictions on accessing fields in inactive non-tagged variants are not enforced. All fields within the <fixed part> and all fields within all variants may be referenced, but only one storage area is allocated. Thus, the variants effectively "remap" the storage area.

A <record type> that includes the designation "PACKED" will be stored in as economical an amount of space as practical, possibly at the expense of speed in accessing the components (please refer to Appendix C, Data Representation).

See also
Data Representation 314

Example

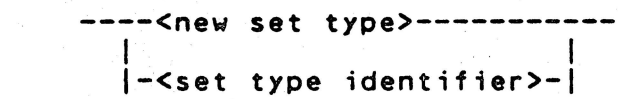
```
type str = packed array [1..20] of char;
     rec = record
           name, firstname : str;
           age : 0..99;
           case married : Boolean of
                true : (spousesname : str);
                false : ();
           end;
```

Type "rec" is a <record type identifier> that defines a <new record type>. The first component of "rec" is "name", which is of type "str". The next component is "firstname", also of type "str". The component "age" is a subrange from 0 to 99, inclusive.

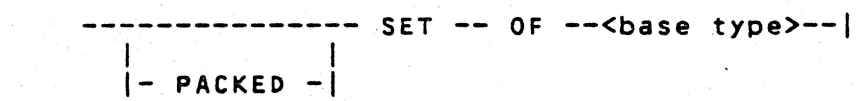
The word "case" introduces a set of two <variant>s, where "married" is a Boolean tag field that is the <variant selector>. If "married" is "true", the next component is "spousesname", of type "str". If "married" is "false", there are no more components.

SET TYPES

<set type>



<new set type>



<base type>



A <set type> is a structured type for which the range of values is all possible subsets of the specified <base type>. In mathematical terms, a <set type> defines the "powerset" of its <base type>. A variable of a <set type> can contain any subset of the set, including the null set and the entire set.

The range of ordinal numbers associated with the <base type> must be within the range 0..255.

The designation "PACKED" has no effect for set types.

Examples

```
type set1 = packed set of char;
     set2 = set of (club, diamond, heart, spade);
```

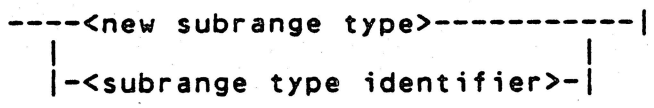
Type "set1" is a <set type identifier> defining a range of values consisting of all possible subsets of the set of type "char".

Type "set2" is a <set type identifier> defining a range of values consisting of all possible subsets of the set that includes the elements "club", "diamond", "heart", and "spade". The following are the possible values a variable declared of type "set2" could assume:

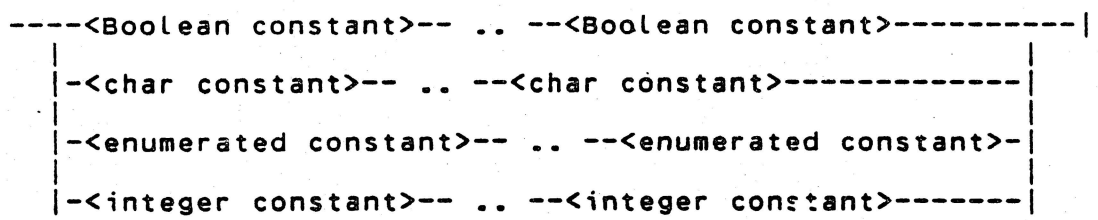
```
[]
[club]
[diamond]
[heart]
[spade]
[club,diamond]
[club,heart]
[club,spade]
[diamond,heart]
[diamond,spade]
[heart,spade]
[club,diamond,heart]
[club,diamond,spade]
[club,heart,spade]
[diamond,heart,spade]
[club,diamond,heart,spade]
```

SUBRANGE TYPES

<subrange type>



<new subrange type>



A <subrange type> is a simple, ordinal type that defines a range of values that is (usually) smaller than the type from which it is derived, called its "host type". The value range includes all values of the host type between the first constant specified and the second constant specified, inclusive. The specified constants must be of the same type, and the second constant must be greater than or equal to the first constant.

The ordinal numbers associated with the values of a <subrange type> are the same as the ordinal numbers associated with those values in the host type.

Examples

```
type letters = 'A'..'Z';
     color   = (red, yellow, blue, green, tartan);
     primary = red..blue;
     mixed   = green..tartan;
     index   = 1..10;
```

Type "letters" is a <subrange type identifier> that selects the subrange of "char" values consisting of the characters from 'A' to 'Z', inclusive.

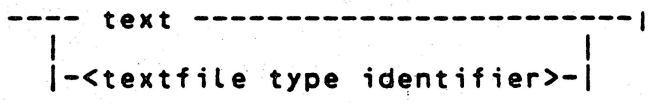
Type "color" is an <enumerated type identifier> whose values are "red", "yellow", "blue", "green", and "tartan". Type "primary" is a <subrange type identifier> that selects the subrange of "color" values from "red" through "blue" (that is, the values "red", "yellow", and "blue"). Type "mixed" is a <subrange type identifier> that selects the subrange of

"color" values from "green" through "tartan"; the ordinal numbers associated with the values of type "mixed" are 3 (green) and 4 (tartan).

Type "index" is a <subrange type identifier> that selects the integer values from 1 to 10, inclusive.

TEXTFILE TYPES

<textfile type>



A <textfile type> is a structured type for which the components are characters grouped into lines. Textfiles are similar to files of characters, but they have a different set of defined operations. As with files, characters are accessed through predefined procedures. Please refer to the I/O Concepts section for additional information on files and textfiles.

See also
I/O Concepts. 134

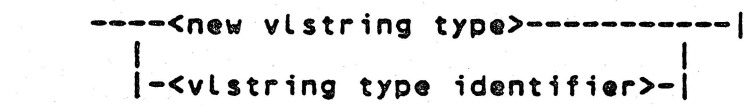
Example

```
type streamfile = text;
```

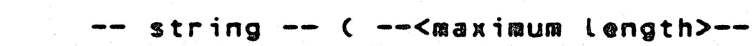
A variable declared to be of type "streamfile" will be a textfile.

VARIABLE-LENGTH STRING (VLSTRING) TYPES

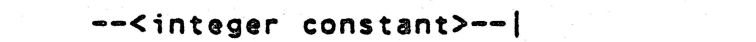
<vlstring type>



<new vlstring type>



<maximum length>



A <vlstring type> (variable length string type) is a structured type for which the range of values includes any string of characters whose length is between 0 and <maximum length>, inclusive. <Maximum length> must be between 1 and 65535, inclusive.

The length of a variable of <vlstring type> is the number of characters it contains; the length of an undefined vlstring is undefined. A variable of <vlstring type> cannot contain defined components after undefined components.

A <vlstring type> is handled as a packed data type even though PACKED is not explicitly stated.

<Vlstring type>s are a Burroughs extension to ANSI Pascal.

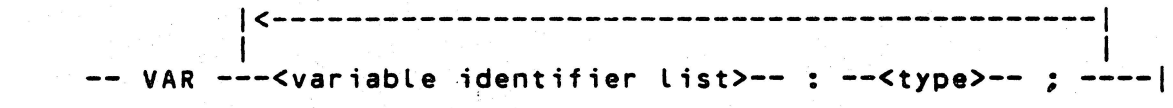
Example

```
type str1 = string(10);
     str2 = string(80);
     str3 = str2;
```

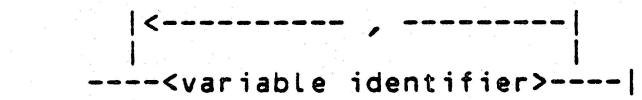
Type "str1" is a <vlstring type identifier> that has a <maximum length> of 10 characters. "Str2" and "st.3" are compatible <vlstring type identifier>s that have a <maximum length> of 80 characters.

3.4 VARIABLE DECLARATIONS

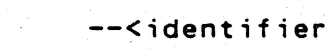
<variable declarations>



<variable identifier list>



<variable identifier>



The <variable declarations> define the variables that are to be used throughout the <block>. Each variable has an associated identifier, by which it is referenced, and an associated <type>, which defines the range of values and the operations applicable to the variable.

The <type> specified can be a predefined type identifier, a type identifier defined in the <type definitions>, or a new type specified in the <variable declarations>. Variables that appear in the same <variable identifier list> are defined to be of the same type. Please refer to the Type Definitions section for additional information on types.

When a block is entered at run time, all variables declared within that block are allocated with undefined values (please refer to Undefined Variables for a description of the implications of undefined variables).

See also

Type Definitions. 34
Undefined Variables 269

Examples

```
type color = (red, yellow, blue, green, tartan);

var  x, y, z, max : real;
     i, j : integer;
     p, q, r : Boolean;
     k : 0..9;
     operator : (plus, minus, times);
     a : array [0..63] of real;
     m, m1, m2 : array [1..10, 1..10] of real;
     f : file of char;
     c : color;
     hue1, hue2 : set of color;
     date : record
            month : 1..12;
            year  : integer;
            end;
     days : array [1..12] of 28..31;
     firstname, lastname : string(15);
```

The variables "x", "y", "z", and "max" are of type "real". The variables "i" and "j" are of type "integer". "P", "q", and "r" are variables of type "Boolean".

"K" is a variable of the <subrange type> 0..9, for which the host type is "integer".

"Operator" is a variable of an <enumerated type>; it can have the value "plus", "minus", or "times".

The variable "a" is a one-dimensional array of type real that may be indexed by an integer from 0 to 63, inclusive. The variables "m", "m1", and "m2" are two-dimensional arrays of type "real". Each dimension may be indexed by an integer between 1 and 10, inclusive.

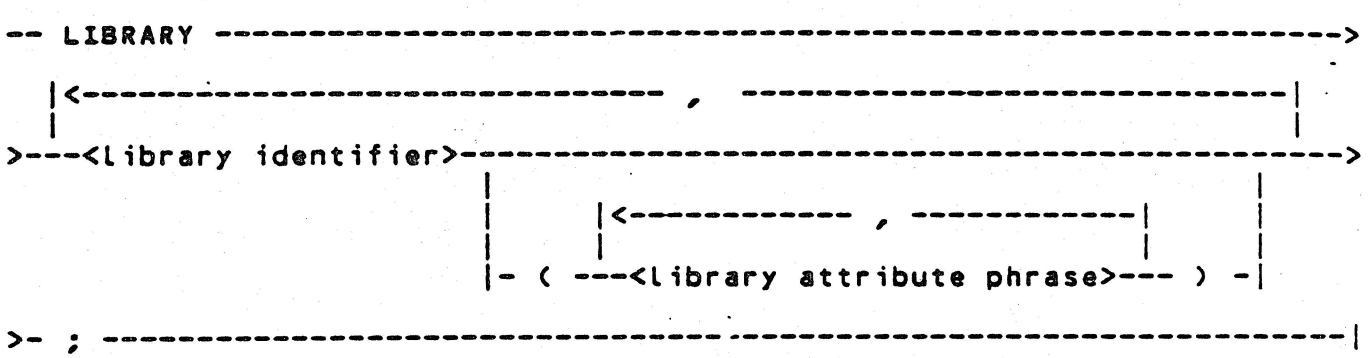
The variable "f" is a file whose component type is "char" (each component is a single character).

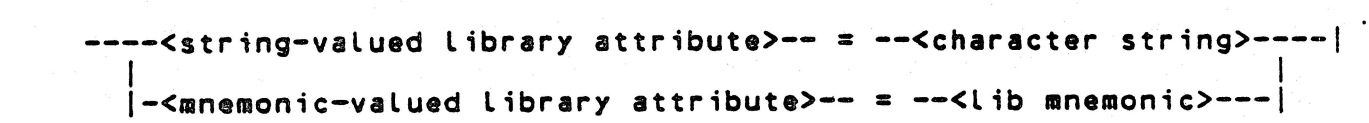
"C" is a variable of the <enumerated type identifier> "color" and may contain a value of "red", "yellow", "blue", "green", or "tartan". "Hue1" and "hue2" are both variables of type "set of color". They may contain any subset of the <enumerated type identifier> "color".

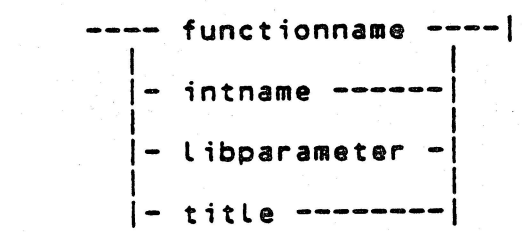
"Date" is a variable of a <new record type>. The field "month" may contain an integer value from 1 to 12, inclusive. The field "year" may contain any value of type "integer". The variable "days" is a one-dimensional array that may contain an integer value from 28 to 31, inclusive; it may be indexed by an integer value between 1 and 12, inclusive.

"Firstname" and "lastname" are variables of a <vlstring type> that has a <maximum length> of 15 characters.

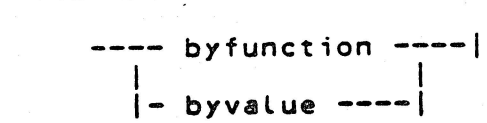
3.5 LIBRARY DECLARATIONS











The <library declarations> identify the libraries that provide entry points (procedures and functions) that this program intends to call.

Library attributes are system-defined variables that define characteristics of libraries. Library attributes can be set in the library declaration and can be set and tested dynamically through the use of the <setattribute procedure> and the <getattribute procedure>. For more details on library attributes, please refer to the System Software Operational Guide, Volume 2, chapter 20.

The decision as to which code file to initiate when the library linkage is being made is controlled by the value of several library attributes. If LIBACCESS has the value BYTITLE (either by assignment or by default), the library is linked to according to the TITLE attribute, which specifies the file title of the library's code file. If TITLE is not assigned, it defaults to the first 17 characters (translated to upper case) of the <library identifier>.

If LIBACCESS has the value BYFUNCTION, the library is linked to according to the value of the FUNCTIONNAME attribute, which specifies the name of an entry in the System Library. These entries are created and changed through the "SL" (System Library) ODT command (refer to the ODT Reference Manual for additional information). The library currently associated with the specified FUNCTIONNAME will be linked to.

See also
Getattribute Procedure. 245
Setattribute Procedure. 251

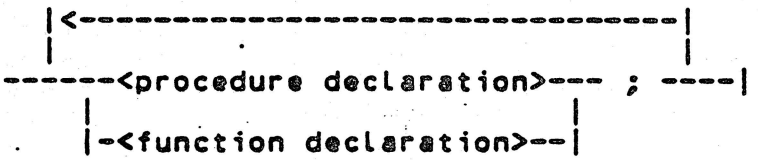
Examples

```
library mylib;

library general_procs (title = 'OBJECT/GENERAL/PROCEDURES');

library invlib (libaccess = byfunction,
                functionname = 'INVENTORYLIB');
```

When an entry point in "mylib" is called, a library code file with the title "MYLIB" will be linked to. The library "general_procs" will be linked to through the title "OBJECT/GENERAL/PROCEDURES". The program using "invlib" will be linked to the library associated with the functionname "INVENTORYLIB" in the System Library tables.

3.6 PROCEDURE AND FUNCTION DECLARATIONS

<procedure and function declarations>



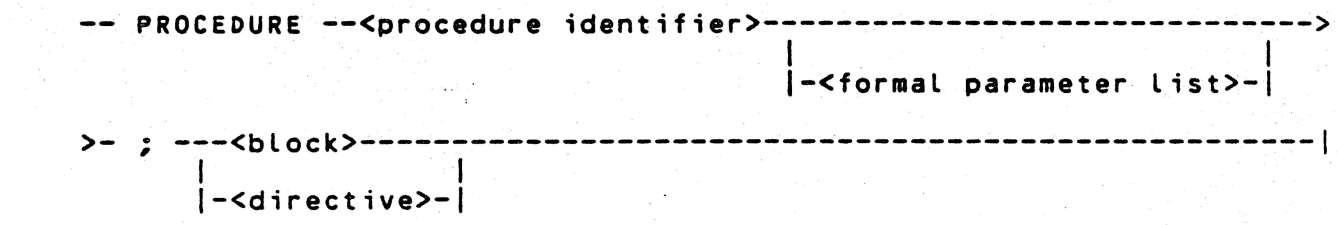
Procedures and functions are subunits of programs and include their own declarations and statements. The major difference between a procedure and a function is that a function returns a value associated with its function identifier; thus, a function is used to generate a value in an expression, whereas a procedure is used as a statement. The Procedure Declaration and Function Declaration sections describe the declarations used to define procedures and functions.

A procedure or function can have an associated list of parameters, which allows the run-time specification of the values and variables on which the procedure or function is to operate. The parameter list occurring in the declaration is called the "formal parameter list" because the parameter names do not refer to actual variables, but they stand in for variables throughout the procedure or function declaration. When the procedure or function is invoked, an "actual parameter list" is supplied, and the actual values and variables take the place of the formal parameters.

The "Formal Parameter Lists" section describes the syntax and semantics of formal parameter lists, which are identical for both procedures and functions. The "Actual Parameter Lists and Parameter Matching" section describes the syntax and semantics of actual parameter lists and how actual parameters are matched with formal parameters when a procedure or function is invoked.

PROCEDURE DECLARATION

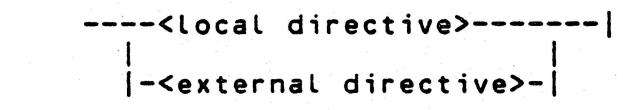
<procedure declaration>



<procedure identifier>



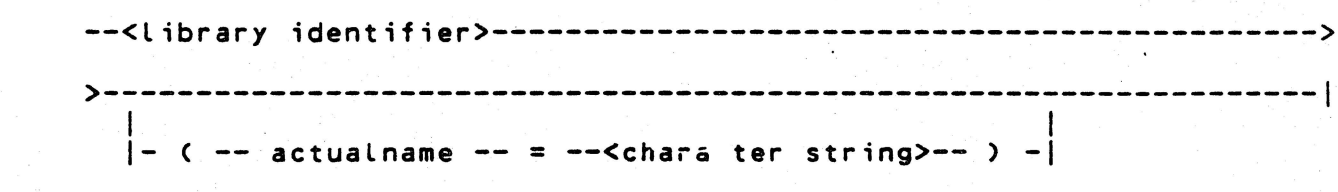
<directive>



<local directive>



<external directive>



The <procedure declaration> defines a procedure identifier, its parameters, and its action. The procedure can then be invoked by a <procedure invocation statement>.

The specification of the procedure's action can be direct, by providing the procedure <block>, or indirect, by providing a <directive> to the procedure algorithm. The <directive> indicates either that the procedure is fully defined later in the program (<local directive>) or that the procedure is to be imported from a library (<external directive>).

When a procedure is declared forward, an actual procedure declaration (that is, a <procedure declaration> that includes a <block>) must appear before the end of the list of <procedure and function declarations> that contains the forward declaration. When a forward declaration is used, the <formal parameter list> (if any) must appear in the forward declaration and cannot appear in the actual declaration.

In some situations, a forward declaration is required. For example, if two procedures each invoke the other, at least one of the procedures must be declared forward.

The use of the <external directive> in the declaration of a procedure identifies that procedure as an entry point of the library specified by the <library identifier>. No actual declaration for the procedure can appear in the <procedure and function declarations> of that <block>.

The <library identifier> must have been declared in the <library declarations> in the same <block> as the entry point. If an "actualname" is specified, its value must be the name specified for the corresponding entry point by the exporting library.

Libraries, which are referenced through the <external directive>, are a Burroughs extension to ANSI Pascal.

See also
Procedure Invocation Statements 97

Examples

```
program procedure_decs;
type arraytype = array [0..10] of integer;
var  x, y : arraytype;
     m, n : integer;
library mylib (title = 'OBJECT/ARRAYARITH');
procedure proc1;
   begin
   display ('in proc1');
   end;
procedure proc2 (i : integer; var j : integer);
   var k : integer;   { local to proc2 }
   begin
   display ('in proc2');
   j := j + i;  { Actual parameter for "j" is changed. }
   end;
```

```
procedure proc3 (procedure procparam(i : integer; var j : integer));
   var n : integer;   { local to proc3 }
   begin
   n := 30;
   display ('in proc3');
   procparam (45, n); { This procedure invocation invokes the
                        actual procedure passed to the "procparam"
                        formal parameter. }
   end;
procedure proc4 (var a : arraytype);
   forward;
procedure proc5;
   begin
   display ('in proc5');
   x[2] := 5;
   proc4 (x);
   end;
procedure proc4;  { The formal parameter list was specified in the
                    "forward" declaration for "proc4". }
   begin
   display ('in proc4');
   if a[2] = 10 then
      proc5;
   end;
procedure array_sum (a1, a2 : arraytype); mylib;
   { This procedure is imported from library "mylib". }

begin
m := 5;
n := 1000;
proc1;
proc2(m,n);
proc3(proc2);  { "Proc2" is passed as the actual parameter.  When
                 "proc3" invokes "procparam", the procedure actually
                 invoked is "proc2". }
proc5;
array_sum(x,y); { The program is linked to library "mylib", and
                  the entry point "array_sum" of that library is
                  invoked. }
end.
```

Procedure "proc1" has no parameters.

Procedure "proc2" has two parameters of type "integer". The first parameter is a <value parameter>, the second is a <variable parameter>.

Procedure "proc3" has a <procedural parameter>.

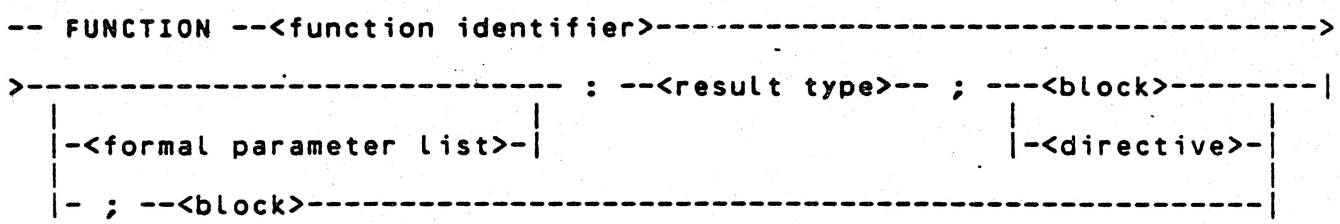
Procedure "proc4" has a <variable parameter> of type "arraytype". Because "proc4" contains a call on "proc5" (and "proc5" has a call on "proc4"), "proc4" was first declared as "forward". The <formal parameter list> for "proc4" is declared only with the forward declaration.

Procedure "proc5" has no parameters. "Proc5" contains a call on "proc4".

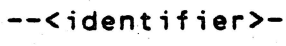
Procedure "array_sum" is provided through a library linkage.

FUNCTION DECLARATION

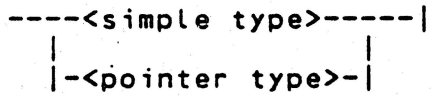
<function declaration>



<function identifier>



<result type>



The <function declaration> defines a function identifier, its type, its parameters, and its action. The function can then be invoked by a <function designator> in an expression.

The <result type> specifies the type associated with the <function identifier>, which is the type of the value returned to the expression invoking the function. The <result type> must be a <simple type> or a <pointer type> (refer to Type Categories). The function result is undefined until and unless the <function identifier> appears as the left-hand side of an <assignment statement> in the function <block>. If a value is never assigned to the <function identifier>, an error occurs.

The specification of the function's action can be direct, by providing the function <block>, or indirect, by providing a <directive> to the function algorithm. The <directive> can indicate either that the function is fully defined later in the program (<local directive>) or that the function is to be imported from a library (<external directive>).

When a function is declared forward, an actual function declaration (that is, a <function declaration> that includes a <block>) must appear before the end of the list of <procedure and function declarations> that contains the forward declaration. When a forward declaration is used, the <formal parameter list> (if any) and <result type> must appear in the forward declaration and cannot appear in the actual declaration.

In some situations, a forward declaration is required. For example, if two functions each invoke the other, at least one of the functions must be declared forward.

The use of the <external directive> in the declaration of a function identifies that function as an entry point of the library specified by the <library identifier>. No actual declaration of the function can appear in the <procedure and function declarations> of that <block>.

Libraries, which are referenced through the <external directive>, are a Burroughs extension to ANSI Pascal.

See also
Type Concepts 36
Function Designators. 109

Examples

```
program function_decs;
type sub1   = 1..10;
     letter = 'A'..'Z';
var  b: Boolean;
     c: letter;
     inx : integer;
     offset : sub1;
library mylib (title = 'OBJECT/ARITHLIB');

function func1 : Boolean;
   begin
   func1 := true;
   end;

function func2 (i : integer) : sub1;
   var k : integer;  { local to func2 }
   begin
   func2 := i - 5;
   end;

function func3 (function funcparam (i : integer) : sub1) : integer;
   var n: integer;  { local to func3 }
```

```
   begin
   func3 := funcparam (10);  { This function invocation invokes the
      actual function passed to the "funcparam" formal parameter. }
   end;

function func4 (var a : letter) : Boolean;
   forward;

function func5 : char;
   begin
   c := 'F';
   b := func4 (c);
   func5 := c;
   end;

function func4;  { The formal parameter list was specified in the
                   "forward" declaration for "func4". }
   begin
   if a < 'D' then
      a := func5;
   func4 := false;
   end;
function fact (f : integer) : integer; mylib;

begin
b := func1;
offset := func2(10);
inx := func3(func2);
c := func5;
inx := fact(5); { The program is linked to library "mylib" and the
                  entry point "fact" is invoked. }
end.
```

"Func1" is a function of type "Boolean" with no parameters.

Function "func2" is of type "sub1" and has one <value parameter> of type "integer".

"Func3" is a function of type "integer" and has one <functional parameter>.

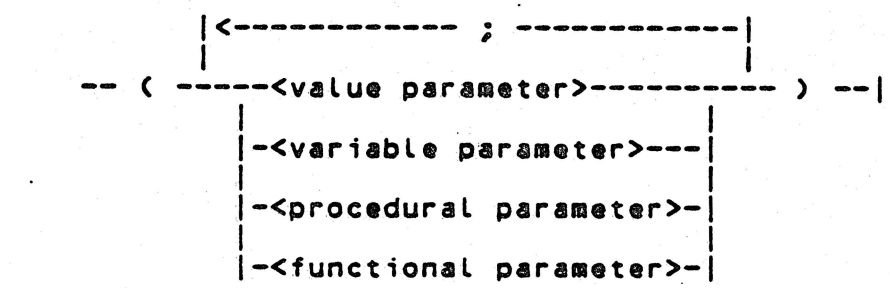
The function "func4" is of type "Boolean" and has one <variable parameter> of type "letter". Because "func4" contains a call on "func5" (and "func5" contains a call on "func4"), "func4" was first declared as "forward". The <formal parameter list> and <result type> for "func4" are declared only with the forward declaration.

Function "func5" is of type "char" and has no parameters.

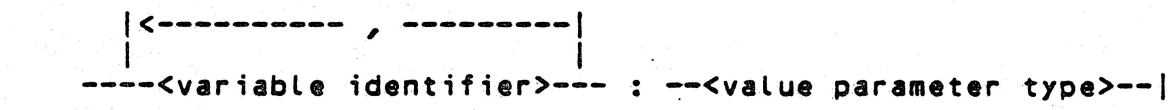
Function "fact" is provided through a library linkage.

FORMAL PARAMETER LISTS

<formal parameter list>



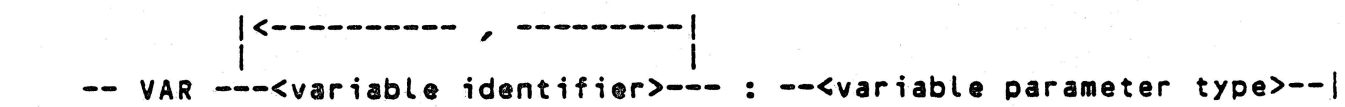
<value parameter>



<value parameter type>

Any <type identifier> that is not a <file type>, a <textfile type>, or a <structured type> containing a <file type> or a <textfile type> as a component.

<variable parameter>



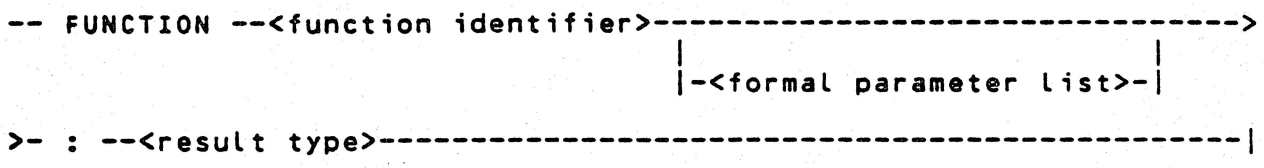
<variable parameter type>



<procedural parameter>



<functional parameter>



The <formal parameter list> appearing in a <procedure declaration> or <function declaration> defines the externally supplied values and variables on which the procedure or function is to operate. The actual values and variables are provided in the <actual parameter list> when the procedure or function is invoked.

Parameters are "declared" by their appearance in a parameter list. They have associated identifiers, which a e valid only within the procedure or function being declared, and associated types, which determine how the parameters can be used within the procedure or function and what type of actual parameters can be matched with the formal parameters. The four kinds of parameters (value, variable, procedural, and functional) also determine the usage of the parameter.

A <value parameter> provides a value to the procedure or function, but an assignment to the formal parameter will not change the value of the actual parameter.

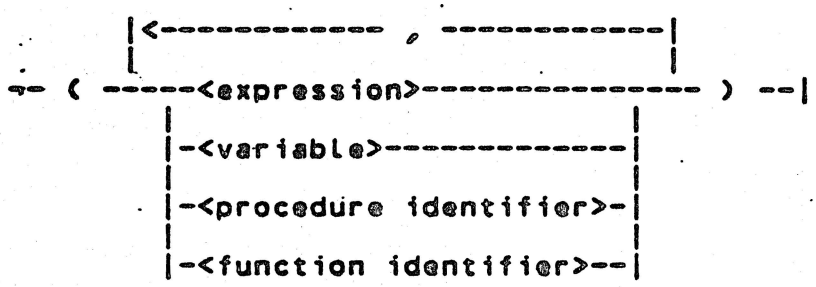
A <variable parameter> provides the procedure or function with a reference to a variable. An assignment to the formal parameter will change the value of the actual parameter.

A <procedural parameter> allows a procedure to be passed dynamically to another procedure or function. When the formal procedural parameter is called, the procedure passed as the actual procedural parameter is actually invoked. The actual procedural parameter is not invoked by its appearance in the actual parameter list; it is invoked only when its corresponding formal parameter is invoked within the procedure or function for which it is a parameter.

A <functional parameter> provides the same dynamic capability for functions as a <procedural parameter> provides for procedures.

ACTUAL PARAMETER LISTS AND PARAMETER MATCHING

<actual parameter list>



If a procedure or function is declared with a <formal parameter list>, an <actual parameter list> must be supplied whenever that procedure or function is invoked. Because the actual parameters will be substituted for the formal parameters in all contexts in which they appear in the <block> of the procedure or function, it is important that the actual and formal parameters have similar characteristics. This similarity is ensured by a mechanism called "parameter matching".

Formal and actual parameters are matched according to their positions in their respective parameter lists. The first formal parameter is matched with the first actual parameter, and so on. There must be the same number of parameters in the <actual parameter list> as were declared in the <formal parameter list>.

A formal <value parameter> must be matched by an <expression> or a <variable> in the <actual parameter list>. The <expression> or <variable> must be assignment compatible with the type of the formal parameter.

A formal <variable parameter> must be matched by a <variable> in the <actual parameter list>. The actual <variable> must be of the same type as the formal parameter. The actual parameter is accessed before the procedure or function is activated, and this access establishes a reference to the <variable> for the entire activation of the procedure or function. The existence of this reference implies that, even if the procedure or function changes a variable (such as an array index) that was used to specify the actual parameter, the actual parameter will not change. For example, if "a[i]" were passed as an actual variable parameter and "i" had the value 5 at the time the procedure was invoked, the actual parameter would always be "a[5]", even if "i" were changed to 7 within the procedure.

A formal <procedural parameter> must be matched by a <procedure identifier> that specifies a previously declared procedure. If either the formal or the actual procedural parameter has an associated <formal parameter list>, the <formal parameter list> associated with the <procedure identifier> in its declaration must be "congruous" with the <formal parameter list> associated with the <procedural parameter>. (Parameter list congruity is described later in this section.)

A formal <functional parameter> must be matched by a <function identifier> that specifies a previously declared function. If either the formal or the actual functional parameter has an associated <formal parameter list>, the <formal parameter list>s must be congruous. In addition, the <result type>s of the formal and actual functional parameters must be the same type.

Predefined procedures and predefined functions cannot be passed as actual parameters. A component of a variable of a PACKED structured type cannot be passed as an actual variable parameter, nor can the tag field of the <variant part> of a record variable.

Note: When passing a <simple type> to a formal by-name parameter (available in ALGOL), it is necessary to ensure that the variable has been initialized. Otherwise, a fault will occur in the ALGOL library when it references the formal parameter.

Parameter List Congruity

Two <formal parameter list>s are congruous if all of the following conditions are true:

a) The <formal parameter list>s contain the same number of parameters.

b) Corresponding parameters are of the same kind (value, variable, procedural, or functional).

c) Corresponding parameters are of the same type.

d) Functional and procedural parameters, if any, have congruous <formal parameter list>s.

e) The <result type>s of functional parameters are the same type.

See also

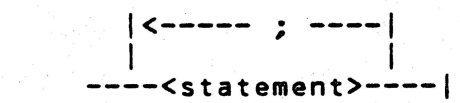
Same Types. 39
Assignment Compatibility. 41
Record Types. 55
Predefined Procedures and Functions 131

4 STATEMENTS

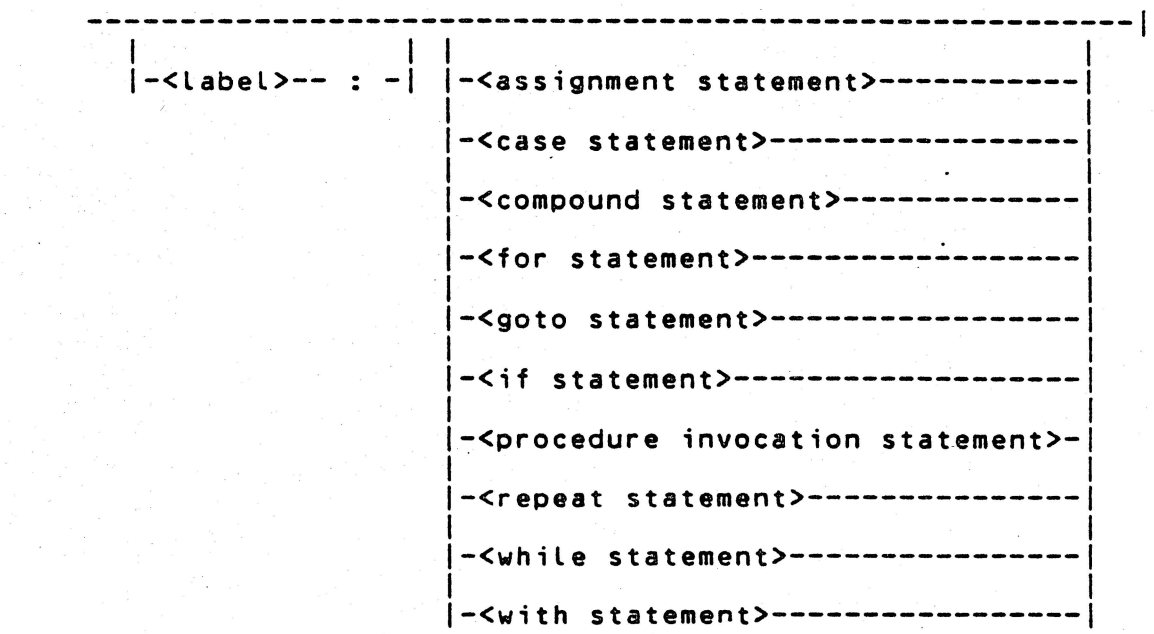
<statement part>



<statement list>



<statement>



Every <block> contains a <statement part>, which is simply a list of statements bracketed by the keywords BEGIN and END. Statements are the executable, or active, components of programs. "Simple" statements perform a single operation once. "Structured" statements contain statements as subcomponents. Depending on the form of the structured statement, the subcomponent statements may be executed sequentially, repetitively, or conditionally.

The <assignment statement>, the <goto statement>, and the <procedure invocation statement> are simple statements. The <compound statement> and the <with statement> are sequential statements. The <for statement>, the <repeat statement>, and the <while statement> are

repetitive statements. The <if statement> and the <case statement> are conditional statements.

The null path through the <statement> syntax diagram represents the "empty statement". The empty statement can be used in situations where a null operation is required. For example, it might be desirable to associate an empty statement with a particular <case constant> in a <case statement>.

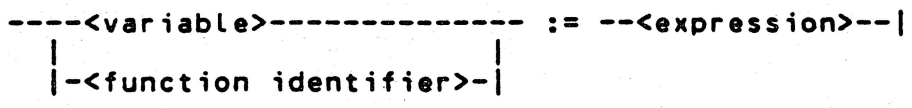
A statement may have an associated <label> that identifies its location for later reference in a <goto statement>. Restrictions on the declaration and placement of labels are described in the Label Declarations section. Restrictions on references to labels in <goto statement>s are described in the Goto Statements section in this chapter.

See also

Label Declarations. 30
Goto Statements 92

ASSIGNMENT STATEMENTS

<assignment statement>



The <assignment statement> assigns the value of the <expression> to the specified <variable> or <function identifier>. The value of the <expression> must be assignment compatible with the type of the <variable> or <function identifier> that is being assigned.

See also
Assignment Compatibility. 41

Examples

```
x := y + z;
```

The variable "x" is assigned the sum of "y" and "z".

```
p := (1 <= i) and (i <= 100);
```

The variable "p" is assigned the Boolean value "true" if "i" is between the values of 1 and 100, inclusive; otherwise, "p" is assigned the Boolean value "false".

```
hue1 := [blue, succ(c)];
```

The set variable "hue1" is assigned the set consisting of the value "blue" and the successor to the value of the variable "c".

```
p1@.mother := true;
```

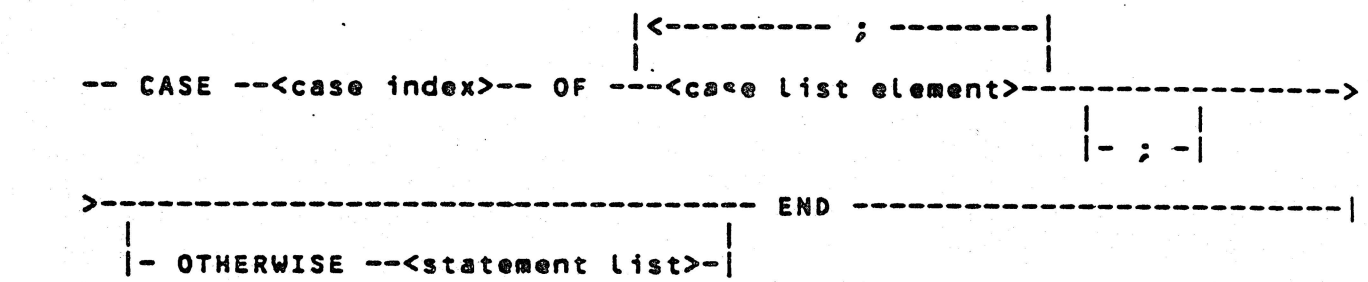
The Boolean "mother", which is a field identifier in a dynamically allocated variable pointed to by "p1", is assigned the value "true".

```
var s : packed array [1..8] of char;
begin
s := 'abc';
end;
```

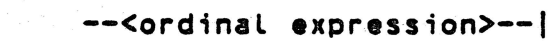
This assignment assigns the value 'abc ' to the string variable "s".

CASE STATEMENTS

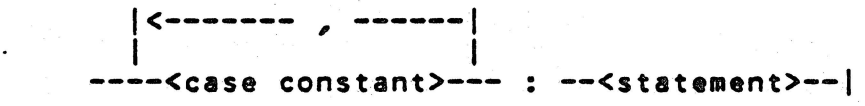
<case statement>



<case index>



<case list element>



The <case statement> allows the selection of one of a group of statements, depending on the value of the specified <case index>. The <case index> is evaluated, and the <statement> associated with the <case constant> of that value is executed.

If no <case constant> has the value of the <case index>, the <statement list> following the reserved word OTHERWISE is executed; if OTHERWISE does not appear, a run-time error occurs.

The values of the <case constant>s must be unique and must be of the same ordinal type as the <case index>.

The OTHERWISE construct is a Burroughs extension to ANSI Pascal.

See also
Ordinal Expressions 105

Examples

```
case operator of
   plus:   x := x + y;
   minus:  x := x - y;
   times:  x := x * y;
end;
```

The value of the enumerated variable "operator" determines the case constant whose statement will be executed.

```
case date.month of
   4,6,9,11:  days [date.month] := 30;
   2:         days [date.month] := 28;
   otherwise  days [date.month] := 31;
end;
```

If "date.month" is a value other than 2, 4, 6, 9, or 11, the statement associated with "otherwise" will be executed.

COMPOUND STATEMENTS

<compound statement>

-- BEGIN --<statement list>-- END --|

The <compound statement> allows a <statement list> to be treated as a single <statement>. A <compound statement> is frequently used as a <statement> within a structured statement (such as an <if statement> or <while statement>).

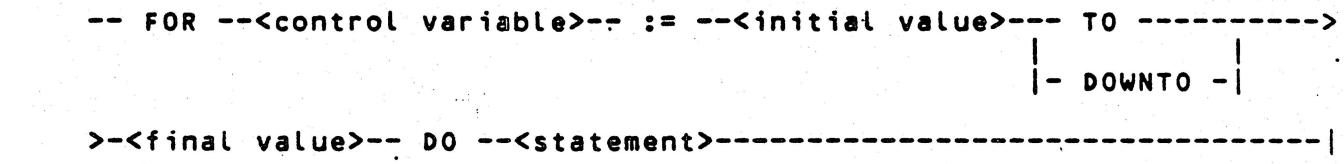
Example

```
if j > k then
   begin
   z := x;
   x := y;
   y := z;
   end;
```

If the value of "j" is greater than the value of "k", "z" will be assigned the value of "x", "x" will be assigned the value of "y", and "y" will be assigned the value of "z".

FOR STATEMENTS

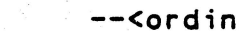
<for statement>



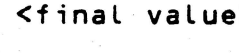
<control variable>

A <Boolean variable>, <char variable>, <enumerated variable>, or <integer variable> that is also an <entire variable>.

<initial value>



<final value>



The <for statement> causes the <statement> to be executed repeatedly, each repetition being performed with the <control variable> assigned to a different value within the specified range of values. The <statement> within the <for statement> is referred to as the "controlled statement".

The range of values is defined by <initial value> and <final value>. If TO is specified, the <control variable> is incremented from <initial value> to <final value>, inclusive. If DOWNTO is specified, the <control variable> is decremented from <initial value> to <final value>, inclusive. The <initial value> and the <final value> are evaluated only once; thus, if one or both are variables, subsequent changes to their values have no effect on the execution of the <for statement>.

Once the <control variable> has been assigned the <final value> and the controlled statement has been executed for the final time, the value of the <control variable> becomes undefined and program control is passed to the statement following the <for statement>. If a <goto statement> within the controlled statement transfers control to a statement outside the controlled statement, the value of the <control variable> remains defined.

The <control variable> must be a locally declared variable of an ordinal type. The <initial value> and <final value> must be assignment compatible with the <control variable>. The value of the <control variable> may be accessed at any time during the execution of the controlled statement, but its value cannot be changed or "threatened". A "threatening" statement is one of the following types of statements occurring in the controlled statement or in any procedure or function declared in the most local block containing the <for statement>:

a) An assignment statement in which the <control variable> appears on the left-hand side.

b) A statement that invokes a procedure or function in which the <control variable> appears as an actual variable parameter in the parameter list.

c) A statement in which either the "read" or the "readln" procedure is invoked with the <control variable> appearing in the parameter list.

d) Another <for statement> in which the <control variable> is also used as the <control variable> for that <for statement>.

See also

Assignment Statements 85
Goto Statements 92
Ordinal Expressions 105
Predefined Procedures and Functions 131

Examples

```
max := a[1];
for i := 2 to 63 do
   if a[i] > max then
      max := a[i];
```

For each value of "i" between 2 and 63, inclusive, "a[i]" will be compared to "max". If the value of "a[i]" is greater than "max", "max" will be assigned the value of "a[i]".

```
for i := 1 to 10 do
   for j := 1 to i - 1 do
      m[i][j] := 0.0;
```

For each value of "i" between 1 and 10, inclusive, "j" is assigned a value of 1 to "i - 1", inclusive. When "i" is 1, "j" is assigned values from 1 to 0. Because there are no values between 0 and 1, the controlled statement of the innermost "for" statement is not executed when "i" is 1. When "i" is 2, "j" is assigned values from 1 to 1, inclusive, so "m[2][1]" is assigned the value "0.0". This process continues for all values of "i" up to, and including, 10.

```
for c := blue downto red do
   q(c);
```

For each value of "c" between "blue" and "red", inclusive, the procedure "q" is called with "c" as a parameter. ("c" is assigned "blue", "pred(c)", ..., until "pred(c)" is the value "red".)

GOTO STATEMENTS

<goto statement>

-- GOTO --<label>--|

The <goto statement> transfers prog.am control to the <statement> associated with the specified <label>.

There are several restrictions on the use of the <goto statement> that depend on the location of the <label> it specifies. In general, the restrictions prohibit branching into a structured statement from outside that statement. Specifically, it is valid for a <goto statement> to reference a <label> only if at least one of the following conditions is true:

a) The <statement> associated with the <label> is in the same <statement list> as the <goto statement> or it is in the same <statement list> as any structured statement containing the <goto statement>.

b) The <statement> associated with the <label> is a <statement> within the <statement part> of any <block> containing the <goto statement>. That is, the <statement> associated with the <label> is a statement at the outermost level of any <block> containing the <goto statement> and is not contained within a structured statement.

Examples

```
program valid_goto_examples;

label 10, 20, 9999;
var counter : integer;

procedure p1;
   label 100;
   var local_loop : integer;
   begin
   local_loop:=1;
100:
   if local_loop > 2 then
      goto 9999;
   local_loop := local_loop + 1;
   goto 100;
   end;

begin
   counter:=0;
10:
   if counter < 10 then
      begin
      counter := counter + 1;
      goto 10;
      end;
   if counter < 20 then
      begin
20:
      counter := counter + 1;
      if counter < 25 then
         begin
         display('looping');
         goto 20;
         end;
      p1;
      end;

9999:
   display('done');
end.
```

In this example, the branches to labels 10, 20, and 100 are valid by rule "a". The branch to label 9999 is valid by rule "b".

```
program invalid_goto_examples;

label 2000, 9000;
var inx : integer;

procedure p1;
   label 100;
   begin
100:
   goto 9000;     {1}
   end;

begin
inx := 3;
if inx = 3 then
   begin
   inx := 4;
   goto 2000;     {2}
   end
else
   begin
2000:
   display ('illegal branch');
   end;

if inx = 4 then
   begin
9000:
   display ('illegal branch');
   end
else
   begin
   goto 100;      {3}
   end;

end.
```

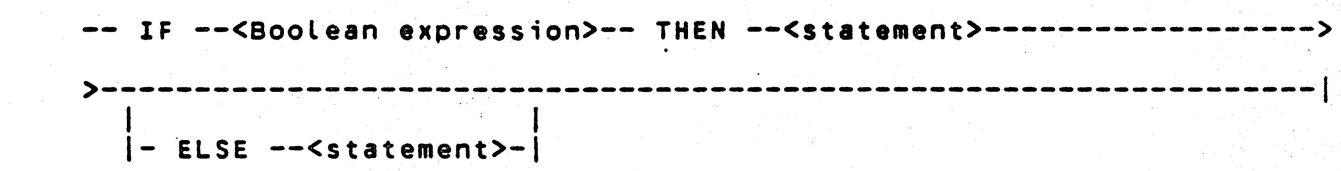
The branch at {1} is invalid because the statement associated with label 9000 is in a containing procedure but is not at the outermost level of the block.

The branch at {2} is invalid because the statement associated with label 2000 is neither in the <statement list> that contains the <goto statement> nor in any structured statement that contains the <goto statement>.

The branch at {3} is invalid because label 100 is not in the scope of the <goto statement>.

IF STATEMENTS

<if statement>



The <if statement> allows the selection of one of two <statement>s, depending upon the value of the <Boolean expression>. If the value of the <Boolean expression> is true, the <statement> following the reserved word THEN is executed. If the value of the <Boolean expression> is false, the <statement> following the reserved word ELSE is executed; if ELSE does not appear, program execution continues with the statement immediately following the <if statement>.

In nested <if statement>s, each ELSE is paired with the nearest preceding unpaired THEN.

Examples

```
if x < 1.5 then
   z := x + y
else
   z := 1.5;
```

If "x" is less than 1.5, "z" will be assigned the sum of "x" and "y". If "x" is greater than or equal to 1.5, "z" is assigned the value 1.5.

```
if p1 <> nil then
   p1 := p1@.father;
```

If the pointer "p1" is referencing a variable, "p1" is updated to the value of the pointer contained in the field "father" in the dynamically allocated record pointed to by "p1".

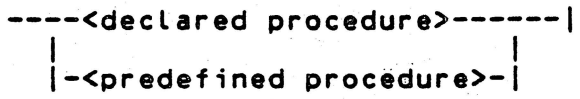
```
if j = 0 then
   if i = 0 then
      writeln('indefinite')
   else
      writeln('infinite')
else
   writeln(i / j);
```

The following table shows what would be written for different values of "i" and "j":

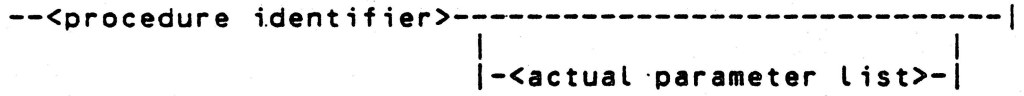
j = 0 and i = 0	indefinite
j = 0 and i <> 0	infinite
j <> 0 and i = 0	i / j
j <> 0 and i <> 0	i / j

PROCEDURE INVOCATION STATEMENTS

<procedure invocation statement>



<declared procedure>



The <procedure invocation statement> activates the specified <declared procedure> or <predefined procedure>. When the procedure activated by the <procedure invocation statement> terminates, the program continues at the point immediately following the <procedure invocation statement>.

The <procedure identifier>s and parameter lists for <declared procedure>s are specified by the programmer in <procedure declaration>s. Procedure identifiers and parameter lists for <predefined procedure>s are described in the Predefined Procedures and Functions chapter.

If the <procedure identifier> was declared with a <formal parameter list>, any <procedure invocation statement> invoking that procedure must include an <actual parameter list>. Please refer to the Actual Parameter Lists and Parameter Matching section for additional information.

See also

Procedure Declaration 71
Actual Parameter Lists and Parameter Matching 80
Predefined Procedures and Functions 131

Examples

```
printheading;
```

The declared procedure "printheading", which has no parameters, is invoked.

```
writeln(f, i, j);
```

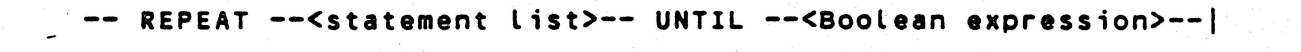
The predefined procedure "writeln" is called to write the values of "i" and "j" to the textfile "f".

```
bisect(fct, -1.0, +1.0, x);
```

The declared procedure "bisect" is called with the actual parameters "fct", "-1.0", "+1.0", and "x".

REPEAT STATEMENTS

<repeat statement>



The <repeat statement> causes the <statement list> to be repeatedly executed until the value of the specified <Boolean expression> is true. The <statement list> will always be executed at least once because the <Boolean expression> is evaluated after each execution of the <statement list>.

See also
Boolean Expressions 111

Example

```
repeat
   k := i mod j;
   i := j;
   j := k;
until j = 0;
```

The variable "k" is assigned the value of "i mod j". The variable "i" is assigned the value of "j". The variable "j" is assigned the value of "k". If "j" is not equal to 0, the three assignment statements are executed again. When "j" is equal to 0, the statement following the repeat statement is executed.

WHILE STATEMENTS

<while statement>



The <while statement> causes the <statement> to be repeatedly executed until the value of the specified <Boolean expression> is false. The <Boolean expression> is evaluated before each execution of the <statement>, so the <statement> will not be executed if the <Boolean expression> is initially false.

See also
Boolean Expressions 111

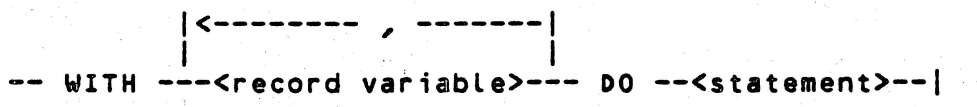
Example

```
while i > 0 do
   begin
   if odd(i) then
      z := z * x;
   i := i div 2;
   x := sqr(x);
   end;
```

The compound statement in the WHILE statement is executed if "i" is greater than 0. After each execution of the compound statement, "i" is compared to 0. If "i" is greater than 0, the compound statement is executed again.

WITH STATEMENTS

<with statement>



The <with statement> establishes a scope within which all <field identifier>s in the <statement> are assumed to be prefixed by the specified <record variable>. Thus, when a <field identifier> is used, the field referenced is actually "<record variable>.<field identifier>". The <with statement> context permits a shorthand notation that is useful when many references are being made to fields within a particular record.

When multiple <record variable>s are specified, the effect is as if the <record variable>s were specified in nested <with statement>s. The left-most <record variable> is assigned the most global scope and the right-most the most local scope. Thus, when two or more records have identically named fields and that field name appears as a <field identifier> in the <statement>, the field is assumed to be the one in the <record variable> associated with the most local <with statement> scope.

Similarly, when a <field identifier> conflicts with an <identifier> whose scope is global to the <with statement>, the <with statement> scope overrides and the field of the record is referenced.

See also

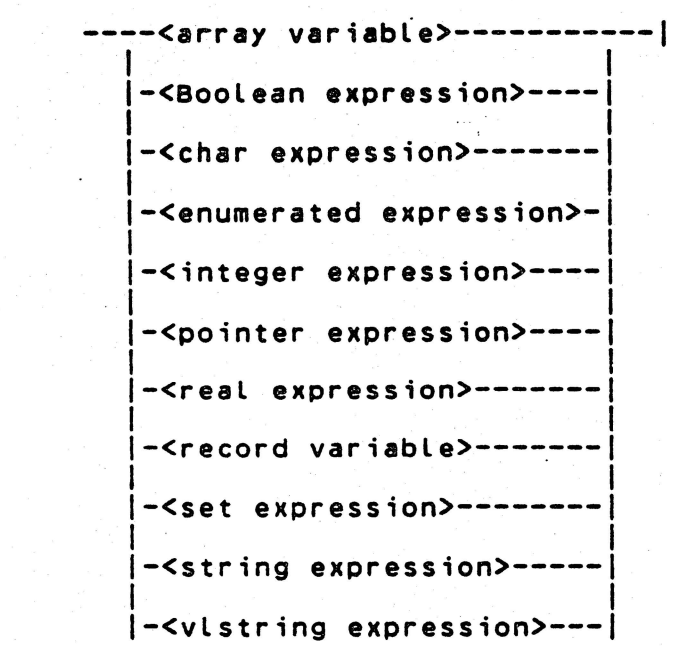
Variables 257

Examples

```
var date : record
           month : 1..12;
           year : 1950..2050;
           end;
begin
with date do
   if month = 12 then
      begin
      month := 1;
      year := year + 1;
      end
   else
      month := month + 1;
end;
```

If "date.month" equals the value 12, "date.month" is assigned the value 1 and "date.year" is incremented by 1. If "date.month" is not equal to 12, "date.month" is incremented by 1.

5 EXPRESSIONS



An <expression> generates a value of a particular type by performing specified operations on specified operands. The operands and operations vary according to type. For example, a <Boolean expression> generates a Boolean value from the application of <Boolean operator>s to <Boolean primary>s (operands).

For most <array type>s and all <record type>s, there are no operations or constants defined; an <expression> of such a type is simply a variable of that type. Arrays of <string type> can be assigned <string expression>s, which are defined in this section. Files and textfiles do not directly generate values, and there are no expressions defined for these types.

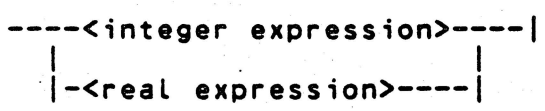
PASCAL / Expressions

The next section describes expression concepts. These include arithmetic expressions, ordinal expressions, precedence of operators (that is, the order in which operators are applied when there are multiple operators in an expression), and the <function designator>, which appears in the syntax for several types of expressions. The final section describes the various expression types, in alphabetical order.

5.1 EXPRESSION CONCEPTS

ARITHMETIC EXPRESSIONS

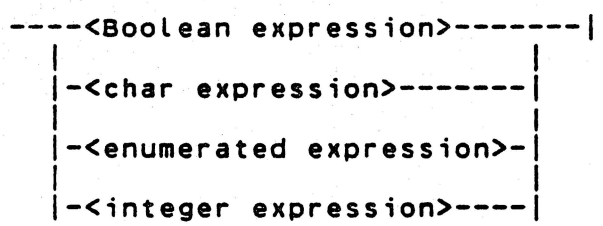
<arithmetic expression>



In some contexts, it is useful to consider <integer expression>s and <real expression>s as <arithmetic expression>s. For example, many arithmetic functions accept <arithmetic expression>s as parameters.

ORDINAL EXPRESSIONS

<ordinal expression>



Boolean, char, enumerated, and integer expressions are grouped as <ordinal expression>s, which are expressions that generate ordinal values. <Ordinal expression>s are frequently used as <case constant>s, array indices, and set components.

PRECEDENCE OF OPERATORS

An "operator" generates a value by performing a defined operation on either one or two data items. The data items on which operators operate are called "operands".

A "unary operator" applies to only one operand. For example, the Boolean NOT operator produces a value that is the logical complement of the Boolean operand to which it is applied.

A "binary operator" applies to two operands, generating a single value by combining or comparing the values of the two items in some way. For example, the arithmetic subtract operator (-) produces a value by subtracting the value of the second operand from the value of the first operand.

An "expression" is a combination of operands and operators that generates a value by applying the operators to the operands according to defined rules. The simplest expression is just an operand, with no operators or other operands specified. A more complicated expression may include many operands and operators.

Theoretically, when there are multiple operators in an expression, there could be multiple interpretations of the expression. For example, "A + B * C" could be interpreted in two ways:

1) first add A and B, then multiply the sum by C, or

2) first multiply B and C, then add the product to A.

If A is 3, B is 5, and C is 7, then the value of the expression is 56 if computed by method 1 and 38 if computed by method 2.

PASCAL / Expressions

Rules that define the "precedence of operators" describe the order in which operations are performed within an expression. Higher precedence operators are applied before lower precedence operators. In Pascal, the precedence of operators is defined by the following table:

```
[highest]     a)  NOT
              b)  *, /, DIV, MOD, AND, CAND
              c)  +, -, OR, COR
[lowest]      d)  =, <>, <=, >=, <, <, IN
```

The highest precedence operator is the Boolean NOT operator.

The "multiplication operators" have the second highest precedence. These operators are integer and real multiply (*), set intersection (*), real division (/), integer division (DIV), integer remainder division (MOD), Boolean AND, and conditional Boolean AND (CAND).

The "addition operators", the next group in precedence, are integer or real unary plus (+), integer or real addition (+), set union (+), integer or real unary minus (-), integer or real subtraction (-), set difference (-), Boolean OR, and conditional Boolean OR (COR).

The lowest precedence operators are the "relational operators", such as greater than (>), equals (=), and so on. These operators apply to several data types and are described under Relational Expressions (please refer to that section for additional details). Note that, in Pascal, the relational operators have the LOWEST precedence. Many other languages, such as FORTRAN and ALGOL, define a higher precedence for these operators. For example, if A, B, C, and D are integer operands, the expression shown below is a valid Boolean expression in FORTRAN and ALGOL (ignoring the minor differences in syntax), but it is not a valid expression in Pascal:

```
 A  =  B  AND  C  =  D
(A  =  B) AND (C  =  D)   {FORTRAN/ALGOL interpretation}
 A  = (B  AND  C) =  D    {Pascal interpretation--INVALID}
```

When an expression contains two or more operators of equal precedence, the operators are applied from left to right. For example, in the expression "X * Y / Z", first X and Y are multiplied, then the product is divided by Z.

The defined precedence of operators can be overridden by enclosing subcomponents of the expression in parentheses. For example, in the expression "A + B * C" mentioned earlier, the precedence rules specify that the multiply operator (*) is to be applied before the addition operator (+). Thus, the result of evaluating this expression is 38 if A is 3, B is 5, and C is 7. The other interpretation can be imposed by enclosing the first part of the expression in parentheses:

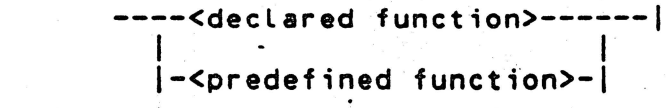
```
(A + B) * C     {Add A and B, then multiply by C -- 56}
 A + (B * C)    {Identical to default interpretation -- 38}
```

See also

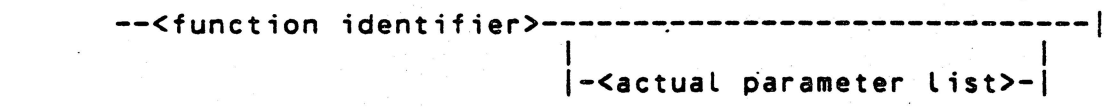
Relational Expressions. 113

FUNCTION DESIGNATORS

<function designator>



<declared function>



The appearance of a <function designator> in an expression activates the specified <declared function> or <redefined function>. When the function activated by the <function designator> terminates, a value is returned and evaluation of the expression continues.

The <function identifier>s and <formal parameter list>s for <declared function>s are specified by the programmer in <function declaration>s. Function identifiers and parameter lists for <predefined function>s are described in the Predefined Procedures and Functions chapter.

If the <function identifier> was declared with a <formal parameter list>, any <function designator> invoking that function must include an <actual parameter list>. Please refer to the Actual Parameter Lists and Parameter Matching section for additional information.

See also

Function Declaration. 75
Actual Parameter Lists and Parameter Matching 80
Predefined Procedures and Functions 131

Examples

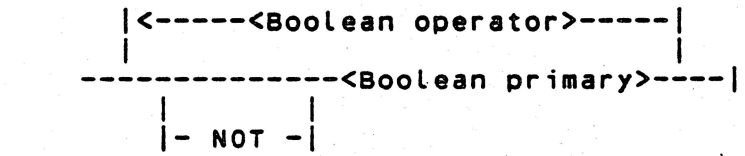
```
program function_example;
var i : integer;
    b : Boolean;
function f1 : integer;
   begin
   f1 := 10;
   end;
function f2 (j : integer) : Boolean;
   begin
   f2 := j > 20;
   end;
begin
i := f1;
b := f2 (i);
end.
```

The variable "i" is assigned the value of the function designator "f1". The variable "b" is assigned the value of the function designator "f2", where "i" is passed as the actual parameter.

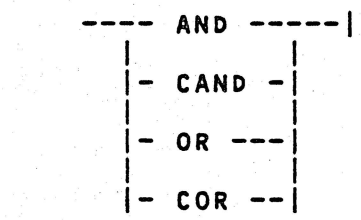
5.2 EXPRESSIONS BY TYPE

BOOLEAN EXPRESSIONS

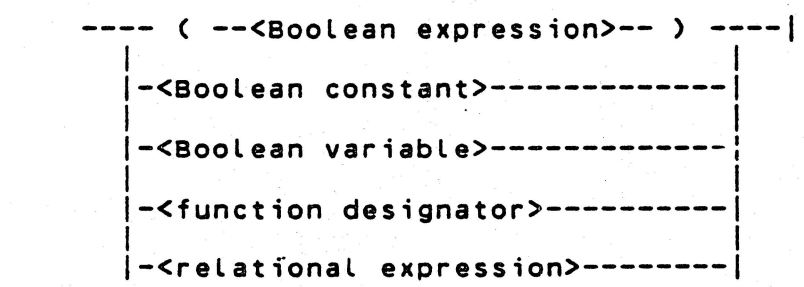
<Boolean expression>



<Boolean operator>



<Boolean primary>



A <Boolean expression> generates a value of the <Boolean type>.

The <Boolean operator>s AND and OR perform the logical AND and logical OR operations, respectively. CAND and COR are conditional operators that perform the same operations as AND and OR, with the following exception: the left-hand <Boolean primary> is always evaluated first and, if the value of the <Boolean expression> can be determined from the value of the left-hand <Boolean primary> alone, the right-hand <Boolean primary> is not evaluated.

<Boolean constant> is defined in the Constant Definitions section, <Boolean variable> in the Variables chapter, and <function designator> at the end of this chapter. <Relational expression> is defined later in this section.

For a <function designator> to return a value of <Boolean type>, it must be declared with <Boolean type> as its <result type>.

CAND and COR are Burroughs extensions to ANSI Pascal.

See also

Constant Definitions. 31
Function Designators. 109
Variables 257

Examples

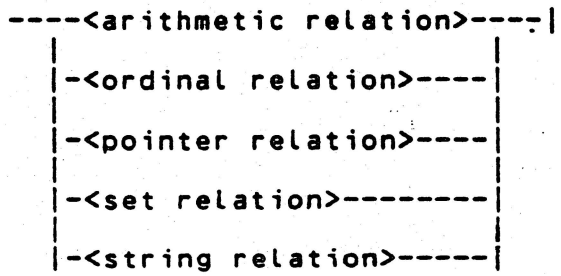
```
var b1, b2, b3 : Boolean;
begin
{ the following 2 expressions are equivalent }
b1 := b1 or b2 and b3;
b1 := b1 or (b2 and b3);
end;
```

```
program cand_example (output);
var i : integer;
    a : array [1..10] of integer;
function f1 (inx : integer) : Boolean;
   begin
   f1 := inx <= 10;
   end;
begin
i := 1;
while f1(i) cand (a[i] = 0) do     { see note below }
   i := i + 1;
end.
```

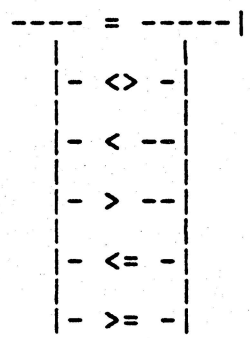
The operator CAND is used in this <Boolean expression> to prevent the evaluation of "a[i]" when "i" has a value that is outside the declared bounds of the array.

Relational Expressions

<relational expression>



<rel op>



A <relational expression> generates a Boolean value by comparing two operands of the same, or similar, types. For relations using the <rel op>s (relational operators), the symbols have the following meanings:

Symbol	Meaning
=	Equals
<>	Not equals
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

<arithmetic relation>



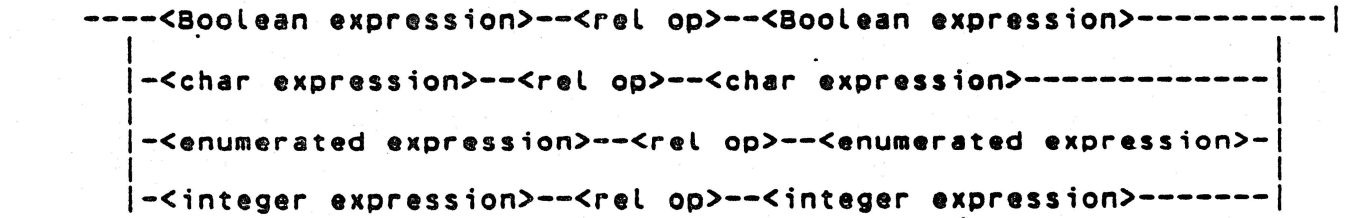
An <arithmetic relation> performs an algebraic comparison of the values of the specified <arithmetic expression>s.

Example

```
var b : Boolean;
    i : integer;
    r : real;
begin
i := 45;
r := 9.0e2;
b := i * 2 >= r;
end;
```

The value of the variable "i" is multiplied by 2 and that result is compared to the value of "r". If "i*2" is greater than or equal to "r", the variable "b" is assigned the value "true"; otherwise, "b" is assigned the value "false".

<ordinal relation>



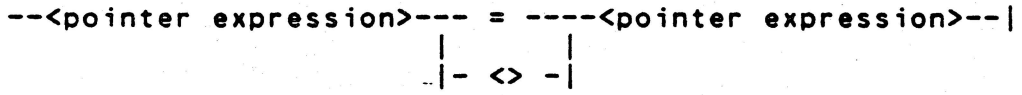
An <ordinal relation> compares the ordinal numbers of the two specified ordinal expressions. The expressions being compared must be of compatible types.

Examples

```
var c : char;
    color : (red, yellow, blue, green, tartan);
    i : integer;
    b : Boolean;
begin
i := 7;
color := tartan;
c := 'Z';
if i > 5 then
   color := blue;
b := color < green;
b := (c = 'Z');
end;
```

In this example, "i > 5", "color < green", and "c = 'Z'" are illustrations of <ordinal relation>s.

<pointer relation>



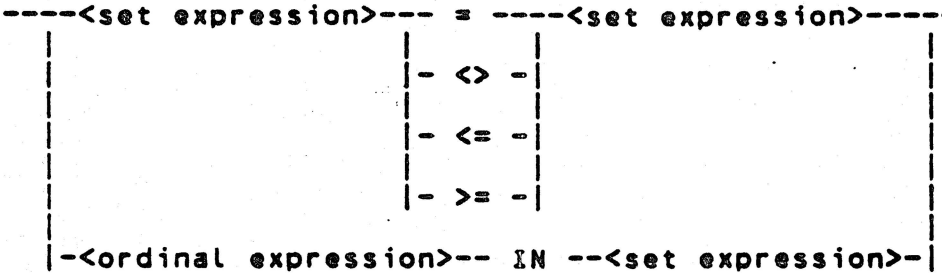
A <pointer relation> compares two <pointer expression>s for equality or inequality. The <pointer expression>s are equal if they refer to the same dynamic variable or are both NIL. When <pointer expression>s are compared, they must be of the same type.

Example

```
program pointer_relation;
type ptr = @rec;
     rec = record
           name : packed array [0..20] of char;
           age  : 0..100;
           end;
var  myptr, yourptr : ptr;
begin
new(myptr);
yourptr := nil;
if (myptr = yourptr) or (yourptr <> nil) then
   display ('Error');
end.
```

This example tests two pointers for equality and then tests a pointer for inequality to NIL.

<set relation>



There are two kinds of <set relation>s. The first compares two <set expression>s for equality (=), inequality (<>), subset relationship (<=), or superset relationship (>=). The second determines whether or not the value of the specified <ordinal expression> is a member of (i.e. is IN) the set specified by the <set expression>. When <set expression>s are compared, they must be of compatible types.

Examples

```
var b1, b2 : Boolean;
    c : set of char;
begin
c := ['a'..'z'];
b1 := ['b','f','A'] <= c;
b2 := 'c' in c;
end;
```

The Boolean variable "b1" is assigned the value "true" if the set containing 'b', 'f', and 'A' is a subset of the set "c"; otherwise, "b1" is assigned the value "false". The Boolean variable "b2" is assigned the value "true" if the character 'c' is a member of the set "c"; otherwise, "b2" is assigned a value of "false".

<string relation>

--<vlstring expression>--<rel op>--<vlstring expression>--|

A <string relation> performs a sequential comparision of the ordinal numbers of corresponding characters in the two <vlstring expression>s.

Two <vlstring expression>s are equal if the length of the <vlstring expression>s are equal and every character in both <vlstring expression>s is identical.

A <vlstring expression> is less than another <vlstring expression> if one of the following conditions is true:

1. In the first character position that differs between the two <vlstring expression>s, the first <vlstring expression> contains a character of a lower ordinal value than the corresponding character in the second <vlstring expression>.

2. The length of the first <vlstring expression> is less than the length of the second <vlstrirg expression> and the expressions compare as equal for the length of the first <vlstring expression>.

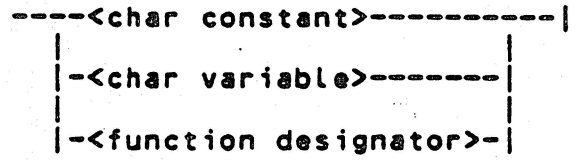
Example

```
var s : string(10);
    b : Boolean;
s := 'abcde';
b := 'abc' < s;
```

The <Boolean variable> "b" is assigned the value "true" because 'abc' is less than the value of the <vlstring variable> "s" (by rule 2, above).

CHAR EXPRESSIONS

<char expression>



A <char expression> generates a value of the <char type>.

<Char constant> is defined in the Constant Definitions section, <char variable> in the Variables chapter, and <function designator> later in this chapter.

For a <function designator> to return a value of <char type>, it must be declared with the <char type>, or a <subrange type> whose host type is the <char type>, as its <result type>.

See also

Constant Definitions. 31
Function Designators. 109
Variables 257

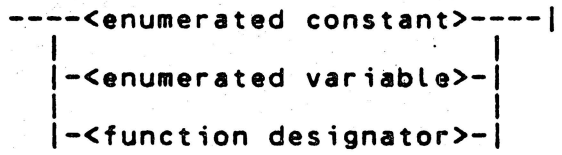
Examples

```
const ch = 'c';
var char1, char2 : char;
function char_function : char;
   begin
   char_function := '?';
   end;
begin
char1 := ch;
char1 := char_function;
char2 := char1;
end;
```

The <char variable> "char1" is assigned the value of the <char constant> "ch" (the character 'c'). "Char1" is then assigned the value of the <function designator> "char_function" (the character '?'). The <char variable> "char2" is assigned the value of "char1" (the character '?').

ENUMERATED EXPRESSIONS

<enumerated expression>



An <enumerated expression> generates a value of an <enumerated type>.

<Enumerated constant> is defined in the Enumerated Types section, <enumerated variable> in the Variables chapter, and <function designator> later in this chapter.

For a <function designator> to return a value of an <enumerated type>, it must be declared with that <enumerated type>, or a <subrange type> whose host type is that <enumerated type>, as its <result type>.

See also

Enumerated Types. 49
Function Designators. 109
Variables 257

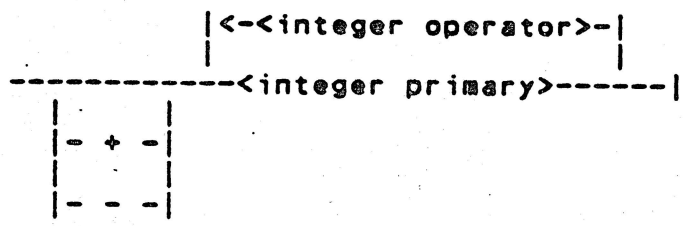
Examples

```
type colortype = (red, yellow, blue, green, tartan);
var color,
    hue : colortype;
function colorwheel : colortype;
   begin
   colorwheel := succ(color);
   end;
begin
color := yellow;
hue := colorwheel;
color := hue;
end;
```

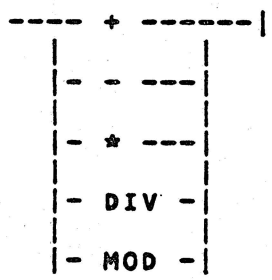
The <enumerated variable> "color" is assigned the <enumerated constant> "yellow". The <enumerated variable> "hue" is assigned the value of the <function designator> "colorwheel" (in this case, the <enumerated constant> "blue"). "Color" is then assigned the value of "hue" (the <enumerated constant> "blue").

INTEGER EXPRESSIONS

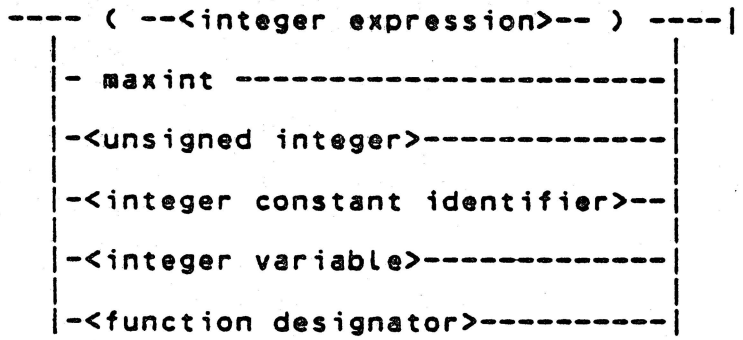
<integer expression>



<integer operator>



<integer primary>



An <integer expression> generates a value of the <integer type>. If the expression generates a value (or an intermediate result) greater than "maxint" or less than "-maxint", an error occurs.

The <integer operator>s are the familiar arithmetic operators for addition (+), subtraction (-), multiplication (*), integer division (DIV), and integer remainder division (MOD).

<Unsigned integer> is defined in the Basic Components chapter, <integer constant identifier> in the Constant Definitions section, <integer variable> in the Variables chapter, and <function designator> later in this chapter.

For a <function designator> to return a value of <integer type>, it must be declared with the <integer type>, or a <subrange type> whose host type is the <integer type>, as its <result type>.

See also

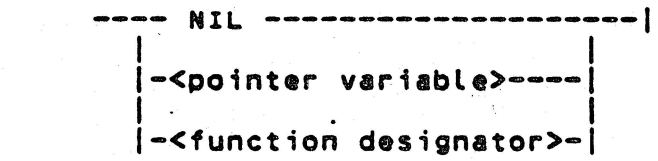
Constant Definitions. 31
Function Designators. 109
Variables 257
Basic Components. 271

Examples

```
var i, j : integer;
begin
j := 79;
i := maxint - (j mod 48);
end;
```

POINTER EXPRESSIONS

<pointer expression>



A <pointer expression> generates a value of a <pointer type>.

The constant NIL denotes a null reference (a pointer that is not currently referencing a variable). <Pointer variable> is defined in the Variables chapter, and <function designator> is defined later in this chapter.

For a <function designator> to return a value of a <pointer type>, it must be declared with that <pointer type> as its <result type>.

See also
Function Designators. 109
Dynamic Allocation Procedures 204
Variables 257

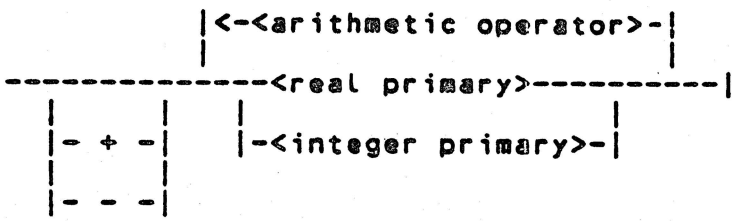
Examples

```
program pointer_exp;
type ptr = @rec;
     rec = record
           name : packed array [1..20] of char;
           age  :  0..100;
           end;
var  myptr, yourptr : ptr;
function allocate : ptr;
   var tempptr : ptr;
   begin
   new(tempptr);
   allocate := tempptr;
   end;
begin
new(myptr);
yourptr := myptr;
myptr := nil;
myptr := allocate;
end.
```

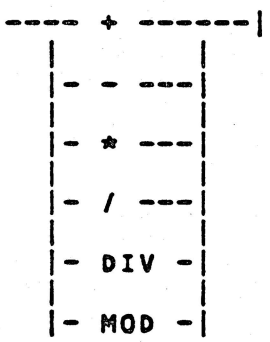
These assignment statements illustrate the three kinds of <pointer expression>s.

REAL EXPRESSIONS

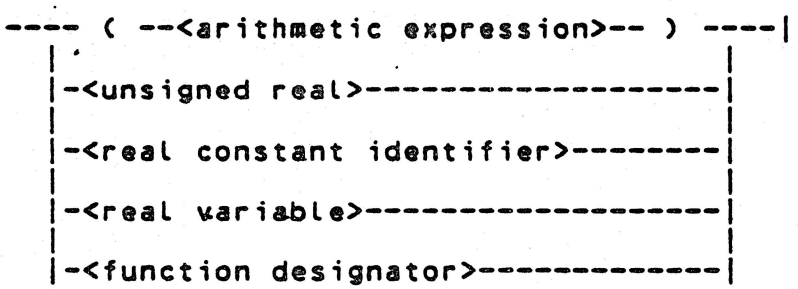
<real expression>



<arithmetic operator>



<real primary>



A. <real expression> generates a value of the <real type>. At least one operand in the expression must be of type real for the expression to be of type real. If the expression generates a value outside the defined range for real values, an error occurs.

The <arithmetic operator>s are the familiar arithmetic operators for addition (+), subtraction (-), multiplication (*), division (/), integer division (DIV), and integer remainder division (MOD). The DIV and MOD operators can be applied only to <integer primary>s.

<Unsigned real> is defined in the Basic Components chapter, <real constant identifier> in the Constant Definitions section, <real variable> in the Variables chapter, and <function designator> later in this chapter.

For a <function designator> to return a value of the <real type>, it must be declared with the <real type> as its <result type>.

See also

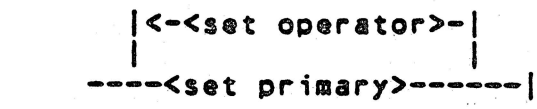
Constant Definitions. 31
Function Designators. 109
Variables 257
Basic Components. 271

Examples

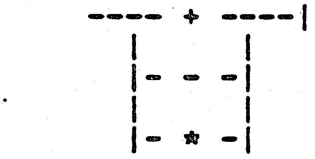
```
const pi = 3.14159;
var a, r : real;
begin
r := 4;
a := pi * sqr(r);
end;
```

SET EXPRESSIONS

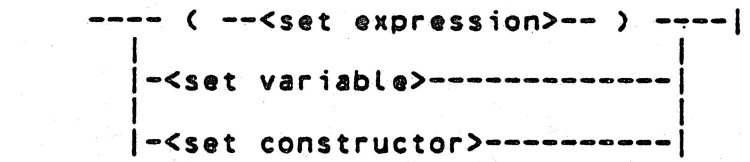
<set expression>



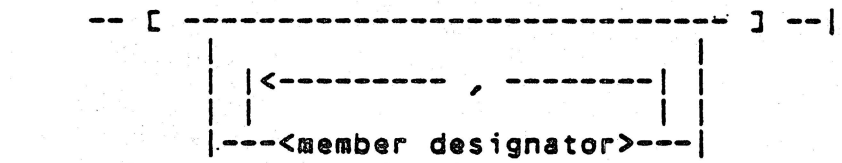
<set operator>



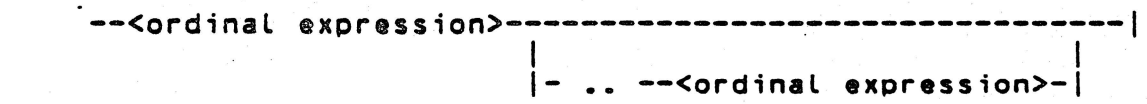
<set primary>



<set constructor>



<member designator>



A <set expression> generates a value of a <set type>. The <set operator>s perform the set operations of union (+), difference (-), and intersection (*).

The operators may be applied to declared <set variable>s or to sets that are defined within the expression by use of the <set constructor> syntax. The <set primary>s within a <set expression> must be of compatible types.

A <set constructor> defines a value of an implied <set type>. The members of the set are specified by the list of <member designator>s, which must all be of the same type or of <subrange type>s of the same host type. <Member designator>s consisting of a single <ordinal expression> denote that <ordinal expression> as a member of the set. If the "<ordinal expression> .. <ordinal expression>" syntax is used, the members denoted are those values from the first <ordinal expression> through the second <ordinal expression>, inclusive. If the second <ordinal expression> is less than the first <ordinal expression>, the set is empty.

The <base type> of the <set type> implied by the <set constructor> is the type (or host type) of the <member designator>s. An empty <set constructor> (that is, "[]") has no specific type and may be used in any <set expression>.

<Set variable> is defined in the Variables chapter.

See also

Same Types. 39
Compatible Types. 40
Variables 257

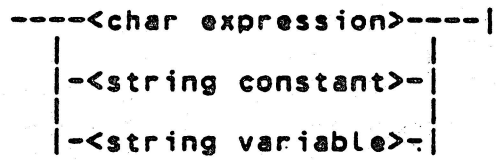
Examples

```
type color = (red, yellow, blue, green, tartan);
var  set1, set2 : set of color;
begin
set1 := [red] + [blue];
set2 := set1 * [yellow, blue, green];
set1 := set1 - set2;
end;
```

"Set1" is assigned the union of the set consisting of the element "red" and the set consisting of the element "blue". "Set2" is assigned the set whose member is the value "blue" (the intersection of the set "set1" and the set containing the elements "yellow", "blue", and "green"). "Set1" is assigned the set difference of "set1" and "set2" or the set whose member is the value "red".

STRING EXPRESSIONS

<string expression>



A <string expression> generates a value of a <string type>.

<String constant> is defined in the Constant Definitions section, and <string variable> is defined in the Variables chapter.

See also
Constant Definitions. 31
Variables 257

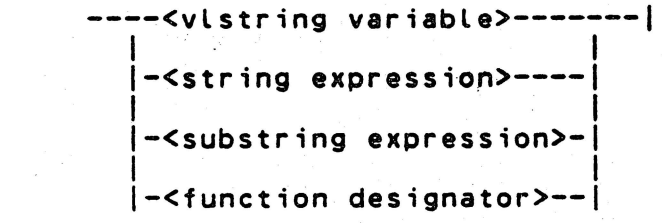
Examples

```
const str1 = 'abcde';
var   str2, str3 : packed array [1..5] of char;
begin
str2 := str1;
str3 := str2;
str2 := '12345';
end;
```

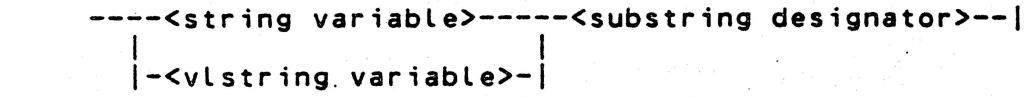
The string variable "str2" is assigned the value of the string constant "str1". The string variable "str3" is assigned the value of the string variable "str2". The string variable "str2" is assigned the character string '12345'.

VLSTRING EXPRESSIONS

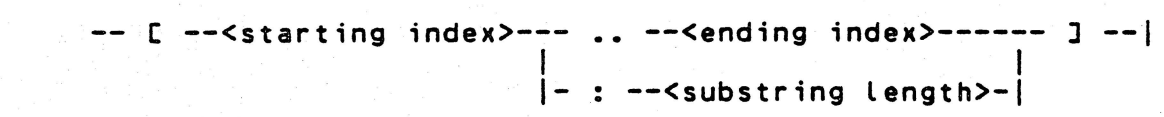
<vlstring expression>



<substring expression>



<substring designator>



<starting index>



<ending index>



<substring length>



A <vlstring expression> returns a value of a <vlstring type>.

A <substring expression> references the portion of the string or vlstring variable specified by the <substring designator>.

The "..<ending index>" form of a <substring designator> references the elements of the string or vlstring from <starting index> to <ending index>, inclusive.

The ":<substring length>" form of a <substring designator> references the elements of the string or vlstring starting with the element specified by the <starting index> and continuing for <substring length> characters.

<Starting index> must be between 1 and the length of the string or vlstring, inclusive. <Ending index> must be greater than or equal to <starting index> and less than or equal to the length of the string or vlstring. <Substring length> must be greater than or equal to 0 and must not reference beyond the current length of the string or vlstring.

<Vlstring variable> is defined in the Variables chapter and <string expression> is defined elsewhere in this chapter.

For a <function designator> to return a value of a <vlstring type>, it must be the <concat function> or the <string function>. These functions are defined in the String Handling Procedures and Functions section.

See also

String Handling Procedures and Functions. 26
Variables 27

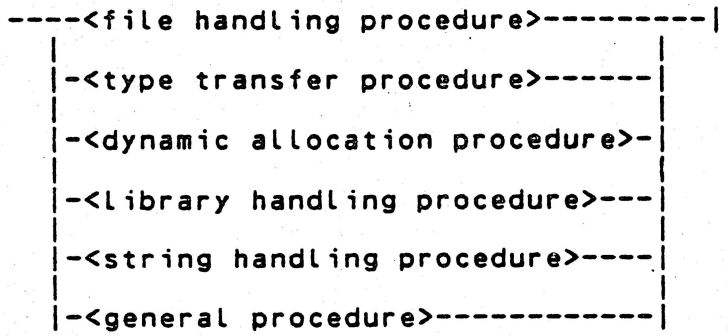
Example

```
var s1, s2 : string(10);
         a : packed array [1..5] of char;
a := 'abc';
s1 := a;
s2 := s1[1..2];
```

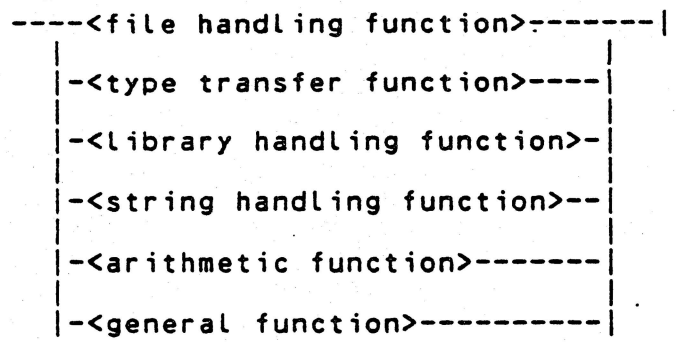
The <string variable> "a" is assigned the value "abc ". The <vlstring variable> "s1" is assigned the value "abc ". The <vlstring variable> "s2" is assigned the value "ab".

6 PREDEFINED PROCEDURES AND FUNCTIONS

<predefined procedure>



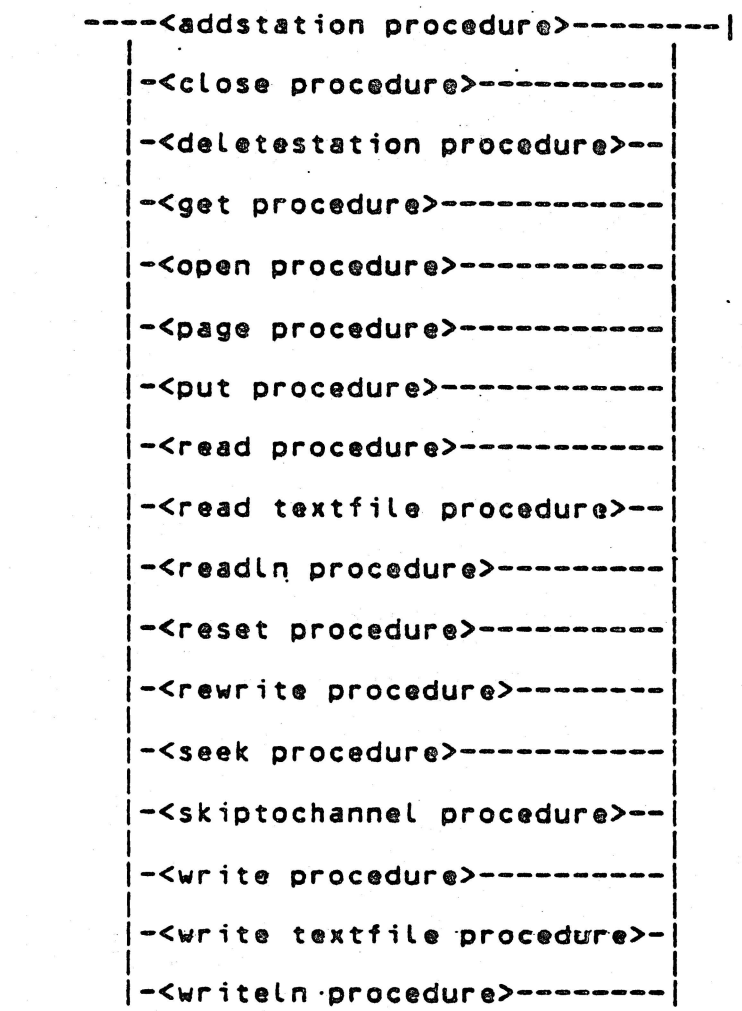
<predefined function>



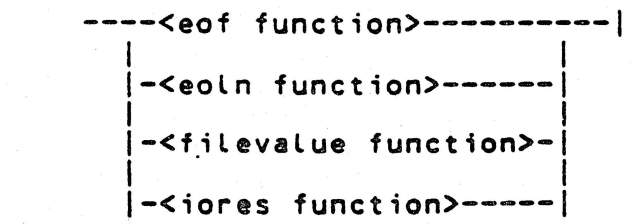
Many Pascal features, including input/output facilities and dynamic variables, are made available through predefined procedures and functions. Although procedures and functions are syntactically different constructs, this difference is not emphasized in this chapter. Instead, procedures and functions relating to a particular program application, such as file handling, are grouped together to provide some continuity in their descriptions. Within each group, the procedures and functions are listed alphabetically.

6.1 FILE HANDLING PROCEDURES AND FUNCTIONS

<file handling procedure>



<file handling function>



The file handling procedures and functions are the basic mechanism for performing input and output operations in Pascal. Some file handling procedures and functions operate on files, some on textfiles, and some on both.

Each procedure and function is defined below, in its own subsection. The first subsection discusses general I/O concepts and should be read before the material in the later subsections.

6.1.1 I/O CONCEPTS

This section describes files, textfiles, file attributes, and input/output operations in Pascal. There are many Burroughs extensions defined in the area of I/O, to enable Pascal programs to access the system-defined I/O subsystem. Programmers who are interested in writing portable programs should pay special attention to the following discussion.

The following topics are discussed in this section:

TERMINOLOGY
- Standard Files and Textfiles
- Inspection Mode and Generation Mode
- Buffer Variables
- File Attributes
- Logical and Physical Files
- Permanent and Temporary Files

STANDARD FILES
- Inspection Mode Operations on Standard Files
- Generation Mode Operations on Standard Files
- Inspection/Generation Operations on Standard Files
- Representation of Standard Files

TEXTFILES
- Inspection Mode Operations on Textfiles
  - Lazy I/O
- Generation Mode Operations on Textfiles
- Representation of Textfiles
- The Textfiles "Input" and "Output"

USE OF FILE ATTRIBUTES

I/O EXCEPTION HANDLING

TERMINOLOGY

The following paragraphs describe some of the basic terms used in defining the kinds of files and input/output operations available in Pascal. In some cases, more detailed information appears in the "Standard Files", "Textfiles", and "Use of File Attributes" discussions later in this section.

Standard Files and Textfiles

In Pascal, there are two types of files: "standard files", which are files of any <component type>, and "textfiles", which are special files of characters. A standard file is declared with a <file type>, and a textfile is declared with a <textfile type>. Note that a variable declared as "file of char" is a standard file, not a textfile.

Standard files are used to transfer data in "machine-readable" form between a program and a physical file. This form of I/O is generally faster and more storage efficient than textfile I/O, but it is not as convenient for use with terminals, line printers, and other character-oriented devices. Textfiles provide translation between the internal representation of data and an external character format. Thus, textfiles are generally better than standard files for representing data in "human-readable" form.

The operations defined for these two types of files are quite different from each other and are described separately throughout this section.

Inspection Mode and Generation Mode

In ANSI Pascal, there are two modes of file operation: "inspection mode", in which the file is being read and not written, and "generation mode", in which the file is being written and not read. In Burroughs Pascal, a third mode, "inspection/generation", is provided for standard files, allowing the file to be both read and written.

Buffer Variables

Associated with each file variable is an implicitly declared buffer variable. The type of the buffer variable is the same as the <component type> of the file ("char" for textfiles). The buffer variable may be used in expressions, assignment statements, and other constructs in just the same fashion as any other variable of the same type. For several predefined operations, data is transferred from the buffer variable to

the file, or vice versa. If the identifier associated with the file is "f", the buffer variable is indicated by "f@".

File Attributes

File attributes are system-defined variables that describe aspects of a file or textfile from the point of view of the I/O subsystem. The compiler assigns appropriate values for the various file attributes when files are declared. In many cases, no further specifications need be made by the programmer. Syntax is provided in the list of <program parameters> and in the <setattribute procedure> to allow programmatic assignment of file attribute values. The <getattribute procedure> allows the programmatic examination of file attribute values.

Logical and Physical Files

A file, as viewed by the program, is a "logical file" that is read or written somewhat independently of the characteristics of the device involved. A file, when thought of in terms of the device used to create it or the medium upon which it is stored, is referred to as a "physical file". Before data can be transferred between a Pascal program and a physical file, a physical file must be assigned to the relevant file or textfile variable. This assignment is made when the file is opened, through a call on the "open", "reset", or "rewrite" procedure.

The decision as to which physical file will be assigned is controlled by the value of several file attributes for the file and by the particular operation used to open the file.

The default value of the KIND attribute in Pascal is DISK. The default value of the TITLE attribute is, as in ALGOL or COBOL, the first 17 characters (translated to upper case) of the <variable identifier> of the file or textfile.

The desired physical file may be a new file or an existing file. If a file is opened using the <reset procedure> or the <open procedure> without the "new" <open option>, an existing file is assumed. If the <rewrite procedure> or the <open procedure> with the "new" <open option> is used, a new file is created. The current setting of the NEWFILE attribute is overridden by these operations; thus, setting NEWFILE has no effect.

Examples

```
program p(f,
          g:f le <kind=tape, title='MASTERFILE', serialno='T123'> );
type employee_record = record
                         name : packed array [1..25] of char;
                         department : 1..9000;
                       end;
var f : file o  employee_record;
    g : file o  employee_record;

begin
{ The following statement creates a new disk file with the title
  "F" and assigns the file to the variable "f". }
rewrite(f);

{ The following statement attempts to locate an existing tape file
  with the title 'MASTERFILE" on a tape whose serial number is
  "T123". If the file is not found, the program will be suspended
  on a "NO FILE" condition, awaiting an operator response. }
reset(g);

end.
```

Permanent and Temporary Files

Files may be further classified as "permanent files" or "temporary files". A file created by a Pascal program is a temporary file unless otherwise specified. A temporary file exists only while the program that created it is running. It is discarded as the result of a "close" that does not specify "save" or "crunch" or when the block in which it is declared is exited. A temporary file cannot be accessed by any other program.

A permanent file, on the other hand, may exist beyond the lifetime of the program and can be accessed by a logical file other than the one used to create it. A permanent file can be created by a Pascal program in one of three ways:

a) If the file name appears in the <program heading>, the file will become a permanent file as soon as it is created (that is, when the <rewrite procedure> or the <open procedure> with the "new" <open option> is invoked). A file name in the <program heading> has the effect of setting the value of the PROTECTION file attribute for that file to SAVE.

b) The PROTECTION file attribute can be explicitly set to SAVE.

c) The file can be closed by a "close" operation that specifies either "save" or "crunch".

Note that, in all cases, an existing permanent file, if any, is replaced. In cases a) and b), the permanent file is replaced as soon as the new file is opened. In case c), the permanent file will not be replaced until the "close" is executed.

A permanent file can be explicitly removed by executing a "close" with the "purge" option.

Examples

```
program p(f);
type employee_record = record
                         name : packed array [1..25] of char;
                         department : 1..9000;
                       end;
var f : file of employee_record;
    g : file of employee_record;

begin
{ The following statement creates a new permanent file.  The file
  is permanent because "f" appears in the program parameter list. }
rewrite(f);

{ The following statement opens a new file.  At this point, the
  file is temporary. }
rewrite(g);

{ The following statement causes "g" to become a permanent file. }
close(g,save);
end.
```

STANDARD FILES

A standard file is a variable of a <file type>. It consists of a (theoretically) unbounded sequence of components of its <component type>. In practice, of course, a file is limited by the size of the device with which it is associated and other system resource limitations.

No special formatting of data is performed for standard files. The operations that can be performed on a standard file depend on the mode of the file (that is, inspection mode, generation mode, or inspection/generation mode).

Inspection Mode Operations on Standard Files

A file is put in inspection mode as th. result of a "reset" operation. The operations that can be performed when the file is in inspection mode are "reset", "get", "read", "seek", "close", and "rewrite".

"Reset" assumes that a file already exists. The file may be open or closed. If the file is open, it is repositioned at the beginning. If the file is closed, it is opened. The first component of the file is assigned to the buffer variable. Immediately following a "reset" operation, the position of the file can be viewed as follows:

```
X0  X1  X2  X3  ...  Xn  eof
*   +
```

"*" indicates the current value of the buffer variable and "+" indicates the next component to be accessed. "Xn" is the last actual component of the file ("eof" can be considered a special component marking the end of the file).

The fundamental input operation is "get". "Get" causes the component of the file indicated by "+" to be transferred to the buffer variable; it then positions the file to the next component. After performing a "get", the file position becomes the following:

```
X0  X1  X2  X3  ...  Xn  eof
    *   +
```

The file can be accessed sequentially by successive "get" operations until the file is positioned at the "eof" component:

```
X0  X1  X2  X3  ...  Xn  eof
                     ↑   ↑
```

At this point, another application of "get" will cause the buffer variable to become undefined. In addition, the <eof function> will return the value "true" if called (until now, the <eof function> would have returned "false"). If "get" is called when the file is at end-of-file, an error occurs.

The "read" operation ("read (f,x)") is defined to be equivalent to the following two statements:

```
x := f@;
get(f);
```

Any errors defined for these two statements are defined for "read" (for example, "f@" must be assignment compatible with the type of "x"). A "read" of multiple values ("read(f,X0,X1,...,Xn)") is equivalent to the following statements:

```
read (f,X0);
read (f,X1);
     .
     .
     .
read (f,Xn);
```

"Seek" is an additional function defined as a Burroughs extension; it allows a file to be accessed randomly. In inspection mode, "seek(f,i)" positions the file such that the next "get" operation will assign the (i+1)th component of the file to the buffer variable.

```
X0  ...  ...  Xi ... ...  eof
              ↑
```

The buffer variable is undefined after a "seek" operation.

A "seek" operation may specify a position that is beyond the end-of-file. The effect in this case is as if each position beyond the last component were occupied by an end-of-file component.

```
Xi ... ... Xn eof eof eof ... ... eof
                +
```

A "get" at this point causes the <eof function> to return "true", leaving the buffer variable undefined. A second "get" results in an error.

The "close" operation terminates the processing of the file and disconnects the logical file from the physical file.

The "rewrite" operation puts the file into generation mode.

Generation Mode Operations on Standard Files

A file is put in generation mode as the result of a "rewrite" operation. The operations that can be performed when the file is in generation mode are "rewrite", "put", "write", "seek", "close", and "reset".

"Rewrite" may be called while the file is open or closed. If the file is open, the attached physical file is released and a new empty file is created. The file is positioned such that an item written will occupy the first position. Following a "rewrite", the <eof function> returns "true" will continue to return "true" as long as the file is in generation mode.

The fundamental operation in generation mode is "put". "Put" causes the contents of the buffer variable to be transferred to the file at the position indicated by "+" and then moves the file to the next position. It is an error if the value of the buffer variable is undefined when "put" is called. Following a "put" operation, the buffer variable becomes undefined. A file following a "rewrite" and "put" would look like this:

```
XO
 +
```

The "seek" operation allows a file in generation mode to be positioned such that a subsequent "put" will transfer the contents of the buffer variable to the specified position in the file (that is, "seek(f,i)" positions the file at the (i+1)th position). The buffer variable is undefined after a "seek" operation; once it has been assigned a value, a subsequent "put" operation would result in the following file structure:

```
  <--undefined-->
X0 ... ... ... ... Xi
                   +
```

A "write" operation ("write(f,x)") is equivalent to the following two statements:

```
f@ := x;
put(f);
```

Any errors defined for these two statements are defined for "write" (for example, "x" must be assignment compatible with the type of "f@"). A multiple-value "write" operation ("write(f,X0,X1,...,Xn)") is equivalent to the following statements:

```
write(f,X0);
write(f,X1);
     .
     .
     .
write(f,Xn);
```

When a file in generation mode is closed, as the result of either a "reset" or "close" operation, and the physical file is retained, a logical end-of-file component is placed following the last position in the file that was assigned a value. At this point, the file might look like this:

```
X0  X1  0 ... Xi  Xi+1  0  Xn  eof
```

"0" marks positions that were never written (because of "seek" operations) and are therefore undefined.

The "close" operation terminates the processing of the file and disconnects the logical file from the physical file.

The "reset" operation puts the file into inspection mode.

Inspection/Generation Operations on Standard Files

A file is placed in inspection/generation mode as the result of an "open" operation. All operations that can be performed in inspection or generation mode can be performed in this mode, with the same effects as before.

"Open" can be called only if the file is closed. A new file or an existing file may be opened, depending on whether or not the "new" <open option> is specified.

The first component of the file is not read as a result of a call to "open", as it is in the case of a "reset". Also, the buffer variable is undefined; therefore, the first I/O operation must not be a "read", as "read" assumes that the buffer variable is defined.

In inspection/generation mode, the <eof function> always returns "false" unless the last operation was a "get" beyond the last component of the file.

The positioning of the file following an "open" depends on whether or not the "atend" option was specified. If it was not, the file is left in the following position:

```
X0  X1 ... ... Xn  eof
+
```

If "atend" was specified, the file is left in the following position:

```
X0  X1 ... ... Xn  eof
                   +
```

Note that the <eof function> will not yet return "true", because a "get" operation has not yet been performed.

Representation of Standard Files

Standard files are bound to files of fixed-length logical records (as if the BLOCKSTRUCTURE file attribute were specified as FIXED). Each Pascal file component corresponds to an I/O subsystem "logical record".

Components are read and written in their raw binary form; no special formatting is performed. A component in the file is represented exactly as a variable of that type is represented in memory (refer to Appendix C, Data Representation, for details). Any pointers written to a file become undefined; pointer values read from a file should never be used.

The file structuring attributes INTMODE, FRAMESIZE, MAXRECSIZE, and BLOCKSIZE are set by the compiler to appropriate values based on the <component type> of the file. If the <component type> is a packed array with 4-bit elements, INTMODE is set to HEX and FRAMESIZE is set to 4. If the <component type> is a packed array of characters, INTMODE is set to EBCDIC or ASCII (depending on the setting of the STRINGS compiler control option) and FRAMESIZE is set to 8. If the <component type> is a packed array of any other type with 8-bit elements, INTMODE is set to EBCDIC and FRAMESIZE is set to 8. All other <component type>s result in INTMODE set to SINGLE and FRAMESIZE set to 48. If INTMODE is set to ASCII, the I/O subsystem will perform translation if the EXTMODE of the associated physical file is EBCDIC.

The MAXRECSIZE attribute is set to the length, in FRAMESIZE units, of the <component type> of the file. If the <component type> is a <string type>, BLOCKSIZE is set to MAXRECSIZE (that is, the file is unblocked). If the <component type> is not a <string type>, BLOCKSIZE is set to a value between 270 and 570 words (expressed in the appropriate units) that minimizes the amount of space that would be wasted if the file were a disk file (based on 30-word disk segments). If the MAXRECSIZE is greater than 570, BLOCKSIZE is set to MAXRECSIZE.

TEXTFILES

Textfiles are intended for "human-readable" input and output. The feature provides for formatting and translation of values between internal system representation and an external character form.

A textfile has some properties in common with a "file of char", but they are not equivalent. A textfile can be viewed as a sequence of characters, but special components and operations exist that allow characters to be grouped into lines. More specifically, a textfile is a sequence of components called "lines", which are separated by logical components called "end-of-line markers". Each line consists of a sequence of characters.

A textfile is denoted by use of the predefined <type identifier> "text". A textfile variable has an associated buffer variable that is defined to be of type "char".

In order to allow textfiles to be used interchangeably with existing system and language file structures, two kinds of textfile lines are supported: fixed-length lines and variable-length lines. The two differ from a functional point of view only in the definition of "writeln". These formats are described under Representation of Textfiles.

The operations that can be performed on a textfile depend on the mode of the textfile. Only inspection mode and generation mode are available for textfiles.

Inspection Mode Operations on Textfiles

A textfile is put in inspection mode as the result of a "reset" operation. The operations that can be performed on a textfile in inspection mode are "reset", "get", "read", "readln", and "close". The operations are described using a three-line, variable-length-line textfile for purposes of illustration.

As with a standard file, "reset" assumes an existing textfile. Following "reset", the file can be viewed as follows:

```
C0  C1  ...  ...  ...  Cn  eoln
*   +

C0  ...  Cm  eoln

C0 ...  ...  Cz  eoln  eof
```

"*" denotes the currently defined value of the buffer variable and "+" denotes the next component to be accessed. "Eoln" represents the end-of-line marker and "eof" marks the end of the file. "Eoln" exists as a functional definition only; such a character is not actually present in the file, but is implied by position (refer to Representation of Textfiles for additional details).

A textfile can only be accessed sequentially. The basic input operation is "get". "Get" operates on a textfile in a manner very similar to a "get" on a "file of char". Each "get" accesses the next component of the file. When the file is in the following position, another "get" will put the file in end-of-line state:

```
C0  C1  ...  ...  ...  Cn  eoln
                     *   +
```

In end-of-line state, the <eoln function>, if called, will return the value "true" and the value of the buffer variable will be ' ' (blank). A second "get" will result in the following file position:

```
C0  C1  ...  ...  ...  Cn  eoln

C0  ...  Cm  eoln
*   +

C0 ...  ...  Cz  eoln  eof
```

When the file is positioned as follows, a "get" will again put the file in end-of-line state, and a second "get" will put the file in end-of-file state:

```
C0  C1  ...  ...  ...  Cn  eoln

C0  ... Cm  eoln

C0 ...  ...  Cz  eoln  eof
             *   +
```

After the second "get", the <eof function>, if called, will return "true" and the value of the buffer variable is undefined. When the file is in the end-of-file state, an error will occur if "get", "read", "readln", or "eoln" is called.

The "read" operation has special semantics for textfiles, and the definition of "read" depends on the type of the variables in the parameter list. The action of "read" on a textfile is described in the section on the <read textfile procedure>.

"Readln" causes the remaining characters in a line to be skipped and positions the file at the beginning of the next line. "Readln" is equivalent to the following statements:

```
while not eoln(f) do
   get(f);
get(f);
```

A multiple-value "readln" operation ("readln(f,X1,...,Xn)") is equivalent to the following statements:

```
read(f,X1,...,Xn);
readln;
```

The "close" operation terminates the processing of the file and disconnects the logical file from the physical file.

Lazy I/O

Textfile input operations require special processing to ensure that the operations are performed in the order that the programmer expects. In particular, a problem arises when reading from a textfile assigned to a remote file. A typical interactive program writes a "prompt" message to the user indicating that it is expecting input and then reads the user's response. Because "reset", "read", and "readln" operations implicitly read one character ahead (that is, the buffer variable is assigned a value that will subsequently be stored into a variable in a "read" or

"readln" parameter list), most interactive programs would end up waiting for the user to provide a response to a prompt that hadn't been displayed yet.

To avoid these potentially frustrating interactions, Burroughs Pascal uses an input technique known as "lazy I/O". With "lazy I/O", data is not transferred to the buffer variable until it is required by the program. Thus, if a "reset", "get", "read", or "readln" operation is performed and the value of the buffer variable following the operation is defined to be the first character of a new line, that line is not read and the value is not actually assigned until one of the following actions occurs:

a) another "get", "read", or "readln" operation is performed.

b) the <eof function> or the <eoln function> is called.

c) the buffer variable is used in an expression or as an actual variable parameter.

Other implementations may use other I/O techniques under these circumstances, and programs may behave differently.

Generation Mode Operations on Textfiles

A textfile is put in generation mode as the result of a "rewrite" operation. The operations that can be performed on a textfile in generation mode are "rewrite", "put", "write", "writeln", and "close".

As with a standard file, "rewrite" creates a new empty textfile.

The basic output operation is "put". "Put" is defined as for a "file of char". At any point, there is a current line that is either empty or partially generated. An error occurs if an attempt is made, through the use of "put", "write", or "writeln", to put more characters in a line than the defined maximum (refer to "Representation of Textfiles" below).

The "write" operation has special semantics for textfiles, and the definition of "write" depends on the type of the variables in the parameter list. The action of "write" on a textfile is described in the section on the <write textfile procedure>.

The current line is terminated by the "writeln" operation. When the file consists of variable-length lines, "writeln" causes a logical end-of-line mark to be placed at the current position, thus immediately

terminating the line. If the file consists of fixed-length lines, the line is padded with blanks from the current position out to the defined line length before the line is terminated.

A multiple-value "writeln" operation ("writeln(f,X1,...,Xn)") is equivalent to the following statements:

```
write(f,X1);
     .
     .
     .
write(f,Xn);
writeln;
```

If a "reset" operation is performed or the file is closed without being released and the current line is not empty, an implicit "writeln" is performed and an end-of-file is written.

The "close" operation terminates the processing of the file and disconnects the logical file from the physical file.

Representation of Textfiles

A textfile is structured as a sequence of components that are called "lines". Each line may contain a fixed or a variable number of characters, depending on the BLOCKSTRUCTURE attribute of the file. For both fixed- and variable-length lines, the maximum length of a line is determined by the MAXRECSIZE attribute. For textfiles, the maximum value that can be assigned to MAXRECSIZE is 9999. The compiler automatically generates values of BLOCKSTRUCTURE and MAXRECSIZE that are designed to match the device associated with the textfile variable. In most cases, the programmer need not be concerned with these details.

A fixed-length-line structure is equivalent to a specification of FIXED for the BLOCKSTRUCTURE file attribute. Fixed-length lines are a Burroughs extension to ANSI Pascal. The assumed length of the lines is defined in Table 6-1.

For fixed-length lines, the end-of-line is assumed to be at the end of the fixed number of characters. When a "writeln" operation is performed, the line is blank filled on the right if necessary. It is an error to attempt to write more characters on the line than the fixed length allows.

A variable-length-line structure is equivalent to BLOCKSTRUCTURE = VARIABLE or BLOCKSTRUCTURE = EXTERNAL, as defined in Table 6-1. For BLOCKSTRUCTURE = VARIABLE, the length of each line is stored in the file as a four-digit decimal number preceding each line. The presence of the length field is transparent to the Pascal program, with the exception that it must be taken into account when changing the maximum line length. For example, if a maximum line length of 256 is desired, the MAXRECSIZE attribute must be set to 260. For BLOCKSTRUCTURE = EXTERNAL, the length of each line is provided by the I/O subsystem as each line is read or written.

The default value for the maximum length of a line and the default value of the BLOCKSTRUCTURE attribute are dependent on the device type associated with the file. These values are assigned when the file is first opened, according to the values shown in Table 6-1. BLOCKSTRUCTURE and MAXRECSIZE can be assigned values by the programmer, and these values will override the compiler-assigned values (please refer to "Use of File Attributes").

Table 6-1. Device-Dependent Textfile Attributes

KIND (Device)	Default Maximum Line Length	Default BLOCKSTRUCTURE	Allowed BLOCKSTRUCTUREs
Disk *	132	VARIABLE	FIXED, VARIABLE
Tape *	132	VARIABLE	FIXED, VARIABLE
Remote	80	EXTERNAL	FIXED, EXTERNAL
Reader	80	FIXED	FIXED
Printer	132	FIXED	FIXED
Punch	80	FIXED	FIXED
Port **	132	EXTERNAL	FIXED, EXTERNAL
Diskette	80	FIXED	FIXED

* When an existing labeled file is opened for input, the length and BLOCKSTRUCTURE assume the values that were used to create the file.

** For port files, the maximum length of a line is negotiated between the two programs communicating through the port file.

The Textfiles "Input" and "Output"

There are two predefined textfiles, called "input" and "output". In order to use these files, the names "input" and "output" must appear in the list of <program parameters>. When they appear, they become implicitly declared; thus, they must not be (re)declared in the program's <variable declarations>. If "input" and "output" do not appear in the list of <program parameters>, the predefined files are not declared and therefore are not available for use. Any subsequent declaration of either "input" or "output" declares a variable other than the predefined one.

In some file handling procedures, such as "readln" and "writeln", the file parameter may be omitted; in these cases, the appropriate predefined textfile (either "input" or "output") is inferred, as specified for each procedure.

If the program is compiled through CANOE, the KIND attributes of the "input" file and the "output" file are set to REMOTE, unless otherwise specified. If the program is compiled through WFL, the KIND attribute for the "input" file is set to READER, and the KIND of the "output" file is set to PRINTER.

USE OF FILE ATTRIBUTES

Burroughs Pascal, together with the I/O subsystem, provides several methods for assigning and interrogating the values of file attributes. File attributes can be assigned in the following ways:

a) through file equation as the program is compiled

b) through file equation as the program is executed

c) by specifying file attributes in the <program parameters>

d) dynamically, through the <setattribute procedure>

When settings from these methods conflict, precedence is determined by the following order, from highest to lowest precedence:

1) the <setattribute procedure>

2) run-time file equation

3) compile-time file equation

4) settings in the <program parameters>

System defaults can always be overridden. Setting attributes used by "rewrite", "reset", or "open", namely NEWFILE, MYUSE, and FILEUSE, will have no effect, because they will be overridden. Setting the OPEN attribute is not allowed because it causes the I/O subsystem to open the file without changing the necessary Pascal program state; a file can be opened at run time using the "open", "reset", or "rewrite" procedure.

File attributes can be dynamically accessed through the <getattribute procedure>. The AVAILABLE and PRESENT attributes cannot be accessed because, as a side effect, accessing either attribute causes the I/O subsystem to open the file without changing the necessary Pascal program state.

A complete list of the attributes available on the system is defined in the I/O Subsystem Manual. Some of these are not available in Pascal, because there is no corresponding data type to represent their value. The mapping between attribute types as specified in the I/O Subsystem Manual and Pascal data types is given in the following table:

I/O type	Pascal type
integer (without mnemonics)	integer
integer (with mnemonics)	integer, for "setattribute" and "getattribute"; mnemonic, otherwise.
Boolean	Boolean
pointer	string, vlstring
real (numeric)	real
real (SERIALNO)	string or vlstring (6 significant characters)
real (bits -- STATE, ATTVALUE, USERINFO)	[not available]
translatetable	[not available]
event	[available through the <wait procedure>]
STATIONLIST	[available through the <addstation procedure> and the <deletestation procedure>]

Pointer-valued attributes are always assumed by the system to have an EBCDIC representation. Thus, if the STRINGS compiler control option has been set to ASCII, which causes the compiler to assume that all string variables are represented in ASCII, it will automatically translate from ASCII to EBCDIC or EBCDIC to ASCII when setting or accessing pointer-valued attributes.

When accessing a pointer-valued attribute, if the value is shorter than the string variable into which it is being stored, the value is blank filled on the right; if the value is being stored into a vlstring variable, it is not blank filled. If the value is longer than the length of the string variable or the maximum length of the vlstring variable into which it is being stored, an error occurs. Note: A period (".") is not returned as part of the value.

If a null pointer-valued attribute is placed into a string variable, the variable will contain all blanks. If a null pointer-valued attribute is placed into a vlstring variable, the variable has a null value and a length of 0.

When setting the SERIALNO attribute, only the first six characters are used. A value less than six characters in length is blank filled on the right to six characters. A null SERIALNO is specified with a string or vlstring variable that is at least six characters long, in which the first six characters are NUL ("chr(0)").

The SERIALNO attribute, when accessed, returns a value that is six characters in length. If the SERIALNO is returned in a string variable that is longer than six characters, the value is blank filled on the right. If the SERIALNO is returned in a vlstring variable, the value is not blank filled. An error occurs if the value is longer than the length of the string variable or the maximum length of the vlstring variable.

I/O EXCEPTION HANDLING

In executing the standard I/O operations, any of several exception conditions may occur. For example, a parity error on a tape file may be encountered, or a specified time limit for data comm input may have elapsed. Normally, the system handles I/O exception conditions by terminating the program. However, Pascal programs requesting an operation may also request to handle the exception. In this case, the system merely passes an "I/O result" back to the program. The I/O result indicates whether or not an exception occurred and, if so, the nature of the exception. If an exception occurred, the program will not be terminated, but will continue executing at the next statement.

Certain exceptions correspond to conditions defined to be errors in the descriptions of the file handling procedures. These conditions are not treated as errors when the exception is handled by the program. Conditions defined to be errors that do not appear in the lists of exceptions below are always treated as errors.

Several of the predefined <file handling procedure>s may be invoked through either a procedure invocation or a function invocation. System handling of exception conditions is indicated by requesting the operation through a procedure invocation. If the operation is requested through a function invocation, then the I/O result is returned as the function result, and no system action is taken.

In all cases, the type of the result is integer. Each exception is represented by an integer constant that has an associated mnemonic value. The <iores function> is provided in order to translate mnemonic values for exception conditions to the appropriate integer constant. The ordering of exception values or the particular integer used to represent such a value should not be relied upon.

The operations that may return results, and the values and meanings of those results, are defined as follows:

Open

The exceptions described here may occur when the I/O subsystem is invoked to open a file:

Iores(Ok)
: The operation was successful.

Iores(Unknown)
: An error occurred, but its nature cannot be determined.

Iores(Unavailable)
The permanent file exists, but is not available to this program. This situation occurs when another program or task has the file open with the EXCLUSIVE attribute set to true.

Iores(Nofile)
The permanent file does not exist or, if the file is a port file and the "available" <open option> is used, a matching subfile cannot be found.

Iores(Genealogy)
The genealogy of the file, as defined by the VERSION and CYCLE attributes, does not match.

Iores(Serialno)
The serial number of the file, as defined by the SERIALNO attribute, does not match.

Iores(Nobackup)
The file is cataloged as a nonresident backup file.

Iores(NoResource)
No resources are available to open the file. This situation can occur, for example, when the number of tape drives available to a program is restricted, and the allotted number are already in use.

Iores(Nopack)
A required pack is not mounted.

Iores(Accesscode)
The file is not available due to ACCESSCODE protection in the user's directory.

Iores(Unreachable)
This value is returned only if the value of the KIND attribute is PORT. If the "available" <open option> was specified, then the host indicated by the value of the HOSTNAME file attribute is unknown or has become unreachable. If the "wait" <open option> is specified, then this value is returned if the host becomes unreachable during the process of opening a subfile.

Iores(Notclosed)
The value of the FILESTATE file attribute for the specified file or subfile is not equal to CLOSED.

Iores(Badindex)
This value is returned only if the value of the KIND attribute is PORT. It is returned when a subfile index is specified that has a value less than zero or greater than the value of the MAXSUBFILES file attribute, or when a subfile index is not specified and the value of MAXSUBFILES is greater than 1.

Iores(Openall)
This value is returned only if the value of the KIND attribute is PORT. It is returned when an attempt is made to open all subfiles and an error occurs on any one of them.

Get, Put, Read, Readln, Write, Writeln

These exceptions can occur when an attempt is made to read or write a file.

Iores(Ok)
The operation was successful.

Iores(Badindex)
This value is returned only if the value of the KIND attribute is PORT. It is returned when a subfile index is specifed that has a value less than zero or greater than the value of the MAXSUBFILES attribute, or when a subfile index is not specified and MAXSUBFILES is greater than 1.

Iores(Dataerr)
A data error occurred, an invalid value was received by a textfile input operation, or a line overflow occurred on textfile output.

Iores(Deleted)
The value of the FILEORGANIZATION attribute is RELATIVE and a deleted or duplicated record was referenced.

Iores(Parity)
A parity error on the associated physical medium was encountered. Note that the I/O subsystem will report this error only after performing several unsuccessful retries.

Iores(Porterr)
This error is returned only if the value of the KIND attribute is PORT. It is returned when one of the following errors occurred: 1) a broadcast write failed for one or more subfiles, 2) a "put" with the "dontwait" option failed because no buffer was available, 3) a "get" with the "dontwait" option failed because no data was available.

Iores(Eof)
End-of-file or end-of-page was encountered. Note that this is "end-of-file" as defined by the I/O subsystem, which does not correspond exactly to the value of the Pascal <eof function>. There are several conditions that can cause this exception to occur without causing the <eof function> to return true. Also, this exception does not always occur in generation mode, whereas the <eof function> will always return true in generation mode. An end-of-page exception occurs when the

PAGESIZE attribute has been set and the end of a logical page has been encountered on output.

Iores(Brk)
A break on output from the associated physical device was encountered.

Iores(Timeout)
The TIMELIMIT file attribute has been set, and the specified time limit expired before the I/O operation completed.

Iores(Security)
A security violation was attempted.

Close

These exceptions can occur when the I/O subsystem close routines are invoked to close the file:

Iores(Ok)
The operation was successful.

Iores(Unknown)
An error has occurred, but its nature cannot be determined.

Iores(Notopen)
The file or subfile is already closed.

Iores(Datalost)
All data was not successfully delivered to the destination.

Iores(Recordcount)
A record count error occurred.

Iores(Blockcount)
A block count error occurred.

Iores(Badindex)
This value is returned only if the value of the KIND attribute is PORT. It is returned when a subfile index less than zero or greater than the value of the MAXSUBFILES file attribute is specified, and when a subfile index is not specified and the value of MAXSUBFILES is greater than 1.

Iores(Closeall)
This value is returned only if the value of the KIND attribute is PORT. It is returned when an attempt is made to close all open subfiles, and an error occurs on at least one of them.

When handling exception conditions, special attention should be paid to the timing of the operations, particularly in the case of textfile I/O (Refer to the discussion of "Textfiles", especially "Lazy I/O").

Under certain rare circumstances, it is possible that an open operation on a tape file may perform an implicit close, in which case a close result may be returned. Similarly, a close operation on a tape file may perform an implicit open.

The ability to return I/O results is a Burroughs extension to ANSI Pascal.

See also

Inspection Mode Operations on Textfiles 145
Iores Function. 170

6.1.2 PROCEDURE AND FUNCTION DESCRIPTIONS

ADDSTATION PROCEDURE

<addstation procedure>

```
-- addstation -- ( --<file variable>-- , --<vlstring expression>--->

>- ) ------------------------------------------------------------|
```

The <addstation procedure> will add new stations (terminals, teletypes, etc.) to the list of stations for the file specified by <file variable>. The <file variable> must be associated with a file that has a value of the KIND file attribute equal to REMOTE.

The <vlstring expression> specifies the station or stations to be added by name. The <vlstring expression> must evaluate to either the name of a valid station in the current data communications network or the name of a file defined in that network. In the latter case, invocation of the <addstation procedure> may result in the addition of more than one station to the file.

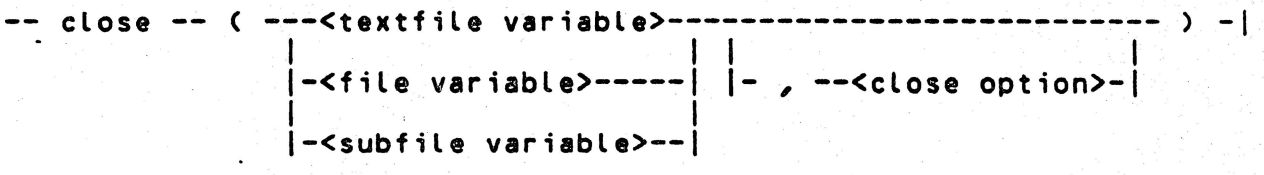
For a given terminal (or other remote device), the station name can be determined through the "?WRU" CANDE command. An error occurs if the <file variable> is not open when the <addstation procedure> is invoked. For more information on adding stations, please refer to the STATIONLIST file attribute description in the I/O Subsystem Reference Manual.

The <addstation procedure> is a Burroughs extension to ANSI Pascal.

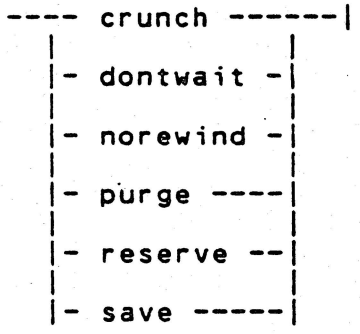
See also
Deletestation Procedure 164

CLOSE PROCEDURE

<close procedure>



<close option>



The <close procedure> terminates processing of the file denoted by <textfile variable>, <file variable>, or <subfile variable>. An error occurs if the file or subfile is not open when the <close procedure> is invoked.

The value of the buffer variable associated with the file becomes undefined. A subsequent attempt to perform any read, write, or seek operation after a "close" operation, without first calling the "open", "reset", or "rewrite" procedure, is an error.

A <subfile variable> is valid only if the value of the KIND attribute of the file is PORT. If a <subfile variable> is used and the associated <subfile index> is zero, all open subfiles are closed. If a <subfile index> greater than zero and less than or equal to the value of the MAXSUBFILES file attribute is specified, only the specified subfile is closed. An error occurs if an invalid <subfile index> is specified, or if MAXSUBFILES is greater than one and a <subfile variable> is not used.

A <close option> may be used to further specify the disposition of the file being closed. If a <close option> is not specified, permanent files remain permanent and are repositioned to the beginning, when the device permits it. Temporary files are released. The connection between the logical file and the physical file is always severed.

The meaning of a particular <close option> depends on the KIND of the file being closed. The valid <close option>s are defined as follows:

Crunch

The "crunch" option causes the file to be made a permanent file. In addition, the value of the file attribute CRUNCHED will be set to true, which has the effect of returning unused storage areas to the system. A file that has been "crunched" can be updated only under certain restricted circumstances. For more information please refer to the description of the CRUNCHED attribute in the I/O Subsystem Reference Manual. The connection between the logical file and physical file is severed. "Crunch" is valid for disk files only.

Dontwait

If the "dontwait" option is specified, control is returned immediately to the program, and the process of actually closing the file takes place in parallel with the execution of the program. "Dontwait" is valid for port files only.

Norewind

The "norewind" option is valid for tape files only and is intended for use when more than one file of the same logical structure is located on a reel. When specified, the tape will not be rewound. Instead, it will be left in a state such that a subsequent open using the same <file variable> or <textfile variable> will open the next file on the tape.

Purge

The "purge" option causes the file to be discarded. A tape file is rewound, and, if - write ring is present, a scratch label is written. A disk file is removed from the directory. The connection between the logical file and the physical file is severed. "Purge" is valid for tape and disk files only.

Reserve

The "reserve" option is valid for tape files only and is intended for use when multiple files with possibly different logical structures are located on a reel. When specified, the tape will not be rewound. In addition, the tape unit will be reserved for use by the calling program only. A subsequent open using any <file variable> or <textfile variable> in the same program will open the next file on the tape. Each file on the tape is required to have a two-level file name (value of the TITLE file attribute), with the first-level name common to all of the files (e.g. 'F/1', 'F/2', 'F/3', ...).

Save

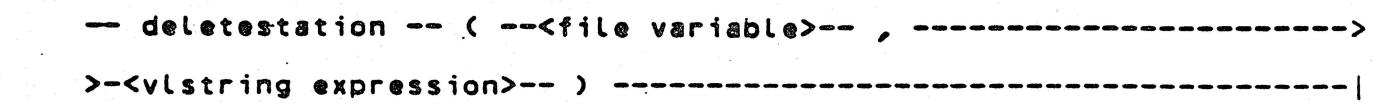
The "save" option repositions the file to the beginning and makes it a permanent file. The connection between the logical file and the physical file is severed. "Save" is valid for tape and disk files only.

If a <close option> that is invalid for the KIND of the file is specified, a simple close appropriate to the device is performed.

The <close procedure> is a Burroughs extension to ANSI Pascal.

DELETESTATION PROCEDURE

<deletestation procedure>



The <deletestation procedure> will delete stations (terminals, teletypes, etc.) from the list of stations of the file specified by <file variable>. The <file variable> must be associated with a file that has a value of the KIND file attribute equal to REMOTE.

The <vlstring expression> specifies the station or stations to be deleted. Please refer to the description of the <addstation procedure> for a description of the values of <vlstring expression>.

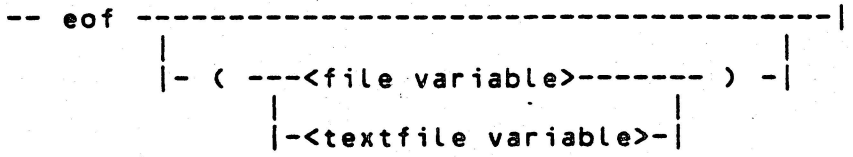
An error occurs if the <file variable> is not open when the <deletestation procedure> is invoked. For more information on deleting stations, please refer to the STATIONLIST file attribute description in the I/O Subsystem Reference Manual.

The <deletestation procedure> is a Burroughs extension to ANSI Pascal.

See also
Addstation Procedure. 160

EOF FUNCTION

<eof function>



The <eof function> returns, as a Boolean value, an indication of whether or not an operation attempted to access beyond the last component of a specified file or the last component of a subfile or station of that file. The circumstances under which the function returns "true" depend on the inspection or generation mode of the file, as follows:

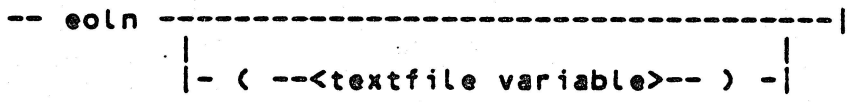
a) If the file is in inspection mode or inspection/generation mode, the function returns "true" if the last operation on the file was a "get", "read", or "reset" beyond the last component.

b) If the file is in generation mode, the function always returns "true".

The file to which the function applies may be specified by including a <file variable> or <textfile variable> in the function call. If no file is specified, the function applies to the textfile "input". If the file is not open, the function returns "false".

EOLN FUNCTION

<eoln function>



The <eoln function> returns, as a Boolean value, an indication of whether or not a particular textfile is positioned at an end-of-line marker. If the file is positioned at an end-of-line marker, the function returns "true"; otherwise, the function returns "false".

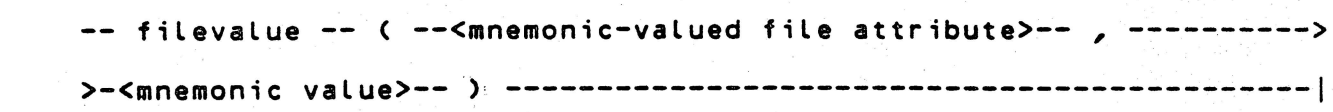
The file to which the function applies may be specified by including a <textfile variable> in the function call. If no file is specified, the function applies to the textfile "input".

If the specified file is not open when the <eoln function> is called, an error occurs. An error also occurs if the <eoln function> is called when the <eof function> would return "true" for that file.

See also
Eof Function. 165

FILEVALUE FUNCTION

<filevalue function>



The <filevalue function> returns the integer value corresponding to the specified <mnemonic value>, which must be a value for the specified <mnemonic-valued file attribute>. The <filevalue function> eliminates the need to permanently embed integer values for mnemonic file attribute values in a program.

The <filevalue function> is a Burroughs extension to ANSI Pascal.

Examples

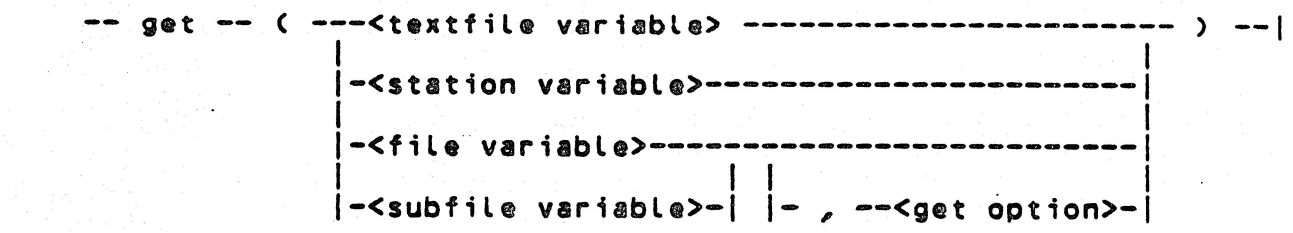
```
var i : integer;
i := filevalue(kind, disk);
i := filevalue(units, characters);
```

The first example returns the integer value associated with the DISK mnemonic of the KIND file attribute and places it into the variable "i".

The next example returns the integer value associated with the CHARACTERS mnemonic of the UNITS file attribute and places it into the variable "i".

GET PROCEDURE

<get procedure>



<get option>



The <get procedure> assigns to the buffer variable of the file denoted by <textfile variable>, <file variable>, <station variable>, or <subfile variable> the value of the component corresponding to the current position of the file. If the file is positioned beyond the last component when the <get procedure> is invoked, the <eof function> becomes "true" and the value of the buffer variable associated with the file becomes undefined.

If a <textfile variable> is specified and the end-of-line marker is reached, the value assigned to the buffer variable is ' ' (blank); at this point, the <eoln function> would return "true". The next call on the <get procedure> will access the first component of the next line or, if there are no more lines, will put the file in end-of-file state.

An error occurs if the file is in generation mode or if the file is not open. If, immediately preceding the invocation of "get", the <eof function> yields the value "true", an error occurs if the <eof function> still yields "true" following the invocation.

A <subfile variable> is meaningful only if the value of the KIND attribute is PORT. If a <subfile variable> is used and the associated <subfile index> is 0, a "nonselective read" (in BNA terminology) is performed; that is, the buffer variable value is assigned from any one of the subfiles. If a <subfile index> greater than 0 and less than or equal to the value of the MAXSUBFILES file attribute is specified, the value is obtained from the specified subfile.

An error occurs if an invalid <subfile index> is specified or if the value of MAXSUBFILES is greater than 1 and a <subfile variable> is not used. In the case of a nonselective read, if an <entire variable> is used to specify the <subfile index>, the variable is updated to contain the index of the subfile from which the value was obtained. The LASTSUBFILE file attribute is updated to this value in all cases, if the operation is performed without error.

A <station variable> is meaningful only if the value of the KIND attribute is REMOTE. If a <station variable> is used and the associated <station index> is 0, a nonselective read is performed. If a <station index> greater than 0 is specified, the data is obtained from the specified station. The LASTSUBFILE file attribute is updated to contain the index of the station from which the data was received. If a <station variable> is not specified, but <file variable> denotes a remote file with multiple stations, the operation behaves as if a <station index> of 0 were specified.

The "dontwait" option is meaningful only if the value of the KIND attribute is PORT. If no data is available, the program is suspended unless the "dontwait" option is specified. If this option is specified and no data is available, control returns to the program, and the operation is not performed. The success or failure of the operation can be determined by examining the associated I/O result. For information concerning I/O results and their use, please refer to the "I/O Exception Handling" section.

The use of <station variable>s, <subfile variable>s, and the "dontwait" option are Burroughs extensions to ANSI Pascal.

See also

I/O Exception Handling. 155

IORES FUNCTION

<iores function>

```
-- iores -- ( --<ioresult mnemonic>-- ) --|
```

<ioresult mnemonic>

One of the <context-sensitive identifier>s defined in the "I/O Exception Handling" section.

The <iores function> returns an integer value that corresponds to the specified <ioresult mnemonic>. The function is completely evaluated at the time of program compilation and behaves as if it were an integer constant. The <iores function> eliminates the need to embed integer values for I/O results in a program. For more information on I/O results and the use of the <iores function>, please refer to the "I/O Exception Handling" section.

The <iores function> is a Burroughs extension to ANSI Pascal.

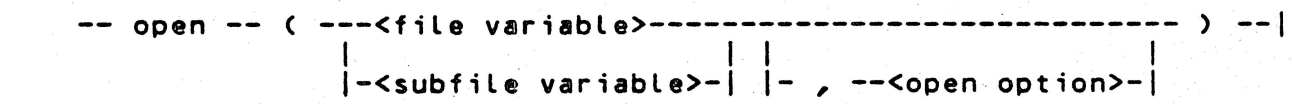
See also
I/O Exception Handling. 155

Example

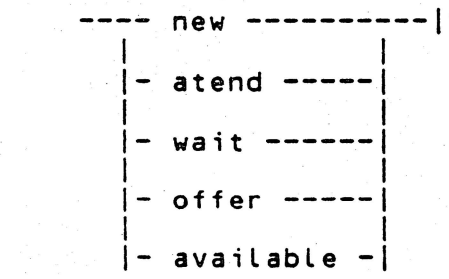
```
type t = packed array[1..80] of char;
var i : integer;
    f : file of t;
begin
open(f);
i := get(f);
case i of
   iores(ok): display('read ok');
   iores(dataerr): display('data error');
   iores(parity): display('parity error');
   iores(eof): display('eof occurred');
   otherwise display('error');
end;
end.
```

OPEN PROCEDURE

<open procedure>



<open option>



The <open procedure> opens the file specified by <file variable> or <subfile variable> in inspection/generation mode. A <get procedure> is not automatically performed, as it is with the <reset procedure>, and the buffer variable remains undefined. (Therefore, it is an error if the first operation on the file is an invocation of any procedure or function that assumes that the buffer variable is defined, such as "read".) An error occurs if a <file variable> is used and the specified file is already open. An error also occurs if a <subfile variable> that specifies a particular subfile is used and that subfile is already open.

Note: Inspection/generation mode is not allowed for textfiles; thus, the <open procedure> cannot be invoked with a <textfile variable> as a parameter. A textfile can be opened in inspection mode by using the <reset procedure> or in generation mode by using the <rewrite procedure>.

A <subfile variable> is valid only if the value of the KIND attribute of the file is PORT. If a <subfile variable> is used and the associated <subfile index> is 0, all closed subfiles are opened. If a <subfile index> greater than 0 and less than or equal to the value of the MAXSUBFILES file attribute is specified, only the specified subfile is opened. An error occurs if an invalid subfile index is specified or if the value of MAXSUBFILES is greater than 1 and a <subfile variable> is not used.

The <open procedure> may be used to open a new file or an existing file. If the "new" <open option> (described below) is specified, a new file will be opened; otherwise, an existing file is assumed.

In the latter case, the <open procedure> invokes the I/O subsystem search logic to find a matching physical file with which to associate the internal Pascal <file variable>. Unless otherwise specified, an attempt is made to locate an existing disk file whose title is given by the first 17 characters (translated to uppercase) of the <file variable> identifier. If a matching file cannot be found, the program is suspended in a system "NO FILE" condition, awaiting an operator response, unless the "available" open-option (described below) is specified.

This search can be modified by changing certain file attributes, such as KIND or TITLE, either through file equation or through Pascal syntax for assigning attributes. For additional information on setting file attributes in Pascal, refer to the Use of File Attributes section of this manual. For a description of the file search logic, please refer to the I/O Subsystem Reference Manual.

The valid <open option>s are defined as follows:

New
: If the "new" option is specified, a new empty file is created. Unless otherwise specified, a disk file with a title given by the first 17 characters (translated to upper case) of the <file variable> is created.

Atend
: If the "atend" option is specified and the file is assigned to a tape or disk file, the file is positioned immediately following the last record of the file. "Atend" has no effect for files that are not tape or disk files.

Wait
: If the "wait" option is specified, control does not return to the program until the file is completely open. "Wait" is the default <open option>.

Offer
: The "offer" option is valid only for port files. When specified, the file or subfile is offered for matching. Control returns to the program without waiting for a matching subfile to be found. The FILESTATE file attribute can be examined to determine whether or not the file is completely open.

Available
: If the "available" option is specified and the attempt to connect the logical and physical file fails, an I/O exception describing the reason for the failure occurs, without

suspending the program. The exception condition can be handled programmatically by examining the associated I/O result. If it is not handled programmatically, the program is terminated. Please refer to the "I/O Exception Handling" section for information concerning I/O results.

If a port file is opened with the "available" option, the file or subfile is matched only to a complementary file or subfile that has already been "offered".

For more information concerning matching of port files and the use of port file contructs, please refer to the I/O Subsystem Reference Manual or the Burroughs Network Architecture (BNA) Architectural Description Reference Manual.

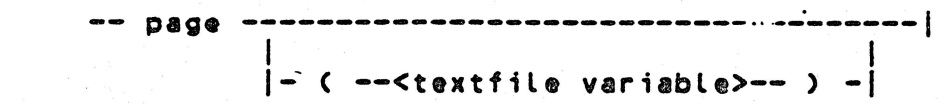
The <open procedure> is a Burroughs extension to ANSI Pascal.

See also

Use of File Attributes. 152
I/O Exception Handling. 155

PAGE PROCEDURE

<page procedure>



The <page procedure> is identical in action to a call on the <skiptochannel procedure> with the channel number specified as 1. That is, it causes a <writeln procedure> without carriage control, followed by a skip-to-top-of-page action. If the <textfile variable> is omitted, the action applies to the textfile "output".

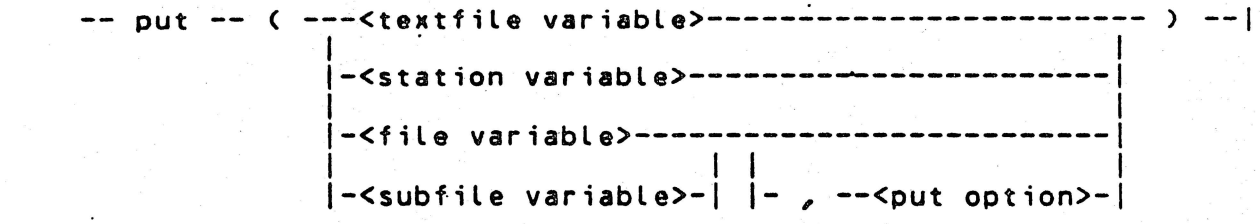
If the <page procedure> is invoked for a file that is not associated with a printer, the effect is equivalent to invoking the <writeln procedure>.

An error occurs if the file is in inspection mode or if the file is not open prior to the execution of the <page procedure>.

See also
Skiptochannel Procedure 188
Writeln Procedure 195

PUT PROCEDURE

<put procedure>



<put option>



The <put procedure> writes to the file denoted by <textfile variable>, <file variable>, <subfile variable>, or <station variable> the value of the buffer variable associated with that file. The value of the buffer variable then becomes undefined.

An error occurs if the file is in inspection mode or if the file is not open prior to execution of the <put procedure>. An error also occurs if a <textfile variable> is specified and the <put procedure> causes the line to exceed the length determined by the value of the MAXRECSIZE file attribute.

A <subfile variable> is meaningful only if the KIND of the file is PORT. If a <subfile variable> is used and the associated <subfile index> is 0, a "broadcast write" (in BNA terminology) is performed; that is, the value of the buffer variable is written to each subfile. If a <subfile index> greater than 0 and less than or equal to the value of the MAXSUBFILES file attribute is specified, the data is written only to the specified subfile. The LASTSUBFILE file attribute is updated to the value of <subfile index> if the operation completes without error. An error occurs if an invalid <subfile index> is specified or if the value of MAXSUBFILES is greater than 1 and a <subfile variable> is not used.

A <station variable> is meaningful only if the KIND of the file is REMOTE. If a <station variable> is used and the associated <station index> is 0, a "broadcast write" is performed. If a <station index> greater than 0 is specified, the data is written only to the specified station. The LASTSUBFILE file attribute will be updated to the value of <station index> when the operation completes. If a <station variable> is not used, but the specified file is a remote file with multiple stations, the write is directed to the station denoted by the current value of the LASTSUBFILE attribute.

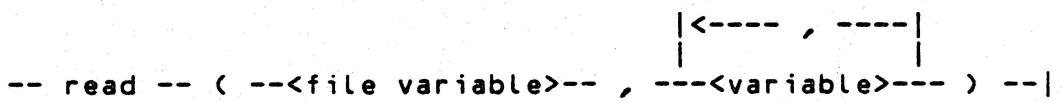
The "dontwait" option is meaningful only if the file is a port file. If the file is a port file and the "dontwait" option is not specified, the program is suspended until buffers are available to perform the write. If the option "dontwait" is specified, the program is not suspended when buffers are not available. Instead, control returns to the program without performing the write. The success or failure of the operation can be determined by examining the associated I/O result. If an exception occurs and the I/O result is not handled programmatically, the program is terminated. Please refer to the "I/O Exception Handling" section for information concerning I/O results.

The use of <subfile variable>s, <station variable>s, and the "dontwait" option are Burroughs extensions to ANSI Pascal.

See also
I/O Exception Handling. 155

READ PROCEDURE

<read procedure>



The <read procedure> causes the specified <variable>s to be assigned sequential values from the file denoted by <file variable>. The action of "read(f,x)" is equivalent to the following statements:

```
x:=f@;    { x is assigned the value of the buffer variable }
get(f);   { f@ is assigned the next value in the file }
```

Thus, the value of the buffer variable (f@) must be assignment compatible with the <variable> being read (x). The action of "read(f,x1,...,xn)" is equivalent to the following statements:

```
read(f,x1);
    .
    .
    .
read(f,xn);
```

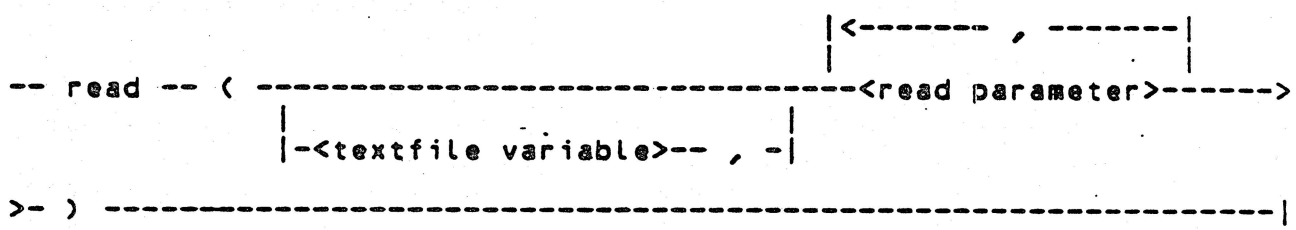
An error occurs if the file is in generation mode, if the file is not open, or if the <eof function> would return "true" prior to the execution of the <read procedure> or any inferred subcomponent of it (such as the "read(f,xn)" statement shown above).

See also

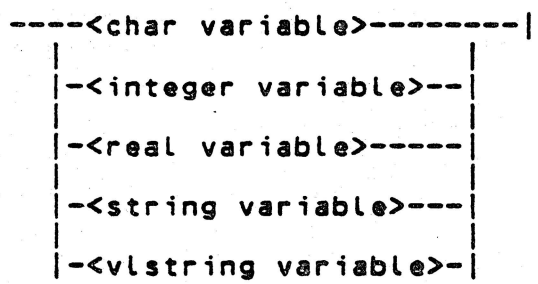
Eof Function. 165
Get Procedure 168

READ TEXTFILE PROCEDURE

<read textfile procedure>



<read parameter>



The <read textfile procedure> is similar to the <read procedure>, except that it applies to textfiles instead of standard files. When the <textfile variable> is not specified, the read is performed on the textfile "input".

The list of <read parameter>s specifies the variables into which the information in the textfile is to be read. As is true of the <read procedure>, reading a list of <read parameter>s is equivalent to reading the variables in successive read statements.

An error occurs if the textfile is in generation mode, if the textfile is not open, or if the <eof function> would return "true" prior to the execution of the <read textfile procedure> or any inferred subcomponent of it.

The action of the <read textfile procedure> depends on the type of the specified <read parameter>:

<char variable>

The action of the <read textfile procedure> with a <char variable> parameter is equivalent to the following two statements, where "c" is the specified <char variable> and "f" is the file to be read:

```
c := f@;
get(f)
```

Example

```
var c1, c2 : char;
    f : text;
begin
read(f,c1,c2);
end;
```

If the textfile contains the characters

```
"defgh"
 *
```

and the buffer variable is at the location indicated by the asterisk, the read procedure assigns the value "d" to variable "c1" and the value "e" to the variable "c2".

<integer variable>

Beginning with the character at the current buffer variable location, characters are scanned, across several lines if necessary, until a nonblank character is encountered. Starting with the first nonblank character, the sequence of nonblank characters is then interpreted as an integer value, which may include a sign. The format of the number must be consistent with the format defined for an <integer constant> appearing in a Pascal program, and the value must be assignment compatible with the type of the parameter.

Following the <read textfile procedure>, the buffer variable is assigned the value of the next character or, if there are no more characters in the line, put in end-of-line state.

Example

```
var i : integer;
    f : text;
begin
read(f,i);
end;
```

If the textfile contains the character sequence

```
"          -123degrees"
 *             *
```

and the buffer variable is positioned at the location indicated by the first asterisk, the read procedure will assign the value "-123" to the variable "i" and will leave the buffer variable positioned at the location indicated by the second asterisk ("d" is not a valid character in an integer).

<real variable>

Beginning with the character at the current buffer variable location, characters are scanned, across several lines if necessary, until a nonblank character is encountered. Starting with the first nonblank character, the sequence of nonblank characters is then interpreted as a real value, which may include a sign and an exponent. The format of the number must be consistent with the format defined for a <real constant> appearing in a Pascal program.

Following the <read textfile procedure>, the buffer variable is assigned the value of the next character or, if there are no more characters in the line, put in end-of-line state.

Example

```
var f : text;
    r : real;
begin
read(f,r);
end;
```

If the textfile contains the character sequence

```
"          98.6degrees"
 *             *
```

and the buffer variable is positioned at the location indicated by the first asterisk, the read procedure will assign the value "98.6" to the variable "r" and will leave the buffer variable positioned at

the location indicated by the second asterisk ("d" is not a valid character in a real value).

If the textfile contains the character sequence

```
"      -1234e-27Mev"
 *              *
```

and the buffer variable is positioned at the location indicated by the first asterisk, the read procedure will assign the value "-1234 times 10 to the power of -27" to the variable "r" and will leave the buffer variable positioned at the location indicated by the second asterisk.

<string variable>

Beginning with the character at the current buffer variable location, characters are assigned into the <string variable>, starting at character position one, until the <string variable> is exhausted or the <eoln function> returns the value "true". If the <eoln function> returns the value "true" before the <string variable> is filled, the remainder of the variable is filled with blanks.

The action of the <read textfile procedure> with a <string variable> parameter is equivalent to the following sequence of statements, where "f" is the file to be read, "i" is an <integer variable>, and "s" is the specified <string variable> that was defined to be a packed array [1..n] of char:

```
i := 1;
while not eoln(f) and (i <= n) do
   begin
   s[i] := f@;
   get(f);
   i := i + 1;
   end;
for i := i to n do
   s[i] := ' ';
```

The reading of a <string variable> from a textfile is a Burroughs extension to ANSI Pascal.

Example

```
var s1, s2 : packed array [1..5] of char;
             f : text;
read(f, s1, s2);
```

If the textfile contains the characters

```
"abcdefgh"
 *
```

and the buffer variable is at the location indicated by the asterisk, the read procedure will assign the value "abcde" to the variable "s1" and the value "fgh " to the variable "s2".

<vlstring variable>

Beginning with the character at the current buffer variable location, characters are assigned into the <vlstring variable>, starting at character position one, until the <vlstring variable> is exhausted or the <eoln function> returns the value "true". The length of the <vlstring variable> is the number of characters that have been assigned into it.

The action of the <read textfile procedure> with a <vlstring variable> parameter is equivalent to the following sequence of statements, where "f" is the file to be read, "i" is an <integer variable>, and "s" is the specified <vlstring variable> that was defined to be string(n):

```
i := 1;
while not eoln(f) and (i <= n) do
   begin
   s := concat(s,f@);
   get(f);
   i := i + 1;
   end;
```

Example

```
var s1, s2 : string(5);
           f : text;
read(f, s1, s2);
```

If the textfile contains the characters

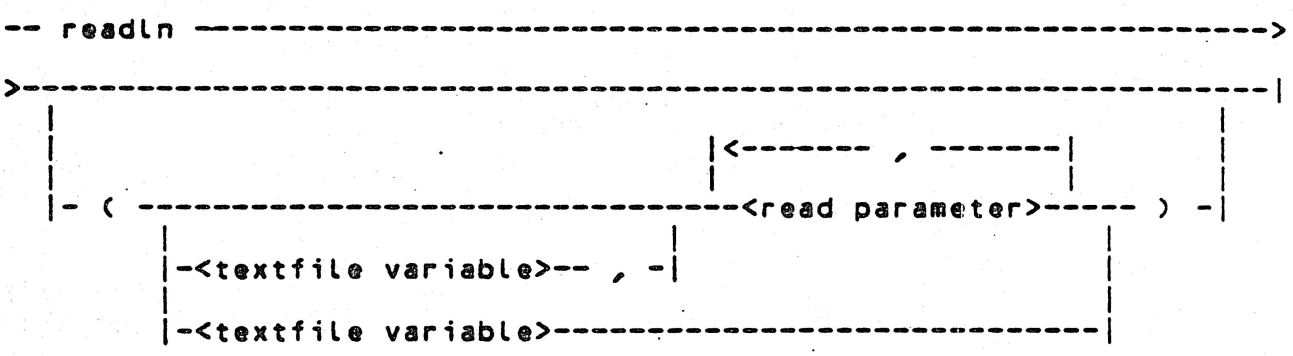
```
"abcdefgh"
 *
```

and the buffer variable is at the location indicated by the asterisk, the read procedure will assign the value "abcde" to the variable "s1", which will have a length of 5. The value "fgh" will be assigned to the variable "s2", which will have a length of 3.

See also
Constant Definitions. 31
Eof Function. 165
Read Procedure. 177
Variables 257

READLN PROCEDURE

<readln procedure>



The <readln procedure> performs the same action as the <read textfile procedure> and then moves the file to the start of the next line. If there is no next line, the file is positioned at end-of-file.

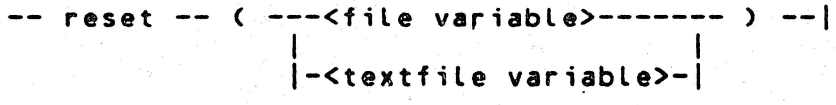
If no <textfile variable> is specified, the <readln procedure> applies to the textfile "input".

An error occurs if the file is in generation mode, if the file is not open, or if the <eof function> would return "true" prior to the execution of the <readln procedure> or any subcomponent of it.

See also
Eof Function. 165
Read Textfile Procedure 178

RESET PROCEDURE

<reset procedure>



The <reset procedure> puts the file denoted by the <file variable> or <textfile variable> in inspection mode. If the file is already open, it is repositioned to the beginning. If the file is closed, it is opened. If the <reset procedure> is applied to a textfile that is currently in generation mode and there is a partially generated line, an automatic <writeln procedure> is performed before the textfile is repositioned.

If the file is not open, the <reset procedure> invokes the I/O subsystem search logic to find a matching physical file with which to associate the internal Pascal <file variable>. Unless otherwise specified, an attempt is made to locate an existing disk file whose title is given by the first 17 characters (translated to uppercase) of the <file variable> or <textfile variable> identifier (if the identifier is the predefined file identifier "input", a remote or card reader file is searched for). This search can be modified by changing certain file attributes, such as KIND or TITLE, either through file equation or through Pascal syntax for assigning attributes. For additional information on setting file attributes in Pascal, refer to the Use of File Attributes section of this manual. For a complete description of the file search logic, please refer to the I/O Subsystem Reference Manual.

When the <reset procedure> is called, an existing file is always assumed. If a matching file cannot be found, the program is suspended in a system "NO FILE" condition, awaiting an operator response. Setting the NEWFILE file attribute has no effect.

Following a <reset procedure>, the file will be in end-of-file state if the file is empty. Otherwise, the buffer variable is defined to have the value of the first component of th file.

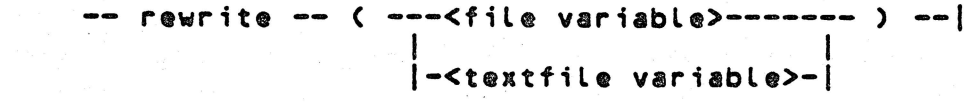
In inspection mode, only input operations can be performed.

See also

Eof Function. 165
Writeln Procedure 195

REWRITE PROCEDURE

<rewrite procedure>



The <rewrite procedure> puts the file denoted by the <file variable> or <textfile variable> into generation mode. If the file is already open, it is discarded, and a new, empty file is created. If the file is closed, a new, empty file is created. Unless otherwise specified, a disk file with a title given by the first 17 characters (translated to uppercase) of the <file variable> or <textfile variable> identifier is created (if the identifier is the predefined file identifier "output", a remote or printer file is created).

Immediately following the invocation of the <rewrite procedure>, the value of the buffer variable is undefined and the <eof function> will return "true". The <eof function> will continue to return "true" as long as the file is in generation mode.

In generation mode, only output operations can be performed.

See also
Eof Function. 165

SEEK PROCEDURE

<seek procedure>

```
-- seek -- ( --<file variable>-- , --<integer expression>-- ) --|
```

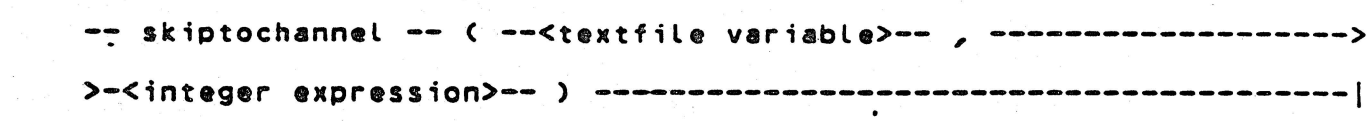
The <seek procedure> positions the file denoted by <file variable> at a specified point in the file. The file is positioned such that the next <get procedure> or <put procedure> is performed on the component specified by the <integer expression>. Components are numbered beginning at 0 (i.e. zero relative). If the value of the specified <integer expression> is less than 0, an error occurs.

Following the execution of the <seek procedure>, the value of the buffer variable is undefined.

The <seek procedure> is a Burroughs extension to ANSI Pascal.

SKIPTOCHANNEL PROCEDURE

<skiptochannel procedure>



The <skiptochannel procedure> performs a <writeln procedure> without carriage control action and then pe forms a skip-to-channel action on the file specified by the <textfile variable>. The channel is specified by the <integer expression>, which must be in the range 1..11.

Following the execution of the <skiptochannel procedure>, the value of the buffer variable is undefined. An error occurs if the file is in inspection mode or if the file is not open when the <skiptochannel procedure> is executed.

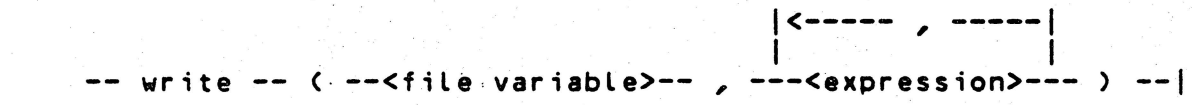
The <skiptochannel procedure> is intended for use with printer files, but may be used with other files, in which case the operation is equivalent to a call on the <writeln procedure>.

The <skiptochannel procedure> is a Burroughs extension to ANSI Pascal.

See also
Integer Expressions 120
Writeln Procedure 195

WRITE PROCEDURE

<write procedure>



The <write procedure> causes the specified <expression>s to be written sequentially to the file denoted by <file variable>. Following the execution of the <write procedure>, the value of the buffer variable is undefined.

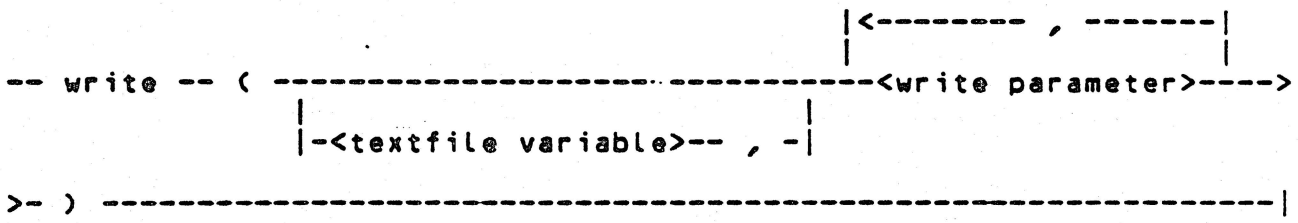
An error occurs if the values of the <expression>s specified in the <write procedure> are not assignment compatible with the file type of the specified <file variable>. An error also occurs if the file is in inspection mode or if the file is not open.

See also

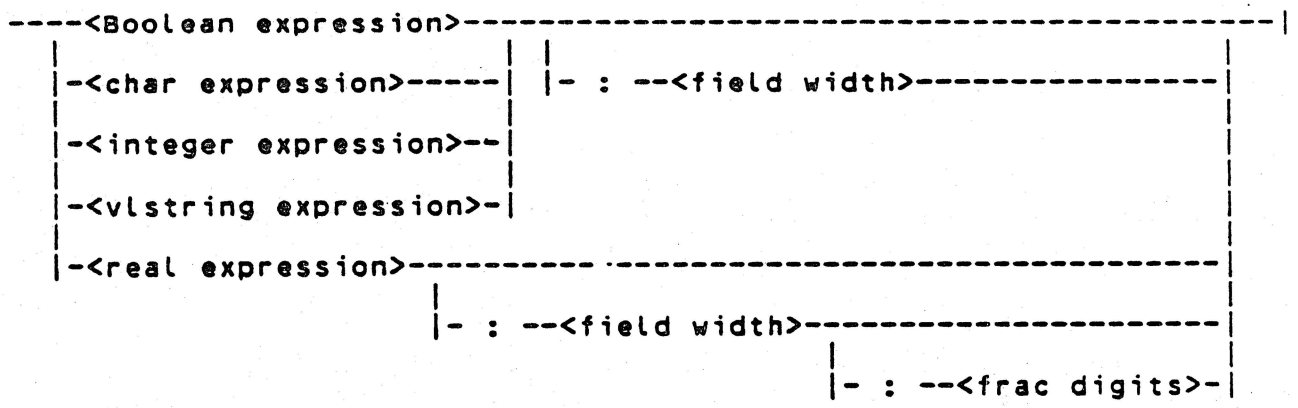
Expressions 103
Eof Function. 165

WRITE TEXTFILE PROCEDURE

<write textfile procedure>



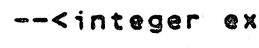
<write parameter>



<field width>



<frac digits>



The <write textfile procedure> is similar to the <write procedure>, except that it applies to textfiles instead of standard files. When the <textfile variable> is not specified, the write is performed to the textfile "output".

An error occurs if the textfile is in inspection mode or if the textfile is not open. Also, an error occurs if the "write" operation causes the length of the current line to exceed the maximum length, which is determined by the value of the MAXRECSIZE file attribute.

The list of <write parameter>s specifies the variables whose values are to be written to the textfile. The <field width> and <frac digits> specifications allow the programmer to control aspects of the formatting of the values written. If these specifications are omitted (where they are allowed), an appropriate representation of the value is chosen by the compiler. If specified, <field width> and <frac digits> must be greater than or equal to one.

The action of the <write textfile procedure> for each type of <write parameter> is described below:

<Boolean expression>

For the values of true and false, the characters strings " TRUE" and "FALSE", respectively, are written. The default <field width> for a <Boolean expression> is five characters. If a <field width> is specified that is smaller than the length of the string to be written, the first <field width> characters are written. If the specified <field width> is larger, leading blanks are written.

Examples

"write(f,b)" produces " TRUE" if "b" is true
and "FALSE" if "b" is false.

"write(f,true:2)" produces "TR".

"write(f,true:10)" produces " TRUE".

<char expression>

For a value of the <char type>, the character is simply moved to the buffer variable and "put" in the file. The default <field width> for a <char expression> is one character. If a <field width> greater than one is specified, leading blanks are written.

Examples

If "c" is a <char variable> that contains the value "$",

"write(f,c)" produces "$" and

"write(f,c:3)" produces " $".

<integer expression>

Values of the <integer type> are formatted with a sign ("-" if the number is negative, blank if the number is positive), followed by the decimal representation of the integer value. The default <field width> for an <integer expression> is ten characters. If a <field width> is specified that is smaller than the length of the number to be written, the <field width> specification is ignored, and the entire number is written. If the specified <field width> is larger, leading blanks are written.

Examples

If "i" is an <integer variable> and its value is "-12345",

"write(f,i)" produces " -12345",

"write(f,i:3)" produces "-12345", and

"write(f,i:12)" produces " -12345".

<vlstring expression>

For a value of a <string type> or <vlstring type>, the characters are simply moved to the buffer variable and "put" in the file one at a time. The default <field width> for a <vlstring expression> is the length of the expression. If a <field width> is specified that is smaller than the length of the string or vlstring, the first <field width> characters are written. If the specified <field width> is larger, leading blanks a.e written.

Examples

If "s" is a <string variable> that contains "abcdefgh",

"write(f,s)" produces "abcdefgh",

"write(f,s:3)" produces "abc", and

"write(f,s:12)" produces " abcdefgh".

<real expression>

Values of the <real type> are written in floating-point or fixed-point format, depending on whether the <frac digits> specification is provided. If it is provided, the number is written in fixed-point format; if it is not, the number is written in floating-point format. The default <field width> for a <real expression> is 15 characters.

In floating-point format, the number will contain the following components:

a) a sign ("-" if the number is negative, blank if it is positive)
b) the first significant digit (or zero, if the number is zero)
c) a decimal point (.)
d) the fractional part (at least one digit)
e) the exponent symbol (E)
f) the sign of the exponent ("+" or "-")
g) two digits of exponent

If the <field width> specified is smaller than the minimum number of characters necessary to represent the number, the <field width> specification is ignored, and the number is written with one digit of fractional part. If the specified <field width> is larger, the number is expanded by adding trailing zeros to the fractional part.

In fixed-point format, the number will contain the following components:

a) a minus sign (-) if the number is negative
b) the integral part of the number (trunc(<real expression>))
c) a decimal point (.)
d) <frac digits> of the fractional part of the number

If a <field width> is specified that is smaller than the minimum number of characters necessary to represent the number in fixed-point format, the <field width> specification is ignored and the entire number is written, including <frac digits> of fractional part. If the specified <field width> is larger, the number is written with leading blanks. If the number of significant digits requested is fewer than the number of significant digits in the system's representation of the number, the number will be rounded at the last digit written.

Examples

"write(f,1.2345:20)" produces " 1.2345000000000E+00".

"write(f,-27.1828e-3:14)" produces "-2.7182800E-02".

"write(f,0.31:3)" produces " 3.1E-01".

"write(f,-96E12:7)" produces "-9.6E+13".

"write(f,0.317269:3)" produces " 3.2E-01".

"write(f,-965E12:7)" produces "-9.7E+14".

"write(f,0.31726e7:7:3)" produces "3172600.000".

"write(f,-965E12:1:7)" produces "-965000000000000.0000000".

"write(f,0.31726e7:13:3)" produces " 3172600.000".

"write(f,-965E-2:12:7)" produces " -9.6500000".

"write(f,3.1776e-1:13:3)" produces " 0.318".

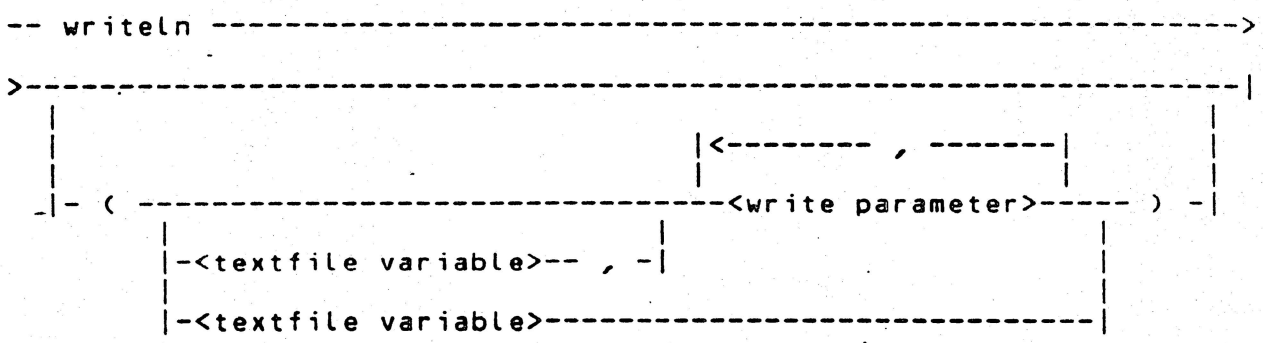
"write(f,-962.5E-2:12:2)" produces " -9.63".

See also

Eof Function 165
Write Procedure 189
Variables 257

WRITELN PROCEDURE

<writeln procedure>



The <writeln procedure> performs the same action as the <write textfile procedure> and then starts a new line. If no <textfile variable> is specified, the <writeln procedure> applies to the textfile "output". If no <write parameter>s are specified, a single blank line is written to the textfile "output". Following the execution of the <writeln procedure>, the value of the buffer variable becomes undefined.

An error occurs if the file is in inspection mode or if the file is not open.

See also

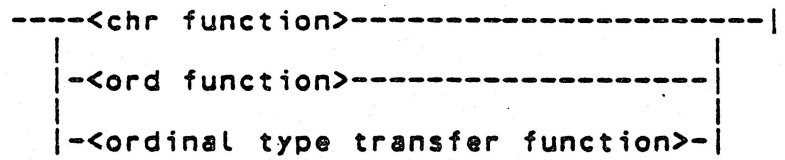
Eof Function. 165
Writeln Procedure 195

6.2 TYPE TRANSFER PROCEDURES AND FUNCTIONS

<type transfer procedure>



<type transfer function>



One of the major reasons for data typing is to allow the compiler to enforce type compatibility restrictions. These restrictions help the programmer ensure that data is handled in a controlled and consistent fashion throughout the program (for example, the compiler will not allow two values of an enumerated type such as "color" to be arithmetically subtracted).

Type transfer functions are provided to allow values of a few data types to be converted to values of certain other data types.

CHR FUNCTION

<chr function>

```
-- chr -- ( --<integer expression>-- ) --|
```

The <chr function> returns the character whose ordinal number is designated by <integer expression>. If the <integer expression> is not a valid ordinal number for the standard character set, an error occurs. Valid ordinal numbers for the EBCDIC character set are in the range 0..255. Valid ordinal numbers for the ASCII character set are in the range 0..127.

Examples

```
var c1, c2 : char;
begin
c1 := chr(129);
c2 := chr(240);
end;
```

If the standard character set is EBCDIC, "c1" is assigned the character "a" and "c2" is assigned the character "0". If the standard character set is ASCII, an error occurs because 129 and 240 are greater than the maximum ordinal number for ASCII.

ORD FUNCTION

<ord function>

```
-- ord -- ( --<ordinal expression>-- ) --|
```

The <ord function> returns, as an integer value, the ordinal number of the specified <ordinal expression>.

Examples

```
var i1, i2 : integer;
begin
i1 := ord('a');
i2 := ord(true);
end;
```

If the standard character set is EBCDIC, "i1" is assigned the integer value 129; if the standard character set is ASCII, "i1" is assigned the integer value 97. "I2" is assigned the integer value 1.

ORDINAL TYPE TRANSFER FUNCTION

<ordinal type transfer function>

--<ordinal type identifier>-- (--<integer expression>--) --|

The <ordinal type transfer function> is actually a set of functions, each of which accepts an <integer expression> and returns the corresponding value of the ordinal type specified by the <ordinal type identifier>. These functions perform the inverse of the <ord function>.

The <ordinal type identifier> may be any declared ordinal type or one of the predefined types "integer", "Boolean", or "char" (the "char" function is identical to the <chr function>). If the <integer expression> is not a valid ordinal number for the specified ordinal type, an error occurs.

The <ordinal type transfer function> is a Burroughs extension to ANSI Pascal.

Examples

```
type color = (red, yellow, blue, green, tartan);
var rslt : Boolean;
    shade : color;
begin
rslt := Boolean(1);
shade := color(3);
end;
```

"Rslt" is assigned the value "true", and "shade" is assigned the color "green".

PACK PROCEDURE

<pack procedure>

```
-- pack -- ( --<unpacked array variable>-- , ---------------------->
>-<ordinal expression>-- , --<packed array variable>-- ) ----------|
```

<unpacked array variable>

```
--<array variable>--|
```

<packed array variable>

```
--<array variable>--|
```

The <pack procedure> transfers data from the specified <unpacked array variable> to the specified <packed array variable>. The unpacked array element whose index is given by the <ordinal expression> is transferred to the low-order element of the packed array. Data transfer continues element by element until the end of the packed array is reached. The arrays must have the same component type.

An error occurs if the value of the <ordinal expression> is not assignment compatible with the <index type> of the <unpacked array variable>.

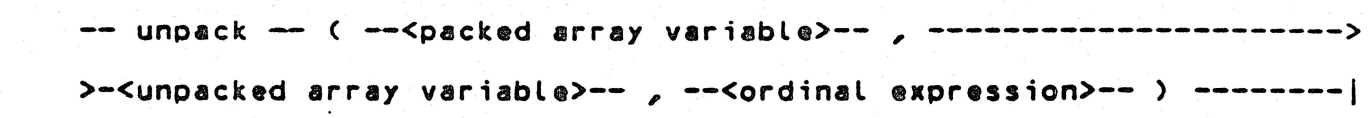
If the end of the unpacked array is reached before the end of the packed array is reached, a run-time error occurs.

Example

```
program pack_example;

  type s1 = 1..100;
       s2 = 1..10;
  var  a : array[s1] of integer;
       z : packed array[s2] of integer;
       u, v, j : s2;
       i : integer;
       k : s1;
  begin

  {Initialize the unpacked array}
  for i := 1 to 100 do
     a[i] := i;

  {The following procedure call will pack into the array "z"
   as many elements as exist in "z" (namely, 10) from the
   unpacked array "a", starting at a[20]}
  i := 20;
  pack(a, i, z);

  {The algorithm below is equivalent to the call on "pack" above. }
  u := 1;
  v := 10;
  k := i;
  for j := u to v do
     begin
     z[j] := a[k];
     if j <> v then
        k := succ(k);
     end;
  end.
```

UNPACK PROCEDURE

<unpack procedure>



The <unpack procedure> transfers data from the specified <packed array variable> to the specified <unpacked array variable>. The low-order element of the packed array is transferred to the element of the unpacked array whose index is given by <ordinal expression>. Data transfer continues element by element until the end of the packed array is reached. The arrays must have the same component type.

An error occurs if the value of the <ordinal expression> is not assignment compatible with the <index type> of the <unpacked array variable>.

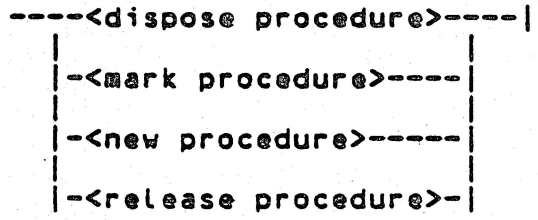
If the end of the unpacked array is reached before the end of the packed array is reached, a run-time error occurs.

Example

```
program unpack_example;

  type s1 = 1..100;
       s2 = 1..10;
  var  a : array [s1] of integer;
       z : packed array [s2] of integer;
       u, v, j : s2;
       i : integer;
       k : s1;
  begin

  {Initialize the packed array}
  for i := 1 to 10 do
     z[i] := i;

  {The following procedure call will unpack all of array "z" into
   elements "a[45]" through "a[54]"}
  i := 45;
  unpack(z, a, i);

  {The algorithm below is equivalent to the call on "unpack" above.}
  u := 1;
  v := 10;
  k := i;
  for j := u to v do
     begin
     a[k] := z[j];
     if j <> v then
        k := succ(k);
     end;
  end.
```

6.3 DYNAMIC ALLOCATION PROCEDURES

<dynamic allocation procedure>



The dynamic allocation procedures, used in conjunction with <pointer variable>s, allow variables to be allocated and deallocated dynamically (that is, independently of the activation of a specific <block>). A variable that is allocated in this way is called a "dynamic variable".

Dynamic variables are allocated in a storage area called the "heap". There are two basic methods of using the heap: 1) as an unordered collection of dynamic variables, which are allocated and deallocated independently of each other, and 2) as a "stack", where the variables are allocated as a sequence and can be deallocated in groups. Creation of dynamic variables and manipulation of the heap is performed through the use of the four predefined procedures "new", "dispose", "mark", and "release".

The "new" procedure is used to allocate a dynamic variable. It accepts a <pointer variable> as a parameter. to which it assigns a "reference value" that can be used to refer to the newly assigned variable. The "new" procedure is the only way to allocate a dynamic variable, and it is used for both the collection and the stack methods of heap management.

The "dispose" procedure deallocates a dynamic variable. The dynamic variable to be deallocated is specified by a <pointer expression> that contains a reference value from the call on the "new" procedure that allocated the variable.

To use the heap as a collection of variables, one simply allocates variables using the "new" procedure and deallocates variables using the "dispose" procedure. Variables are allocated in the heap in a manner designed to make best use of the available space; space that becomes available through the "dispose" procedure may be used when the "new" procedure is called again.

Example

```
program new_dispose;
type ptr_to_node = @node;
     node = record
              name : packed array [1..20] of char;
              next_node : ptr_to_node;
              end;
var person1,
    person2,
    person3 : ptr_to_node;
    head_ptr : ptr_to_node;
begin
   { Build a linked list of three people, where the first person
     is pointed to by head_ptr. }
new(person1);
head_ptr := person1;
new(person2);
person1@.next_node := person2;
new(person3);
person2@.next_node := person3;
person3@.next_node := nil;
   { Delete the node pointed to by person2 and connect the pointers
     to point from person1 to person3. }
person1@.next_node := person2@.next_node;
dispose(person2);
end.
```

This example creates a linked list and deallocates an item in the list. As each dynamic variable is allocated, a copy of its pointer is placed in the previously allocated node. When the list is finished, the "next_node" field of the last item has the value NIL.

In order to delete an entry from the list, the links are changed to exclude this entry, and the dynamic variable is deallocated, making its storage area available for later allocation.

The "mark" and "release" procedures are used to manage the heap as a stack. A stack can be viewed as a time-ordered sequence of variables, where the most recently allocated variables are "on top of" variables allocated earlier. Stack management is particularly useful when the lifetime of a group of variables is identical.

The "mark" procedure stores a reference to the dynamic variable that is the top-of-stack variable at the time the procedure is called. A "mark value" is assigned to the <pointer variable> that is passed as a parameter. This value cannot be used to access the top-of-stack variable; instead, it is used to indicate a position in the stack for later use by the "release" procedure. Once the "mark" procedure has been called, the "new" procedure allocates all new variables such that they are logically "above" the mark in the stack.

The "release" procedure deallocates all variables that were allocated above the mark specified by the <pointer expression> passed as its parameter. The pointer must contain a mark value (that is, a value assigned by the "mark" procedure). The variable that was the top-of-stack variable at the time "mark" was called again becomes the top-of-stack variable.

To maintain the heap as a stack, one typically calls the "mark" procedure, then the "new" procedure one or more times, then the "release" procedure. The "mark" procedure may be called several times before the "release" procedure is finally called. When "release" is called, it deallocates variables down to the mark it is passed as a parameter, regardless of whether or not there exist marks above that one in the stack.

Example

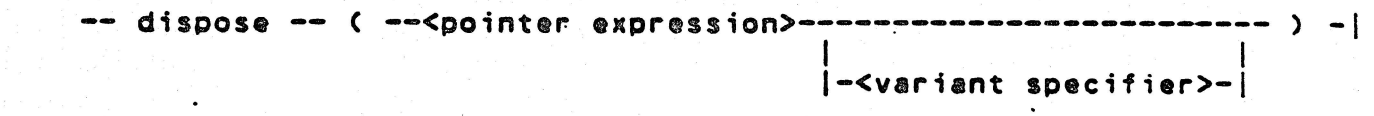
```
program mark_release;

type ptr_to_node = @node;
     node = record
            name : packed array [1..20] of char;
            next_node : ptr_to_node;
            end;
var marker : ptr_to_node;
    person1,
    person2,
    person3 : ptr_to_node;

begin
mark(marker);
new(person1);
new(person2);
new(person3);
release(marker);
end.
```

The call on the <mark procedure> "marks" the heap at the point of the call. After new items have been created in the heap, the call on the <release procedure> causes all three dynamic variables to be deallocated. The three pointers "person1", "person2", and "person3" are undefined after the execution of the <release procedure>.

Dynamic variables can be very useful for certain applications. They can also cause a great deal of programmatic, and programmer, confusion when used incorrectly. In particular, care should be exercised to ensure that the correspondence between pointers and variables is properly maintained. If a variable is deallocated while a pointer to the variable still exists, the pointer becomes a "dangling reference" (a reference to a nonexistent variable). If a variable exists but all references to it have been lost (for example, because a new value was assigned to the only pointer that referenced the variable), the variable is inaccessible and its space is wasted. In ANSI Pascal, the use of a dangling reference in an attempt to access a nonexistent dynamic variable is defined to be invalid, but in this implementation, as in most others, these errors are not always detected.

DISPOSE PROCEDURE

<dispose procedure>



The <dispose procedure> deallocates the dynamic variable identified by the <pointer expression>. If the <pointer expression> has a value of NIL, if the <pointer expression> is undefined, or if the <pointer expression> identifies a marked variable, an error occurs (refer to the <mark procedure>).

The value of the <pointer expression> is undefined after execution of the <dispose procedure>, as are all pointer variables that have the same value as the <pointer expression> (i.e. other pointers referencing the same dynamic variable).

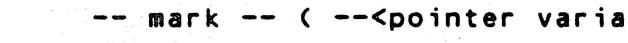
If the variable to be deallocated is a record variable that was created by a call on the <new procedure> with a <variant specifier>, the call on the <dispose procedure> must include the identical <variant specifier> (please refer to the <new procedure> for the definition of <variant specifier>).

See also

Pointer Expressions 122
Mark Procedure. 209
New Procedure 210

MARK PROCEDURE

<mark procedure>



The <mark procedure> assigns to the <pointer variable> a "mark value", a value that corresponds to the location of the most recently allocated dynamic variable (i.e. the current top-of-stack variable). Subsequent calls to the <new procedure> will allocate dynamic variables "above" this mark; such variables are referred to as "marked variables".

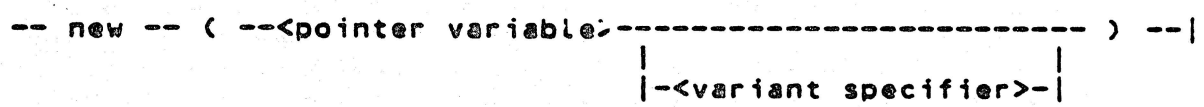
The <pointer variable> can later be used in a call on the <release procedure>, which simultaneously deallocates all variables above the mark. Because the mark value identifies a set of variables rather than a single variable, an error occurs if a variable that contains a mark value is used in any other context (for example, as a reference to a variable).

The <mark procedure> is a Burroughs extension to ANSI Pascal.

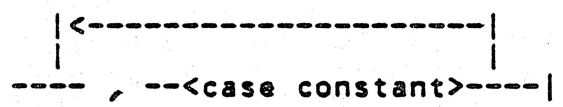
See also
Release Procedure 211

NEW PROCEDURE

<new procedure>



<variant specifier>



The <new procedure> allocates space for a new dynamic variable of the type with which the <pointer variable> is associated. The <pointer variable> then becomes a reference to the location of the new variable.

The <variant specifier> applies only when allocating a record variable with variants. If the <variant specifier> appears, the <case constant>s are interpreted as values for successive <variant selector>s within the record. The <new procedure> allocates sufficient space only for a record of the selected variants. If fewer <case constant>s are specified than there are <variant selector>s in the record, sufficient space is allocated to accommodate the largest possible variant for each remaining <variant selector> in the record.

When the <variant specifier> form is used, the allocated record variable is constrained by the variants selected; the values of the <case constant>s must not be changed for that variable. Also, only fields within the record variable, as opposed to the entire record variable, can be referenced (for example, if P is a pointer to the dynamic record variable, "P@:=x" is illegal because "P@" references the entire variable, but "P@.f:=y" is legal if "f" is a field within the record). These restrictions do not apply when the <variant specifier> form is not used.

RELEASE PROCEDURE

<release procedure>

```
-- release -- ( --<pointer expression>-- ) --|
```

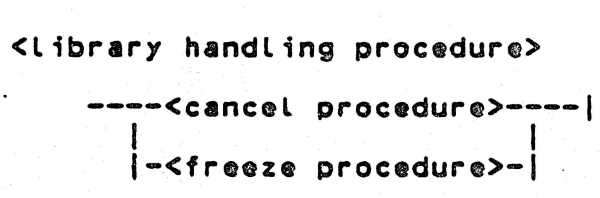
The <release procedure> deallocates the marked variables denoted by the <pointer expression>. An error occurs if the <pointer expression> does not contain a mark value (refer to the Mark Procedure).

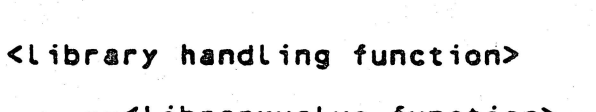
Following the execution of the <release procedure>, all pointer variables and functions that reference the variables that have been deallocated become undefined.

The <release procedure> is a Burroughs extension to ANSI Pascal.

See also
Pointer Expressions 122
Mark Procedure. 209

6.4 LIBRARY HANDLING PROCEDURES AND FUNCTIONS





The library handling procedures and functions provide various additional features used by libraries and the programs that reference them. The following example illustrates the use of these procedures and functions.

Example

```
program call_lib;
   type str     = packed array [1..25] of char;
        vector = array [1..20] of integer;
   var libtitle : str;
       sine, angle : real;
       m, n : integer;
       vector1, vector2 : vector;
   library mylib (title = 'OBJECT/ARITHLIB');
   procedure vector_sum (vect1, vect2 : vector;
                         var vectorresult : vector); mylib;
   function fact (n : integer) : integer; mylib;
   function sin (angle : real) : real; mylib;

begin
n := libraryvalue (libaccess, bytitle);
   { returns the integer value of "bytitle" }
getattribute (mylib, libaccess, m);
   { assigns the value of "libaccess" for "mylib" to "m" }
if n <> m then
   setattribute (mylib, libaccess, n);
n := fact(5); { links the program to the library "mylib" }
cancel (mylib); { delinks the program from the library "mylib" }
setattribute (mylib, title, 'OBJECT/TRIGLIB');
sine := sin(angle);
   { relinks the program to the library "mylib", which is now
     a different code file because the "setattribute" call above
     changed the "title" attribute of the library }
end.
```

CANCEL PROCEDURE

<cancel procedure>

```
-- cancel -- ( --<library identifier>-- ) --|
```

The effect of this procedure is to sever the linkage between the program and the library denoted by <library identifier>. Thereafter, if an entry point from the library is invoked, the program is linked to the library based on the current values of the library attributes, by the same process as the original linkage was made, as described in the "Library Declarations" section.

The <cancel procedure> is a Burroughs extension to ANSI Pascal.

See also

Library Declarations. 68

FREEZE PROCEDURE

<freeze procedure>

-- freeze --|

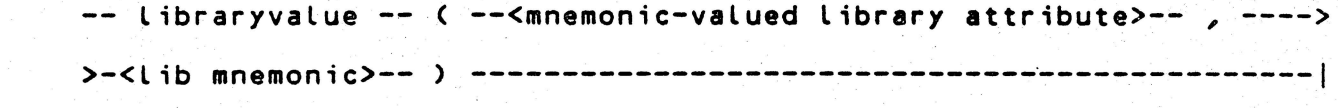
A <freeze procedure> may appear only in a <library>; it is an error if it is used in a <program>. The execution of the <freeze procedure> suspends the execution of the library and causes the entry points of the library to be made available for import by other programs. Those procedures and functions affected by the <freeze procedure> are the ones that are exported via the <interface part> of the library.

If a <freeze procedure> does not appear in the body of a <library>, an implicit freeze will be done at the end of the outermost block.

The <freeze procedure> is a Burroughs extension to ANSI Pascal.

LIBRARYVALUE FUNCTION

<libraryvalue function>

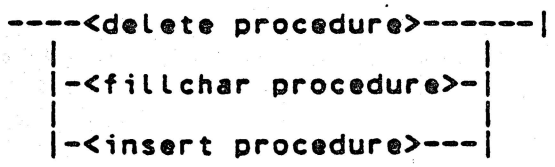


The <libraryvalue function> returns the integer value corresponding to the mnemonic value of the mnemonic-valued library attribute.

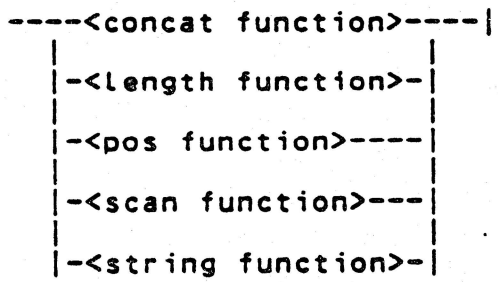
The <libraryvalue function> is a Burroughs extension to ANSI Pascal.

6.5 STRING HANDLING PROCEDURES AND FUNCTIONS

<string handling procedure>



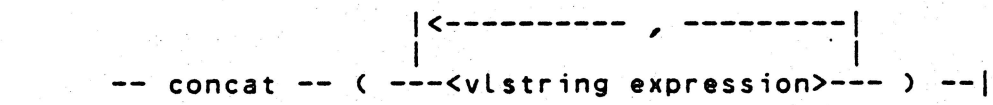
<string handling function>



The string handling procedures and functions provide concatenation, scanning, and various other capabilities for manipulating string variables and vlstring variables.

CONCAT FUNCTION

<concat function>



The <concat function> returns, as a value of type vlstring, the concatenation of all of the <vlstring expression>s in the order they occur in the parameter list.

The <concat function> is a Burroughs extension to ANSI Pascal.

Example

```
var s1, s2 : string(20);
s1 := 'there';
s2 := 'hello';
s1 := concat (s2, ' ', s1);
```

The <vlstring variable> "s1" is assigned the value "hello there".

DELETE PROCEDURE

<delete procedure>

```
-- delete -- ( --<vlstring variable>-- , --<delete index>-- , ----->
>-<delete length>-- ) ------------------------------------------|
```

<delete index>

```
--<integer expression>--|
```

<delete length>

```
--<integer expression>--|
```

The <delete procedure> deletes <delete length> characters from <vlstring variable> starting at the <delete index>th element.

The <delete procedure> is a Burroughs extension to ANSI Pascal.

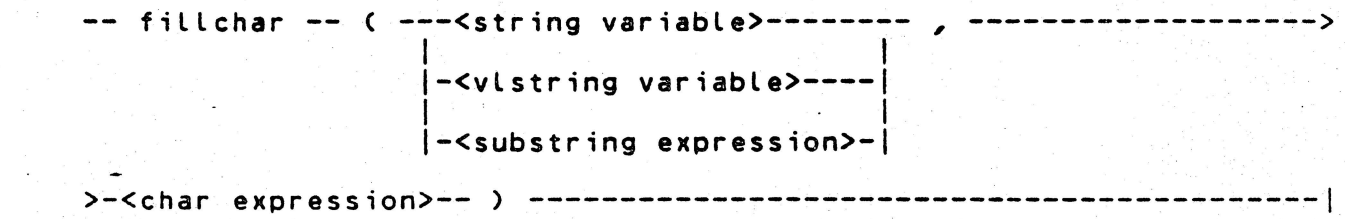
Example

```
var s : string(80);
s := 'This is an extremely difficult task.';
delete (s,10,11);
```

The vlstring variable "s" is assigned the string "This is an extremely difficult task." and has a length of 36. Eleven characters are then deleted from "s", starting at position 10. "S" now contains the string "This is a difficult task." and has a length of 25.

FILLCHAR PROCEDURE

<fillchar procedure>



The <fillchar procedure> assigns the value of the <char expression> to the <string variable>, <vlstring variable>, or <substring expression>. The <char expression> is assigned to successive characters of the string or vlstring, starting at the first element and continuing for the length of the <string variable> or the maximum length of the <vlstring variable>. When a <substring expression> is used, the <char expression> is assigned to the elements specified by the <substring designator>. An error occurs if the <substring expression> begins beyond the current length of the <vlstring variable>.

The <fillchar procedure> is a Burroughs extension to ANSI Pascal.

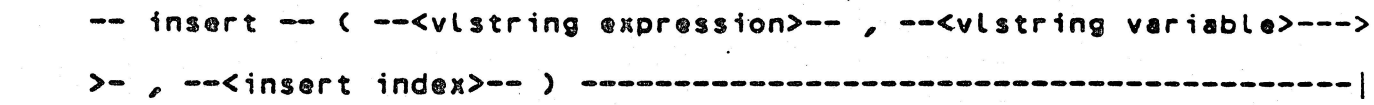
Example

```
const blank = ' ';
var s : string(20);
    i : integer;

{ The following procedure call will assign "blank" to the first
  five elements of the string "s" ("s[1]", "s[2]", "s[3]", "s[4]",
  and "s[5]"). }
fillchar(s[1..5], blank);

{ The following algorithm is equivalent to the call on "fillchar"
  above. }
for i := 1 to 5 do
   s[i] := blank;
```

The length of "s" is 5.

INSERT PROCEDURE

<insert procedure>



<insert index>



The <insert procedure> inserts the <vlstring expression> into the <vlstring variable> starting at the <insert index>th position of the <vlstring variable>. It is an error if the <insert index> is more than 1 greater than the length of the vlstring or if an attempt is made to insert a <vlstring expression> that would cause the length of the <vlstring variable> to be greater than its <maximum length>.

The <insert procedure> is a Burroughs extension to ANSI Pascal.

Example

```
var s : string(10);
s := '1256';
insert('34', s, 3);
```

"S" is assigned the value "1256" and has a length of 4. The <insert procedure> inserts the string "34" at position 3 of "s". "S" now contains the string "123456" and has a length of 6.

LENGTH FUNCTION

<length function>

```
-- length -- ( --<vlstring expression>-- ) --|
```

The <length function> returns, as a value of type integer, the number of characters in the <vlstring expression>.

The <length function> is a Burroughs extension to ANSI Pascal.

Examples

```
var s : string(10);
    a : packed array [1..10] of char;
    i1, i2 : integer;
s := 'abc';
a := s;
i1 := length(s);
i2 := length(a);
```

"I1" is assigned the value 3 and "i2" is assigned the value 10 (because "a" is a fixed-length string, it is always of length 10).

POS FUNCTION

<pos function>

```
-- pos -- ( --<pattern>-- , --<pos source>-- ) --|
```

<pattern>

```
--<vlstring expression>--|
```

<pos source>

```
--<vlstring expression>--|
```

The <pos function> scans <pos source> for the first occurrence of <pattern> and returns, as a value of type integer, the character position of the first character of the <pattern>. The character positions are numbered from 1 to n, where n is the length of the <pos source>. The function returns 0 if the <pattern> does not occur in the <pos source>; it returns 1 if the <pattern> is a null string.

The <pos function> is a Burroughs extension to ANSI Pascal.

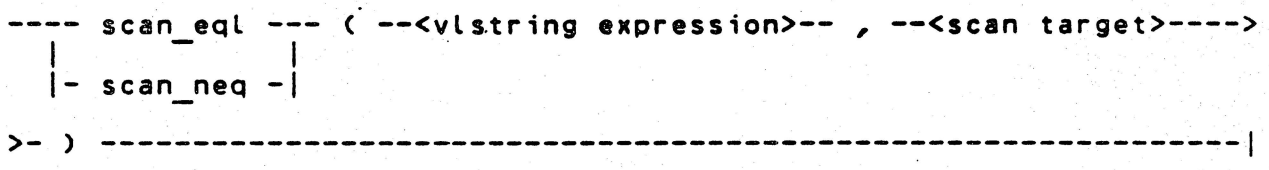
Examples

```
var s : string(10);
    i : integer;
s := 'ababc';
i := pos('bc', s);
s := 'abcdede';
i := pos('f', s);
```

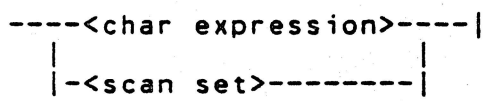
In the first example, the variable "i" is assigned the value 4, the starting location of the first occurrence of "bc" in "s". In the next example, "i" is assigned the value 0 because the character "f" does not occur in "s".

SCAN FUNCTION

<scan function>



<scan target>



<scan set>

A <set constructor> whose members are constants of type char.

The <scan function> scans the <vlstring expression> searching for characters that are in or not in the <scan target> and returns, as a value of type integer, the number of characters skipped.

"Scan_eql" will search the <vlstring expression> for the first occurrence of any of the characters in the <scan target>. "Scan_neq" will search <vlstring expression> for the first character that is not in the <scan target>.

If all the characters in the <vlstring expression> are scanned, the number of characters in the expression is returned. It is an error if the <scan target> is the empty set.

The <scan function> is a Burroughs extension to ANSI Pascal.

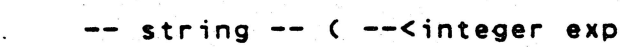
Example

```
program scan_example (output);
const blank = ' ';
var s : string(80);
    inx, skipped, len : integer;
  begin
  s := '   find all    the  words          ';
  len := length(s);
  inx := 1;

  {This will find the words in th. variable s.}
  while inx < len do
     begin
     skipped := scan_neq(s[inx..len], blank);
     inx := inx + skipped;
     if inx < len then
        begin
        skipped := scan_eql(s[inx..len], blank);
        writeln(s[inx:skipped]);
        inx := inx + skipped;
        end;
     end;
  end.
```

This program writes the words in the <vlstring variable> "s" to the file "output".

STRING FUNCTION

<string function>



The <string function> returns, as a value of type vlstring, the decimal representation of <integer expression>. If the <integer expression> is negative, a minus sign (-) will be included immediately to the left of the decimal representation of the number.

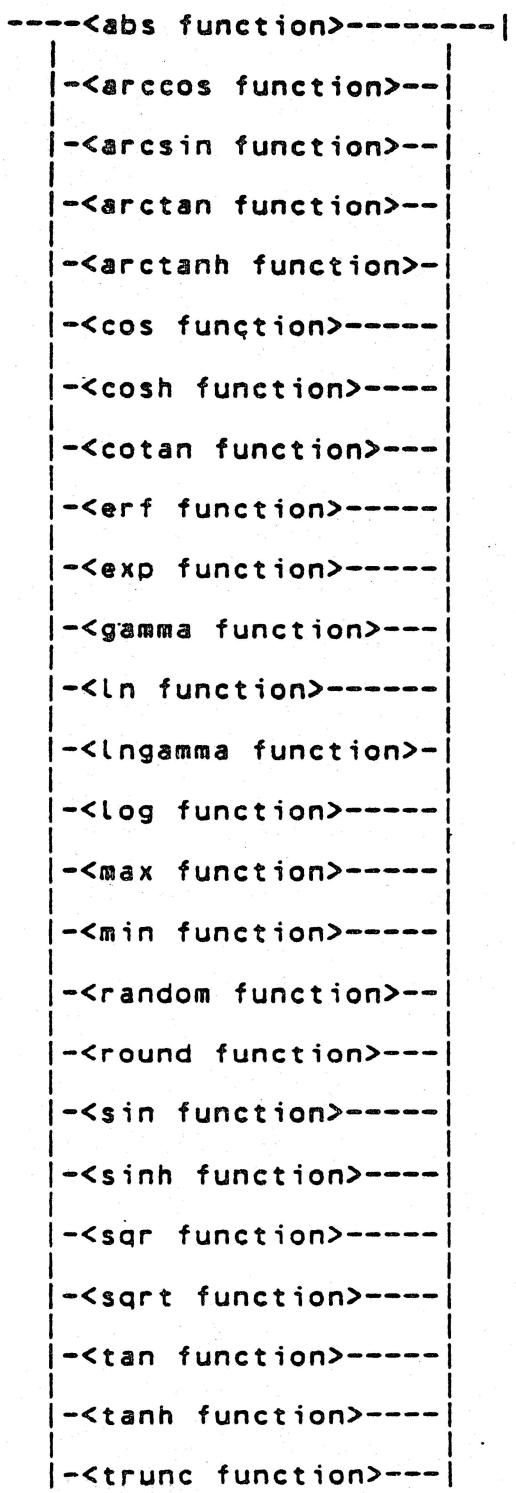
The <string function> is a Burroughs extension to ANSI Pascal.

Example

```
var s : string(10);
    i : integer;
i := 5;
s := string(i+34);
```

The <vlstring variable> "s" is assigned the value "39". The length of "s" is 2.

6.6 ARITHMETIC FUNCTIONS



The <arithmetic function>s provide trigonometric functions, pseudo-random numbers, minimum and maximum functions, and other functions useful in <arithmetic expression>s.

ABS FUNCTION

<abs function>

```
-- abs -- ( --<arithmetic expression>-- ) --|
```

The <abs function> returns the absolute value of the specified <arithmetic expression>. The result returned is of the same type as the specified <arithmetic expression>.

ARCCOS FUNCTION

<arccos function>

```
-- arccos -- ( --<arithmetic expression>-- ) --|
```

The <arccos function> returns, as a real value in radians, the principal value of the arccosine function at the specified <arithmetic expression>. The <arithmetic expression> must be in the range -1 to 1, inclusive.

The <arccos function> is a Burroughs extension to ANSI Pascal.

ARCSIN FUNCTION

<arcsin function>

```
-- arcsin -- ( --<arithmetic expression>-- ) --|
```

The <arcsin function> returns, as a real value in radians, the principal value of the arcsine function at the specified <arithmetic expression>. The <arithmetic expression> must be in the range -1 to 1, inclusive.

The <arcsin function> is a Burroughs extension to ANSI Pascal.

ARCTAN FUNCTION

<arctan function>

```
-- arctan -- ( --<arithmetic expression>-- ) --|
```

The <arctan function> returns, as a real value in radians, the principal value of the arctangent function at the specified <arithmetic expression>.

ARCTANH FUNCTION

<arctanh function>

```
-- arctanh -- ( --<arithmetic expression>-- ) --|
```

The <arctanh function> returns, as a real value, the hyperbolic arctangent of the specified <arithmetic expression>.

The <arctanh function> is a Burroughs extension to ANSI Pascal.

COS FUNCTION

<cos function>

```
-- cos -- ( --<arithmetic expression>-- ) --|
```

The <cos function> returns, as a real value, the cosine of the angle specified by the <arithmetic expression>, which is assumed to be in radians.

COSH FUNCTION

<cosh function>

```
-- cosh -- ( --<arithmetic expression>-- ) --|
```

The <cosh function> returns, as a real value, the hyperbolic cosine of the value specified by the <arithmetic expression>.

The <cosh function> is a Burroughs extension to ANSI Pascal.

COTAN FUNCTION

<cotan function>

```
-- cotan -- ( --<arithmetic expression>-- ) --|
```

The <cotan function> returns, as a real value, the cotangent of the angle specified by the <arithmetic expression>, which is assumed to be in radians.

The <cotan function> is a Burroughs extension to ANSI Pascal.

ERF FUNCTION

<erf function>

```
-- erf -- ( --<arithmetic expression>-- ) --|
```

The <erf function> returns, as a real value, the value of the standard error function at the specified <arithmetic expression>.

The <erf function> is a Burroughs extension to ANSI Pascal.

EXP FUNCTION

<exp function>

```
-- exp -- ( --<arithmetic expression>-- ) --|
```

The <exp function> returns, as a real value, "e" (the base of the natural logarithms) raised to the <arithmetic expression> power.

GAMMA FUNCTION

<gamma function>

```
-- gamma -- ( --<arithmetic expression>-- ) --|
```

The <gamma function> returns, as a real value, the value of the gamma function at the specified <arithmetic expression>.

The <gamma function> is a Burroughs extension to ANSI Pascal.

LN FUNCTION

<ln function>

```
-- ln -- ( --<arithmetic expression>-- ) --|
```

The <ln function> returns, as a real value, the natural logarithm of the specified <arithmetic expression>.

LNGAMMA FUNCTION

<lngamma function>

```
-- lngamma -- ( --<arithmetic expression>-- ) --|
```

The <lngamma function> returns, as a real value, the natural logarithm of the gamma function at the specified <arithmetic expression>.

The <lngamma function> is a Burroughs extension to ANSI Pascal.

LOG FUNCTION

<log function>

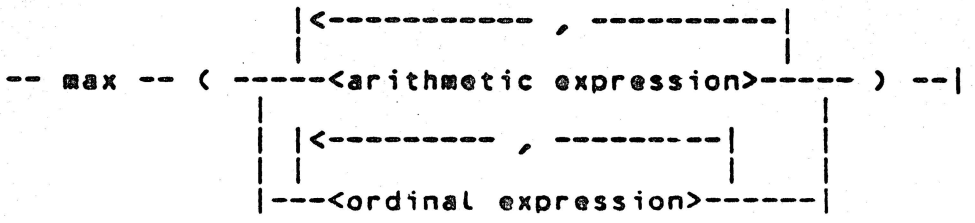
```
-- log -- ( --<arithmetic expression>-- ) --|
```

The <log function> returns, as a real value, the base 10 logarithm of the <arithmetic expression>.

The <log function> is a Burroughs extension to ANSI Pascal.

MAX FUNCTION

<max function>



The <max function> returns the value of the <arithmetic expression> or <ordinal expression> that has the largest arithmetic or ordinal value of those in the list. Each expression must be of a <simple type>, and all expressions must be of compatible types, except that expressions of type real and of type integer may be mixed.

If any expression in the parameter list is of type real, the result returned will be of type real. Otherwise, the result returned will be of the same type as the type (or, for subranges, the host type) of the parameters.

The <max function> is a Burroughs extension to ANSI Pascal.

Examples

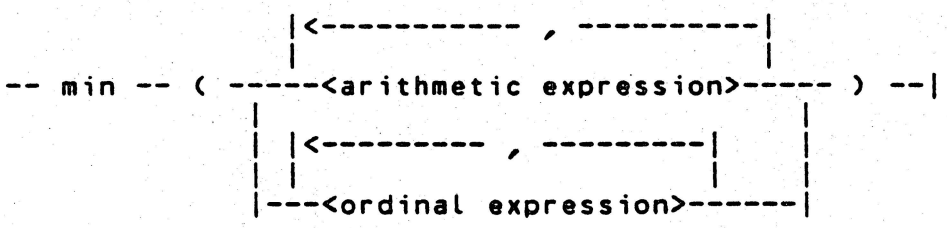
```
var c : (red, yellow, blue, green, tartan);
    i, j, k : integer;
begin
c := max(red, yellow, green);
i := max(i, j, k);
end;
```

In the first example, the value "green" will be returned and placed in the variable "c".

In the second example, the greatest value of the variables "i", "j", and "k" will be returned and placed in the variable "i".

MIN FUNCTION

<min function>



The <min function> returns the value of the <arithmetic expression> or <ordinal expression> that has the smallest arithmetic or ordinal value of those in the list. Each expression must be of a <simple type>, and all expressions must be of compatible types, except that expressions of type real and of type integer may be mixed.

If any expression in the parameter list is of type real, the result returned will be of type real. Otherwise, the result returned will be of the same type as the type (or, for subranges, the host type) of the parameters.

The <min function> is a Burroughs extension to ANSI Pascal.

Examples

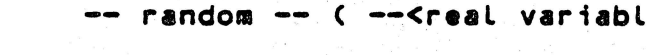
```
var c : (red, yellow, blue, green, tartan);
    i, j, k : integer;
begin
c := min(green, red, yellow);
i := min(i, j, k);
end;
```

In the first example, the value "red" will be returned and placed in the variable "c".

In the second example, the smallest value of the variables "i", "j", and "k" will be returned and placed in the variable "i".

RANDOM FUNCTION

<random function>



The <random function> returns, as a real value, a pseudo-random number that is greater than or equal to 0 and less than 1. The <real variable> is a seed that is used to generate the pseudo-random number. The <random function> assigns a new value to the <real variable>, so successive calls to the <random function> will generate different pseudo-random numbers. The function will generate the same series of pseudo-random numbers if started with the same seed value.

The <random function> is a Burroughs extension to ANSI Pascal.

Example

```
var seed,
    r    : real;
    i    : integer;
begin
{Initialize the random sequence}
seed := 0.5;
for i := 1 to 100 do
   begin
   r := random(seed);
   writeln(r);
   end;
end;
```

ROUND FUNCTION

<round function>

```
-- round -- ( --<real expression>-- ) --|
```

The <round function> returns the nearest integer value to the specified <real expression>. If the value of the <real expression> is positive or zero, the result of the <round function> is equivalent to the value of "trunc(<real expression>+0.5)". If the value of the <real expression> is negative, the result of the <round function> is equivalent to the value of "trunc(<real expression>-0.5)".

It is an error if the nearest integer to the <real expression> is greater than "maxint" or less than "-maxint".

Examples

round(3.5) yields the value 4

round(-3.5) yields the value -4

SIN FUNCTION

<sin function>

```
-- sin -- ( --<arithmetic expression>-- ) --|
```

The <sin function> returns, as a real value, the sine of the angle specified by the <arithmetic expression>, which is assumed to be in radians.

SINH FUNCTION

<sinh function>

```
-- sinh -- ( --<arithmetic expression>-- ) --|
```

The <sinh function> returns, as a real value, the hyperbolic sine of the value specified by the <arithmetic expression>.

The <sinh function> is a Burroughs extension to ANSI Pascal.

SQR FUNCTION

<sqr function>

```
-- sqr -- ( --<arithmetic expression>-- ) --|
```

The <sqr function> returns the square of the value of the specified <arithmetic expression>. The result returned is of the same type as the <arithmetic expression>.

If the result value is out of range for its type, an error occurs.

SQRT FUNCTION

<sqrt function>

```
-- sqrt -- ( --<arithmetic expression>-- ) --|
```

The <sqrt function> returns, as a real value, the square root of the value of the specified <arithmetic expression>. The <arithmetic expression> must be greater than or equal to 0.

TAN FUNCTION

<tan function>

```
-- tan -- ( --<arithmetic expression>-- ) --|
```

The <tan function> returns, as a real value, the tangent of the angle specified by the <arithmetic expression>, which is assumed to be in radians.

The <tan function> is a Burroughs extension to ANSI Pascal.

TANH FUNCTION

<tanh function>

```
-- tanh -- ( --<arithmetic expression>-- ) --|
```

The <tanh function> returns, as a real value, the hyperbolic tangent of the value specified by the <arithmetic expression>.

The <tanh function> is a Burroughs extension to ANSI Pascal.

TRUNC FUNCTION

<trunc function>

```
-- trunc -- ( --<real expression>-- ) --| .
```

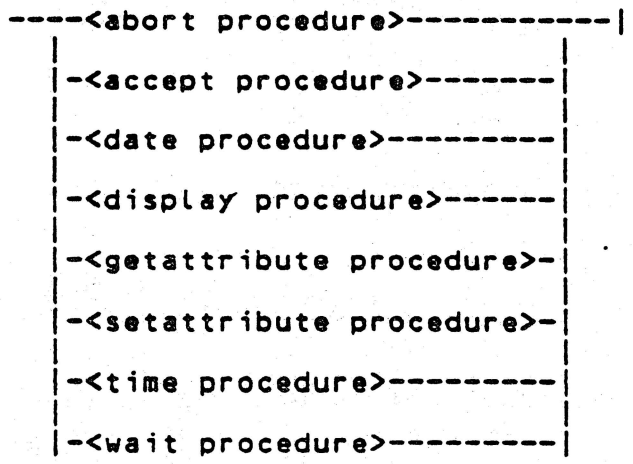
The <trunc function> returns the integer value, computed by truncation, of the specified <real expression>. If the result is greater than "maxint" or less than "-maxint", an error occurs.

Examples

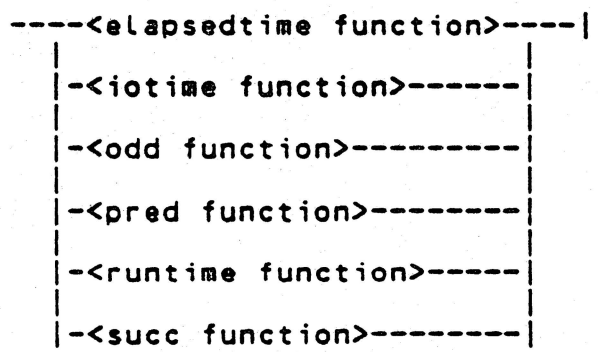
```
trunc(3.5)   yields the value   3

trunc(-3.5)  yields the value  -3
```

6.7 GENERAL PROCEDURES AND FUNCTIONS

<general procedure>



<general function>



Many general procedures and functions are extensions to ANSI Pascal to allow the program to access system-specific features, such as file attributes, the program's accumulated run time, I/O time, and elapsed time, the interface to the Operator Display Terminal (ODT), and the system's time and date values. Other general procedures and functions are part of ANSI Pascal and provide features that are not described elsewhere in this manual.

ABORT PROCEDURE

<abort procedure>

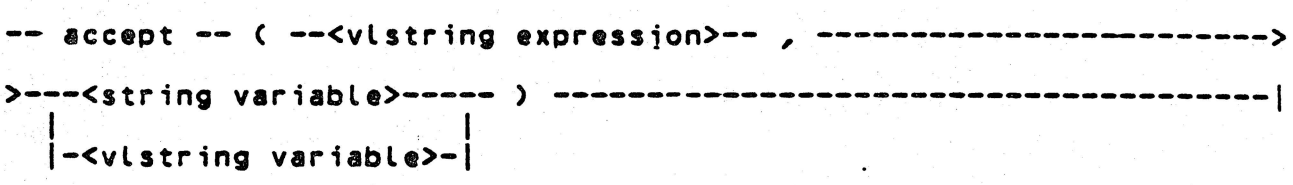


The <abort procedure> forces an immediate, abnormal termination of the program.

The <abort procedure> is a Burroughs extension to ANSI Pascal.

ACCEPT PROCEDURE

<accept procedure>



The <accept procedure> displays the contents of the <vlstring expression> on the Operator Display Terminal (ODT), suspends the program until a response from the operator is entered (through the "AX" ODT command), and then places the operator's response into the <string variable> or <vlstring variable>. If the <vlstring expression> to be displayed contains a NUL character ("chr(0)"), only the characters preceding the NUL character are displayed; otherwise, the entire <vlstring expression> is displayed.

The length of the <vlstring expression> to be displayed and the length of a <string variable> used to receive the response must be less than or equal to 430 characters. If the operator's response is longer than the <string variable> or longer than the maximum length of the <vlstring variable>, an error occurs. If a <string variable> that is longer than the operator's response is used, the response is blank filled on the right. A <vlstring variable> is not blank filled.

If the STRINGS compiler control option has been set to ASCII, the internal representation of the <vlstring expression> is assumed to be ASCII. Thus, before the <vlstring expression> is displayed on the ODT, it is translated to EBCDIC. The response from the ODT is then translated to ASCII before it is stored in the <string variable> or <vlstring variable>.

The <accept procedure> is a Burroughs extension to ANSI Pascal.

Example

```
var str : packed array [1..3] of char;
begin
accept('Do you want to continue? (yes or no)',str);
end;
```

The string "Do you want to continue? (yes or no)" is displayed on the ODT. The response will be placed in "str".

DATE PROCEDURE

<date procedure>

```
-- date -- ( --<year>-- , --<month>-- , --<day>-- ) --|
```

<year>

```
--<variable>--|
```

<month>

```
--<variable>--|
```

<day>

```
--<variable>--|
```

The <date procedure> returns the current date in the parameters <year>, <month>, and <day>. The values returned will be of <integer type> and will be in the following ranges:

parameter	range
<year>	0..9999
<month>	1..12
<day>	1..31

The types of the <variable>s must be such that the values returned will be assignment compatible with those <variable>s.

The <date procedure> is a Burroughs extension to ANSI Pascal.

Example

```
var year  : real;
    month : integer;
    day   : 1..31;
begin
date (year, month, day);
end;
```

The year is placed in the variable "year", the month is placed in the variable "month", and the day of the month is placed in the variable "day".

DISPLAY PROCEDURE

<display procedure>

```
-- display -- ( --<vlstring expression>-- ) --|
```

The <display procedure> displays the contents of the <vlstring expression> on the Operator Display Terminal (ODT). If the <vlstring expression> contains a NUL character ("chr(0)"), only the characters preceding the NUL character are displayed; otherwise, the entire <vlstring expression> is displayed. The length of the <vlstring expression> must be less than or equal to 430 characters.

If the STRINGS compiler control option has been set to ASCII, the internal representation of the <vlstring expression> is assumed to be ASCII. Thus, before the <vlstring expression> is displayed on the ODT, it is translated to EBCDIC.

The <display procedure> is a Burroughs extension to ANSI Pascal.

ELAPSEDTIME FUNCTION

<elapsedtime function>

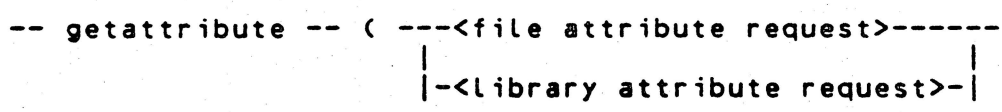
```
-- elapsedtime --|
```

The <elapsedtime function> returns, as a real value in units of seconds, the total time that has elapsed since the program was initiated.

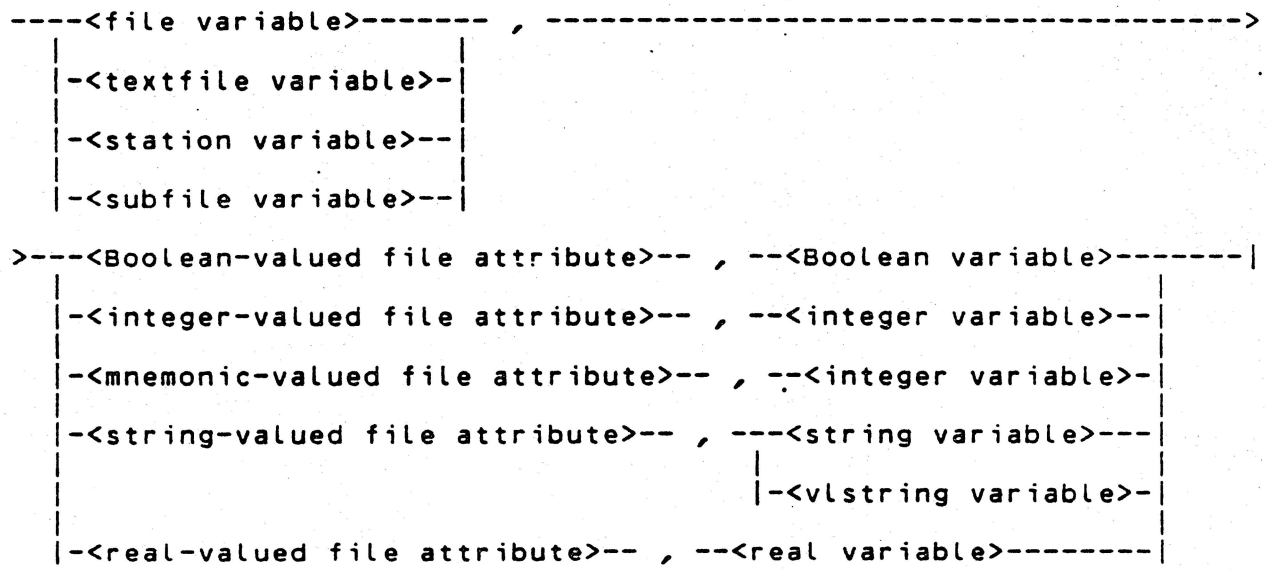
The <elapsedtime function> is a Burroughs extension to ANSI Pascal.

GETATTRIBUTE PROCEDURE

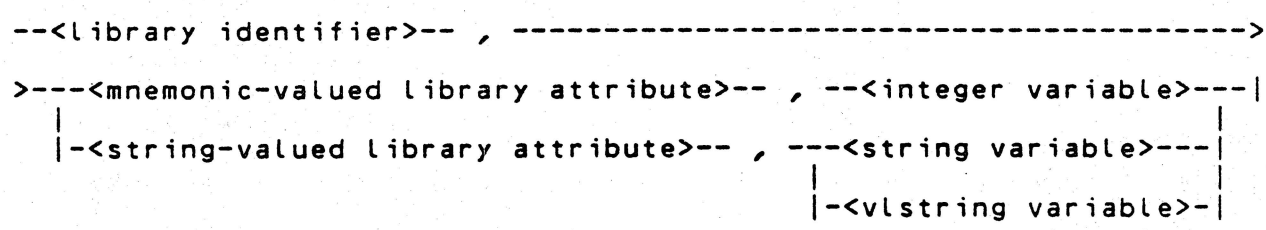
<getattribute procedure>



<file attribute request>



<library attribute request>



The <getattribute procedure> returns the value of the specified file or library attribute.

For a <file attribute request>, the file attribute is returned for the file denoted by <textfile variable>, <file variable>, <subfile variable>, or <station variable>. The attribute value is returned in the variable provided as the third parameter. An error occurs if a <subfile variable> or <station variable> is used and an invalid <subfile index> or an invalid <station index> is specified.

For a <library attribute request>, the library attribute is returned for the library denoted by <library identifier>. The attribute value is returned in the variable provided as the third parameter.

The file or library attribute specified must be a valid attribute. The mapping between file attribute types and Pascal types is described in the "Use of File Attributes" section of this manual. Library attribute types are defined in the "Library Declarations" section of this manual.

The AVAILABLE and PRESENT file attributes cannot be accessed because, as a side effect, accessing either attribute causes the I/O subsystem to open the file without changing the necessary Pascal program state.

The <getattribute procedure> is a Burroughs extension to ANSI Pascal.

See also
Library Declarations. 68
Use of File Attributes. 152

Examples

```
var i : integer;
    b : Boolean;
    f : file of integer;
begin
getattribute(f, crunched, b);
getattribute(input, maxrecsize, i);
end;
```

The first example places the value of the CRUNCHED file attribute for the file variable "f" into the variable "b".

The next example places the value of the MAXRECSIZE file attribute for the file variable "input" into the variable "i".

IOTIME FUNCTION

<iotime function>

```
-- iotime --|
```

The <iotime function> returns, as a real value in units of seconds, the total I/O time that has been charged to the program.

The <iotime function> is a Burroughs extension to ANSI Pascal.

ODD FUNCTION

<odd function>



The <odd function> returns, as a Boolean value, a result indicating whether or not the value of the <integer expression> is odd. The function returns "true" if the value is odd and "false" if it is even.

Example

```
var b : Boolean;
begin
b := odd(79 mod 27);
end;
```

PRED FUNCTION

<pred function>

```
-- pred -- ( --<ordinal expression>-- ) --|
```

The <pred function> returns the value whose ordinal number is one less than that of the <ordinal expression> (that is, its "predecessor"). If the <ordinal expression> has no predecessor value, an error occurs.

The function returns a result of the same type as the <ordinal expression>.

Examples

```
type color = (red, yellow, blue, green, tartan);
var swatch : color;
    i : integer;
begin
swatch := pred(blue);
i := pred(7);
end;
```

The first example assigns "yellow" to the variable "swatch".

The second example assigns 6 to the variable "i".

RUNTIME FUNCTION

<runtime function>

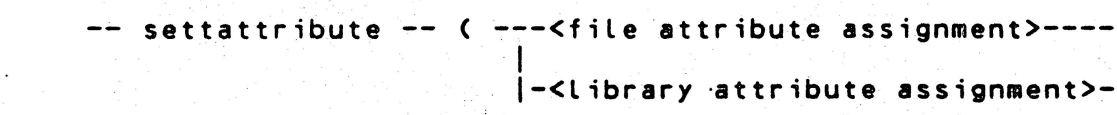


The <runtime function> returns, as a real value in units of seconds, the processor time that has been charged to the program.

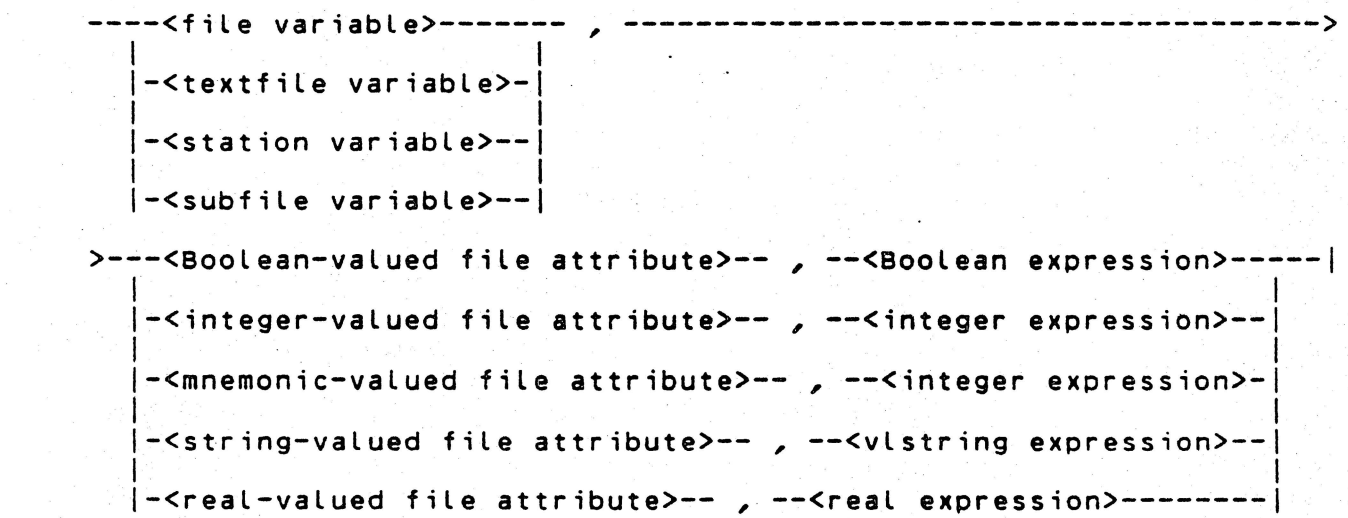
The <runtime function> is a Burroughs extension to ANSI Pascal.

SETATTRIBUTE PROCEDURE

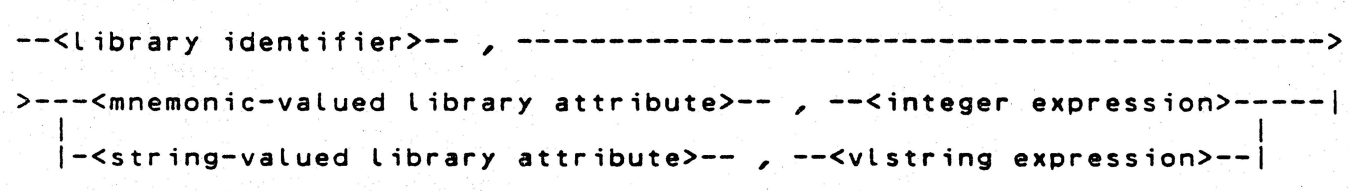
<setattribute procedure>



<file attribute assignment>



<library attribute assignment>



The <setattribute procedure> assigns the value of the expression to the specified file or library attribute.

For a <file attribute assignment>, the attribute is assigned for the file denoted by the <textfile variable>, <file variable>, <station variable>, or <subfile variable>. An error occurs if a <subfile variable> or <station variable> is used and and invalid <subfile index> or an invalid <station index> is specified.

For a <library attribute assignment>, the attribute is assigned for the library denoted by the <library identifier>.

The file or library attribute specified must be a valid attribute. The mapping between file attribute types as specified in the I/O Subsystem Manual and Pascal types is described in the "Use of File Attributes" section of this manual. Library attribute types are defined in the "Library Declarations" section of this manual.

Setting the OPEN file attribute is not allowed because setting OPEN causes the I/O subsystem to open the file without changing the necessary Pascal program state; a file can be opened at run time using the "open", "reset", or "rewrite" procedures.

The <setattribute procedure> is a Burroughs extension to ANSI Pascal.

See also
Library Declarations. 68
Use of File Attributes. 152

Examples

```
var f : file of integer;
begin
setattribute(f, kind, filevalue(kind, disk));
setattribute(f, title, 'A/B/C');
end;
```

The first example sets the KIND file attribute for the file "f" to the value of the DISK mnemonic.

The second example sets the TITLE file attribute for the file "f" to the value "A/B/C".

SUCC FUNCTION

<succ function>



The <succ function> returns the value whose ordinal number is one greater than that of the <ordinal expression> (that is, its "successor"). If the <ordinal expression> does not have a successor value, an error occurs.

The function returns a value of the same type as the <ordinal expression>.

Examples

```
type color = (red, yellow, blue, green, tartan);
var wool_dye : color;
    alpha : char;
begin
wool_dye := succ(blue);
alpha := succ('y');
end;
```

The first example assigns "green" to the variable "wool_dye".

The second example assigns 'z' to the variable "alpha".

TIME PROCEDURE

<time procedure>

```
-- time -- ( --<hours>-- , --<minutes>-- , --<seconds>-- ) --|
```

<hours>

```
--<variable>--|
```

<minutes>

```
--<variable>--|
```

<seconds>

```
--<variable>--|
```

The <time procedure> returns the current time of day (based on a 24-hour clock) in the parameters <hours>, <minutes>, and <seconds>. The values returned will be of <integer type> and will be in the following ranges:

parameter	range
<hours>	0..23
<minutes>	0..59
<seconds>	0..59

The types of the <variable>s must be such that the values returned will be assignment compatible with those <variable>s.

The <time procedure> is a Burroughs extension to ANSI Pascal.

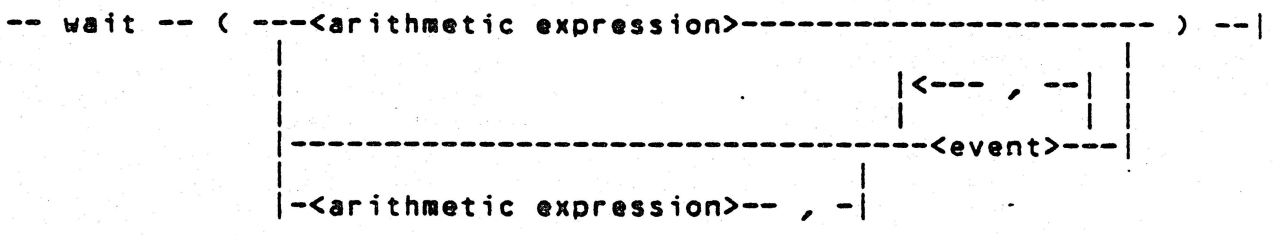
Example

```
var hours   : real;
    minutes : integer;
    seconds : 0..59;
begin
time (hours, minutes, seconds);
end;
```

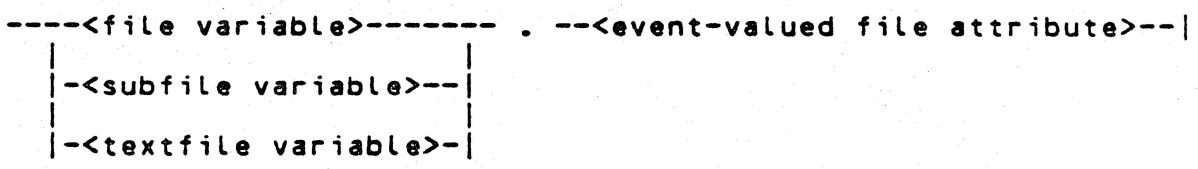
The hour will be placed in the variable "hours", the number of minutes past the hour will be placed in the variable "minutes", and the number of seconds into the minute will be placed in the variable "seconds".

WAIT PROCEDURE

<wait procedure>



<event>



The <wait procedure> causes the program to be suspended until the time specified, in seconds, by <arithmetic expression> has elapsed or one of the <event>s has been caused. The <wait procedure> can also be treated as a function, in which case a value of integer type is returned indicating which item in the parameter list caused the program to be resumed. If the cause is associated with the first parameter, a value of 1 is returned, a value of 2 is returned for the second parameter, and so on. Please refer to the I/O Subsystem Manual and the Burroughs Network Architecture (BNA) Architectural Description Reference Manual for a description of <event-valued file attribute>s and their use.

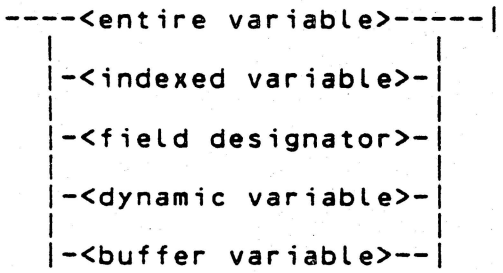
The <wait procedure> is a Burroughs extension to ANSI Pascal.

7 VARIABLES

A <variable> is a declared item that, unlike a constant, can be assigned a value during the execution of the program. Every <variable> has an associated type, which determines the values that it can be assigned. Variables of specific types, such as <array variable>s and <Boolean variable>s, are described later in the "Variables by Type" portion of this chapter.

7.1 VARIABLES BY ACCESS

<variable>



Another characteristic of a <variable> is its "access", which refers to the method by which the <variable> is identified when its value is to be referenced or changed. The access characteristic is basically independent of the variable's type. In general, it depends on whether or not the variable is a component of a structured variable and, if so, the type of the structured variable of which it is a component. The variables described below (entire, indexed, field, dynamic, and buffer variables) define the possible access characteristics.

ENTIRE VARIABLES

<entire variable>

```
--<variable identifier>--|
```

An <entire variable> is a <variable identifier> that was declared in a <variable identifier list> in a group of <variable declarations> or was defined as a formal parameter. An <entire variable> can be accessed simply by its name.

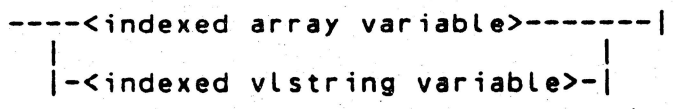
Example

```
var x   : real;
    str : packed array [1..5] of char;
```

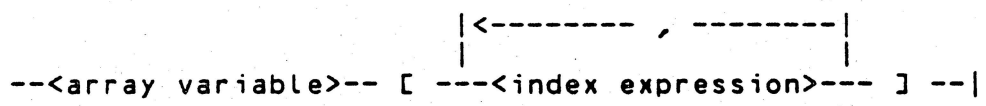
"X" and "str" are <entire variable>s; "str[1]", "str[2]", "str[3]", "str[4]", and "str[5]" are not <entire variable>s.

INDEXED VARIABLES

<indexed variable>



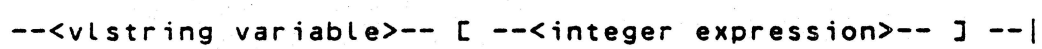
<indexed array variable>



<index expression>



<indexed vlstring variable>



An <indexed variable> denotes a variable that is a component of an array or vlstring. In order to access an <indexed array variable>, the <array variable> of which it is a component must be identified and the variable's location within that array must be specified by providing an <index expression> for each dimension of the array. The value of each <index expression> must be assignment compatible with the <index type> of the array dimension it specifies.

In order to access a component of a <vlstring variable>, the <vlstring variable> of which it is a component must be identified and the variable's location within that vlstring must be specified by providing an <integer expression>. The value of the <integer expression> must be less than or equal to the current length of the <vlstring variable> and greater than or equal to 1. An <indexed vlstring variable> is of the <char type>.

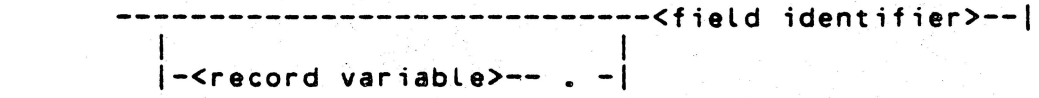
Examples

```
var s : string(5);
    a : array [Boolean] of 1..10;
```

"s[1]", "s[2]", "s[3]", "s[4]", "s[5]", "a[true]", and "a[false]" are <indexed variable>s.

FIELD DESIGNATORS

<field designator>



A <field designator> is a <variable> that denotes a <field identifier> in a <record variable>. The <record variable> of which the field is a component must be specified unless the <field identifier> appears in a <with statement> that designates the appropriate <record variable>.

It is an error to change the active <variant> of a record while a <field designator> within the currently active <variant> is being referenced in one of the following ways:

a) as the <record variable> of a <with statement>.

b) as an actual variable parameter in an <actual parameter list>.

c) as the left-hand side of an <assignment statement>.

See also

Actual Parameter Lists and Parameter Matching 80
Assignment Statements 85
With Statements 101

Example

```
var r1, r2 : record
             i : integer;
             b : Boolean;
             end;
```

"R1.i", "r1.b", "r2.i", and "r2.b" are <field designator>s.

DYNAMIC VARIABLES

<dynamic variable>

```
--<pointer variable>-- @ --|
```

A <dynamic variable> is a <variable> accessed through a <pointer variable> declared as a pointer to the type of the <variable>. in order for a variable to be a <dynamic variable>, it must have been allocated dynamically, through the <new procedure>.

An error occurs if the <pointer variable> is NIL, is undefined, contains a "mark value" (refer to the <mark procedure>), or references a dynamic variable that has been deallocated through the use of the <dispose procedure> or the <release procedure>.

It is an error to "dispose" or "release" a dynamic variable while any one of the following references to that variable exists:

a) as the <record variable> of a <with statement>.

b) as an actual variable parameter in an <actual parameter list>.

c) as the left-hand side of an <assignment statement>.

See also

Actual Parameter Lists and Parameter Matching 80
Assignment Statements 85
With Statements 101
Dynamic Allocation Procedures 204

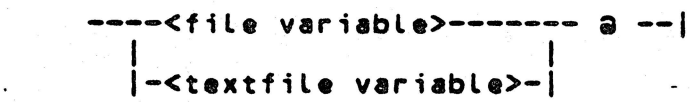
Example

```
type ptr  = @node;
     node = record
              name : packed array [1..20] of char;
              next : ptr;
              end;
var  p1, p2 : ptr;
     person : node;
begin
new(p1);
p1@.name := 'Robert Smith';
p1@.next := nil;
person := p1@;
end;
```

"P1" is a pointer to a dynamically allocated record of type "node". "P1@" is a record of type "node" and is assignment compatible with "person".

BUFFER VARIABLES

<buffer variable>



A <buffer variable> is automatically associated with each declared <file variable> and <textfile variable>. The <buffer variable> for a file or textfile is the means by which the file component associated with the current file position can be examined or modified. The type of the <buffer variable> is the <component type> of the file. For textfiles, the <buffer variable> is of type char.

It is an error to alter the position of a file while the buffer variable is in use in one of the following ways:

a) as the <record variable> of a <with statement>.

b) as an actual variable parameter in an <actual parameter list>.

c) as the left-hand side of an <assignment statement>.

See also
Actual Parameter Lists and Parameter Matching 80
Assignment Statements 85
With Statements 101

Example

```
var myfile : file of integer;
    inx : integer;
begin
rewrite(myfile);
myfile@ := 3;
put(myfile);
reset(myfile);
inx := myfile@;
end;
```

The type of <buffer variable> "myfile@" is the same as the component type of the file. Therefore, in this example, "myfile@" may be used as a variable of type "integer".

7.2 VARIABLES BY TYPE

ARRAY VARIABLE

<array variable>

A <variable> declared of an <array type>.

BOOLEAN VARIABLE

<Boolean variable>

A <variable> declared of the <Boolean type> or of a <subrange type> whose host type is the <Boolean type>.

CHAR VARIABLE

<char variable>

A <variable> declared of the <char type> or of a <subrange type> whose host type is the <char type>.

ENUMERATED VARIABLE

<enumerated variable>

A <variable> declared of an <enumerated type> or of a <subrange type> whose host type is an <enumerated type>.

FILE VARIABLE

<file variable>

An <entire variable> declared of a <file type>.

INTEGER VARIABLE

<integer variable>

A <variable> declared of the <integer type> or of a <subrange type> whose host type is the <integer type>.

POINTER VARIABLE

<pointer variable>

A <variable> declared of a <pointer type>.

REAL VARIABLE

<real variable>

A <variable> declared of the <real type>.

RECORD VARIABLE

<record variable>

A <variable> declared of a <record type>.

SET VARIABLE

<set variable>

A <variable> declared of a <set type>.

STRING VARIABLE

<string variable>

A <variable> declared of a <string type>.

STATION VARIABLE

<station variable>

```
--<file variable>-- ( -- station --<station index>-- ) --|
```

<station index>

```
--<integer expression>--|
```

A remote file (a file whose KIND attribute is REMOTE) may have one or more associated "stations", each of which may be connected to a separate terminal or remote device. A <station variable> can be used instead of a <file variable> in certain predefined procedures and functions in order to refer to a particular station. In all cases, the type of the <station variable> is the same as the type of the <file variable> that it qualifies. Note also that all stations share the buffer variable of their associated <file variable>. For more information on the use of stations in Pascal, please refer to the description of the <get procedure>, <put procedure>, <addstation procedure>, and <deletestation procedure> in the "File Handling Procedures and Functions" section of this manual, and the <getattribute procedure> and <setattribute procedure> in the "General Procedures and Functions" section.

The <station variable> is a Burroughs extension to ANSI Pascal.

See also
File Handling Procedures and Functions. 132
General Procedures and Functions. 238

SUBFILE VARIABLE

<subfile variable>

```
--<file variable>-- ( -- subfile --<subfile index>-- ) --|
```

<subfile index>

```
--<integer expression>--|
```

A port file (a file whose KIND attribute is PORT) may have one or more associated "subfiles", each of which may be connected to a different complementary subfile. A <subfile variable> can be used instead of a <file variable> in certain predefined procedures and functions in order

to refer to a particular subfile. In all cases, the type of the <subfile variable> is the same as the type of the <file variable> that it qualifies. Note also that all subfiles share the buffer variable of their associated <file variable>. For more information on the use of subfiles in Pascal, please refer to the description of the <open procedure>, <close procedure>, <get procedure>, and <put procedure> in the "File Handling Procedures and Functions" section of this manual, and the <wait procedure>, <getattribute procedure> and <setattribute procedure> in the "General Procedures and Functions" section.

The <subfile variable> is a Burroughs extension to ANSI Pascal.

See also
File Handling Procedures and Functions. 132
General Procedures and Functions. 238

TEXTFILE VARIABLE

<textfile variable>

An <entire variable> of the <textfile type>.

VLSTRING VARIABLE

<vlstring variable>

A <variable> declared of a <vlstring type>.

7.3 UNDEFINED VARIABLES

An undefined variable is a variable whose value is invalid for some reason and therefore must not be examined. For example, when a block is entered at run time, all variables declared within that block are allocated as undefined variables. The use of any undefined variable in an expression is an error.

An undefined variable becomes defined when it is assigned a valid value, for example, when it appears as the left-hand side of an <assignment statement> or as an actual variable parameter to a procedure or function that will assign it a value (such as "read").

Example

```
var i : integer;
    j : integer;
begin
j := i;   { ERROR -- the value of "i" is undefined. }
end;
```

8 BASIC COMPONENTS

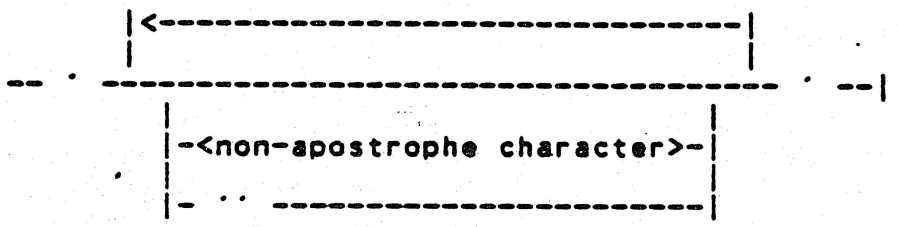
The basic components defined in this section are syntactic items that appear in the syntax diagrams in previous sections of the manual, but are simple enough, and widely distributed enough, that explaining them where they appear would be redundant or cumbersome. These items include characters, identifiers, and numbers.

The next chapter, Interpretation of Program Text, describes a different set of basic items -- those that relate to the representation of the program and the compiler's interpretation of it. Those items include reserved words, comments, context-sensitive identifiers, and special symbols (and their notational synonyms, if any).

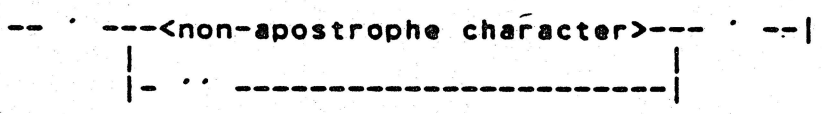
A special convention for the "Railroad" syntax notation is used in this section: the basic components described here must not contain embedded blanks, comments, or record boundaries, even though the standard interpretation of "Railroad" diagrams permits these token separators between any two distinct items in a diagram. Of course, blanks are allowed as <character>s within a <character string>, but they are significant in that context and are not treated as token separators.

CHARACTERS AND CHARACTER STRINGS

<character string>



<character literal>



<non-apostrophe character>

Any <character> except the apostrophe (').

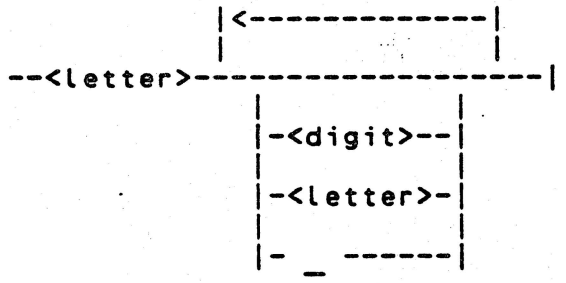
<character>

Any one of the characters in the "standard character set". The standard character set is EBCDIC, unless specified as ASCII using the STRINGS compiler control option.

A <character string> represents a constant of the <string type>, and a <character literal> represents a constant of the <char type>. A single apostrophe (') character contained within a <character string> or <character literal> is represented by two successive apostrophes (e.g. '''A''' is a <character string> containing the three characters 'A'). A <character string> that contains no values ('') is a null string.

IDENTIFIERS

<identifier>



<letter>

Any one of the letters A through Z or a through z. The lowercase characters (a through z) are synonymous with the uppercase characters (A through Z).

<digit>

Any one of the decimal numbers 0 through 9.

Identifiers may be of any length (greater than 0), subject to the constraint that an identifier cannot be split across source records. All characters, including underscores, are significant in distinguishing identifiers. An <identifier> must not have the same spelling as a <reserved word> (please refer to Interpretation of Program Text).

Allowing underscores in identifiers is a Burroughs extension to ANSI Pascal.

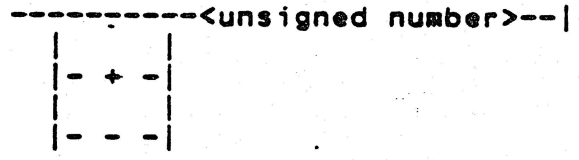
See also
Interpretation of Program Text. 277

Examples

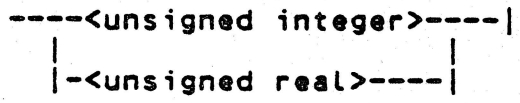
```
Index       MESSAGE_COUNT      item_3   { Three valid identifiers }
BEGIN       { INVALID -- reserved word }
1776        { INVALID -- doesn't start with a letter }
W2 form     { INVALID -- embedded blanks not allowed }
```

NUMBERS

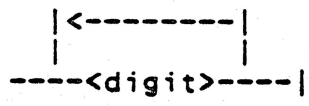
<number>



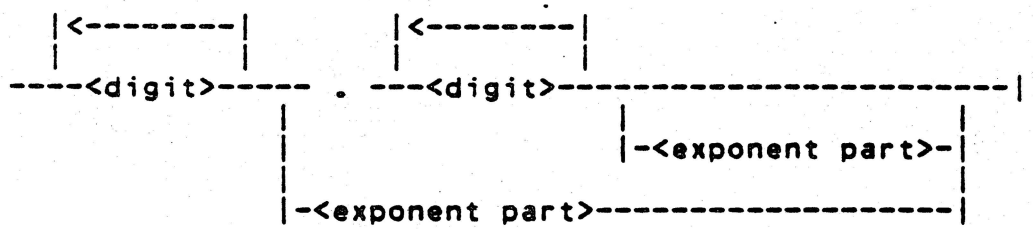
<unsigned number>



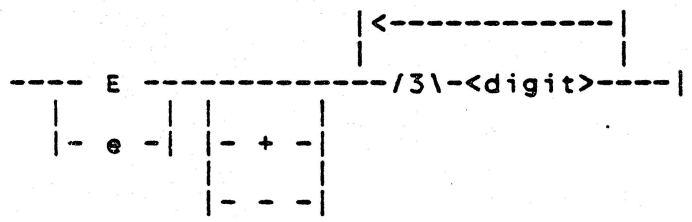
<unsigned integer>



<unsigned real>



<exponent part>



A <number> is an integer or real value optionally preceded by a sign. If no sign is specified, "+" is assumed. Numbers are symmetrical around zero; that is, any magnitude that can be represented as a positive value can also be represented as a negative value, and vice versa.

The type of a <number> is determined by its format. A simple string of one or more digits is an <unsigned integer>. The largest <unsigned integer> can be referred to by the predefined <integer constant identifier> "maxint".

A number that includes a fractional part and/or an <exponent part> is an <unsigned real>. Up to twelve significant digits of precision are maintained (please refer to Appendix C, Data Representation).

In the <exponent part>, the letter "E" introduces a decimal exponent ("E" has the meaning "times 10 to the power of"). The absolute value of an <unsigned real> must not exceed 4.31359146674E+68. The smallest absolute value of an <unsigned real> is 8.75811540203E-47.

See also
Data Representation 314

Examples

```
123     -1000     +2              0      { integers }
0.0     -23.45    124567.4e-20    9E15   { reals }
```

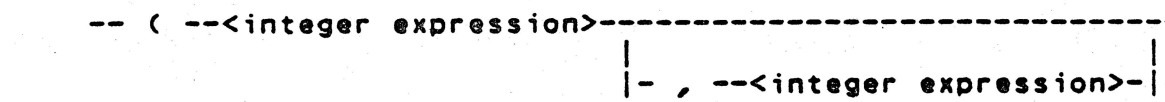
FILE ATTRIBUTES AND MNEMONIC VALUES

<Boolean-valued file attribute>
<event-valued file attribute>
<integer-valued file attribute>
<mnemonic-valued file attribute>
<string-valued file attribute>
<real-valued file attribute>
<mnemonic value>

File attributes and values are system-defined identifiers describing characteristics of files. Most of the file attributes and values described in the I/O Subsystem Reference Manual are available in Pascal; please refer to the Use of File Attributes section for additional information.

Certain file attributes may either require or allow parameters in order to further qualify the property of the file that is to be modified or queried. In order to access such attributes, an <attribute parameter list> may be used in the <getattribute procedure> and the <setattribute procedure>. An <attribute parameter list>, if used, must immediately follow the name of the attribute to be accessed.

<attribute parameter list>



Example

```
type t = packed array [1..80] of char;
var f, g : file of t;
    i : integer;
begin
i := 1;
setattribute(f, serialno(1), 'TAPE1');
setattribute(g, familyindex(i,2), 2);
end.
```

See also
Use of File Attributes. 152

9 INTERPRETATION OF PROGRAM TEXT

The Pascal program to be compiled is presented to the compiler as one or more files in a particular format. The merging of multiple files, and the files themselves, are described in Appendix A, Compiler Control Records, and Appendix B, Compiler Files. This section describes how the compiler interprets its input during the compilation process.

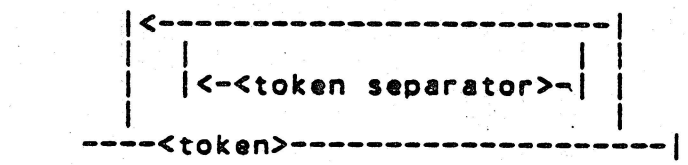
For purposes of this discussion, the program input file can be considered a sequence of records (from whatever source) that the compiler reads during compilation. Each record includes the following fields:

Columns	Contents
1-72	<program text> and <compiler control record>s
73-80	sequence number (optional)
81-90	mark information (optional)

Records containing a "$" in column 1 are <compiler control record>s, which are not part of the Pascal program; they are described in Appendix A. Records that do not contain a "$" in column 1 are assumed to contain <program text>, that is, the Pascal program to be compiled. Optionally, there can be sequence information in columns 73-80 (refer to the SEQUENCE compiler control option) and mark information in columns 81-90. These fields are not discussed further here.

PROGRAM TEXT

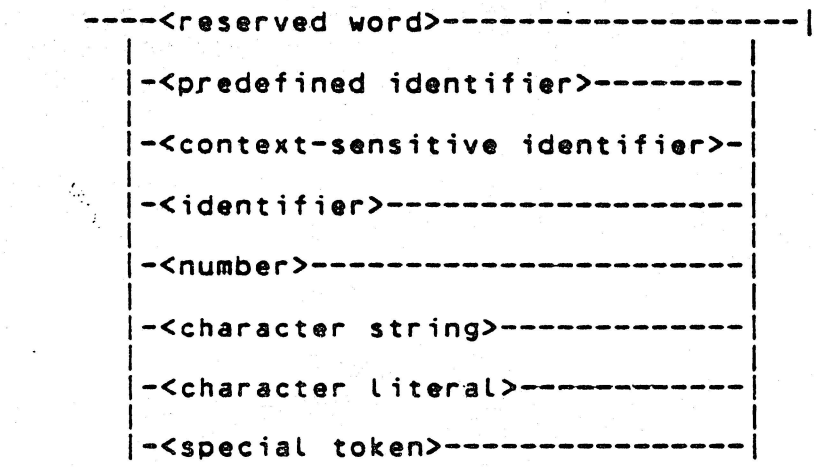
<program text>



The Pascal <program text> can be considered a continuous stream of <token>s, all of which may be, and some of which must be, separated by <token separator>s.

TOKEN

<token>



A token is a sequence of characters in the program text that the compiler recognizes as a syntactic unit. Every pair of tokens must be separated by a <token separator> unless one token in the pair is a <special token>.

Reserved Word

<reserved word>

AND	ARRAY	BEGIN	CAND	CASE	CONST
COR	DIV	DO	DOWNTO	ELSE	END
FILE	FOR	FUNCTION	GOTO	IF	IN
INTERFACE	LABEL	LIBRARY	MOD	NIL	NOT
OF	OR	OTHERWISE	PACKED	PROCEDURE	PROGRAM
RECORD	REPEAT	SET	THEN	TO	TYPE
UNTIL	USAGE	VAR	WHILE	WITH	

<Reserved word>s are language keywords that cannot be redefined by the programmer. In general, these are words the compiler uses to recognize declarations, statements, and operators.

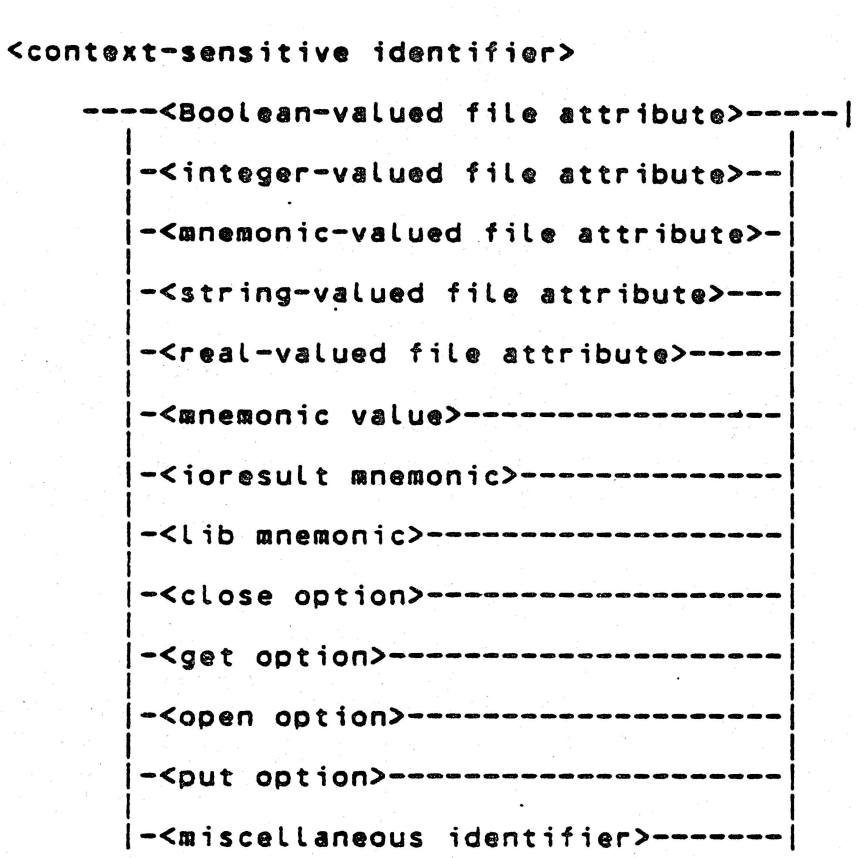
Predefined Identifier

<predefined identifier>

abort	abs	accept	addstation
arccos	arcsin	arctan	arctanh
Boolean	cancel	char	chr
close	concat	cos	cosh
cotan	date	delete	deletestation
display	dispose	elapsedtime	eof
eoln	erf	exp	false
filevalue	fillchar	freeze	gamma
get	getattribute	input	insert
integer	iores	iotime	length
libraryvalue	ln	lngamma	log
mark	max	maxint	min
new	odd	open	ord
output	pack	page	pos
pred	put	random	read
readln	real	release	reset
rewrite	round	runtime	scan_eql
scan_neq	seek	setattribute	sin
sinh	skiptochannel	sqr	sqrt
string	succ	tan	tanh
text	time	true	trunc
unpack	wait	write	writeln

<Predefined identifier>s are <identifier>s that have a predefined meaning in Pascal; as with user-defined <identifier>s, <predefined identifier>s can be redefined, but the former definition becomes unavailable within the scope of the redefinition.

Context-Sensitive Identifier



<miscellaneous identifier>

actualname	dontcare	duration	forward
functionname	intname	libaccess	permanent
private	sharing	sharedbyall	sharedbyrununit
station	subfile	temporary	title

A <context-sensitive identifier> is a predefined <identifier> that is recognized only within a certain context. If there is a user-declared <identifier> with the same name, an occurrence of the <identifier> outside its special context refers to the user-declared item, and an occurrence of the <identifier> inside its special context carries the predefined meaning. The valid values for these items are defined in the sections that describe their uses.

<Identifier>, <number>, <character string>, and <character literal> are user-specified items that are defined in the Basic Components chapter.

Special Token

<special token>

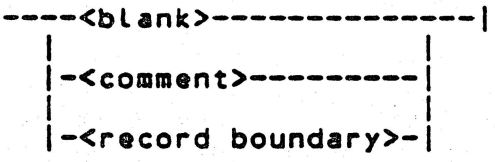
+	-	*	/	=	<	>	[	]
.	,	:	;	@	(	)	<>	<=
>=	:=	..						

The <special token>s are the various symbols that are used as operators and brackets within the language. Because these symbols are usually readily distinguishable from alphanumeric tokens, a token separator is not required either before or after a <special token>.

Some special tokens have character synonyms defined because of the differences between the ASCII and EBCDIC character codes and graphics. "(." and ".)" are synonyms for "[" and "]" (left and right square bracket). " " (circumflex) is a synonym for "@" (commercial "at"). [As shown in the <comment> diagram below, "(*" and "*)" are synonyms for "{" and "}" (left and right braces).]

TOKEN SEPARATOR

<token separator>



<Token separator>s are required as delimiters for alphanumeric tokens, to separate tokens so that the compiler will interpret them properly. However, this function is incidental for <comment>s; their purpose is to allow the programmer to interleave descriptive text with the program text.

Blank

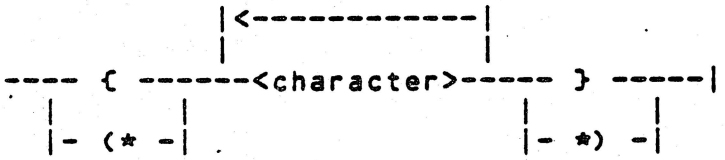
<blank>

One or more blank characters.

Blanks can be used freely throughout the program text to improve readability and to separate tokens that must be separated so that the compiler will interpret them property.

Comment

<comment>



Comments are used to include documentation in a program. A <comment> can appear anywhere that a <blank> can appear; a <comment> cannot appear in a <character string> or in another <comment>. Comments may contain any <character>s except the delimiting characters "}" and "*)".

Note that compiler control records that appear between the record containing the beginning of a comment and the record containing the end of that comment are processed as normal compiler control records; they are not treated as part of the comment.

Examples

```
{ This is a comment. }
(* This comment uses the two-character synonyms for braces. *)
```

Record Boundary

<record boundary>

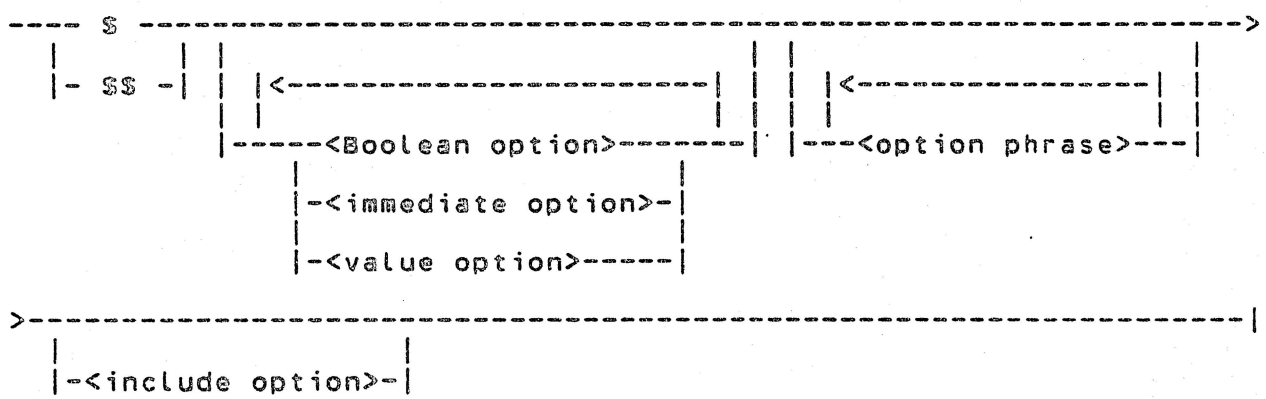
A theoretical boundary between column 72 of one record and column 1 of the next record.

The <record boundary> acts as an implicit token separator. Thus, a token cannot appear in part up through column 72 of one record and then be continued beginning in column 1 of the next record. The compiler interprets a split item as two separate items.

A COMPILER CONTROL RECORDS

<Compiler control record>s contain options that direct the compilation process. For example, the user can instruct the compiler to accept input from several different files or to produce a printer listing of the program text. Familiarity with the information contained in Appendix B, Compiler Files, is helpful in understanding the information in this appendix.

<compiler control record>



<Compiler control record>s contain a dollar sign ("$") in column one of the input record. Records with a dollar sign only in column one are "temporary compiler control records": they affect only the current compilation, they are not saved in the NEWSOURCE file (if any), and (unless the LISTDOLLAR option is TRUE) they are not printed in the program listing. Records containing a dollar sign in both columns one and two are "permanent compiler control records": they remain permanently associated with the language portion of the input and output files.

A <compiler control record> that consists of a "$" character and no option information (a "$" followed by all blanks) is called a "null option record" and has no effect unless the MERGE option is TRUE. If MERGE is TRUE and a record with the same sequence number as the null option record exists in the secondary input file (SOURCE), the record from SOURCE will be ignored.

Options and keywords appearing on compiler control records should appear in uppercase; if lowercase words are found, the compiler gives a warning that the compiler control records may not be compatible with other system software.

Three types of options may be specified: (1) Boolean, (2) value, and (3) immediate.

A <Boolean option> is either enabled (set to TRUE) or disabled (set to FALSE). When enabled, a <Boolean option> causes the compiler to apply an associated function to all subsequent processing until the option is disabled.

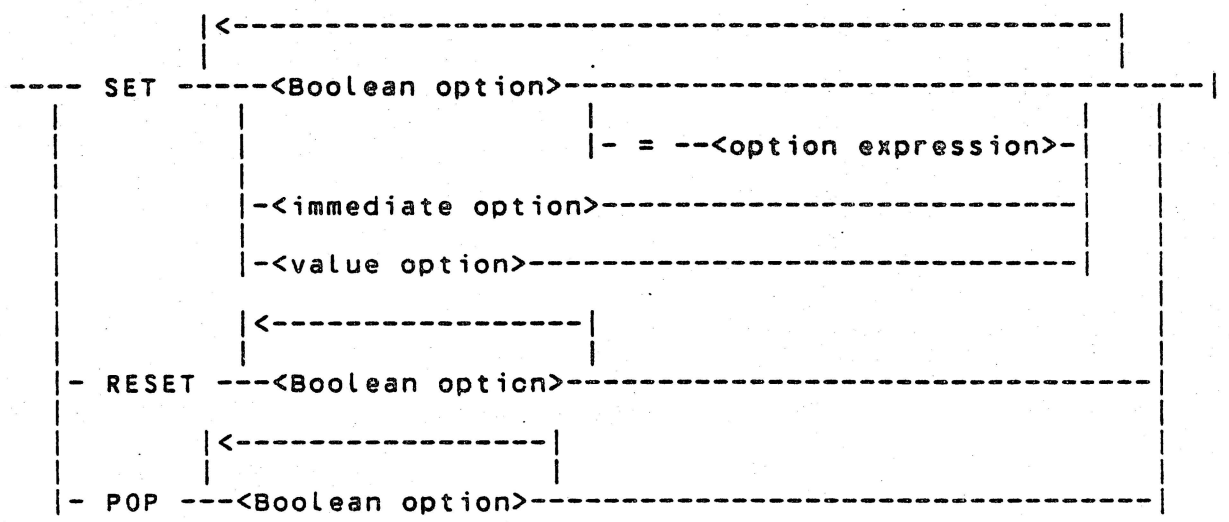
A <value option> causes the compiler to store a value associated with a given function.

An <immediate option> causes the compiler to perform a function independent of subsequent processing. The <include option> is an <immediate option>, but is not grouped with those options because of its special syntactic requirements.

The <option phrase> provides additional features that apply to Boolean options.

A.1 OPTION PHRASE

<option phrase>



The <option phrase> may include the option actions SET, RESET, and POP, and may contain an <option expression>, which is described later. The keywords SET, RESET, and POP affect the setting of <Boolean option>s. Each <Boolean option> has an associated "stack", in which up to 47 previous settings of the option are saved. The management of the option setting and the stack is described below.

1. If a <Boolean option> is not the object of an explicit SET, RESET, or POP, it is implicitly enabled and the previous setting is pushed onto the stack (that is, the previous setting of the option is saved).

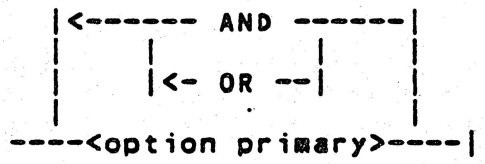
2. If a <Boolean option> is the object of a SET or RESET, the option is enabled or disabled, respectively, and the previous setting is pushed onto the stack.

3. If a <Boolean option> is the object of a POP, the current setting of the option is discarded, and the previous setting is popped off the stack (the setting is restored to its last previous value).

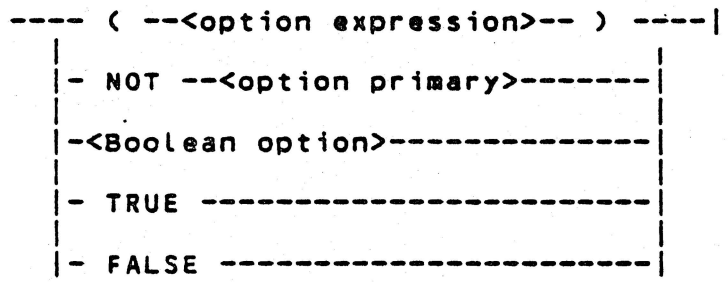
4. If the <immediate option> CLEAR appears, all <Boolean option>s, except MERGE, NEW, and <user option>s, are disabled, and all <Boolean option> stacks, except for <user option>s, are discarded.

A.2 OPTION EXPRESSIONS

<option expression>



<option primary>

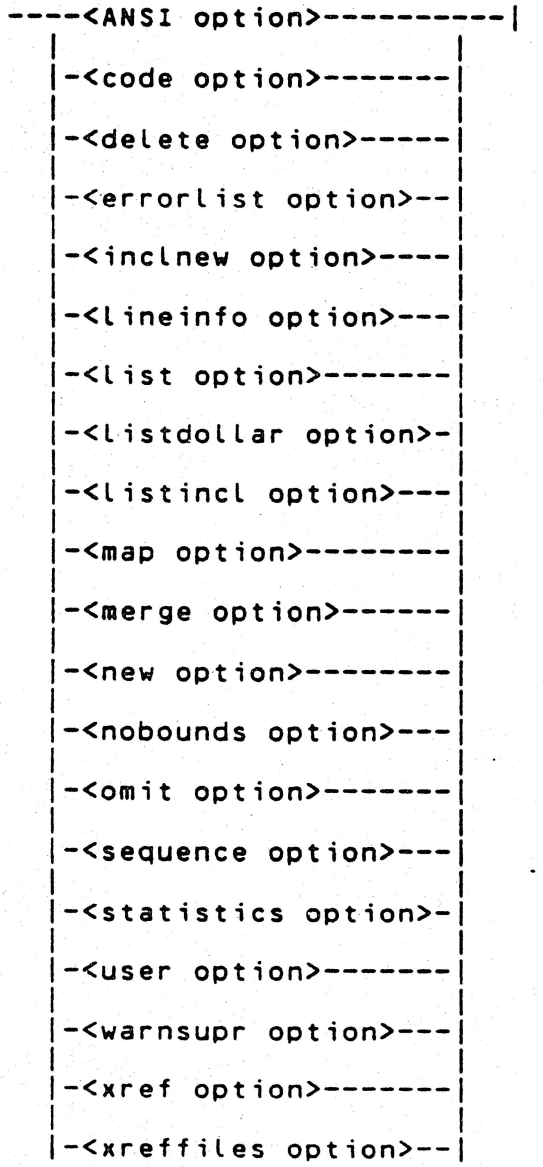


An <option expression> phrase assigns a value to a <Boolean option> based on the settings of other <Boolean option>s. The operators AND, OR, and NOT have the familiar meanings for those logical operators. The order of precedence of evaluation within an <option expression> is as follows:

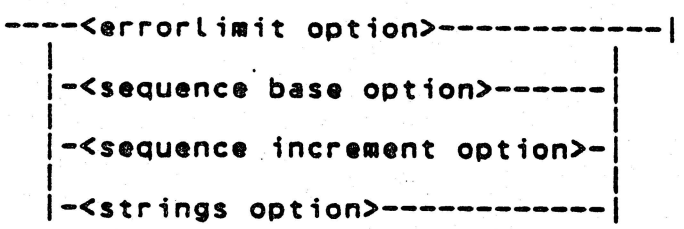
[highest]	Parenthesized expressions
	NOT
	AND
[lowest]	OR

A.3 OPTIONS

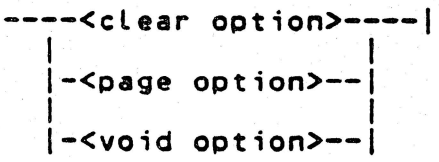
<Boolean option>



<value option>



<immediate option>



ANSI OPTION

<ANSI option>



(Type: Boolean, Default: FALSE)

The ANSI option, when TRUE, causes a syntax error to be given for uses of extensions to the ANSI Pascal Standard. ANSI is FALSE by default.

CLEAR OPTION

<clear option>

```
-- CLEAR --|
```

(Type: immediate)

The CLEAR option disables (sets to FALSE) all <Boolean option>s except MERGE, (conditionally) NEW, and <user option>s. It then discards all <Boolean option> stacks except those for <user option>s. The NEW option is set to FALSE only if no source language records have been written to the new symbolic file (NEWSOURCE).

See also
MERGE Option. 298
NEW Option. 298

CODE OPTION

<code option>

```
-- CODE --|
```

(Type: Boolean, Default: FALSE)

The CODE option, when enabled, causes the object code produced by the compilation process to be included in the program listing (file LINE).

DELETE OPTION

<delete option>

```
-- DELETE --|
```

(Type: Boolean, Default: FALSE)

The DELETE option, when enabled, causes source language records from the secondary input file (SOURCE) to be discarded until the option is disabled. DELETE is ignored if the MERGE option has not been enabled.

This option does not alter the normal merging process. However, it causes all source language records selected from SOURCE, including compiler control records, to be unconditionally discarded.

The source language records discarded as a result of this option are not carried forward to the output symbolic file (NEWSOURCE) when the NEW option is enabled; they are also not included in the program listing.

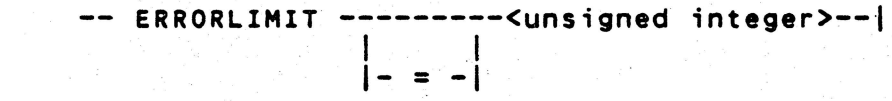
This option may appear only on a compiler control record in the primary input file (CARD).

See also

MERGE Option. 298
NEW Option. 298

ERRORLIMIT OPTION

<errorlimit option>



(Type: Value, Default: 6 if the compile was initiated through CANDE; otherwise 100)

The ERRORLIMIT option specifies the maximum number of errors that may occur before a compilation is terminated. If the error limit is exceeded when the NEW option is enabled, the NEWSOURCE file is purged.

If the ERRORLIMIT is set to zero, no limit is applied.

The ERRORLIMIT option counts only syntax errors, not warnings.

See also
NEW Option. 298

ERRORLIST OPTION

<errorlist option>



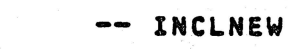
(Type: Boolean, Default: TRUE if the compile was initiated through CANDE; otherwise FALSE)

The ERRORLIST option, when TRUE, causes error and warning messages to be written to the error file (ERRORS).

If this option is TRUE and the <list option> is FALSE, error and warning messages appear only in the error file. If both this option and the <list option> are TRUE, all error and warning messages appear in both the error file (ERRORS) and the output listing (LINE).

INCLNEW OPTION

<inclnew option>



(Type: Boolean, Default: FALSE)

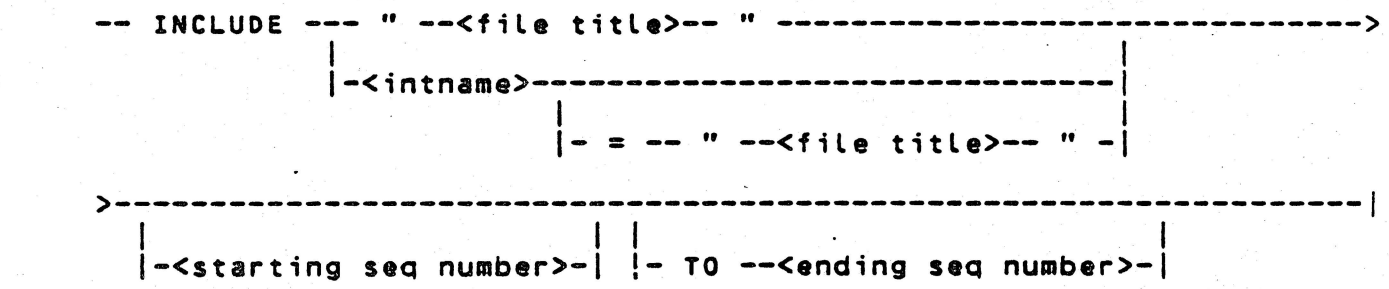
The INCLNEW option, when enabled, writes any source language records read as a result of the enabling of the INCLUDE option to the output symbolic file (NEWSOURCE). If the NEW option is disabled, the INCLNEW option has no effect.

If this option is enabled and the SEQUENCE option is also enabled, the source records from the included file are assigned new sequence numbers. However, if the SEQUENCE option is disabled, the sequence numbers of the source records from the included file remain unchanged.

See also
INCLUDE Option. 294
NEW Option. 298
SEQUENCE (SEQ) Option 301

INCLUDE OPTION

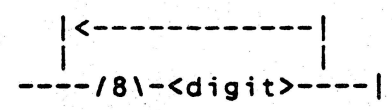
<include option>



<file title>

A valid value for the TITLE file attribute, which is described in the I/O Subsystem Reference Manual.

<starting seq number>
<ending seq number>



<intname>

An <identifier> that contains no underscore characters.

(Type: immediate)

The INCLUDE option suspends input from the primary and secondary input files (CARD and SOURCE) and accepts input from the file specified by <file title> or <intname>.

If an <intname> is specified, it defines an <identifier> that can be used to supply the <file title> dynamically through file equation or, if the "= <file title>" phrase is included, to refer to the specified file in subsequent occurrences of the <include option>. If both file equation and the "= <file title>" phrase are used, the title supplied through file equation overrides.

If a <starting seq number> is specified, the inclusion of the source records begins when a record in the included file has a sequence number greater than or equal to the specified sequence number. If a <starting seq number> is not specified, inclusion begins with the first record of the included file. If an <ending seq number> is specified, the inclusion of the source records ends when a record in the included file has a sequence number that is greater than the <ending seq number>. If an <ending seq number> is not specified, the inclusion of source records terminates following the last record in the included file.

A file that is included may itself contain compiler control records with INCLUDE options. The maximum depth of this nesting is five.

The source language records included as a result of the INCLUDE option are not listed unless the LISTINCL option is enabled, nor are they carried forward to the output symbolic file (NEWSOURCE) unless the INCLNEW option is TRUE.

No compiler control options may follow the INCLUDE parameters on a given compiler control record.

See also
INCLNEW Option. 294
LISTINCL Option 297

LINEINFO OPTION

<lineinfo option>

-- LINEINFO --|

(Type: Boolean, Default: TRUE)

The LINEINFO option, when enabled, causes the compiler to save program sequence numbers in the CODE file. If the program terminates abnormally, the source language sequerce number associated with the point of program termination is displayed.

LIST OPTION

<list option>

-- LIST --|

(Type: Boolean, Default: FALSE if the compile was initiated through CANDE; otherwise TRUE)

The LIST option, when enabled, creates a listing of the program (compiler file LINE). This listing includes the program text accepted for compilation, all compiler error and warning messages, and a summary of information about the compilation itself.

LISTDOLLAR OPTION

<listdollar option>

```
-- LISTDOLLAR --|
```

(Type: Boolean, Default: FALSE)

The LISTDOLLAR option, when enabled, causes the compiler to list all temporary compiler control records encountered during the compilation.

LISTINCL OPTION

<listincl option>

```
-- LISTINCL --|
```

(Type: Boolean, Default: FALSE)

The LISTINCL option, when enabled, lists all source language input that was accepted for compilation as a result of an INCLUDE option.

See also
INCLUDE Option. 294

MAP OPTION

<map option>

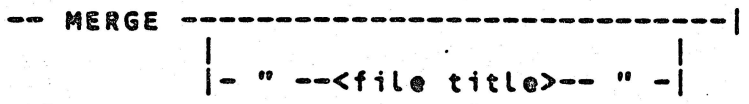
```
-- MAP --|
```

(Type: Boolean, Default: FALSE)

The MAP option, when enabled, includes, as part of the output listing, information concerning the allocation of variables within the object code produced by the compiler.

MERGE OPTION

<merge option>



(Type: Boolean, Default: FALSE)

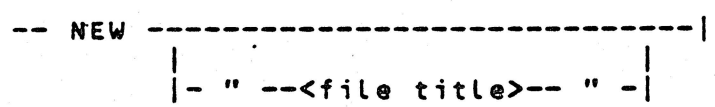
The MERGE option, when enabled, merges source language records from the primary input file (CARD) with source language records from the secondary input file (SOURCE). The <file title>, if present, causes the specified file to be read as the SOURCE file (the <file title> is used to assign the TITLE attribute of the SOURCE file). If no <file title> is specified, SOURCE is assumed.

This option remains enabled throughout a compilation and may not be disabled. Subsequent attempts to SET or RESET this option are treated as errors and are ignored.

This option may appear only on a compiler control record in the CARD file.

NEW OPTION

<new option>



(Type: Boolean, Default: FALSE)

The NEW option, when enabled, creates a new source language symbolic (NEWSOURCE) of all source language records accepted for compilation. This new symbolic includes any input records that were omitted by the OMIT option and any permanent compiler control records, but it does not include any records that were discarded by enabling the DELETE option or the VOID option.

The <file title>, if present, causes the specified file to be written as the NEWSOURCE file (the <file title> is used to assign the TITLE attribute of the NEWSOURCE file). If no <file title> is specified, NEWSOURCE is assumed.

This option remains enabled throughout a compilation and may not be disabled. Subsequent attempts to SET or RESET this option are treated as errors and are ignored.

See also
DELETE Option 292
OMIT Option 300
VOID Option 305

NOBOUNDS OPTION

<nobounds option>



(Type: Boolean, Default: FALSE)

The NOBOUNDS option, when enabled, eliminates all range-checking code that the compiler would normally generate to verify the assignment compatibility of set variables, subrange variables, and vlstring variables. Range checking that is performed by the system, such as the range check on an array index when the array is an <entire variable> (not part of a record), cannot be disabled.

See also
Variables 257

OMIT OPTION

<omit option>

```
-- OMIT --|
```

(Type: Boolean, Default: FALSE)

The OMIT option, when enabled, causes all source language records to be ignored for purposes of compilation (but not discarded) until the option is disabled. This option may appear on a compiler control record in either the primary (CARD) or the secondary (SOURCE) source language input.

The source language records omitted as a result of enabling this option are carried forward to the output symbolic file (NEWSOURCE) when the NEW option is enabled, but they are not listed.

Compiler control records encountered in the source language input while this option is enabled will be processed in the normal fashion.

See also
NEW Option. 298

PAGE OPTION

<page option>

```
-- PAGE --|
```

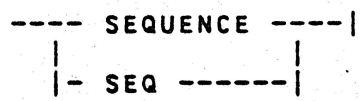
(Type: immediate)

The PAGE option causes the output listing to skip to the top of a new page. If the LIST option is disabled, this option is ignored. If the OMIT option is enabled, this option is ignored.

See also
LIST Option 296
OMIT Option 300

SEQUENCE (SEQ) OPTION

<sequence option>



(Type: Boolean, Default: FALSE)

The SEQUENCE option, when enabled, assigns new sequence numbers to the source language records accepted for compilation. This option affects only those input source language records encountered by the compiler following the merging process and includes any records that were omitted due to the enabling of the OMIT option.

The SEQUENCE option assigns the current sequence base (see The Sequence Base Option) to the current source language record and increments the sequence base by the sequence increment (see The Sequence Increment Option). Sequencing occurs before the record is written to the new symbolic and before the source language record is listed. If the sequence number exceeds 99999999 when the compiler increments the sequence base by the sequence increment, then the SEQUENCE option is disabled and a sequencing error is produced.

See also

OMIT Option 300
Sequence Base Option. 302
Sequence Increment Option 302

SEQUENCE BASE OPTION

<sequence base option>

--<unsigned integer>--|

(Type: Value, Default: 1000)

The sequence base option stores the specified <unsigned integer> value as the sequence base associated with the SEQUENCE option. This option must appear on a compiler control record in such a way that it will not be associated with any other option requiring an <unsigned integer> as a parameter or a value. The maximum value of <unsigned integer> is 99999999. This option may be specified independently of the SEQUENCE option.

See also
SEQUENCE (SEQ) Option 301

SEQUENCE INCREMENT OPTION

<sequence increment option>

-- + --<unsigned integer>--|

(Type: Value, Default: 1000)

The sequence increment option stores the specified <unsigned integer> value as the sequence increment associated with the SEQUENCE option. This option must appear on a compiler control record in such a way that it will not be associated with any other option requiring "+ <unsigned integer>" as a parameter or a value. The maximum value of <unsigned integer> is 99999999. This option may be specified independently of the SEQUENCE option.

See also
SEQUENCE (SEQ) Option 301

STATISTICS OPTION

<statistics option>

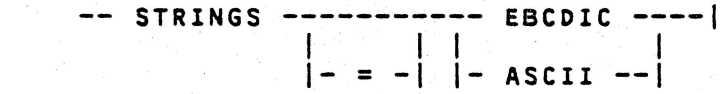


(Type Boolean, Default: FALSE)

The STATISTICS option, when enabled, generates code to gather frequency counts, processor times, and other run-time statistical data about the program.

STRINGS OPTION

<strings option>



(Type: Value, Default: EBCDIC)

The STRINGS option specifies the character set to be used to represent strings in the object program. The STRINGS option must appear before any program text is encountered.

Use of ASCII as the Standard Character Set

The default character set for B 5000/B 6000/B 7000 systems is EBCDIC.

When the STRINGS option is set to ASCII, all internal values of type "char" or "packed array of char" are assumed to be in ASCII. <Character literal>s and <character string>s that appear in the program text are stored in the code file as ASCII values.

External interfaces that require EBCDIC will cause an automatic ASCII-to-EBCDIC or EBCDIC-to-ASCII translation. For example, when "display(s)" is executed, "s" is assumed to have an ASCII internal representation, and the characters in "s" will be translated to EBCDIC for display on the ODT. Similarly, when "accept(s1,s2)" is executed, "s1" is translated to EBCDIC for display on the ODT, and the operator's response is translated to ASCII before it is stored in "s2".

The INTMODE attribute for textfiles is set to ASCII and all values are appropriately translated to ASCII if the associated physical file has an EXTMODE of EBCDIC. Integer and real values in textfiles are handled specially to insure that the correct values are maintained through the translation process.

The INTMODE attribute for files of "packed array of char" is also set to ASCII, so that translation will be performed automatically by the I/O Subsystem when required. The INTMODE of files of records is not set to ASCII, however, because translation would destroy any non-character values in the record. Thus, in order to read or write a file of records, the external representation must match the internal representation. That is, any characters in the physical file must be in ASCII or they will not be interpreted properly when read into a program variable.

USER OPTIONS

<user option>

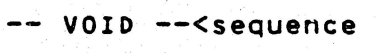


(Type: Boolean, Default: FALSE)

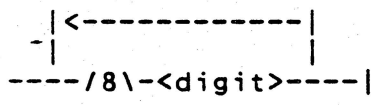
If an identifier on a compiler control record is not recognized as one of the standard options, it is considered to be a <user option>. A <user option> can be manipulated exactly like any other Boolean option; that is, it can be SET, RESET, or POPped. In addition, it can be used as a variable in <option expression>s to assign values to other Boolean options.

VOID OPTION

<void option>



<sequence number>



(Type: immediate)

The VOID option causes all input records in the secondary input file (SOURCE) to be discarded until the secondary input sequence number exceeds the specified <sequence number>. The <sequence number> must be an unsigned integer that is less than or equal to 99999999. The <sequence number> may not be less than the sequence number of the compiler control record on which the VOID option is specified. The VOID option is ignored if the MERGE option has not been enabled.

The source language records discarded as a result of the VOID option are not carried forward to the output symbolic file (NEWSOURCE) when the NEW option is enabled and are not listed.

This option may appear only on a compiler control record in the primary input file (CARD).

See also

MERGE Option. 298
NEW Option. 298

WARNSUPR OPTION

<warnsupr option>

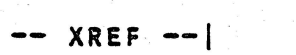


(Type: Boolean, Default: FALSE)

The WARNSUPR option, when enabled, suppresses the printing of warning messages, but does not suppress the printing of syntax errors.

XREF OPTION

<xref option>



(Type: Boolean, Default: FALSE)

The XREF option, when enabled, causes the compiler to generate a printed listing containing cross-reference information. This information includes an alphabetized list of the identifiers that appear in the program and, for each identifier, the structural type of that identifier, the sequence number of the input record on which the identifier was declared, the sequence numbers of the input records on which the identifier was referenced, and other relevant information.

The compiler saves cross-reference information if either the XREF or the XREFFILES option or both options are TRUE. This information is discarded if any syntax errors occur during compilation. The XREF option must appear before any program text is encountered. Once set to TRUE, the XREF option cannot be set to FALSE.

XREFFILES OPTION

<xreffiles option>



(Type: Boolean, Default: FALSE)

The XREFFILES option, when enabled, causes cross-reference information to be saved by the compiler and files containing this information to be generated. These files may be used with either the Editor or INTERACTIVEXREF.

The titles of the generated files are XREFFILES/<code file name>/DECS and XREFFILES/<code file name>/REFS, where <code file name> is the name of the code file the compiler is generating.

The compiler saves cross-reference information if either the XREF or the XREFFILES option or both options are TRUE. This information is discarded if any syntax errors occur during compilation. The XREFFILES option must appear before any program text is encountered. Once set to TRUE, the XREFFILES option cannot be set to FALSE.

When the <xreffiles option> and the <xref option> are both TRUE, both the files and the printed output will be produced.

See also

XREF Option 306

B COMPILER FILES

The Pascal compiler accepts source input from one or more input files. If more than one source file is used, the program text from these files may be merged on the basis of sequence numbers or may be inserted at a specific location regardless of sequence numbers, depending on the programmer's specifications.

The Pascal compiler can generate four types of output files. The compiler generates an object code file if the source input compiles successfully; the object code file is the file that is actually executed when the program is run. If the compilation is initiated through CANDE and there are compile-time errors to report, the compiler will also generate information to be displayed on the CANDE terminal to describe the errors that were detected. The programmer may request the following additional output files: a file containing an updated symbolic and/or a listing of the program text.

Many compiler control options are used to specify the compiler's use of its files. These options are frequently referred to in the following discussion; they are fully described in Appendix A, Compiler Control Records.

COMPILER FILES

Internal File Name	Input/ Output	Usage
CARD	Input	Primary input file for program text; KIND=READER unless file-equated.
SOURCE	Input	Optional secondary input file for program text (refer to the MERGE Option); KIND=DISK unless file-equated.
INCLUDE**	Input	Optional additional input files for program text (refer to the INCLUDE Option); KIND=DISK unless file-equated.
CODE	Output	Object code file; KIND=DISK.
NEWSOURCE	Output	Optional updated program symbolic (refer to the NEW Option); KIND=DISK unless file-equated.
LINE	Output	Optional listing of program text and related compilation information (refer to the LIST Option); KIND=PRINTER unless file-equated. For WFL-initiated compiles, compile-time errors are written to file LINE, whether or not LIST is enabled. The default TRAINID for the LINE file is EBCDIC96.
ERRORS	Output	Optional information about compile-time errors (refer to the ERRORLIST Option). For CANDE-initiated compiles, KIND=REMOTE unless file-equated. For WFL-initiated compiles, KIND=PRINTER

** INCLUDE files do not actually have the internal name "INCLUDE"; there are several INCLUDE files, each with a different name. Titles are supplied using the INCLUDE option syntax.

See also

ERRORLIST Option. 293
INCLUDE Option. 294
LIST Option 296
MERGE Option. 298
NEW Option. 298

INTERACTIVE (CANDE) COMPILATION

When a CANDE workfile of type PASCAL is compiled with the CANDE "COMPILE" command, the compiler reads the workfile as its CARD file. If a SOURCE file is required (because the MERGE compiler control option is TRUE), the programmer is expected to provide the title of the file by including a "FILE" <modifier> on the COMPILE command or by including a <file title> on the compiler control record on which MERGE appears. These two methods are illustrated in the following examples:

```
Compiler Control Record
    in the workfile          CANDE "COMPILE" Command
-----------------------      -----------------------
$ SET MERGE                  COMPILE;
                               PASCAL FILE SOURCE(TITLE=MY/FILE)
$ SET MERGE "MY/FILE"        COMPILE
```

File titles for INCLUDE files are generally specified as part of the INCLUDE compiler control option syntax.

```
Compiler Control Record
    in the workfile          CANDE "COMPILE" Command
-----------------------      -----------------------
$ INCLUDE "MY/INCLUDE/FILE"  COMPILE
$ INCLUDE STD                COMPILE;
                               PASCAL FILE STD(TITLE="STANDARD")
```

When a workfile is compiled through CANDE, the CODE file is given the title "OBJECT/workfile-name" when the workfile is saved. Unless the LIST compiler control option is specifically set to TRUE in the workfile, no program listing will be generated. If the NEW option is set to TRUE in the workfile, a NEWSOURCE file will be created; the programmer is expected to supply the title for the NEWSOURCE file either on the NEW option record or through file equation (if the title is not file-equated, it defaults to NEWSOURCE).

```
Compiler Control Record
    in the workfile          CANDE "COMPILE" Command
-----------------------      -----------------------
$ SET MERGE NEW              COMPILE;
                               PASCAL FILE SOURCE(TITLE=MY/FILE);
                               PASCAL FILE NEWSOURCE(TITLE=MY/NEW)
$ SET NEW "MY/NEWFILE"       COMPILE
```

BATCH (WFL) COMPILATION

When a file is compiled with Pascal through WFL, the programmer must provide a CARD file through file equation. If a SOURCE file is required (because the MERGE option is TRUE), the title can be provided either through file equation or through the MERGE option syntax. If one or more INCLUDE files are used, their titles can be provided through the INCLUDE option syntax. All files, except CARD, are assumed to be KIND=DISK, unless otherwise specified through file equation; CARD is assumed to be KIND=READER unless otherwise specified.

Compiler Control Record in the CARD file	WFL "COMPILE" Statement
$ SET MERGE	COMPILE MY/PROGRAM PASCAL LIBRARY; PASCAL FILE CARD(TITLE=MY/PROG); PASCAL FILE SOURCE(TITLE=SYM/PROG)

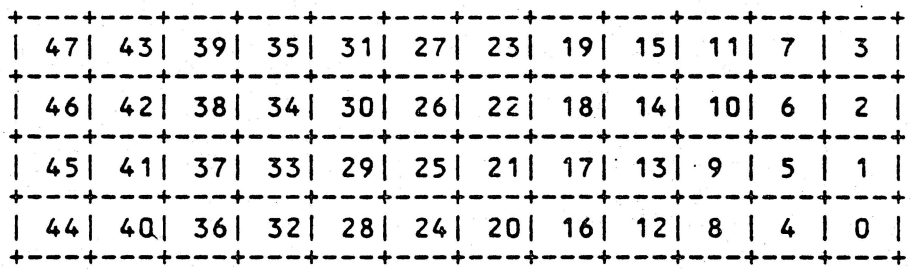
When a program is compiled through WFL, the CODE file is given the title specified in the WFL "COMPILE" statement. Unless the LIST compiler control option is specifically set to FALSE in the CARD file, a program listing will be generated. If the NEW option is set to TRUE in the CARD file, a NEWSOURCE file will be created; the programmer is expected to supply the title for the NEWSOURCE file either on the NEW option record or through file equation (if the title is not file-equated, it defaults to NEWSOURCE).

Compiler Control Record in the CARD file	WFL "COMPILE" Statement
$ SET MERGE NEW	COMPILE MY/PROGRAM PASCAL LIBRARY; PASCAL FILE CARD(TITLE=MY/PROG); PASCAL FILE SOURCE(TITLE=SYM/PROG); PASCAL FILE NEWSOURCE(TITLE=NEW/PROG)

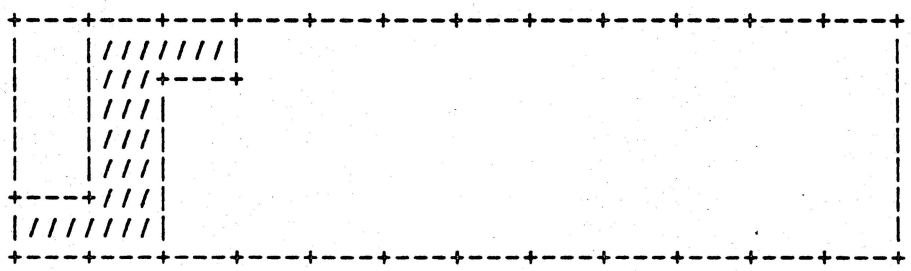
C DATA REPRESENTATION

This section describes the actual machine representation of the data types declarable in Pascal. It is not generally necessary for Pascal programmers to be aware of data structuring at this level. However, when storage space becomes a problem and PACKED arrays, sets, files, and records are being considered, this information can help to determine the amount of space that would be conserved (if any). Also, if a permanent file is created and is to be read by a program written in another language, the details of data representation may be important for designing a file structure that is compatible with both languages.

Data representation is described here in terms of portions and multiples of 48-bit words. The bits within a word are numbered as shown in the following diagram:



A partial word (a "field", in this context) is defined by the bit number of its highest-order bit and the number of bits it comprises, usually written as an "n:m" pair. For example, the notation "44:6" represents the field indicated by "/" characters in the following diagram:



The representation of each type of variable is described separately, in the following order:

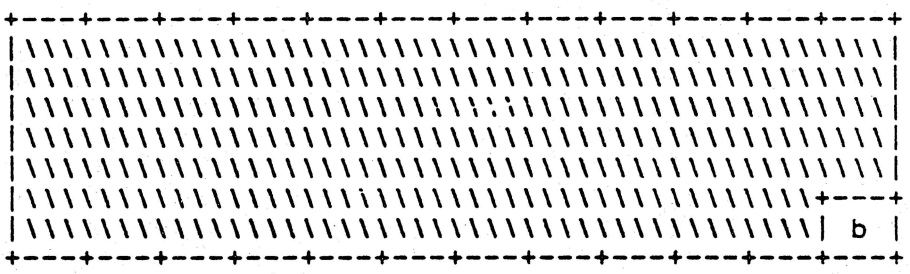
- Simple Types
  - Booleans
  - Characters
  - Enumerations
  - Integers
  - Reals
  - Subranges
- Structured Types
  - Arrays
  - Files
  - Records
  - Sets
  - Textfiles
  - Vlstrings
- Pointers
- Undefined Operands

C.1 SIMPLE TYPES

The data formats described here refer to the representation of items of these data types when the items are not components of PACKED structured types. Their formats within a PACKED data type are described in the section describing the particular structured type.

BOOLEANS

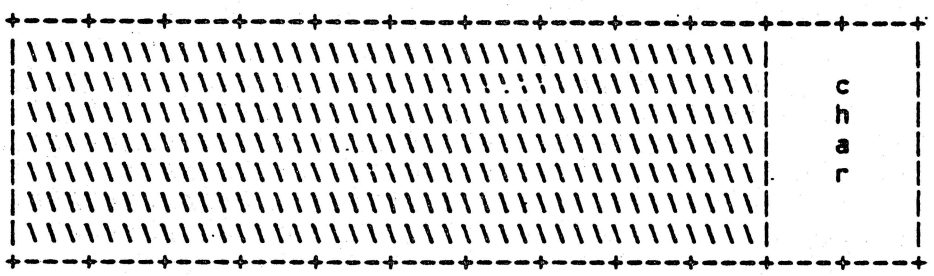
Only the low-order bit (bit 0) is used in a Boolean operand, as shown in the following diagram:



Key	Field	Contents
\\\	47:47	Not used
b	0:1	Boolean value: 0 = false 1 = true

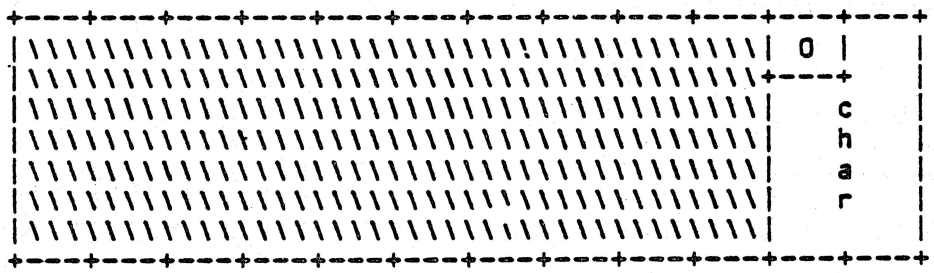
CHARACTERS

Characters are represented for storage as values assigned to fields of bits, regardless of whether they are EBCDIC or ASCII. The followin diagram shows how an EBCDIC character is stored within word:



Key	Field	Contents
\\\	47:40	Not used
char	7:8	The EBCDIC character

An ASCII character is also stored in an 8-bit field, but the high-order bit of the field is stored as 0, as shown in the following diagram:



Key	Field	Contents
\\\	47:40	Not used
0	7:1	Contains zero
char	6:7	The ASCII character

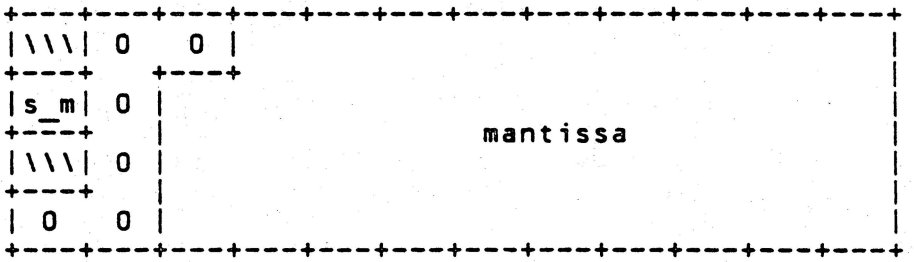
The character codes and ordinal numbers for the ASCII and EBCDIC character sets are described in Appendix D, EBCDIC and ASCII Character Sets.

ENUMERATIONS

An enumerated type of n elements is stored as the subrange 0..n-1, where n is an integer (refer to Subranges below).

INTEGERS

Integer values require a full word of storage and are represented in signed-magnitude notation. The sign of the value is denoted by bit 46 of the data word. This bit is 0 for positive values and 1 for negative values. The magnitude of the value is stored, right-justified, in bits 38 through 0, as illustrated below:



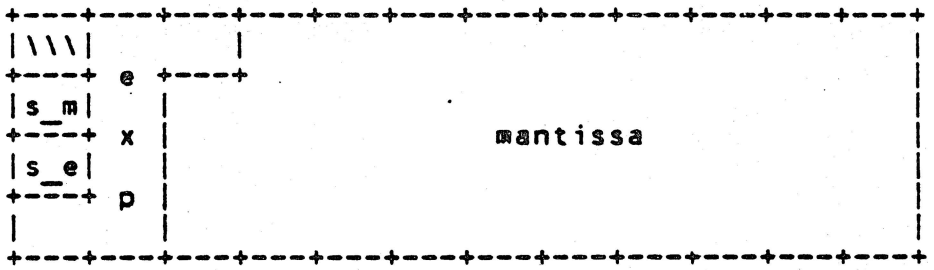
Key	Field	Contents
\\\	47:1	Not used
s_m	46:1	Sign bit: 1 = negative 0 = positive
\\\	45:1	Not used
0	44:6	Contains zero
mantissa	38:39	The magnitude of the number

REALS

Real (floating-point) values require a full word of storage and are represented internally as a signed-magnitude mantissa and an exponent. The value represented by a real data word may be obtained by the following formula:

(mantissa) * 8**(exponent)

The sign of the mantissa is in bit 46, and the sign of the exponent is in bit 45. The exponent (base 8) is contained in bits 44 through 39; thus, the value of the exponent field cannot exceed 2**6 - 1 (63), representing an actual exponent of 8**63. The magnitude of the mantissa is contained in bits 38 through 0, with the radix point assumed to be after bit 0. The word format for a real value is shown in the following diagram:



Key	Field	Contents
\\\	47:1	Not used
s_m	46:1	Sign of mantissa: 1 = negative 0 = positive
s_e	45:1	Sign of exponent: 1 = negative 0 = positive
exp	44:6	Exponent
mantissa	38:39	The magnitude of the number

SUBRANGES

A subrange is stored in the same format as its host type. For example, a subrange of type "char" is stored as a char and a subrange of type "integer" is stored as an integer.

C.2 STRUCTURED TYPES

Structured types contain one or more components of either simple or structured types. Because there may be many components in one variable of a structured type, the specification PACKED is provided to allow the programmer to specify that storage space is to be economized, possibly at the expense of speed in retrieval. PACKED can be specified for arrays, files, records, and sets, but it has no effect for files and sets.

ARRAYS

Unpacked Arrays

Unpacked arrays of simple or pointer types are stored in consecutive words in the same manner that the type would be stored if it were not the component of an array.

In an array of records or packed records, the records are stored as consecutive records. Each record starts on a word boundary and is stored as it would be if it were not in an array.

In an array of sets, the sets are stored as consecutive sets. Each set is stored as it would be if it were not in an array.

In an array of arrays, the component arrays are stored as consecutive arrays (in row-major order), where each array starts on a word boundary. For example, the elements of an array declared as

```
var a : array [1..2, 1..3] of real
```

are stored in the following order:

a[1,1]	a[1,2]	a[1,3]	a[2,1]	a[2,2]	a[2,3]

Packed Arrays

Packed arrays of integers, reals, or pointers are stored the same as unpacked arrays of the same type.

Packed arrays whose component type is a structured type are stored the same as unpacked arrays of the same type.

The following storage rules apply to packed arrays of Boolean, char, subrange, and enumeration:

1. If the type is a subrange, it is biased to 0. (Biasing to 0 means to adjust, by addition or subtraction, the subrange so that the lower bound of the subrange is 0. For example, a subrange 10..20 becomes 0..10 when biased to 0.) A value is stored as its biased value.

2. The representation of the packed array depends on the number of bits (b) it takes to represent the maximum (biased) ordinal value of the type.

b > 8	A full word is used for each element in the array.
4 < b <= 8	A byte (8 bits) is used for each element in the array.
b <= 4	One hex character (4 bits) is used for each element in the array.

FILES

The representation of files is described in detail in the I/O Concepts section. There is no difference in the representation of packed and unpacked files.

See also
Representation of Standard Files. 144

RECORDS

Unpacked Records

In an unpacked record, the fields are stored in consecutive words, in the same manner as each field would be stored if it were not in a record. Each field starts on a word boundary.

Packed Records

An integer, real, or pointer in a packed record is stored in a full word, in the same manner the type would be stored if it were not a field in a record.

A structured type in a packed record starts on the next word boundary. A field following a structured type in a packed record starts on the next word boundary after the structure.

The following storage rules apply to packed records that contain fields of type Boolean, char, subrange, and/or enumeration:

1. If the type is a subrange, it is biased to 0. (Biasing to 0 means to adjust, by addition or subtraction, the subrange so that the lower bound of the subrange is 0. For example, a subrange 10..20 becomes 0..10 when biased to 0.) A value is stored as its biased value.

2. The representation of the packed record depends on the number of bits (b) it takes to represent the maximum (biased) ordinal value of the type.

   If b is greater than 39, the field is stored in the next full word and the value that is stored is not biased.

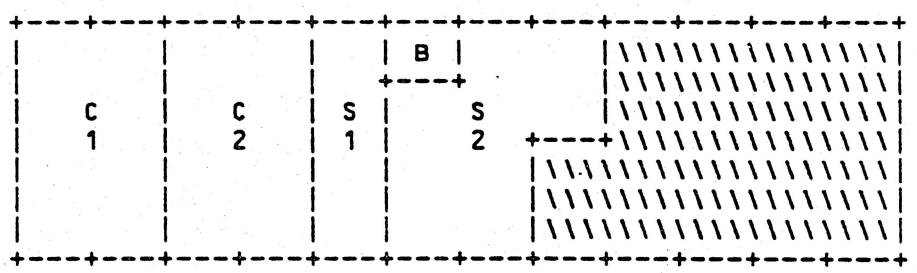
   If b is less than 39, the field is stored in the next available b bits. If fewer than b bits remain in the current word, the field is stored in the left-most b bits of the next word.

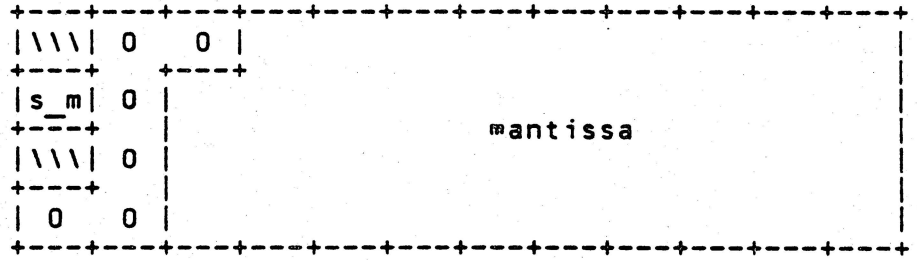
   An ASCII character is stored in 8 bits.

When a record has a variant part, enough storage space is allocated to hold the largest variant, unless the variable is allocated dynamically using the <variant specifier> form of the <new procedure>. In this case, only enough space to hold the selected variants is allocated.

Example of Storage of Packed Records

```
var r : packed record
                c1, c2 : char;
                s1 : 10..20;
                b : Boolean;
                s2 : 0..300;
                i : integer;
                end;
```





	Key	Field	Contents
First Word ->	C1	47:8	C1 (Char)
	C2	39:8	C2 (Char)
	S1	31:4	S1 (Subrange 10..20)
	B	27:1	B (Boolean)
	S2	26:9	S2 (Subrange 0..300)
	\\\	17:18	Not used
Second Word ->	\\\	47:1	Not used
	s_m	46:1	Sign bit: 1 = negative 0 = positive
	\\\	45:1	Not used
	0	44:6	Contains zero
	mantissa	38:39	The magnitude of "i"

SETS

Unpacked Sets

Each element in a set is indicated by a bit in a word. The number of words used for a set is determined as follows:

1. The maximum ordinal value (m) of all elements in the set is determined.

2. The number of words is (m DIV 48) + 1.

Each value from 0 to m is represented by a single bit. If the value of the bit is 0, that element is not in the set. If the value of the bit is 1, that value is in the set. The bit position for a particular element (n) is determined by the following formula:

```
WORD   (n DIV 48) + 1
BIT    (n MOD 48)
```

Sets are stored in consecutive words, 1 to (m DIV 48) + 1, where the first 48 elements of the set are stored in the first word.

Packed Sets

There is no difference in the storage of packed and unpacked sets.

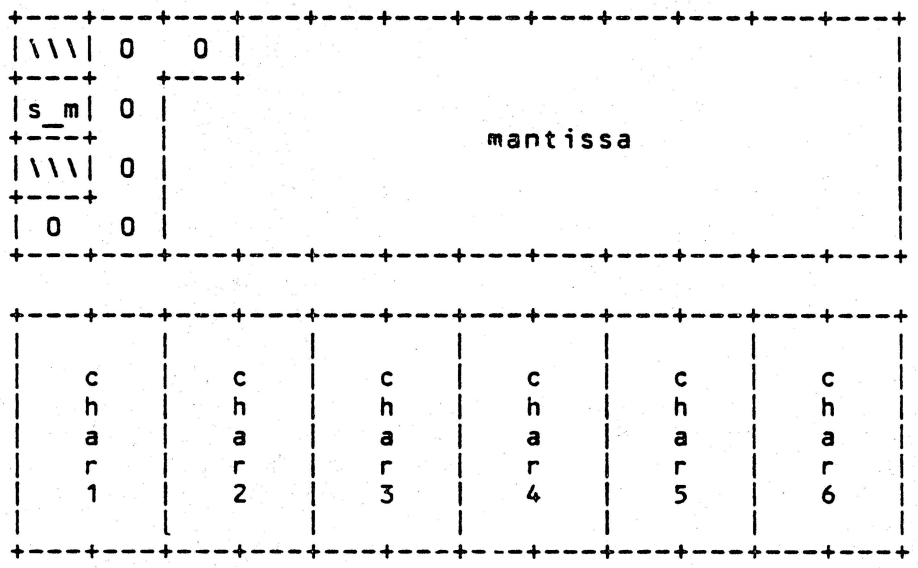
TEXTFILES

The representation of textfiles is described in detail in the I/O Concepts section.

See also
Representation of Textfiles 149

VLSTRINGS

A vlstring is stored in a word array, where the length of the vlstring, in characters, is stored as an integer in the first word, and the character string starts in the first byte of the second word.



	Key	Field	Contents
First Word ->	\\\	47:1	Not used
	s_m	46:1	Sign bit: 0 (always positive)
	\\\	45:1	Not used
	0	44:6	Contains zero
	mantissa	38:39	The length of the vlstring
Second Word ->	char1	47:08	First character of vlstring
	char2	39:08	Second character of vlstring
	char3	31:08	Third character of vlstring
	char4	23:08	Fourth character of vlstring
	char5	15:08	Fifth character of vlstring
	char6	7:08	Sixth character of vlstring

C.3 POINTERS

A pointer takes a full word for storage.

C.4 UNDEFINED OPERANDS

When a variable of a simple type is declared, its tag is initialized to 6. Accessing the value of a tag-6 word causes a run-time error. When a value is stored into the variable, its tag is changed to 0, and the value can safely be accessed.

D EBCDIC AND ASCII CHARACTER SETS

The tables below list the hexadecimal representation and ordinal number for each ASCII and EBCDIC character. The first table is sorted by EBCDIC ordinal number and represents the EBCDIC-to-ASCII translation that is performed when necessary. The second table is sorted by ASCII ordinal number and represents the ASCII-to-EBCDIC translation that is performed when necessary.

E B C D I C		A S C I I			
Hex Code	Ordinal Number	Hex Code	Ordinal Number		
00	0	00	0	NUL	Null
01	1	01	1	SOH	Start of Heading
02	2	02	2	STX	Start of Text
03	3	03	3	ETX	End of Text
04	4	9C	156		
05	5	09	9	HT	Horizontal Tabulation
06	6	86	134		
07	7	7F	127	DEL	Delete
08	8	97	151		
09	9	8D	141		
0A	10	8E	142		
0B	11	0B	11	VT	Vertical Tabulation
0C	12	0C	12	FF	Form Feed
0D	13	0D	13	CR	Carriage Return
0E	14	0E	14	SO	Shift Out
0F	15	0F	15	SI	Shift In
10	16	10	16	DLE	Data Link Escape
11	17	11	17	DC1	Device Control 1
12	18	12	18	DC2	Device Control 2
13	19	13	19	DC3	Device Control 3
14	20	9D	157		
15	21	85	133		
16	22	08	8	BS	Backspace
17	23	87	135		
18	24	18	24	CAN	Cancel
19	25	19	25	EM	End of Medium
1A	26	92	146		
1B	27	8F	143		
1C	28	1C	28	FS	File Separator
1D	29	1D	29	GS	Group Separator
1E	30	1E	30	RS	Record Separator
1F	31	1F	31	US	Unit Separator
20	32	80	128		
21	33	81	129		
22	34	82	130		
23	35	83	131		
24	36	84	132		
25	37	0A	10	LF	Line Feed

PASCAL / EBCDIC & ASCII Character Sets

E B C D I C		A S C I I			
Hex Code	Ordinal Number	Hex Code	Ordinal Number		
26	38	17	23	ETB	End of Transmission Block
27	39	1B	27	ESC	Escape
28	40	88	136		
29	41	89	137		
2A	42	8A	138		
2B	43	8B	139		
2C	44	8C	140		
2D	45	05	5	ENQ	Enquiry
2E	46	06	6	ACK	Acknowledge
2F	47	07	7	BEL	Bell
30	48	90	144		
31	49	91	145		
32	50	16	22	SYN	Synchronous Idle
33	51	93	147		
34	52	94	148		
35	53	95	149		
36	54	96	150		
37	55	04	4	EOT	End of Transmission
38	56	98	152		
39	57	99	153		
3A	58	9A	154		
3B	59	9B	155		
3C	60	14	20	DC4	Device Control 4
3D	61	15	21	NAK	Negative Acknowledge
3E	62	9E	158		
3F	63	1A	26	SUB	Substitute
40	64	20	32	SP	Space
41	65	A0	160		
42	66	A1	161		
43	67	A2	162		
44	68	A3	163		
45	69	A4	164		
46	70	A5	165		
47	71	A6	166		
48	72	A7	167		
49	73	A8	168		
4A	74	5B	91	[	Opening Bracket
4B	75	2E	46	.	Period
4C	76	3C	60	<	Less Than
4D	77	28	40	(	Opening Parenthesis
4E	78	2B	43	+	Plus
4F	79	21	33	!	Exclamation Point
50	80	26	38	&	Ampersand
51	81	A9	169		
52	82	AA	170		
53	83	AB	171		
54	84	AC	172		
55	85	AD	173		

PASCAL / EBCDIC & ASCII Character Sets

E B C D I C		A S C I I			
Hex Code	Ordinal Number	Hex Code	Ordinal Number		
56	86	AE	174		
57	87	AF	175		
58	88	B0	176		
59	89	B1	177		
5A	90	5D	93	]	Closing Bracket
5B	91	24	36	$	Dollar Sign
5C	92	2A	42	*	Asterisk
5D	93	29	41	)	Closing Parenthesis
5E	94	3B	59	;	Semicolon
5F	95	5E	94		Circumflex (ASCII); Not Sign (EBCDIC)
60	96	2D	45	-	Hyphen (Minus)
61	97	2F	47	/	Slant (Slash)
62	98	B2	178		
63	99	B3	179		
64	100	B4	180		
65	101	B5	181		
66	102	B6	182		
67	103	B7	183		
68	104	B8	184		
69	105	B9	185		
6A	106	7C	124	|	Vertical Line
6B	107	2C	44	,	Comma
6C	108	25	37	%	Percent
6D	109	5F	95		Underline
6E	110	3E	62	>	Greater Than
6F	111	3F	63	?	Question Mark
70	112	BA	186		
71	113	BB	187		
72	114	BC	188		
73	115	BD	189		
74	116	BE	190		
75	117	BF	191		
76	118	C0	192		
77	119	C1	193		
78	120	C2	194		
79	121	40	96	`	Grave Accent (Opening Single Quote Mark)
7A	122	3A	58	:	Colon
7B	123	23	35	#	Number Sign
7C	124	60	64	@	Commercial At
7D	125	27	39	'	Apostrophe (Closing Single Quotation Mark)
7E	126	3D	61	=	Equals
7F	127	22	34	"	Quotation Marks
80	128	C3	195		
81	129	61	97	a	Lowercase a
82	130	62	98	b	Lowercase b

PASCAL / EBCDIC & ASCII Character Sets

EBCDIC Hex Code	EBCDIC Ordinal Number	ASCII Hex Code	ASCII Ordinal Number		
83	131	63	99	c	Lowercase c
84	132	64	100	d	Lowercase d
85	133	65	101	e	Lowercase e
86	134	66	102	f	Lowercase f
87	135	67	103	g	Lowercase g
88	136	68	104	h	Lowercase h
89	137	69	105	i	Lowercase i
8A	138	C4	196		
8B	139	C5	197		
8C	140	C6	198		
8D	141	C7	199		
8E	142	C8	200		
8F	143	C9	201		
90	144	CA	202		
91	145	6A	106	j	Lowercase j
92	146	6B	107	k	Lowercase k
93	147	6C	108	l	Lowercase l
94	148	6D	109	m	Lowercase m
95	149	6E	110	n	Lowercase n
96	150	6F	111	o	Lowercase o
97	151	70	112	p	Lowercase p
98	152	71	113	q	Lowercase q
99	153	72	114	r	Lowercase r
9A	154	CB	203		
9B	155	CC	204		
9C	156	CD	205		
9D	157	CE	206		
9E	158	CF	207		
9F	159	D0	208		
A0	160	D1	209		
A1	161	7E	126	~	Overline (Tilde)
A2	162	73	115	s	Lowercase s
A3	163	74	116	t	Lowercase t
A4	164	75	117	u	Lowercase u
A5	165	76	118	v	Lowercase v
A6	166	77	119	w	Lowercase w
A7	167	78	120	x	Lowercase x
A8	168	79	121	y	Lowercase y
A9	169	7A	122	z	Lowercase z
AA	170	D2	210		
AB	171	D3	211		
AC	172	D4	212		
AD	173	D5	213		
AE	174	D6	214		
AF	175	D7	215		
B0	176	D8	216		
B1	177	D9	217		
B2	178	DA	218		

PASCAL / EBCDIC & ASCII Character Sets

E B C D I C		A S C I I			
Hex Code	Ordinal Number	Hex Code	Ordinal Number		
B3	179	DB	219		
B4	180	DC	220		
B5	181	DD	221		
B6	182	DE	222		
B7	183	DF	223		
B8	184	E0	224		
B9	185	E1	225		
BA	186	E2	226		
BB	187	E3	227		
BC	188	E4	228		
BD	189	E5	229		
BE	190	E6	230		
BF	191	E7	231		
C0	192	7B	123	{	Opening Brace
C1	193	41	65	A	Uppercase A
C2	194	42	66	B	Uppercase B
C3	195	43	67	C	Uppercase C
C4	196	44	68	D	Uppercase D
C5	197	45	69	E	Uppercase E
C6	198	46	70	F	Uppercase F
C7	199	47	71	G	Uppercase G
C8	200	48	72	H	Uppercase H
C9	201	49	73	I	Uppercase I
CA	202	E8	232		
CB	203	E9	233		
CC	204	EA	234		
CD	205	EB	235		
CE	206	EC	236		
CF	207	ED	237		
D0	208	7D	125	}	Closing Brace
D1	209	4A	74	J	Uppercase J
D2	210	4B	75	K	Uppercase K
D3	211	4C	76	L	Uppercase L
D4	212	4D	77	M	Uppercase M
D5	213	4E	78	N	Uppercase N
D6	214	4F	79	O	Uppercase O
D7	215	50	80	P	Uppercase P
D8	216	51	81	Q	Uppercase Q
D9	217	52	82	R	Uppercase R
DA	218	EE	238		
DB	219	EF	239		
DC	220	F0	240		
DD	221	F1	241		
DE	222	F2	242		
DF	223	F3	243		
E0	224	5C	92	\	Reverse Slant
E1	225	9F	159		
E2	226	53	83	S	Uppercase S

PASCAL / EBCDIC & ASCII Character Sets

E B C D I C		A S C I I			
Hex Code	Ordinal Number	Hex Code	Ordinal Number		
E3	227	54	84	T	Uppercase T
E4	228	55	85	U	Uppercase U
E5	229	56	86	V	Uppercase V
E6	230	57	87	W	Uppercase W
E7	231	58	88	X	Uppercase X
E8	232	59	89	Y	Uppercase Y
E9	233	5A	90	Z	Uppercase Z
EA	234	F4	244		
EB	235	F5	245		
EC	236	F6	246		
ED	237	F7	247		
EE	238	F8	248		
EF	239	F9	249		
F0	240	30	48	0	Zero
F1	241	31	49	1	One
F2	242	32	50	2	Two
F3	243	33	51	3	Three
F4	244	34	52	4	Four
F5	245	35	53	5	Five
F6	246	36	54	6	Six
F7	247	37	55	7	Seven
F8	248	38	56	8	Eight
F9	249	39	57	9	Nine
FA	250	FA	250		
FB	251	FB	251		
FC	252	FC	252		
FD	253	FD	253		
FE	254	FE	254		
FF	255	FF	255		

PASCAL / EBCDIC & ASCII Character Sets

A S C I I		E B C D I C			
Hex Code	Ordinal Number	Hex Code	Ordinal Number		
00	0	00	0	NUL	Null
01	1	01	1	SOH	Start of Heading
02	2	02	3	STX	Start of Text
03	3	03	4	ETX	End of Text
04	4	37	55	EOT	End of Transmission
05	5	2D	45	ENQ	Enquiry
06	6	2E	46	ACK	Acknowledge
07	7	2F	47	BEL	Bell
08	8	16	22	BS	Backspace
09	9	05	5	HT	Horizontal Tabulation
0A	10	25	37	LF	Line Feed
0B	11	0B	11	VT	Vertical Tabulation
0C	12	0C	12	FF	Form Feed
0D	13	0D	13	CR	Carriage Return
0E	14	0E	14	SO	Shift Out
0F	15	0F	15	SI	Shift In
10	16	10	16	DLE	Data Link Escape
11	17	11	17	DC1	Device Control 1
12	18	12	18	DC2	Device Control 2
13	19	13	19	DC3	Device Control 3
14	20	3C	60	DC4	Device Control 4
15	21	3D	61	NAK	Negative Acknowledge
16	22	32	50	SYN	Synchronous Idle
17	23	26	38	ETB	End of Transmission Block
18	24	18	24	CAN	Cancel
19	25	19	25	EM	End of Medium
1A	26	3F	63	SUB	Substitute
1B	27	27	39	ESC	Escape
1C	28	1C	28	FS	File Separator
1D	29	1D	29	GS	Group Separator
1E	30	1E	30	RS	Record Separator
1F	31	1F	31	US	Unit Separator
20	32	40	64	SP	Space
21	33	4F	79	!	Exclamation Point
22	34	7F	127	"	Quotation Marks
23	35	7B	123	#	Number Sign
24	36	5B	91	$	Dollar Sign
25	37	6C	108	%	Percent
26	38	50	80	&	Ampersand
27	39	7D	125	'	Apostrophe (Closing Single Quotation Mark)
28	40	4D	77	(	Opening Parenthesis
29	41	5D	93	)	Closing Parenthesis
2A	42	5C	92	*	Asterisk
2B	43	4E	78	+	Plus
2C	44	6B	107	,	Comma
2D	45	60	96	-	Hyphen (Minus)
2E	46	4B	75	.	Period

PASCAL / EBCDIC & ASCII Character Sets

A S C I I		E B C D I C			
Hex Code	Ordinal Number	Hex Code	Ordinal Number		
2F	47	61	97	/	Slant (Slash)
30	48	F0	240	0	Zero
31	49	F1	241	1	One
32	50	F2	242	2	Two
33	51	F3	243	3	Three
34	52	F4	244	4	Four
35	53	F5	245	5	Five
36	54	F6	246	6	Six
37	55	F7	247	7	Seven
38	56	F8	248	8	Eight
39	57	F9	249	9	Nine
3A	58	7A	122	:	Colon
3B	59	5E	94	;	Semicolon
3C	60	4C	76	<	Less Than
3D	61	7E	126	=	Equals
3E	62	6E	110	>	Greater Than
3F	63	6F	111	?	Question Mark
40	64	7C	124	@	Commercial At
41	65	C1	193	A	Uppercase A
42	66	C2	194	B	Uppercase B
43	67	C3	195	C	Uppercase C
44	68	C4	196	D	Uppercase D
45	69	C5	197	E	Uppercase E
46	70	C6	198	F	Uppercase F
47	71	C7	199	G	Uppercase G
48	72	C8	200	H	Uppercase H
49	73	C9	201	I	Uppercase I
4A	74	D1	209	J	Uppercase J
4B	75	D2	210	K	Uppercase K
4C	76	D3	211	L	Uppercase L
4D	77	D4	212	M	Uppercase M
4E	78	D5	213	N	Uppercase N
4F	79	D6	214	O	Uppercase O
50	80	D7	215	P	Uppercase P
51	81	D8	216	Q	Uppercase Q
52	82	D9	217	R	Uppercase R
53	83	E2	226	S	Uppercase S
54	84	E3	227	T	Uppercase T
55	85	E4	228	U	Uppercase U
56	86	E5	229	V	Uppercase V
57	87	E6	230	W	Uppercase W
58	88	E7	231	X	Uppercase X
59	89	E8	232	Y	Uppercase Y
5A	90	E9	233	Z	Uppercase Z
5B	91	4A	74	[	Opening Bracket
5C	92	E0	224	\	Reverse Slant
5D	93	5A	90	]	Closing Bracket

PASCAL / EBCDIC & ASCII Character Sets

A S C I I Hex Code	A S C I I Ordinal Number	E B C D I C Hex Code	E B C D I C Ordinal Number		
5E	94	5F	95		Circumflex (ASCII); Not Sign (EBCDIC)
5F	95	6D	109	_	Underline
60	96	79	121	`	Grave Accent (Opening Single Quote Mark)
61	97	81	129	a	Lowercase a
62	98	82	130	b	Lowercase b
63	99	83	131	c	Lowercase c
64	100	84	132	d	Lowercase d
65	101	85	133	e	Lowercase e
66	102	86	134	f	Lowercase f
67	103	87	135	g	Lowercase g
68	104	88	136	h	Lowercase h
69	105	89	137	i	Lowercase i
6A	106	91	145	j	Lowercase j
6B	107	92	146	k	Lowercase k
6C	108	93	147	l	Lowercase l
6D	109	94	148	m	Lowercase m
6E	110	95	149	n	Lowercase n
6F	111	96	150	o	Lowercase o
70	112	97	151	p	Lowercase p
71	113	98	152	q	Lowercase q
72	114	99	153	r	Lowercase r
73	115	A2	162	s	Lowercase s
74	116	A3	163	t	Lowercase t
75	117	A4	164	u	Lowercase u
76	118	A5	165	v	Lowercase v
77	119	A6	166	w	Lowercase w
78	120	A7	167	x	Lowercase x
79	121	A8	168	y	Lowercase y
7A	122	A9	169	z	Lowercase z
7B	123	C0	192	{	Opening Brace
7C	124	6A	106	|	Vertical Line
7D	125	D0	208	}	Closing Brace
7E	126	A1	161	~	Overline (Tilde)
7F	127	07	7	DEL	Delete
80	128	20	32		
81	129	21	33		
82	130	22	34		
83	131	23	35		
84	132	24	36		
85	133	15	21		
86	134	06	6		
87	135	17	23		
88	136	28	40		
89	137	29	41		
8A	138	2A	42		
8B	139	2B	43		

PASCAL / EBCDIC & ASCII Character Sets

A S C I I		E B C D I C	
Hex Code	Ordinal Number	Hex Code	Ordinal Number
8C	140	2C	44
8D	141	09	9
8E	142	0A	10
8F	143	1B	27
90	144	30	48
91	145	31	49
92	146	1A	26
93	147	33	51
94	148	34	52
95	149	35	53
96	150	36	54
97	151	08	8
98	152	38	56
99	153	39	57
9A	154	3A	58
9B	155	3B	59
9C	156	04	4
9D	157	14	20
9E	158	3E	62
9F	159	E1	225
A0	160	41	65
A1	161	42	66
A2	162	43	67
A3	163	44	68
A4	164	45	69
A5	165	46	70
A6	166	47	71
A7	167	48	72
A8	168	49	73
A9	169	51	81
AA	170	52	82
AB	171	53	83
AC	172	54	84
AD	173	55	85
AE	174	56	86
AF	175	57	87
B0	176	58	88
B1	177	59	89
B2	178	62	98
B3	179	63	99
B4	180	64	100
B5	181	65	101
B6	182	66	102
B7	183	67	103
B8	184	68	104
B9	185	69	105
BA	186	70	112
BB	187	71	113

PASCAL / EBCDIC & ASCII Character Sets

A S C I I		E B C D I C	
Hex Code	Ordinal Number	Hex Code	Ordinal Number
BC	188	72	114
BD	189	73	115
BE	190	74	116
BF	191	75	117
C0	192	76	118
C1	193	77	119
C2	194	78	120
C3	195	80	128
C4	196	8A	138
C5	197	8B	139
C6	198	8C	140
C7	199	8D	141
C8	200	8E	142
C9	201	8F	143
CA	202	90	144
CB	203	9A	154
CC	204	9B	155
CD	205	9C	156
CE	206	9D	157
CF	207	9E	158
D0	208	9F	159
D1	209	A0	160
D2	210	AA	170
D3	211	AB	171
D4	212	AC	172
D5	213	AD	173
D6	214	AE	174
D7	215	AF	175
D8	216	B0	176
D9	217	B1	177
DA	218	B2	178
DB	219	B3	179
DC	220	B4	180
DD	221	B5	181
DE	222	B6	182
DF	223	B7	183
E0	224	B8	184
E1	225	B9	185
E2	226	BA	186
E3	227	BB	187
E4	228	BC	188
E5	229	BD	189
E6	230	BE	190
E7	231	BF	191
E8	232	CA	202
E9	233	CB	203
EA	234	CC	204
EB	235	CD	205

PASCAL / EBCDIC & ASCII Character Sets

A S C I I		E B C D I C	
Hex Code	Ordinal Number	Hex Code	Ordinal Number
EC	236	CE	206
ED	237	CF	207
EE	238	DA	218
EF	239	DB	219
F0	240	DC	220
F1	241	DD	221
F2	242	DE	222
F3	243	DF	223
F4	244	EA	234
F5	245	EB	235
F6	246	EC	236
F7	247	ED	237
F8	248	EE	238
F9	249	EF	239
FA	250	FA	250
FB	251	FB	251
FC	252	FC	252
FD	253	FD	253
FE	254	FE	254
FF	255	FF	255

E COMPARISON WITH ANSI PASCAL

This section describes the relationship of Burroughs Pascal to ANSI Pascal. In particular, specifications are given for implementation-defined features and implementation-dependent features. Also, Burroughs extensions to ANSI Pascal are summarized.

The violations of the requirements of ANSI Pascal and the ANSI Pascal features that are not implemented in Burroughs Pascal are described in the Compliance Statement in the Introduction to this manual. A list of errors defined in ANSI Pascal that are not detected in Burroughs Pascal is given in Appendix F, Error Detection and Reporting.

See also

Compliance Statement. 2
Error Detection and Reporting 347

IMPLEMENTATION-DEFINED FEATURES

The following list of "implementation-defined" features was derived from occurrences of the term "implementation-defined" in the ANSI Standard. The term "implementation-defined" is used to mean possibly differing between implementations, but defined for any particular implementation.

The numbers in brackets refer to sections in the ANSI Standard.

[6.1.7] Char type values of string characters.

See "[6.4.2.2] Values and ordinal values of the type char" defined below.

[6.4.2.2] Values of the type real.

The range of real values is +/-4.31359146673E68. The smallest positive real value is 8.75811540203E-47.

[6.4.2.2] Values and ordinal values of the type char.

The values and ordinal values of the type char consist of either the EBCDIC or 8-bit ASCII character sets (see Appendix D, EBCDIC and ASCII Character Sets). The default character set is EBCDIC. The default character set can be overridden with the compiler control option STRINGS (see Appendix A, Compiler Control Records).

See also
Compiler Control Records. 285
EBCDIC and ASCII Character Sets 327

[6.6.5.2] Activities corresponding to the post assertions which are defined for the file handling procedures, and the point at which they occur.

These activities are described in the File Handling Procedures and Functions section.

See also
File Handling Procedures and Functions. 132

[6.7.2.2] The value of the required constant identifier "maxint".

The value of "maxint" is 549,755,813,887.

[6.7.2.2] Accuracy of real arithmetic operators and functions.

Thirteen octal digits of precision are maintained in the B 5000/B 6000/B 7000 floating-point representation of a real number. Intermediate results in a floating-point operation (+, -, *, /) requiring more than thirteen digits are rounded to thirteen digits. The "round-off" error thus incurred can be described as follows:

Let "x op y" represent the exact result of performing the operation "op" on the operands "x" and "y", and let "fl(x op y)" represent the result of the floating-point computation. "fl(x op y) = (x op y)*(1+E)" where "abs(E) <= (8**-12) / 2" (that is, the absolute value of epsilon is less than or equal to one half of the quantity 8 to the -12 power). The accuracy of the predefined functions "sin", "cos", "exp", etc., is affected both by accumulated round-off errors and by the approximation methods used in their computation. These factors vary with the system intrinsics used to provide these functions.

[6.9.4.2] The default value of totalwidth when writing expressions of type real, integer, or Boolean.

The default field widths are real = 15, integer = 10, and Boolean = 5.

[6.9.4.5.1] The total number of digit characters written in an exponent.

The total number of digit characters written in an exponent is 2.

[6.9.4.5.1] The exponent character used when writing a real number.

The exponent character is "E".

[6.9.4.6] The case of "true" or "false" when writing Boolean values.

Uppercase is used for type Boolean output.

[6.9.6] The effect of page (f) on a textfile f.

The standard procedure page (f) is defined to be equivalent to skiptochannel (f,1).

See also
Page Procedure. 174
Skiptochannel Procedure 188

[6.10] The binding of file type program parameters to external entities.

The association of logical and physical files is described in the I/O Concepts section under the heading "Use of File Attributes".

See also
Use of File Attributes. 152

[6.10] The effect of an explicit reset or rewrite on the standard files "input" or "output".

An explicit reset or rewrite is allowed on the standard files "input" or "output" with the same effect which would occur for ordinary files. However, there may be system restrictions depending upon the particular device associated with a file.

IMPLEMENTATION-DEPENDENT FEATURES

All features defined to be "implementation-dependent" by the ANSI Standard are listed below. The term "implementation-dependent" is used to mean possibly differing between implementations, but not necessarily defined for any given implementation.

The numbers in brackets refer to sections in the ANSI Standard.

[6.1.4] The directives, other than forward, permitted as a replacement for a procedure block or function block.

No other directives are permitted.

[6.7.2.1] The evaluation of the operands of a dyadic operator.

The order of evaluation of operands of a dyadic operator is undefined.

[6.7.3] The order of evaluation and binding of actual parameters for a function designator.

The order of evaluation of the actual parameters of a function designator is undefined.

[6.8.2.2] The order of evaluation of the variable on the left-hand side and the expression on the right-hand side of an assignment statement.

The order of evaluation of the variable and expression of an assignment statement is undefined.

[6.8.2.3] The order of evaluation and binding of actual parameters for a procedure statement.

The order of evaluation of the actual parameters of a procedure statement is undefined.

[6.9.6] The effect of inspecting a textfile to which the page procedure was applied during generation.

The effect of inspecting a textfile to which the "page" procedure was applied is the same as the effect of inspecting a textfile to which a "writeln" procedure was applied.

See also
Page Procedure. 174
Writeln Procedure 195

[6.10] The binding of non-file type program parameters to external entities.

Non-file-type program parameters are not allowed.

EXTENSIONS TO STANDARD PASCAL

The following features are Burroughs extensions to ANSI Pascal:

1) The underscore (_) as a significant character in <identifier>s.

2) The OTHERWISE clause in the <case statement>. The word OTHERWISE has been added to the list of reserved words.

3) The file attribute specification for program parameters.

4) The conditional Boolean operators CAND and COR. The words CAND and COR have been added to the list of reserved words.

5) The file handling procedures and functions "addstation", "close", "deletestation", "filevalue", "iores", "open", "seek", and "skiptochannel".

6) The <ordinal type transfer function>.

7) The dynamic allocation procedures "mark" and "release".

8) Libraries, and the library handling procedures and functions "cancel", "freeze", and "libraryvalue".

9) The string handling procedures and functions "concat", "delete", "fillchar", "insert", "length", "pos", "scan_eql", "scan_neq", and "string".

10) The arithmetic functions "arccos", "arcsin", "arctanh", "cosh", "cotan", "erf", "gamma", "lngamma", "log", "max", "min", "random", "sinh", "tan", and "tanh".

11) The general procedures and functions "abort", "accept", "date", "display", "elapsedtime", "getattribute", "iotime", "runtime", "setattribute", "time", and "wait".

12) Vlstring types as a structured type.

13) The ability to handle exception conditions on I/O operations.

14) Additional options in the <get procedure> and <put procedure>.

15) The concepts of <subfile variable>s and <station variable>s.

F ERROR DETECTION AND REPORTING

One area in which Pascal compilers tend to differ significantly is in the detection and reporting of errors. An "error" in this context is one of the following occurrences:

a) A violation of the defined syntax rules of Pascal. For example, it is an error to omit the word "DO" following the <Boolean expression> in a WHILE statement.

b) A violation of the defined semantic rules of Pascal. For example, it is an error to specify "11" as the index of an array when the valid index range was declared to be "1..10".

c) An attempt to exceed a system architecture or resource limitation. For example, it is an error to attempt to store a real number that is larger than the maximum real number that the system can represent.

There are several ways that an error can be detected and reported:

a) The compiler may detect the error at compile time. The compiler has enough information to detect all syntax errors, some semantic errors, and a few system errors at compile time.

b) The compiler may generate code to detect the error at run time. This method is used to detect semantic errors that the compiler cannot detect at compile time, usually because the construct in error is a variable whose value is not available at compile time.

c) The system may detect the error at run time. This method is commonly used for semantic errors of a general nature (such as an attempt to divide by zero, which is invalid in all languages). It is also the usual method for detecting system architecture and resource errors.

Errors that occur at compile time cause a message to be displayed and the code file to be discarded at the end of the compilation. Errors that occur at run time cause a message to be displayed and the program to be terminated. Note that the compile-time and run-time messages for similar errors may differ. Usually, the compiler's messages are more specific and complete, because the compiler has more contextual information at its disposal than is available at run time.

Some semantic errors are not detected at all. Even if the error is not detected, the program is invalid in that aspect, and the results of the program are undefined. Known violations of Pascal semantics should be avoided because different implementations detect different errors and

because later versions of the same implementation may be able to detect the error or may change the results without notification.

The following errors are not currently detected by the Burroughs Pascal Compiler:

ANSI Standard Reference	Error Not Detected
[6.4.6]	A set value cannot be assigned to a variable of a certain <set type> or passed as a value parameter to a formal parameter of that <set type> if it contains a member that is not included in that <set type>. This error is detected only when the <set type> contains 48 elements or fewer.
[6.5.3.3]	A field of a <variant> in the <variant part> of a record cannot be accessed if it is not "active" (please refer to Record Types). Also, if a reference to a field in a <variant> is established, the active <variant> cannot be changed as long as that reference exists. A reference, in this context, is a usage of the field in one of the following ways: a) as the left-hand side of an <assignment statement>. b) as an actual variable parameter. c) as the <record variable> of a <with statement>.
[6.5.4]	A dynamic variable cannot be deallocated, either through the <dispose procedure> or through the <release procedure>, while a reference to that variable exists. A reference, in this context, is a use of its <pointer variable> (for example, "p@") in one of the ways defined above (refer to 6.5.5.3).
[6.5.5]	The position of a file cannot be changed while a reference to the buffer variable of that file exists. A reference, in this context, is a use of the file's buffer variable in one of the ways defined above (refer to 6.5.5.3).

[6.6.5.2] The "put" procedure cannot be called when the value of the buffer variable is undefined. This error is currently detected only for textfiles.

[6.6.5.3] When the <case constant> form of the "new" procedure is used to allocate a dynamic variable, only the <variant>s specified by those <case constant>s can be made "active". That is, a new variant cannot subsequently be activated.

[6.6.5.3] When the <case constant> form of the "new" procedure is used to allocate a dynamic variable, the same <case constant> list must be used in the "dispose" procedure if and when that variable is deallocated through the "dispose" procedure.

[6.6.5.3] When the <case constant> form of the "new" procedure is used to allocate a dynamic variable, that variable cannot be used as a whole. That is, if "p" is a pointer to the variable, "p@" cannot be used in an <assignment statement>, parameter list, or expression. However, a component of the variable can be referenced (for example, "p@.f" is valid if "f" is a field of that variable).

[6.7.1] A variable cannot be used in an expression when its value is undefined. Many, but not all, of these cases are detected.

G RAILROAD DIAGRAMS

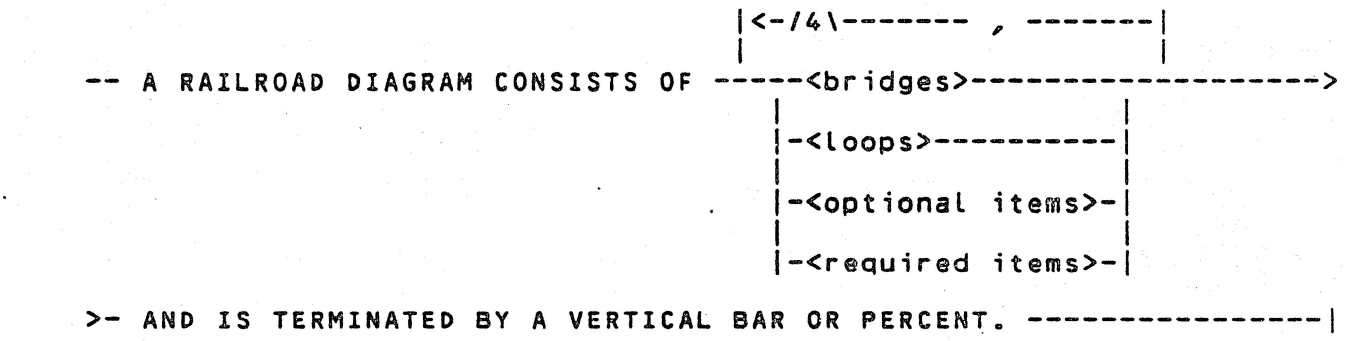
Railroad diagrams graphically represent the syntax of software commands.

The railroad diagrams are traversed left to right or in the direction of the arrowhead. Adherence to the limits illustrated by bridges produces a syntactically valid statement. Continuation from one line of a diagram to another is represented by a right arrow (>) appearing at the end of the current line and the beginning of the next line. The complete syntax diagram is terminated by a vertical bar (|).

Items contained in broken brackets (<>) are syntactic variables that are defined in the manual or are information that the user is required to supply.

Uppercase items not enclosed in broken brackets must appear literally; the minimum abbreviations are underlined.

Example



The following are some of the syntactically valid constructs that may be generated from the preceding diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <optional items> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

A RAILROAD DIAGRAM CONSISTS OF <optional items>, <required items>, <optional items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR OR PERCENT.

Railroad Components

<required items>

No alternate path through the railroad diagram exists for required items or required punctuation.

Example

```
-- REQUIRED ITEM -- . --|
```

<optional items>

Items shown as a vertical list indicate that the user may make a choice of the items specified. An empty path through the list allows the optional item to be absent.

Example

```
-- REQUIRED ITEM ------------------------|
                 |                       |
                 |-<optional item-1>-|
                 |                       |
                 |-<optional item-2>-|
```

The following valid constructs may be generated from the preceding diagram:

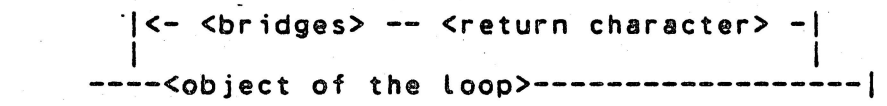
REQUIRED ITEM

REQUIRED ITEM <optional item-1>

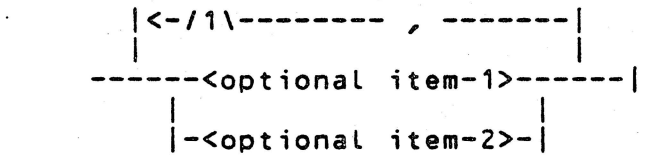
REQUIRED ITEM <optional item-2>

<loops>

A loop is a recurrent path through a railroad diagram and has the following general format:



Example



The following are some of the valid constructs that may be generated from the preceding diagram:

<optional item-1>

<optional item-1>,<optional item-1>

<optional item-2>,<optional item-1>

The loop must be traversed in the direction in which the arrowhead points, and the limits specified by bridges cannot be exceeded.

<bridges>

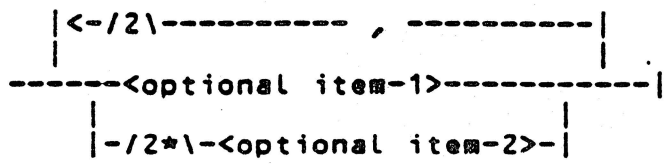
A bridge illustrates the minimum or maximum number of times a path may be traversed in a railroad diagram.

Two forms of bridges exist:

/n\ n is an integer that specifies the maximum number of times the path may be traversed.

/n*\ n is an integer that specifies the maximum number of times the path may be traversed. The asterisk (*) indicates that the path must be traversed at least once.

Example



The loop may be traversed a maximum of two times, and the path for <optional item-2> must be traversed at least once but no more than twice.

The following are some of the valid constructs that may be generated from the preceding diagram:

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-2>,<optional item-1>

<optional item-2>

H GLOSSARY

actual declaration

A <procedure declaration> or <function declaration> that includes a <block> instead of the "formal" <directive>.

See also
Procedure Declaration 71
Function Declaration. 75

actual parameter

A variable or expression that is passed to an invocation of a procedure or function.

See also
Actual Parameter Lists and Parameter Matching 80

ASCII

American Standard Code for Information Interchange. A 7- or 8-level code representing a set of 128 control and graphic characters.

See also
EBCDIC and ASCII Character Sets 327

assignment compatible

A term that is used to indicate that a value may be assigned to a variable because the value is compatible with the declared type and value range for the variable.

See also
Assignment Compatibility. 41

base type

The type for which a <set type> is the "powerset" (that is, the type for which a <set type> is the set of all possible subsets).

See also
Set Types 59

CANDE

A Burroughs MCS that provides generalized file preparation and updating capabilities in an interactive, terminal-oriented environment. (Refer to the CANDE Language Manual for additional information.)

compatible type

A term used to indicate that the types of two or more variables are the same or similar enough to consider a particular operation defined and valid.

See also
Compatible Types. 40

compile time

The time during which the compiler is interpreting the Pascal program text and is generating a code file (compare "run time").

compiler control record

A record that appears in the source input to the compiler and contains a "$" (dollar sign) in column 1 to distinguish it from records containing Pascal program text. These records contain information that directs the compiler to perform some action relating to the compilation process.

See also
Compiler Control Records. 285

component

An item within a variable of a structured type. The component itself may be structured.

See also
Type Concepts 36

conditional statement

A structured statement that includes a Boolean condition (explicit or implied) that determines whether or not a subcomponent statement will be executed.

See also
Statements. 83

constant expression

An expression that can be fully evaluated at compile time.

context-sensitive identifier

An identifier that the compiler recognizes within a certain specialized programmatic context but that is otherwise available for use by the programmer.

See also
Interpretation of Program Text. 277

control variable

The variable that controls the repetitive execution of a FOR statement.

See also
For Statements. 89

controlled statement

The statement that a FOR statement contains and that is executed repetitively, under the control of the "control variable".

See also
For Statements. 89

dynamic variable

A variable that is allocated at run time through the "new" procedure.

See also
Dynamic Allocation Procedures 204

EBCDIC

Extended Binary Coded Decimal Interchange Code. An 8-bit code representing 256 graphic and control characters.

See also
EBCDIC and ASCII Character Sets 327

element

A "component" of an array variable.

See also
Array Types 44

enumeration

A user-defined type consisting of the values listed in the type or variable declaration.

See also
Enumerated Types. 49

expression

A combination of operands and, possibly, operators that results in the generation of a single value.

See also
Expressions 103

external file name

The name used to identify a physical file. The external name of a file is accessed through the TITLE file attribute (please refer to the I/O Subsystem Reference Manual for additional information).

field

A "component" of a record.

See also
Record Types. 55

file attribute

A system-defined variable that describes a characteristic of a physical or logical file. Many file attributes can be accessed through the "getattribute" procedure and assigned through the "setattribute" procedure and the <program parameters> syntax. (Please refer to the I/O Subsystem Reference Manual for a description of the file attributes.)

See also
Program Parameters. 10
Getattribute Procedure. 245
Setattribute Procedure. 251

formal declaration

A <procedure declaration> or <function declaration> that includes the "forward" <directive> instead of a <block>.

See also
Procedure Declaration 71
Function Declaration. 75

formal parameter

An identifier that represents a procedure or function parameter that is to be provided as an "actual parameter" when the procedure or function is invoked.

See also
Formal Parameter Lists. 78

heap

A conceptual memory area in which dynamic variables are allocated.

See also
Dynamic Allocation Procedures 204

host type

The type from which a subrange type is derived.

See also
Subrange Types. 61

identified variable

A "dynamic variable".

index

A value that is used to specify a particular element of an array variable.

See also
Variables 257

internal file name

An identifier that is declared to be of a <file type>. The system assigns that identifier to the file's INTNAME file attribute; the INTNAME can then be used in WFL file equations. (Please refer to the I/O Subsystem Reference Manual for additional information on the INTNAME attribute and the Work Flow Language (WFL) Reference Manual for information on file equation.)

See also
File Types. 51

mark value

A special value assigned to a pointer variable through the use of the "mark" procedure.

See also
Mark Procedure. 209

marked variable

A dynamically allocated variable that was allocated after the "mark" procedure was used.

See also
Mark Procedure. 209

ordinal type

A data type in which the values have a simple, ordered relationship to each other.

See also
Ordinal Types 37

predecessor

In an "ordinal type", the value having the next lowest ordinal number.

See also
Pred Function 249

repetitive statement

A structured statement that specifies that its subcomponent statements are to be repeated.

See also
Statements. 83

reserved word

An identifier whose spelling is recognized by the compiler as a Pascal language keyword and cannot be redeclared by the programmer.

See also
Interpretation of Program Text. 277

run time

The time during which the code file generated by the compiler from the Pascal program text is being executed by the system (compare "compile time").

same type

A term applied to two types when they have been defined as equivalent to each other or both equivalent to a third type.

See also
Same Types. 39

sequential statement

A structured statement in which the subcomponent statements are executed in the order in which they appear, without conditions or repetitions.

See also
Statements. 83

simple statement

A statement that performs a single action and contains no other statements.

See also
Statements. 83

source program

The Pascal program text.

standard character set

The character set that the compiler is using as the representation for the "char" type and string types. The standard character set is EBCDIC unless changed to ASCII using the STRINGS compiler control option.

See also
STRINGS Option. 303

structured statement

A statement that can have statements as subcomponents.

See also
Statements. 83

structured type

A data type that defines variables that can have multiple components.

See also
Type Concepts 36

substring

A portion of a string or vlstring variable.

See also
Vlstring Expressions. 129

successor

In an "ordinal type", the value having the next highest ordinal number.

See also
Succ Function 253

<abort procedure> 238, *239
<abs function> 226, *227
<accept procedure> 238, *240
activation record 16, 17, 20, 21, 22
actual declaration 72, 76, 355
actual parameter 41, 45, 70, 73, 79, 80, 97, 109, 110, 261, 262, 264, 355, 360
<actual parameter list> 79, *80, 97, 109, 261, 262, 264
<addstation procedure> 132, 153, *160, 164, 267
<ANSI option> 289, *290
<arccos function> 226, *227
<arcsin function> 226, *228
<arctan function> 226, *228
<arctanh function> 226, *228
<arithmetic expression> . . . *105, 114, 124, 227, 228, 229, 230, 231, 232, 233, 235, 236, 237, 256
<arithmetic function> 131, *227
<arithmetic operator> *124
<arithmetic relation> 113, *114
<array type> 34, 37, *44, 103, 265
<array type identifier> *34, 38, 44
<array variable> 103, 200, 257, 259, *265
ASCII 4, 48, 144, 153, 197, 198, 240, 243, 272, 281, 303, 304, 316, 322, 327, 329, 335, 340, 355, 358, 363
assignment compatible . . . 41, 42, 43, 80, 85, 90, 140, 142, 177, 179, 189, 200, 202, 241, 254, 259, 263, 355
<assignment statement> 75, 83, *85, 261, 262, 264, 269, 348, 349
<attribute parameter list> *276
<attribute phrase> *10, 11

<base type> 40, *59, 127
<blank> *282
<block> 8, 11, *12, 16, 18, 24, 25, 29, 30, 65, 71, 72, 75, 76, 80, 83, 92, 204, 355, 359
<Boolean constant> *31, 33, 56, 61, 111, 112
<Boolean constant identifier> *31, 33
<Boolean expression> 95, 99, 100, 103, 105, *111, 112, 114, 190, 191, 251, 347
<Boolean operator> 103, *111
<Boolean option> 285, 286, 287, 288, *289, 291
<Boolean primary> 103, *111
<Boolean type> 32, 34, 36, 37, *47, 56, 111, 112, 265
<Boolean type identifier> *34, 38, 47
<Boolean-valued file attribute> 10, 245, 251, *276, 280
<Boolean variable> 89, 111, 112, 117, 245, 257, *265
<buffer variable> 257, *264

<cancel procedure> 212, *213
<case constant> 2, 40, *56, 57, 84, 86, 105, 210, 349
<case index> *86
<case list element> *86
<case statement> 83, 84, *86, 345

<char constant> *31, 56, 61, 118
<char constant identifier> *31, 33
<char expression> 103, 105, 114, *118, 128, 190, 191, 219
<char type> 34, 36, 37, 42, *48, 56, 118, 191, 259, 265, 272
<char type identifier> *34, 38, 48
<char variable> 89, 118, 178, 179, 191, *265
<character> 271, *272, 282
<character literal> 31, 32, *272, 278, 280
<character string> 10, 32, 45, 68, 71, 271, *272, 278, 280, 282, 303
<chr function> 196, *197, 199
<clear option> 290, *291
<close-option> *161, 162, 163, 280
<close procedure> 132, *161, 163, 268
<code option> 289, *291
<comment> 281, *282
compatible type 356
<compiler control record> 277, *285
<component type> 45, *51, 135, 139, 144, 264
<compound statement> 2, 83, *88
<concat function> 130, 216, *217
<constant definitions> 24, 29, *31, 32
<context-sensitive identifier> 170, 278, *280
<control variable> *89, 90
controlled statement 89, 90, 91, 358
<cos function> 226, *229
<cosh function> 226, *229
<cotan function> 226, *229

<date procedure> 238, *241
<day> *241
<declaration part> 12, *29, 30
<declared function> *109
<declared procedure> *97
<delete option> 289, *292
<delete procedure> 216, *218
<deletestation procedure> 132, 153, *164, 267
<digit> 30, *273, 274, 295, 305
<directive> *71, 75, 355, 359
<display procedure> 238, *243
<dispose procedure> 204, *208, 262, 348
<domain type> 14, *53
<duration option> 19, 20, *22
<dynamic allocation procedure> 131, *204
<dynamic variable> 257, *262

<elapsedtime function> 238, *244
<element type> *44
<ending index> *129
<ending seq number> 294, *295
<entire variable> 89, 169, 257, *258, 265, 268, 299
<enumerated constant> 13, *49, 50, 56, 61, 119
<enumerated expression> 103, 105, 114, *119

<enumerated type> 13, 34, 36, 37, *49, 66, 119, 265
<enumerated type identifier> *34, 38, 49, 50, 56, 61, 66
<enumerated variable> 89, 119, *265
<eof function> . . . 132, 140, 141, 143, 147, 148, 157, *165, 166, 168, 177, 178, 184, 186
<eoln function> 132, 146, 148, *166, 168, 181, 182
<erf function> 226, *230
<errorlimit option> 290, *293
<errorlist option> 289, *293
<event> *256
<event-valued file attribute> 256, *276
<exp function> 226, *230
<exponent part> *274, 275
<expression> 41, 80, 85, *103, 189
<external directive> 25, *71, 72, 75, 76
<external file identifier> *10, 11
<external file specification> *10

<field designator> 15, 257, *261
<field identifier> 15, *55, 56, 57, 101, 261
<field list> *55, 56
<field type> *55, 57
<field width> *190, 191, 192, 193, 194
file attribute 10, 11, 136, 137, 144, 149, 156, 158, 160, 161, 162, 164, 167, 168, 169, 171, 172, 175, 185, 190, 245, 246, 251, 252, 256, 276, 280, 294, 345, 359, 361
<file attribute assignment> *251
<file attribute request> 245, 246
<file handling function> 131, *132
<file handling procedure> 131, *132, 155
<file title> *294, 295, 298, 299, 311
<file type> 11, 34, 37, 42, 44, *51, 53, 55, 78, 135, 139, 265, 361
<file type identifier> *34, 38, 51, 53
<file variable> . . . 160, 161, 162, 164, 165, 168, 169, 171, 172, 175, 177, 185, 186, 187, 189, 245, 246, 251, 256, 264, *265, 267, 268
<filevalue function> 132, *167
<fillchar procedure> 216, *219
<final value> *89, 90
<fixed part> *55, 57
<for statement> 41, 83, *89, 90
formal declaration 359
formal parameter 12, 16, 24, 45, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 97, 109, 258, 348, 360
<formal parameter list> . . . 12, 16, 24, 71, 72, 74, 75, 76, 77, *78, 79, 80, 81, 97, 109
<frac digits> *190, 191, 193, 194
<freeze procedure> 18, 19, 212, *214
<function declaration> 70, *75, 76, 79, 109, 355, 359
<function designator> . . 75, 104, *109, 111, 112, 118, 119, 120, 121, 122, 124, 125, 129, 130
<function identifier> 24, *75, 79, 80, 81, 85, 109
<functional parameter> 77, 78, *79, 81

<gamma function> 226, *230
<general function> 131, *238
<general procedure> 131, *238
<get option> *168, 280
<get procedure> 132, *168, 171, 187, 267, 268, 345
<getattribute procedure> 69, 136, 152, 238, *245, 246, 267, 268, 276
<goto statement> 30, 83, 84, 89, *92, 94

host type . . . 26, 44, 61, 66, 118, 119, 121, 127, 232, 233, 265, 266, 319, 360
<hours> *254

<identifier> . . . 8, 10, 31, 32, 34, 38, 49, 55, 65, 71, 75, 101, *273, 278, 279, 280, 295, 304, 345
<if statement> 83, 84, 88, *95
<immediate option> 285, 286, 287, *290
<inclnew option> 289, *294
<include option> 285, 286, *294, 295
<index expression> *259
<index type> *44, 45, 200, 202, 259
<indexed array variable> *259
<indexed variable> 257, *259, 260
<indexed vlstring variable> *259
<initial value> *89, 90
<insert index> 220
<insert procedure> 216, *220
<integer constant> *31, 56, 61, 64, 179
<integer constant identifier> *31, 32, 33, 120, 121, 275
<integer expression> . . . 103, 105, 114, *120, 129, 187, 188, 190, 192, 197, 199, 218, 220, 225, 248, 251, 259, 267, 276
<integer operator> *120
<integer primary> *120, 124
<integer type> 34, 36, 37, 42, 44, *52, 56, 120, 121, 192, 241, 254, 266
<integer type identifier> *34, 38, 52
<integer-valued file attribute> 10, 245, 251, *276, 280
<integer variable> 89, 120, 121, 178, 179, 181, 182, 192, 245, *266
<interface function declaration> 24
<interface part> 18, *24, 214
<interface procedure and function declarations> 24
<interface procedure declaration> 24
<intname> 294, *295
<iores function> 132, 155, *170
<ioresult mnemonic> *170, 280
<iotime function> 238, *247

<label> *30, 83, 84, 92
<label declarations> 29, *30
<length function> 216, *221
<letter> *273

<lib mnemonic> *68, 215, 280
<library> 7, *18, 19, 24, 214
<library attribute assignment> *251, 252
<library attribute phrase> *68
<library attribute request> *245, 246
<library declarations> 19, 25, 29, *68, 72
<library handling function> 131, *212
<library handling procedure> 131, *212
<library heading> 18
<library identifier> 18, 68, 69, 71, 72, 76, 213, 245, 246, 251, 252
<libraryvalue function> 212, *215
<lineinfo option> 289, *296
<list option> 289, 293, *296
<listdollar option> 289, *297
<listincl option> 289, *297
<ln function> 226, *231
<lngamma function> 226, *231
<local directive> *71, 75
<log function> 226, *231

<map option> 289, *297
<mark procedure> 204, 207, 208, *209, 262
mark value 206, 209, 211, 262, 361
marked variable 208, 361
<max function> 226, *232
<maximum length> 42, *64, 67, 220
<member designator> *126, 127
<merge option> 289, *298
<min function> 226, *233
<minutes> *254
<miscellaneous identifier> *280
<mnemonic value> 10, 167, *276, 280
<mnemonic-valued file attribute> 10, 167, 245, 251, *276, 280
<mnemonic-valued library attribute> 68, 215, 245, 251
<month> *241

<new array type> 38, *44
<new enumerated type> 38, *49
<new file type> 38, *51
<new option> 289, *298
<new pointer type> 38, *53
<new procedure> 204, 208, 209, *210, 262, 322
<new record type> 38, *55, 58, 67
<new set type> 38, *59
<new subrange type> 38, *61
<new vlstring type> 38, *64
<nobounds option> 289, *299
<non-apostrophe character> *272
<number> 10, *274, 275, 278, 280

<odd function> 238, *248
<omit option> 289, *300

<open option> 136, 137, 143, 156, *171, 172, 280
<open procedure> 132, 136, 137, *171, 172, 173, 268
<option expression> 287, *288, 304
<option phrase> 285, 286, *287
<option primary> *288
<ord function> 196, *198, 199
<ordinal expression> 86, 89, *105, 116, 126, 127, 198, 200, 202,
232, 233, 249, 253, 259
<ordinal relation> 113, *114, 115
<ordinal type> *37, 44, 59
<ordinal type identifier> 45, *56, 57, 199
<ordinal type transfer function> 196, *199, 345

<pack procedure> 196, *200
<packed array variable> *200, 202
<page option> 290, *300
<page procedure> 132, *174
<pattern> *222
<pointer expression> 103, 115, *122, 123, 204, 206, 208, 211
<pointer relation> 113, *115
<pointer type> 14, 34, 36, 37, *53, 75, 122, 266
<pointer type identifier> *34, 38, 53
<pointer variable> 122, 204, 206, 209, 210, 262, *266, 348
<pos function> 216, *222
<pos source> *222
<pred function> 238, *249
<predefined function> 109, *131
<predefined identifier> 278, *279
<predefined procedure> 97, *131
<procedural parameter> 73, *78, 79, 81
<procedure and function declarations> 19, 25, 29, *70, 72, 76
<procedure declaration> 70, *71, 72, 79, 97, 355, 359
<procedure identifier> *71, 78, 80, 81, 97
<procedure invocation statement> 71, 83, *97
<program> 7, *8, 12, 18, 214
<program heading> *8, 18, 137
<program identifier> *8
<program parameters> 8, *10, 11, 18, 136, 151, 152, 359
<program text> 277, *278
<program unit> *7, 22
<put option> *175, 280
<put procedure> 132, *175, 187, 267, 268, 345

<random function> 226, *234
<read parameter> *178, 184
<read procedure> 132, *177, 178
<read textfile procedure> . . . 132, 147, *178, 179, 180, 181, 182, 184
<readln procedure> 132, *184
<real constant> 31, *32, 180
<real constant identifier> *31, 32, 33, 124, 125
<real expression> 103, 105, *124, 190, 193, 235, 237, 251
<real primary> *124
<real type> 34, 36, 37, 42, *54, 124, 125, 193, 266

<real type identifier> *34, 38, 54
<real-valued file attribute> 10, 245, 251, *276, 280
<real variable> 124, 125, 178, 180, 234, 245, *266
<record boundary> 282, *283
<record type> 13, 34, 37, 40, *55, 56, 57, 103, 266
<record type identifier> *34, 38, 55, 58
<record variable> 15, 101, 103, 261, 262, 264, *266, 348
<rel op> *113, 114, 117
<relational expression> 111, *113
<release procedure> 204, 207, 209, *211, 262, 348
<repeat statement> 83, *99
<reserved word> 273, 278, *279
<reset procedure> 132, 136, 171, *185
<result type> 24, *75, 76, 77, 79, 81, 112, 118, 119, 121, 122, 125
<rewrite procedure> 132, 136, 137, 171, *186
<round function> 226, *235
<runtime function> 238, *250

same type . . . 27, 39, 40, 42, 47, 48, 61, 65, 80, 81, 115, 127, 135, 227, 232, 233, 236, 249, 253, 321, 362
<scan function> 216, *223
<scan set> *223
<scan target> 223
<seconds> *254
<seek procedure> 132, *187
<sequence base option> 290, *302
<sequence increment option> 290, *302
<sequence number> *305
<sequence option> 289, *301
<set constructor> *126, 127, 223
<set expression> 103, 116, *126, 127
<set operator> *126
<set primary> *126
<set relation> 113, *116
<set type> 34, 37, 40, *59, 126, 127, 266, 348, 356
<set type identifier> *34, 38, 59, 60
<set variable> 126, *266
<setattribute procedure> 69, 136, 152, 238, *251, 252, 267, 268, 276
<sharing option> *20
<simple type> *36, 75, 81, 232, 233
<sin function> 226, *235
<sinh function> 226, *236
<skiptochannel procedure> 132, 174, *188
<special token> 278, *281
<sqr function> 226, *236
<sqrt function> 226, *236
standard character set 48, 197, 198, 272, 363
<starting index> *129, 130
<starting seq number> 294, *295
<statement> *83, 84, 86, 88, 89, 92, 95, 100, 101
<statement list> *83, 86, 88, 92, 94, 99

<statement part> 12, *83, 92
<station index> 169, 175, 246, 251, *267
<station variable> 168, 169, 175, 176, 245, 246, 251, *267, 345
<statistics option> 289, *303
<string constant> 31, *32, 128
<string constant identifier> *31, 32, 33
<string expression> 103, *128, 129, 130
<string function> 130, 216, *225
<string handling function> 131, *216
<string handling procedure> 131, *216
<string relation> 113, *117
<string type> 40, 41, 42, *45, 46, 103, 128, 144, 192, 266, 272
<string-valued file attribute> 10, 245, 251, *276, 280
<string-valued library attribute> *68, 245, 251
<string variable> 128, 129, 130, 178, 181, 192, 219, 240, 245, *266
<strings option> 290, *303
<structured type> 36, *37, 44, 51, 53, 55, 78
<subfile index> 161, 168, 169, 171, 175, 246, 251, *267
<subfile variable> 161, 168, 169, 171, 175, 176, 245, 246, 251, 256, *267, 268, 345
<subrange type> . . . 34, 36, 37, 44, *61, 66, 118, 119, 121, 127, 265, 266
<subrange type identifier> *34, 38, 56, 61, 62
<substring designator> *129, 130, 219
<substring expression> *129, 219
<substring length> *129, 130
<succ function> 238, *253

<tan function> 226, *237
<tanh function> 226, *237
<textfile type> . . . 11, 34, 37, 42, 44, 51, 53, 55, *63, 78, 135, 268
<textfile type identifier> *34, 38, 53, 63
<textfile variable> . . . 161, 162, 165, 166, 168, 171, 174, 175, 178, 184, 185, 186, 188, 190, 195, 245, 246, 251, 256, 264, *268
<time procedure> 238, *254
<token> *278
<token separator> 278, *282
<trunc function> 226, *237
<type> *36, 44, 51, 55, 65
<type definitions> 14, 24, 29, *34, 35, 38, 65
<type identifier> *38, 39, 40, 53, 78, 145
<type transfer function> 131, *196
<type transfer procedure> 131, *196

<unpack procedure> 196, *202
<unpacked array variable> *200, 202
<unsigned integer> 10, 31, 32, 120, *274, 275, 293, 302
<unsigned number> *274
<unsigned real> 32, 124, *274, 275
<usage clause> 18, *20
<user option> 287, 289, 291, *304

<value option> 285, 286, 287, *290
<value parameter> 16, 25, 73, 77, *78, 79, 80
<value parameter type> *78
<variable> 80, 85, 177, 241, 254, *257, 261, 262, 265, 266, 268
<variable declarations> 11, 16, 29, 38, *65, 151, 258
<variable identifier> *65, 78, 136, 258
<variable identifier list> 42, *65, 258
<variable parameter> 25, 39, 73, 74, 77, *78, 79, 80
<variable parameter type> *78
<variant> 40, *56, 57, 58, 261, 348, 349
<variant part> 2, 55, *56, 57, 81, 348
<variant selector> 40, *56, 57, 58, 210
<variant specifier> 208, 210, 322
<vlstring expression> 103, 117, *129, 160, 164, 190, 192, 217, 220, 221, 222, 223, 240, 243, 251
<vlstring type> 34, 37, 40, 42, *64, 67, 129, 130, 192, 268
<vlstring type identifier> *34, 38, 64
<vlstring variable> . . . 117, 129, 130, 178, 182, 217, 218, 219, 220, 224, 225, 240, 245, 259, *268
<void option> 290, *305

<wait procedure> 153, 238, *256, 268
<warnsupr option> 289, *306
<while statement> 83, 88, *100
<with statement> 83, *101, 261, 262, 264, 348
<write parameter> *190, 191, 195
<write procedure> 132, *189, 190
<write textfile procedure> 132, 148, *190, 191, 195
<writeln procedure> 132, 174, 185, 188, *195

<xref option> 289, *306, 307
<xreffiles option> 289, *307

<year> *241