Announcement Letter

???? ??, 1985

B1000 SYSTEMS

PASCAL LANGUAGE MANUAL

With this letter, we are announcing the availability of the B 1000 Systems Pascal Language Manual, form 1152048, dated ?????????.

This manual includes Burroughs extensions to the ANSI Pascal programming language.

All technical communication relative to this document should be directed to:

Burroughs Corporation
Manager, B1000 Software Documentation
6300 Hollister Avenue
Goleta, California 93117 USA

Copies of this publication may be ordered from the Publications Center, Dearborn, Michigan U.S.A.

Raymond J. Renzullo, Manager
Documentation - West

TITLE: B1000 SYSTEM SOFTWARE RELEASE MARK 12.0 (DOCUMENT)

FILE ID: DOCUMENT/PASCAL TAPE ID: PASCALSRC

RELATIVE TO THE MARK 12.0.000 RELEASE

Comments or suggestions regarding this document should be submitted to PA&S Santa Barbara at the following address:

Burroughs Corporation
PA&S/Santa Barbara
6300 Hollister Ave.
Santa Barbara, CA 93117

TABLE OF CONTENTS

SECTION 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 1900 Pascal Compiler complies with this standard with the restrictions described later in this section. Throughout the remainder of this manual, the Burroughs B 1900 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 1900 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. A complete description of railroad syntax is provided in appendix B, Railroad Diagrams.

The remainder of this introduction 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.

IMPLEMENTATION RESTRICTIONS

The following items are restrictions in the initial Pascal implementation. Many will be removed or changed in future releases.

DISPOSE Procedure
Not implemented. Dynamic memory is managed by using the MARK and RELEASE procedures.

Variant Record Declarations
Do not require all labels to be present.

Procedural Parameters
Not implemented.

Non-local GOTOs
Not implemented.

PACK, UNPACK
Not implemented.

NEW
Tag constants not permitted is parameter list.

The following is a list of limits imposed by the B 1000 implementation.

!bu Labels in CASE statements must be in the range 0 to 255 inclusive.

!bu Labels in variant parts of records must be in the range 0 to 23 inclusive.

!bu REAL numbers have a precision of approximately eight decimal digits. The exponents can be within the range -47 to +68. The routines that print REAL numbers print a maximum of seven significant digits. This is done so that the last digit can be guaranteed to be accurate.

!bu Maxint is 8388607.

!bu Routines with local file variables cannot be used recursively.

!bu A file must not be a component of any structured type.

!bu The maximum nesting of lexic levels is eight.

ERRORS DURING EXECUTION

The following errors can be detected during the execution of a program.

Integer overflow
Real overflow
Stack limit exceeded
Heap limit exceeded
Text file buffer overflow
Division by zero
Value of end of file wrong for file operation
Operation on improperly defined file
Nil pointer dereference

Undefined pointer dereference
Released pointer dereference
Array index out of range
No label corresponding to case selector
Record variant accessed with incorrect tag
Value out of subrange

Some operations may cause values to go out of range with no error reported. Complete checking is not guaranteed, but data will not be altered or lost as a result of incomplete checking. The following errors are not checked:

Changing variables in the list of a WITH statement
GOTOs from outside to inside a structured statement
Side effects, especially those thwarting run-time checks
Dangling pointers as a result of a RELEASE operation
Operations on an uninitialized variable
Record variable accessed with incorrect tag type

STRUCTURE OF MANUAL

The structure of this manual is top-down; that is, larger or higher-level syntactic components such as programs, declarations, and statements, are described first and smaller or lower-level components such as variables and identifiers are described last. A brief description of each section and appendix follows.

Section 1, Introduction, introduces the language and the manual.

Section 2, Program Structure, describes Pascal programs, program parameters, and blocks. This section also describes the concept of scope as it applies to identifiers and activations.

Section 3, Declarations and Definitions, contains a description of the declaration part of a block, including type definitions and variable declarations. Concepts relating to data types in Pascal are covered under Type Definitions.

Section 4, Statements, describes the statement constructs available in Pascal.

Section 5, Expressions, describes all expression types and includes a discussion of the precedence of operators within expressions.

Section 6, Predefined Procedures and Functions, explains the ready-made procedures and functions that are available. These procedures and functions provide facilities for file handling, type transfer, dynamic variable allocation, arithmetic functions, and other general features. A detailed description of Pascal input/output concepts and how they relate to the Burroughs B 1900 system is included under File Handling Procedures.

Section 7, Variables, describes variables of various types and how they are referenced within the program.

Section 8, Basic Components, defines some of the small, frequently used components of the syntax of Pascal, such as identifiers and numbers.

Section 9, Interpretation of Program Text, describes how the Burroughs Pascal compiler interprets the program information it reads from its input files. This section includes lists of reserved words, predefined identifiers, and context-sensitive identifiers. A description of the use of comments within the program text is also included.

APPENDIX A, Compiling, Executing and Analyzing a Pascal Program, defines the syntax and semantics of the options that can be used to direct certain aspects of the compilation and execution of Pascal programs.

Appendix B, Railroad Diagrams, contains a description of the notation used throughout this manual to represent the syntax of the Pascal language.

Appendix C, EBCDIC and ASCII Character Sets, provides two tables, the first in EBCDIC sequence and the second in ASCII sequence, of the B 1000 codes. Each table includes the hexadecimal and ordinal numbers for the EBCDIC and ASCII codes as well as the assigned graphics and their meanings.

RELATED DOCUMENTS

The following documents contain information of interest to the users of this manual:

American National Standards Institute (ANSI)
Programming Language Pascal (X3J9/81-093) -- Proposed

Pascal User Manual and Report by K Jensen and N. Wirth
Springer-Verlag, New York, 1978

B 1000 Systems System Software Operation Guide, Volume 1
Form No. 1151982

B 1000 Systems System Software Operation Guide, Volume 2
Form No. 1152097

Burroughs CSG Standard for Compiler Control Images
Burroughs No. 1955 2959

SECTION 2

PROGRAM STRUCTURE

Syntax diagrams for all the Pascal program elements discussed in this section are presented in figure 2-1.

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. The form of the <program unit> is very similar to the forms of the procedures and functions that can be defined within it.

The <program heading> includes a program <identifier>, which is not used for any subsequent purpose, and the optional <program paramenters>.

The other major component of the <program> is the <block>. This contains the data definitions and algorithm descriptions of the program. Details of the syntax and semantics of the program block begin later in this section and continue through the remainder of this manual.

```
Program Unit syntax:

---- <program> ---------------------------------------------------

<program> syntax:

---- <program heading> ---- ; ---- <block> ---- . ----------------

<program heading> syntax:

---- PROGRAM -- <program identifier> --+--------------------------+-
                                       !                          !
                                       +- ( <program parameters> ) -+

<program identifier> syntax:

---- <identifier> ------------------------------------------------

<program parameters> syntax:

        +<------------- , -----------------+
        !                                  !
--------+-- <external file specification> --+---------------------

<external file specification> syntax:

---- <external file identifier> ----------------------------------

>---------+------------------------------------------------+-----
          !                 +<--------- , -----------+     !
          !                 !                        !     !
          +---- : -- FILE < --+-- <attribute phrase> --+-- > --+

<external file identifier> syntax:

---- <identifier> ------------------------------------------------
```

Figure 2-1. Syntax Diagrams: Pascal Program Elements

<attribute phrase> syntax:

```
--+ <Boolean-value file attribute>   = --+ TRUE ---------------------+-
  !                                      !                           !
  !                                      ! FALSE -------------------+
  !                                                                  !
  + <integer-value file attribute>   = --+---+-- <unsigned integer> +
  !                                      !   !                       !
  !                                      + + +                       !
  !                                      !   !                       !
  !                                      + - +                       !
  !                                                                  !
  + <mnemonic-valued file attribute> = <mnemonic value> ------------+
  !                                                                  !
  + <string-valued file attribute>   = <character string> ----------+
  !                                                                  !
  + <real-valued file attribute>     = <number> --------------------+
```

<block> syntax:

```
----+----------------------------+---- <statement part> -------------
    !                            !
    +---- <declaration part> ----+
```

Figure 2-1 Continued.

An example of a program follows.

Example:

```
program EXAMPLE(INPUT, OUT_FILE : file <maxrecsize= 132>);

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, named EXAMPLE, program computes the factorial of a number entered through a file named INPUT. The factorial is computed by recursively calling the procedure FACT. The answer is written to file OUT_FILE, which may be label-equated to a printer file.

NOTE

> The names EXAMPLE, INPUT, OUT_FILE, and FACT are spelled in upper case here for ease of identification. Pascal does not distinguish between upper-case and lower-case spelling except in literals.

PROGRAM PARAMETERS

The <program parameters> specify permanent 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 a <textfile type>. The predefined files named INPUT and OUTPUT are exceptions to this rule; their appearance in the <program parameters> is equivalent to declaring them in the outer block of the program, they must not appear in the <variable declarations> of the program.

When a file is named in the list of <program parameters>, the PROTECTION file attribute for that file is automatically set to SAVE. Thus, a file created by the program becomes a permanent file.

For further information on files, textfiles, and file attributes, please refer to I/O Concepts in section 6.

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

PROGRAM BLOCKS

A <block> is a set of related declarations and statements. The declarations describe data and the statements describe actions. The <declaration part> and the <statement part> of blocks are described in sections 3 and 4.

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

Scope

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

Scope: Blocks

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, a nested function or a nested record definition, the former definition is unavailable in that region and the new definition applies.

Figure 2-2 illustrates the concept of scope for blocks. In viewing the figure, note that a reference to an identifier or label is always to its closest (most local) definition.

```
program p;

   var a, {declaration of a and b} <----!         <--!
       b : real;                        !            ! scope of
                                        !            !  b of p
   procedure q;                         !         <--!
      var b : real;       <--!          !
      begin                  !          !
        .                    ! scope of !
      {statements of q}      !  b of q  ! scope of
        .                    !          ! a, q of p
      end;                <--!          !
   begin                                !         <--!
     .                                  !            ! scope of
   {statements of p}                    !            !  b of p
     .                                  !            !
   end.                            <----!         <--!
```

Figure 2-2. Illustrations of the Scope of Blocks

```
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.                                             <----!
```

Figure 2-3. Scope of Record Definitions

Scope: Record Definitions

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 figure 2-3, the <enumerated constant>s red, green, and blue can be referenced within the block in which type r is defined.

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:

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

!bu 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.

!bu 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 invalid forward reference even if the same identifier is defined in an enclosing scope.

Scope: WITH Statements

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.

Scope: Record Variables

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 period (.) to the end of the <field designator>.

Activation Records

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 from ALGOL OWN variables; those retain their previous values when the block is re-entered.)

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 shown in figure 2-4.

A procedure or function can call itself, either directly or indirectly. If, in the previous example, procedure q calls procedure p, the stack will contain the activation records shown in figure 2-5.

Logically, this process could continue indefinitely; however, the system would 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 Dynamic Allocation Procedures in section 6.

```
                           ----------
                           !    d    !
                           !    c    !
   ----------              !---------!              ----------
   !    b    !             !    b    !              !    b    !
   !    a    !             !    a    !              !    a    !
   ----------              ----------               ----------

   Before                  While in                 After
   procedure q             procedure                procedure
   is called               q                        q is exited
```

Figure 2-4. Procedure p Calls Procedure q.

```
----------
! b ! second activation record of p
! a !
----------
! d ! activation record of q
! c !
!---------!
! b ! first activation record of p
! a !
----------
```

Figure 2-5. Procedure q Calls Procedure p.

SECTION 3

DECLARATIONS AND DEFINITIONS

Following is the syntax diagram for the <declaration part> of a <block>.

Syntax:

```
----+------------------------------------------------+-------------->
    !                                                !
    +---- <label declarations> ----------------------+

>---+------------------------------------------------+-------------->
    !                                                !
    +---- <constant definitions> --------------------+

>---+------------------------------------------------+-------------->
    !                                                !
    +---- <type definitions> ------------------------+

>---+------------------------------------------------+-------------->
    !                                                !
    +---- <variable declarations> -------------------+

>---+------------------------------------------------+--------------!
    !                                                !
    +---- <procedure and function declarations> -----+
```

The declarations and definitions are all optional, but when two or more are used, they must appear in the sequence shown in the diagram.

The <constant definitions>, <type definitions>, and <variable declarations> primarily are used to describe the data on which the program is to act. The <label declarations> and <procedure and function declarations> are tools used in describing the program algorithm. These components are described in the following sections, in the order in which the components appear in the <declaration part>.

LABEL DECLARATIONS

<label declarations> identify <label>s for use within the <block>. The <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> may have up to four significant digits. (Leading zeros are not significant digits.) Therefore, <label> range is 0 through 9999.

<label declarations> syntax:

```
               +<---- ; -----+
               !             !
---- LABEL --+-- <label> --+-- ; ------------------------------------
```

<label> syntax

```
     +<---/ 4 \----+
     !             !
----+-- <digit> --+--------------------------------------------------
```

CONSTANT DEFINITIONS

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 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 8,388,607 (2 raised to the 23rd power minus 1). TRUE and FALSE are predefined values of the <Boolean type>. <identifier>, <character literal>, <unsigned integer>, <unsigned real>, and <character string> are defined in section 8, Basic Components.

Examples:

```
1. always    = TRUE;
2. a         = 'a';
3. maxbits   = 48;
4. minvalue  = -4.5;
5. greeting  = 'Hello';
6. intro     = greeting;
7. warning   = 'Don''t do it';
```

In example 1 always is a <Boolean constant identifier> with the value TRUE; thus, always may be used wherever a <Boolean constant> is valid.

In example 2, the letter a is a <char constant identifier> with a as its value.

In example 3, maxbits is an <integer constant identifier> with the value 48.

In example 4, minvalue is a <real constant identifier> with the value -4.5.

In example 5, greeting is a <string constant identifier> with the value 'Hello'.

In example 6, intro is a <string constant identifier> with the same value as greeting (example 5).

In example 7, warning is a <string constant identifier> with the value 'Don't do it'.

<constant definitions> syntax:

```
---- CONST -------------------------------------------------------------

  +<--------------------------------------------------------------+
  !                                                               !
>-+--+ <Boolean constant identifier> = <Boolean constant> +-- ; --+---
     !                                                    !
     + <char constant identifier>    = <char constant>    +
     !                                                    !
     + <integer constant identifier> = <integer constant> +
     !                                                    !
     + <real constant identifier>    = <real constant>    +
     !                                                    !
     + <string constant identifier>  = <string constant>  +
```

<Boolean constant> syntax.

```
----+-- TRUE -------------------------+---------------------------
    !                                 !
    +-- FALSE ------------------------+
    !                                 !
    +-- <Boolean constant identifier> --+
```

<char constant> syntax:

```
----+-- <character literal>----------+--------------------------------
    !                                !
    +-- <char constant identifier> --+
```

<integer constant> syntax:

```
----+-----+---+-- MAXINT -------------------------+------------------
    !     !   !                                   !
    !  +  !   +-- <unsigned integer>--------------+
    !     !   !                                   !
    +  -  +   +-- <integer constant identifier> --+
```

<real constant> syntax:

```
----+-----+---+-- <unsigned real> ---------------+-----------------
    !     !   !                                  !
    !  +  !   +-- <real constant identifier> -----+
    !     !
    +  -  +
```

<string constant> syntax:

```
----+-- <character string>-------------+---------------------------
    !                                  !
    +-- <string constant identifier> --+
```

TYPE DEFINITIONS

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 to be specified.

Discussions of some general concepts that apply to types are presented next, followed by descriptions of all the types, presented in alphabetical order.

<type definitions> syntax:

```
-- TYPE ---------------------------------------------------------->

  +<--------------------------------------------------------------+
  !                                                               !
>-+-+ <array type identifier>       = <array type> ---------+-- ; --+--
    !                                                       !
    + <Boolean type identifier>     =  <Boolean type> ------+
    !                                                       !
    + <char type identifier>        =  <char type> ---------+
    !                                                       !
    + <enumerated type identifier> =  <enumerated type> ---+
    !                                                       !
    + <file type identifier>        =  <file type> ---------+
    !                                                       !
    + <integer type identifier>     =  <integer type> ------+
    !                                                       !
    + <pointer type identifier>     =  <pointer type> ------+
    !                                                       !
    + <real type identifier>        =  <real type> ---------+
    !                                                       !
    + <record type identifier>      =  <record type> -------+
    !                                                       !
    + <set type identifier>         =  <set type> ----------+
    !                                                       !
    + <subrange type identifier>    =  <subrange type> -----+
    !                                                       !
    + <textfile type identifier>    =  <textfile type> -----+
```

Simple, Structured, and Pointer Types

Types may be classified into three categories that reflect their structure.

<type> syntax:

```
----+-- <simple type> ------+-------------------------------------
    !                       !
    +-- <structured type> --+
    !                       !
    +-- <pointer type> -----+
```

Simple Types

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.

<simple type> syntax:

```
----+-- <Boolean type>-------+---------------------------------------
    !                        !
    +-- <char type>----------+
    !                        !
    +-- <enumerated type> ---+
    !                        !
    +-- <integer type> ------+
    !                        !
    +-- <real type> ---------+
    !                        !
    +-- <subrange type> -----+
```

Structured Types

Variables of structured types are composed of multiple components, which may be of one or more simple types or may be structured themselves.

<structured type> syntax:

```
----+-- <array type> -------+----------------------------------------
    !                       !
    +-- <set type> ---------+
    !                       !
    +-- <record type> ------+
    !                       !
    +-- <file type> --------+
    !                       !
    +-- <textfile type> ----+
```

Pointer Type

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

<pointer type> syntax:

```
---- <pointer type> -------------------------------------------------
```

Ordinal Types

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 (for example, greater than, less than) based on their ordinal numbers.

The only simple type that is not an ordinal type is the <real type>.

<ordinal type> syntax:

```
----+-- <Boolean type>------+-------------------------------------
    !                       !
    +-- <char type> --------+
    !                       !
    +-- <enumerated type> --+
    !                       !
    +-- <integer type>------+
    !                       !
    +-- <subrange type> ----+
```

Type Identifiers

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

(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; previously defined <type identifier>s must be used.

<type identifier> syntax:

```
----+-- BOOLEAN--------------------------+----------------------------------
    !                                    !
    +-- CHAR ----------------------------!
    !                                    !
    +-- INTEGER -------------------------+
    !                                    !
    +-- REAL ----------------------------+
    !                                    !
    +-- TEXT ----------------------------+
    !                                    !
    +-- <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> -----+
```

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 to assignment compatibility. See Compatible Types, later in this section.

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

Rule 1 One type is defined to be equal to the other.

Rule 2 Both types are of the same type as a third type.

In the simplest case of same type, T1 is defined to be equal to T2, as shown in the following example:

```
TYPE T1 = T2;        {Rule 1}
```

Rule 2 describes the situation in which T1 and T2 have a common ancestor. The simplest case is the following:

```
TYPE T3 = INTEGER;
     T1 = T3;        {Rule 1}
     T2 = T3;        {Rule 1}
```

T1 is the same type as T2 by rule 2. In the following example, T1 and T2 are also of the same type:

```
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 rules are true:

Rule 1 T1 and T2 are the same type.

Rule 2 One type is a subrange of the other, or both types are subranges of the same type.

Rule 3 T1 and T2 are <set type>s with compatible <base type>s and both T1 and T2 are packed or both T1 and T2 are not packed.

Rule 4 T1 and T2 are <string type>s with identical character counts.

Examples:

```
type t1 = real;
     t2 = t1;
     {t1 and t2 are compatible by rule 1.}

     t3 = 1..10;
     t4 = 5..7;
     t5 = 20..30;
     {t3, t4, and t5 are compatible by rule 2.}

     t6 = set of char;
     t7 = set of 'a'..'z';
     {t6 and t7 are compatible by rule 3.}

     t8 = packed array [1..10] of char;
     t9 = packed array [1..7]  of char;
     {t8 and t9 are compatible by rule 4.}
```

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, runtime, and date procedures must be assignment compatible with the parameters passed to those procedures.

In the definition of assignment compatibility that follows, 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 is true:

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

2. 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).

3. T2 is the <real type> and T1 is the <integer type>.

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

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

6. T1 and T2 are compatible <string type>s.

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 1.}

var  v1,
     v2 : array [1..10] of Boolean;
     {All values of v1 are assignment-compatible with v2, and vice
       versa, by rule 2.}

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

     v5 : 7..10;
     v6 : 1..20;
     {All values of v5 are assignment-compatible with v6 by rule
       4, 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
       5, but only some values of v8 are assignment-compatible
       with v7, namely those set values that contain only characte
       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 6.}
```

Type Descriptions

Array Types

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 multidimensional, 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 is practical, possibly at the expense of speed in accessing the components. When a multidimensional 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).

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.

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

Example:

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

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

<array type> syntax:

```
----+-- <new array type>-----------+-----------------------------------
    !                              !
    +-- <array type identifier> ---+
```

<new array type> syntax:

```
                                      +<-----  ,  -----+
                                      !                !
----+---------+- ARRAY  [ --+ <index type> +-- ]  OF -- <element type>
    !         !
    + PACKED +
```

<index type> syntax:

```
---- <ordinal type> ------------------------------------------------------
```

<element type> definition:

An <element type> is 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.

<string type> definition:

A <string type> is an array that is defined as PACKED ARRAY [1..n] OF CHAR, where n is greater than or equal to 1.

Boolean Types

Boolean is a predefined ordinal type that comprises the values TRUE (value = ordinal 1) and FALSE (value = ordinal 0). All <Boolean type>s are of the same type.

Example:

```
    type b = Boolean;
```

Type b is a <Boolean type identifier>.

<Boolean type> syntax:

```
----+-- BOOLEAN -------------------+-----------------------------------
    !                              !
    +-- <Boolean type identifier> --+
```

Character Types

The character type (<char type>) is a predefined ordinal type that comprises the standard character set (EBCDIC unless changed to ASCII using the STRINGS compiler control option. The mapping of characters to ordinal numbers is defined in appendix C, EBCDIC and ASCII Character Sets.

All <char type>s are the same type.

Examples:

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

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

<char type> syntax:

```
----+-- CHAR ----------------------+-----------------------------------
    !                              !
    +-- <char type identifier> ----+
```

Enumerated Types

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.

<enumerated type> syntax:

```
----+-- <new enumerated type>----------+------------------------------
    !                                  !
    +-- <enumerated type identifier> --+
```

<new enumerated type> syntax:

```
        +<----------- , ------------+
        !                           !
---- [ +-- <enumerated constant> --+ ] ------------------------------
```

<enumerated constant> syntax:

```
---- <identifier> ----------------------------------------------------
```

File Types

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 I/O Concepts in section 6.

The designation PACKED has no effect for file types.

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 following fields: name, firstname, department_code, employee_no.

<file type> syntax:

```
----+-- <new file type>----------+------------------------------------
    !                            !
    +-- <file type identifier> --+
```

<new file type> syntax:

```
----+--------+---- FILE -- OF -- <component type> -------------------
    !        !
    + PACKED +
```

<component type> definition:

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.

Integer Types

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>.

<integer type> syntax:

```
----+-- INTEGER --------------------+---------------------------------
    !                               !
    +-- <integer type identifier> --+
```

Pointer Types

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, may not be currently referencing a variable. Please refer to Dynamic Allocation Procedures in section 6 for details on dynamic variables.

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.

<pointer type> syntax:

```
----+-- <new pointer type>----------+---------------------------------
    !                               !
    +-- <pointer type identifier> --+
```

<new pointer type> syntax:

```
---- @ --- <domain type> ---------------------------------------------
```

<domain type> definition:

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.

Real Types

Real is a predefined simple type that comprises the range of floating-point approximations. Real numbers in B 1000 Pascal have a precision of approximately seven decimal digits. The routines that print real numbers print a maximum of seven significant digits in order to guarantee the accuracy of the last digit. The exponent range is from E-47 to E+68.

Example:

```
    type r = real;
```

Type r is a <real type identifier>.

<real type> syntax:

```
----+-- REAL --------------------+-------------------------------------
    !                            !
    +-- <real type identifier> --+
```

Record Types

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> or a <variant part> or both or neither. 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 (that is, 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 is stored in as economical an amount of space as practical, possibly at the expense of speed in accessing the components.

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,TRUE, type str. If married is FALSE, there are no more components.

<record type> syntax:

```
----+-- <new record type>----------+-------------------------------------
    !                              !
    +-- <record type identifier> --+
```

<new record type> syntax:

```
----+--------+-- RECORD -- <field list> -- END -------------------------
    !        !
    + PACKED +
```

<field list> syntax:

```
----+------------------------------------+---+---+-----------------
    !                                    !   !   !
    +-- <fixed part> --+-----------------+   + ; +
    !                  !                 !
    !                  + ; <variant part> --+
    !                                    !
    +-- <variant part> ------------------+
```

<fixed part> syntax:

```
    +<---------------------------------------------------+
    !                                                    !
    !    +<-------- , ------------+                      !
    !    !                        !                      !
----+----+-- <field identifier> --+-- : --<field type> --+------------
```

<field identifier> syntax:

```
---- <identifier> ---------------------------------------------------
```

<field type> definition:

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> syntax:

```
                                        +<----- ; ------+
                                        !            !  !
---- CASE -- <variant selector> -- OF --+-- <variant> --+------------
```

<variant selector> syntax:

```
----+----------------------------+-- <ordinal type identifier> ------
    !                            !
    +-- <field identifier> -- : --+
```

<ordinal type identifier> syntax:

```
----+-- <Boolean type>----------------+--------------------------
    !                                 !
    +-- <char type>-------------------+
    !                                 !
    +-- <enumerated type identifier> ---+
    !                                 !
    +-- <integer type> ----------------+
    !                                 !
    +-- <subrange type identifier> -----+
```

<variant> syntax:

```
    +<------- , ----------+
    !                     !
----+-- <case constant> --+-- : -- ( <field list> ) ---------------
```

<case constant> syntax:

```
----+-- <Boolean constant>-------+--------------------------------
    !                            !
    +-- <char constant>----------+
    !                            !
    +-- <enumerated constant> ---+
    !                            !
    +-- <integer constant> ------+
```

Set Types

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> is 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]
```

<set type> syntax:

```
----+-- <new set type>----------+-----------------------------------
    !                           !
    +-- <set type identifier> --+
```

<new set type> syntax:

```
----+--------+-- SET -- OF -- <base type> --------------------------
    !        !
    + PACKED +
```

<base type> syntax:

```
---- <ordinal type> ------------------------------------------------
```

Subrange Types

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.

<subrange type> syntax:

```
----+-- <new subrange type>--------+----------------------------------
    !                              !
    +-- <subrage type identifer> --+
```

<new subrange type> syntax:

```
----+-- <Boolean constant> .. <Boolean constant> ---------+-----------!
    !                                                     !
    +-- <char constant> .. <char constant> ---------------+
    !                                                     !
    +-- <enumerated constant> .. <enumerated constant> ---+
    !                                                     !
    +-- <integer constant> .. <integer constant> ---------+
```

Textfile Types

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.

Example:

```
    type streamfile = text;
```

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

<textfile type> syntax:

```
----+------------ TEXT --------------+------------------------------
    !                                !
    +-- <textfile type identifier> --+
```

VARIABLE DECLARATIONS

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 in this 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.

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;
```

Variables x, y, z, and max are of type real, variables i and j are of type integer, and variables p, q, and r are of type Boolean.

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

The variable operator is 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. 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.)

The variable c is a variable of the <enumerated type identifier> color and may contain a value of red, yellow, blue, green, or tartan. Variables hue1 and hue2 are both of type "set of color." They may contain any subset of the <enumerated type identifier> color.

The variable date is 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.

<variable declarations> syntax:

```
           +<--------------------------------------------------+
           !                                                   !
---- VAR --+-- <variable identifier list> -- : -- <type> -- ; --+-----
```

<variable identifier list> syntax:

```
     +<----------- , ------------+
     !                   !       !
----+-- <variable identifier> --+-----------------------------------
```

<variable identifier> syntax:

```
---- <identifier> ----------------------------------------------------
```

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.

<procedure and functions declarations> syntax:

```
     +<-----------------------------------+
     !                                    !
----+--+-- <procedure declaration> ---+--+-- ; ----------------------
        !                         !     !
        +-- <function declaration> ----+
```

The declarations used to define procedures and functions are described under the headings Procedure Declaration and Function Declaration in the pages that follow.

A procedure or function can have an associated list of parameters. This allows the the values and variables on which the procedure or function is to operate to be specified at run time. The parameter list occurring in the declaration is called the formal parameter list because the parameter names do not refer to actual variables; 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 syntax and semantics of formal parameter lists are provided under the heading Formal Parameter Lists, later in this section. Formal parameter lists are identical for both procedures and functions. functions.

The syntax and semantics of actual parameter lists and information on the matching of actual parameters with formal parameters when a procedure or function is invoked are provided under Actual Parameter Lists and Parameter Matching.

Procedure Declaration

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

<procedure declaration> syntax:

```
---- PROCEDURE  <procedure identifier> --+-------------------------+--
                                         !                         !
                                         + <formal parameter list> +

>--- ; ----+---------------+----------------------------------------!
           !               !
           +-- <directive> --+
```

<procedure identifier> syntax:

```
---- <identifier> ----------------------------------------------------
```

<directive> syntax:

```
---- <forward> -------------------------------------------------------
```

Before a procedure is invoked by a <procedure invocation statement>, the <procedure identifier> and the formal parameters of the procedure must be defined. Such a definition can be provided either in a forward declaration or in an actual declaration for the procedure. A forward declaration is a <procedure declaration> that includes the forward <directive>. When a procedure is forward-declared, an actual procedure declaration 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; it must not 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.

Examples:

```
program procedure_decs;
type arraytype = array [0..10] of integer;
var  x, y : arraytype;
     m, n : integer;
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 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;

begin
m := 5;
n := 1000;
proc1;
proc2(m,n);
proc5;
end.
```

Procedure proc1 has no parameters.

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

Procedure proc4 has a <variable parameter> of type arraytype. Because procedure proc4 contains a call on procedure proc5 (and proc5 has a call on proc4), procedure 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.

Function Declaration

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.

<function declaration> syntax:

```
---- FUNCTION -- <function identifier> -----------------------------

>-+--------------------------+ : <result type> ; --+---------------+-
  !                          !                     !               !
  +- <formal parameter list> -+                    +- <directive> -+
```

<function identifier> syntax:

```
---- <identifier> --------------------------------------------------
```

<result type> syntax:

```
----+-- <simple type>----+------------------------------------------
    !                    !
    +-- <pointer type> --+
```

<directive> syntax:

```
---- <forward> -----------------------------------------------------
```

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 Concepts.) 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.

Before a function is invoked by a <function designator>, the <function identifier>, the formal parameters, and the <result type> of the function must be defined. This definition can be provided either in a forward declaration or in an actual declaration for the function. A forward declaration is a <function declaration> that includes the forward <directive>. When function is declared forward, an actual function declaration (that is, a <function declaration> 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.

Examples:

```
program function_decs;
type sub1   = 1..10;
     letter = 'A'..'Z';
var  b: Boolean;
     c: letter;
     inx : integer;
     offset : sub1;

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

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

begin
b := func1;
offset := func2(10);
c := func5;
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.

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

Function func5 is of type char and has no parameters.

Formal Parameter Lists

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.

<formal parameter list> syntax:

```
           +<----------- , ---------------+
           !                              !
---- ( --+--+--- <value parameter> ----+--+-- ) --------------------
            !                          !
            +-- <variable parameter> --+
```

<value parameter> syntax

```
    +<---------- , -------------+
    !                           !
----+-- <variable identifier> --+-- : -- <value parameter type> ------
```

<value parameter type> definition:

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> syntax:

```
            +<---------- , -------------+
            !                           !
---- VAR --+-- <variable identifier> --+ : <variable parameter type> -
```

<variable parameter type> syntax:

```
---- <type identifier> ---------------------------------------------
```

Parameters are declared by their appearance in a parameter list. They have associated identifiers, which are 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 two kinds of parameters, value and variable, 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.

ACTUAL PARAMETER LISTS AND PARAMETER MATCHING

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.

<actual parameter list> syntax:

```
          +<---------  ,  -----------+
          !                          !
---- ( --+--+-- <expression> --+--+-- ) ------------------------------
            !                  !
            +--- <variable> ---+
```

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 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. io.parameter list congruity

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

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

2. Corresponding parameters are of the same kind (value and variable).

3. corresponding parameters are of the same type.

SECTION 4

STATEMENTS

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.

<statement part> syntax:

```
---- BEGIN -- <statement list> -- END --------------------------------
```

<statement list> syntax:

```
    +<------ , -------+
    !                 !
----+-- <statement> --+-------------------------------------------------
```

<statement> syntax:

```
----+-----------------+-+---------------------------------------+-----
    !                 ! !                                       !
    +-- <label> -- : --+ +-- <assignment statement> ------------+
                         !                                       !
                         +-- <case statement> ------------------+
                         !                                       !
                         +-- <compound statement> --------------+
                         !                                       !
                         +-- <for statement> -------------------+
                         !                                       !
                         +-- <goto statement> ------------------+
                         !                                       !
                         +-- <if statement> --------------------+
                         !                                       !
                         +-- <procedure invocation statement> --+
                         !                                       !
                         +-- <repeat statement> ----------------+
                         !                                       !
                         +-- <while statement> -----------------+
                         !                                       !
                         +-- <with statement> ------------------+
```

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 under Label Declarations in section 3. Restrictions on references to labels in <goto statement>s are described under GOTO Statements in this section.

ASSIGNMENT STATEMENTS

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

<assignment statement> syntax:

```
---- <variable> -------------- := --+-- <expression> -----------+-----
                                    !                           !
                                    +-- <function identifier> --+
```

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..3] of char;
```

```
begin
s := 'abc';
end;
```

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

CASE STATEMENTS

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.

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.

<case statement> syntax:

```
                                   +<------ ; -----------+
                                   !                !    !
---- CASE <case index>  OF --+ <case list element> +--+---+-----------
                                                      !   !
                                                      + ; +

>---+---------------------------------+-------- END ---------------
    !                                 !
    +-- OTHERWISE -- <statement list> --+
```

<case index> syntax:

```
---- <ordinal expression> -----------------------------------------
```

<case list element> syntax:

```
     +<------- , ----------+
     !                     !
----+-- <case constant> --+-- : -- <statement> ----------------------
```

COMPOUND STATEMENTS

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>).

<compound statement> syntax:

```
---- BEGIN -- <statement list> -- END -------------------------------
```

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

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."

<for statement> syntax:

```
---- FOR -- <control variable> -- := -- <initial value> --+--- TO ---+
                                                          !          !
                                                          +- DOWNTO -+

>--- <final value> -- DO -- <statement> -----------------------------
```

<control variable> definition:

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

<initial value> syntax:

```
---- <ordinal expression> -------------------------------------------
```

<final value> syntax:

```
---- <ordinal expression> -------------------------------------------
```

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>:

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

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

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

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

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

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

<goto statement> syntax:

```
---- GOTO -- <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:

1. 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>.

2. 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.

Example 1:

```
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 example 1, the branches to labels 10, 20, and 100 are valid by rule 1. The branch to label 9999 is valid by rule 2.

Example 2:

```
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.
```

In example 2, 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

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>.

<if statement> syntax:

```
---- IF -- <Boolean expression> -- THEN -- <statement>---------------

>----+------------------------+-------------------------------------
     !                        !
     +-- ELSE -- <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 named 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 various 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

STRING RELATION

A <string relation> performs a sequential comparison of the ordinal numbers of corresponding characters in the two <string expression>s. The <string expression>s must be of the same length.

<string relation> syntax:

---- <string expression> -- <rel op> -- <string expression> ----------

Two <string expression>s are equal if every character in both strings is identical. A <string expression> is less than another <string expression> if, in the first character position that differs between the two <string expression>s, the first <string expression> contains a character of a lower ordinal number than the corresponding character in the second string.

Example:

```
var b : Boolean;
    sl, s2 : packed array [1..10] of char;
begin
s1 := 'abcdefghij';
s2 := 'abcdefghiz';
b  := s1 < s2;
end;
```

The string s1 is compared, character by character, to string s2. The variable b is assigned the value true because, at the first character position at which the strings differ (j and z at character 9), the ordinal number of j is less than the ordinal number of z.

PROCEDURE INVOCATION STATEMENTS

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>.

<procedure invocation statement> syntax:

```
----+-- <declared procedure> ----+-----------------------------------
    !                            !
    +-- <predefined procedure> --+
```

<declared procedure> syntax:

```
---- <procedure identifier> --+----------------------------+---------
                              !                            !
                              +-- <actual parameter list> --+
```

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 section 6.

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 in section 3 for additional information.

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

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

<repeat statement> syntax:

```
-- REPEAT --<statement list>-- UNTIL --<Boolean expression>--!
```

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

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.

<while statement> syntax:

```
---- WHILE -- <Boolean expression> -- DO -- <statement> -------------
```

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

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.

<with statement> syntax:

```
---- WITH -- <record variable> -- DO -- <statement> -----------------
```

When multiple <record variable>s are specified, the effect is as if the <record variable>s were specified in nested <with statement>s. The leftmost <record variable> is assigned the most global scope and the rightmost 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.

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.

SECTION 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).

<expression> syntax:

```
----+-- <array variable> --------+-----------------------------------
    !                            !
    +-- <Boolean expression> ----+
    !                            !
    +-- <char expression> -------+
    !                            !
    +-- <enumerated expression> -+
    !                            !
    +-- <integer expression> ----+
    !                            !
    +-- <pointer expression> ----+
    !                            !
    +-- <real expression> -------+
    !                            !
    +-- <record expression> -----+
    !                            !
    +-- <set expression> --------+
    !                            !
    +-- <string expression> -----+
```

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.

ARITHMETIC EXPRESSIONS

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.

<arithmetic expression> syntax:

```
----+-- <integer expression> --+-------------------------------------
    !                          !
    +-- <real expression> -----+
```

ORDINAL EXPRESSIONS

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.

<ordinal expression> syntax:

```
----+-- <Boolean expression> -----+----------------------------------
    !                             !
    +-- <char expression> --------+
    !                             !
    +-- <enumerated expression> --+
    !                             !
    +-- <integer expression> -----+
```

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.

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. The precedence of operators is defined in 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 and set intersection (*), real division (/), integer division (DIV), integer remainder division (MOD), Boolean AND, and Boolean conditional 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 Boolean conditional OR (COR).

The lowest precedence operators are the relational operators. These operators, which apply to several data types, are described under Relational Expressions in this section.

Other languages, such as FORTRAN and ALGOL, define a higher precedence for the relational 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 yields 56}
 A + (B * C)     {Identical to default interpretation yields 38}
```

FUNCTION DESIGNATORS

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

<function designator> syntax:

```
----+-- <declared function> ----+-----------------------------------
    !                           !
    +-- <predefined function> --+
```

<declared function> syntax:

```
---- <function identifier> --+----------------------------+----------
                             !                            !
                             +-- <actual parameter list> --+
```

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 section 6, Predefined Procedures and Functions.

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 Actual Parameter Lists and Parameter Matching in section 3 for additional information.

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.

EXPRESSIONS BY TYPE

Expression types, in alphabetical sequence, are described in the paragraphs that follow.

Boolean and Relational Expressions

A <Boolean expression> generates a value of the <Boolean type>. A relational expression generates a Boolean value by comparing two operatands of the same type or of similar types.

Boolean Expressions

Following are syntax diagrams for Boolean expressions.

<Boolean expression> syntax:

```
     +<--------- <Boolean operator> ------+
     !                                    !
----+--+---------+-- <Boolean primary> --+----------------------------
       !         !
       +-- NOT --+
```

<Boolean operator> syntax:

```
----+-- AND ---+--------------------------------------------------
    !          !
    +-- CAND --+
    !          !
    +-- OR ----+
    !          !
    +-- COR ---+
```

<Boolean primary> syntax:

```
----+-- ( <Boolean expression> ) --+-------------------------------!
    !                              !
    +-- <Boolean constant>---------+
    !                              !
    +-- <Boolean variable>---------+
    !                              !
    +-- <function designator>------+
    !                              !
    +-- <relational expression> ---+
```

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 Constant Definitions in section 3, <Boolean variable> is defined in section 7, Variables, and <function designator> and <relational operator> are defined 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>.

The CAND and COR operators are Burroughs extensions to ANSI Pascal.

Examples:

```
var b1, b2, b3 : Boolean;
begin
{The following two 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.
```

NOTE

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

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

<relational expression> syntax:

```
----+-- <arithmetic relation>--+------------------------------------
    !                          !
    +-- <ordinal relation> ----+
    !                          !
    +-- <set relation>---------+
    !                          !
    +-- <string relation>------+
```

<rel op> syntax:

```
----+-- =  --+-------------------------------------------------------
    !        !
    +-- <> --+
    !        !
    +-- <  --+
    !        !
    +-- >  --+
    !        !
    +-- <= --+
    !        !
    +-- >= --+
```

<arithmetic relation> syntax:

```
---- <arithmetic expression> -- <rel op> -- <arithmetic expression> --
```

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> syntax:

```
----+----- <Boolean expression> <rel op> <Boolean expression> -----+--
    !                                                               !
    +--------- char expression> <rel op> <char expression> --------+
    !                                                               !
    +-- <enumerated expression> <rel op> <enumerated expression> --+
    !                                                               !
    +----- <integer expression> <rel op> <integer expression> -----+
```

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 the above, i > 5, color < green, and c = 'Z' are illustrations of <ordina relation>s.

<pointer relation> syntax:

```
----+-- <pointer expression> --+ =  +-- <pointer expression> ---------
                               !    !
                               + <> +
```

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> syntax:

```
----+-- <set expression> --+-- = ---+-- <set expression> ----+-------
    !                      !        !                        !
    !                      +-- <> --+                        !
    !                      !        !                        !
    !                      +-- <= --+                        !
    !                                                        !
    +-- <ordinal expression> -- IN -- <set expression> ------+
```

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 (that is, 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.

CHAR Expressions

A <char expression> generates a value of the <char type>. <char constant> is defined in the Constant Definitions section, <char variable> in the Variables introduction, and <function designator> later in that introduction.

<char expression> syntax:

```
----+-- <char constant>---------+-----------------------------------
    !                           !
    +-- <char variable> --------+
    !                           !
    +-- <function designator> --+
```

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>.

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

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

<enumerated expression> syntax:

```
----+-- <enumerated constant> --+-----------------------------------
    !                           !
    +-- <enumerated variable> --+
    !                           !
    +-- <function designator> --+
```

The <enumerated constant> is defined under Enumerated Types in section 3, <enumerated variable> under Variables, section 7, and <function designator> in this section.

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>.

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

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).

<integer expression> syntax:

```
                   +<-- <integer operator> --+
                   !                         !
----+-------+--+--- <integer primary> ---+--------------------------
    !       !
    +-- + --+
    !       !
    +-- - --+
```

<integer operator> syntax:

```
----+-- +   --+-------------------------------------------------
    !         !
    +-- -   --+
    !         !
    +-- *   --+
    !         !
    +-- DIV --+
    !         !
    +-- MOD --+
```

<integer primary> syntax:

```
----+-- ( <integer expression> ) ---------+-------------------------
    !                                     !
    +-- MAXINT ---------------------------+
    !                                     !
    +-- <unsigned integer>----------------+
    !                                     !
    +-- <integer constant identifier> ----+
    !                                     !
    +-- <integer variable> ---------------+
    !                                     !
    +-- <function designator> ------------+
```

The <insigned integer> is defined in section 8, Basic Components, <integer constant identifier> under Constant Definitions in section 3, <integer variable> in section 7, and <function designator> in this section.

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>.

Examples:

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

Pointer Expressions

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

<pointer expression> syntax:

```
----+-- NIL ---------------------+-------------------------------------
    !                            !
    +-- <pointer variable> ------+
    !                            !
    +-- <function designator> ---+
```

The constant NIL denotes a null reference (a pointer that is not currently referencing a variable). The <pointer variable> is defined in section 7 and <function designator> is defined in this section.

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

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

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.

<real expression> syntax:

```
              +<---- <arithmetic operator> --+
              !                              !
----+-----+--+--+-- <real primary> -----+---+------------------------
    !     !     !                       !
    +- + -+     +-- <integer primary> --+
    !     !
    +- - -+
```

<arithmetic operator> syntax:

```
----+--  +  --+--------------------------------------------------
    !         !
    +--  -  --+
    !         !
    +--  *  --+
    !         !
    +-- DIV --+
    !         !
    +-- MOD --+
```

<real primary> syntax:

```
----+-- ( <arithmetic expression>-- ) --+--------------------------
    !                                   !
    +-- <unsigned real> ----------------+
    !                                   !
    +-- <real constant identifier> -----+
    !                                   !
    +-- <real variable> ----------------+
    !                                   !
    +-- <function designator> ----------+
```

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 section 8, Basic Components, <real constant identifier> under Constant Definitions in section 3, <real variable> in section 7, and <function designator> in this section.

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

Examples:

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

Set Expressions

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

<set expression> syntax:

```
    +<-- <set operator> --+
    !                     !
----+--- <set primary> ---+--------------------------------------------
```

<set operator> syntax:

```
----+--   +   --+-------------------------------------------------------
    !           !
    +--   -   --+
    !           !
    +--   *   --+
```

<set primary> syntax:

```
----+-- ( <set expression> ) --+-----------------------------------------
    !                          !
    +-- <set variable> --------+
    !                          !
    +-- <set constructor> -----+
```

<set constructor> syntax:

```
---- [ --+-----------------------------+-- ] -------------------------
         !                             !
         !  +<--------  ,  ------------+
         !  !                          !
         +--+-- <member designator> --+
```

<member designator> syntax:

```
---- <ordinal expression> --+-----------------------------------+--------
                            !                                   !
                            +-- .. -- <ordinal expression> --+
```

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>.

The <set variable> is defined in section 7.

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

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

<string expression> syntax:

```
----+-- <char expression> --+------------------------------------
    !                       !
    +-- <string constant> --+
    !                       !
    +-- <string variable> --+
```

The <string constant> is defined under Constant Definitions in section 3, and <string variable> is defined in section 7.

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'.

SECTION 6

PREDEFINED PROCEDURES AND FUNCTIONS

Following this introduction, this section has two major parts: INPUT/OUTPUT AND FILE-HANDLING CONCEPTS and PROCEDURE AND FUNCTION DESCRIPTIONS.

The first part presents input/output (I/O) concepts pertaining to Pascal. Some basic terminology is covered and information is presented on files (standard files and textfiles) and related I/O operations, and file attributes. Many of the Burroughs extensions to ANSI Pascal pertain to I/O to enable Pascal programs to access the system-defined I/O subsystem. Programmers who are interested in writing portable programs are advised to become familiar with this material.

The second part is a glossary of all the procedures and functions, grouped according to program application and, within that grouping, in alphabetic order.

Many Pascal features, including I/O facilities and dynamic variables, are made available through predefined procedures and functions. Although procedures and functions are syntactically different constructs, that difference is not emphasized in this section.

<predefined procedure> syntax:

```
----+-- <file handling procedure>--------+---------------------------
    !                                    !
    +-- <dynamic allocation procedure> --+
    !                                    !
    +-- <general procedure>--------------+
```

<predefined function> syntax:

```
----+-- <file handling function> --------+---------------------------
    !                                    !
    +-- <type transfer function> --------+
    !                                    !
    +-- <arithmetic function> -----------+
    !                                    !
    +-- <general function> --------------+
```

INPUT/OUTPUT AND FILE-HANDLING CONCEPTS

The file handling procedures and functions are the basic mechanisms 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 in the second part of this section, under the heading File Handling Procedures and Functions. The general syntax is presented here.

<file handling procedure> syntax:

```
----+-- <close procedure> ----------+----------------------------------
    !                               !
    +-- <get procedure> ------------+
    !                               !
    +-- <page procedure> -----------+
    !                               !
    +-- <put procedure> ------------+
    !                               !
    +-- <read procedure> -----------+
    !                               !
    +-- <read textfile procedure> --+
    !                               !
    +-- <readln procedure> ---------+
    !                               !
    +-- <reset procedure> ----------+
    !                               !
    +-- <rewrite procedure> --------+
    !                               !
    +-- <seek procedure> -----------+
    !                               !
    +-- <write procedure> ----------+
    !                               !
    +-- <write textfile procedure> -+
    !                               !
    +-- <writeln procedure>---------+
```

<file handling function> syntax:

```
----+-- <eof function>----+--------------------------------------------
    !                     !
    +-- <eoln function> --+
```

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 in this section.

Standard Files and Textfiles

In Pascal, there are two types of files: standard files (files of any <component type>), and textfiles (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 treated 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 and textfiles, allowing the files to be both read and written. The B 1900 implementation uses the inspection/generation mode only.

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.

Logical and Physical Files

As viewed by a program, a file is a logical entity that is read or written somewhat independently of the characteristics of the device involved. In terms of the device used to create it or the medium upon which it is stored, however, a file 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 either the reset procedure or the rewrite procedure.

The desired physical file may be a new file or an existing file. If a file is opened using the <reset procedure> an existing file is assumed. If the <rewrite procedure> is used, a new file is created.

The decision as to which physical file will be assigned is controlled by the values 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 10 characters (translated to upper case) of the <variable identifier> of the file or textfile.

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 operation that does not specify the save or crunch option. 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 two ways:

(1) If the file name appears in the <program heading>, the file will become a permanent file when it is closed.

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

In both cases, an existing permanent file replaced by a saved file with the same name, but it is not replaced until the close operation is executed.

A permanent file can be explicitly removed by executing a close operation 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 the file 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 file 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.

Operations on standard files are described next.

Reset Operation

The reset operation assumes that a file already exists. The file may be open or closed. If the file is open, it is repositioned at the beginning of the file. 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
*    +

*      current value of the buffer variable
+      next component to be accessed
Xn     last component of the file
eof    special component marking end of file
```

Get Operation

Get, the fundamental input operation, causes the file component indicated by + to be transferred to the buffer variable; it then positions the file to the next component. After performing a get operation, the file is positioned as follows:

```
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 causes the buffer variable to become undefined. In addition, the <eof function> returns the value true if called. (Until now, the <eof function> returned false.) If get is called when the file is at end-of-file, an error occurs.

Read Operation

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.

Seek Operation

The seek operation is an additional function defined as a Burroughs extension; it allows a file to be accessed randomly. The command 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.

```
XO  ...  ...  Xi ... ...  eof
              +
```

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

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

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

Rewrite Operation

A rewrite operation 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.

Put Operation

The put operation 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 to be positioned such that a subsequent put operation 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-->
XO ... ... ... ... Xi
                     +
```

Write Operation

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 the write operation. For example, x must be assignment-compatible with the type of f@).

When a file 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:

```
XO  X1  O ... Xi  Xi+1  O  Xn  eof
```

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

Close Operation

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

Textfiles (Including Predefined 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.

Textfiles in General

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.

Predefined Textfiles (Input, Output)

There are two predefined textfiles with the names "input" and "output." In order to use these files, their names must appear in the list of <program parameters>. When they appear, they become implicitly declared; thus, they must not be declared again in the <variable declarations> of the program. If the names 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.

Operations on textfiles are described next.

Reset Operation

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

```
CO  C1  ...   ...   ...  Cn  eoln
*   +

CO  ... Cm  eoln

CO ...   ...  Cz  eoln  eof

*       currently defined value of the buffer variable
+       next component to be accessed.
eoln    end-of-line marker
eof     end-of-file marker
```

Eoln exists as a functional definition only; such a character is not actually present in the file, but is implied by position.

Get Operation

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 operation accesses the next component of the file. When the file is in the following position, another get operation will put the file in end-of-line state:

```
CO  C1  ...  ...  ...  Cn  eoln
                        *    +
```

In end-of-line state, the <eoln function>, if called, returns the value true and the value of the buffer variable is ' ' (blank). A second get operation results in the following file position:

```
CO  C1  ...  ...  ...  Cn  eoln

CO  ... Cm  eoln
*    +

CO ...  ...  Cz  eoln  eof
```

When the file is positioned as follows, a get operation again puts the file into end-of-line state, and a second get operation puts the file into end-of-file state:

```
CO  C1  ...  ...  ...  Cn  eoln

CO  ... Cm  eoln

CO ...  ...  Cz  eoln  eof
             *     +
```

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

Read Operation

The read operation has special semantics for textfiles. The definition of a read operation depends on the type of the variables in the parameter list. The action of the read operation on a textfile is described under Read Teaxfile Procedure.

Readln Operation

The readln operation 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 such as readln(f,X1,...,Xn) is equivalent to the following statements:

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

Rewrite Operation

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

Put Operation

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.

Write Operation

The write operation has special semantics for textfiles. The definition of write depends on the type of the variables in the parameter list. The action of write on a textfile is described under Write Textfile Procedure.

Writeln Operation

The current line is terminated by the writeln operation. A multiple-value writeln operation such as 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.

Close Operation

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 prompts a user for 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 thus have to wait for the user to respond to a prompt that has not yet been displayed.

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 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 another get, read, or readln operation is performed.

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

Use of File Attributes

Burroughs Pascal, together with the B 1000 I/O subsystem, provides several methods for assigning and interrogating the values of file attributes. File attributes can be assigned in the following ways:

1. Through file equation as the program is executed.

2. By specification of the file attributes in the <program parameters>.

3. Dynamically, through the <setattribute procedure>.

When settings from these methods conflict, precedence is determined by the following sequence (highest to lowest): (1) <setattribute procedure>, (2) run-time file equation, (3) settings in the <program parameters>.

PROCEDURE AND FUNCTION DESCRIPTIONS

Described next, in alphabetic order within groups, are all the procedures and functions available in B 1000 Pascal. The groups are

- File-Handling Procedures and Functions
- Type Transfer Functions
- Dynamic Allocation Procedures
- Arithmetic Functions
- General Procedures and Functions

File-Handling Procedures and Functions

Following are descriptions of all the file-handling procedures and functions.

Close Procedure

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

<close procedure> syntax:

```
---- CLOSE -- ( -+- <textfile variable> -+-+--------------------+- ) -
                 !                       ! !                    !
                 +- <file variable> -----+ +- , <close option> -+
```

<close option> syntax:

```
----+-- CRUNCH --+-----------------------------------------------
    !            !
    +-- PURGE ---+
    !            !
    +-- SAVE ----+
```

After a close operation, 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 <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 of the file if the device permits this. 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 is set to true, which has the effect of returning unused storage areas to the system. The connection between the logical file and physical file is severed. The crunch option is valid for disk files only.

purge — The purge option causes the file to be discarded. A tape file is rewound, and, if a 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. The purge option is valid for tape and disk files only.

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. The save option 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.

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. The function returns true if the last operation on the file was a get, read, or reset beyond the last component.

<eof function> syntax:

```
---- EOF --+--------------------------------+-------------------------
           !                                !
           +-- ( -- <file variable> -- ) --+
           !                                !
           +---- <textfile variable> ------+
```

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 named input If the file is not open, the function returns false. If the specified file is not open when the <eof function> is called, an error occurs.

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.

<eoln function> syntax:

```
---- EOLN --+-------------------------------------+--------------------
            !                                     !
            +-- ( -- <textfile variable> -- ) --+
```

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 named input.

If the specified file is not open when the <eoln function> is called, an error occurs.

Get Procedure

The <get procedure> assigns to the buffer variable of the file denoted by <textfile variable> or <file 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.

<get procedure> syntax:

```
---- GET -- ( --+-- <textfile variable> --+-- ) ---------------------
                !                          !
                +-- <file variable>-------+
```

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 not open. If, immediately preceding the invocation of the get procedure, the <eof function> yields the value true, an error occurs if the <eof function> still yields true following the invocation.

Page Procedure

The <page procedure> 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.

<page procedure> syntax:

```
---- PAGE --+-----------------------------------+---------------------
            !                                   !
            +-- ( -- <textfile variable> -- ) --+
```

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 not open prior to the execution of the <page procedure>.

Put Procedure

The <put procedure> writes to the file denoted by <textfile variable> or <file variable> the value of the buffer variable associated with that file. The value of the buffer variable then becomes undefined.

<put procedure> syntax:

```
---- PUT -- ( --+-- <textfile variable> --+-- ) ---------------------
                !                         !
                +-- <file variable>-------+
```

An error occurs 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.

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).

<read procedure> syntax:

```
---- READ -- ( -- <file variable> -- , -- <variable> -- ) -----------
```

Read Textfile Procedure

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 predefined textfile named input.

<read textfile procedure> syntax:

```
                                          +<------- , ---------+
                                          !                    !
---- READ -- ( -+----------------------+-+-- <read parameter> -+-- ) -
                !                      !
                + <textfile variable> , +
```

<read parameter> syntax:

```
----+-- <char variable>------+-------------------------------------
    !                        !
    +-- <integer variable> --+
    !                        !
    +-- <real variable> -----+
```

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 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>, as explained next.

Type = <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.

Type = <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, it is put into eol 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 assigns the value -123 to the variable i and leaves the buffer variable positioned at the location indicated by the second asterisk. (d is not a valid character in an integer.)

Type = <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, it is put into eol 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 assigns the value 98.6 to the variable r and leaves 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 assigns the value -1234 times 10 to the power of -27 to the variable r and leaves the buffer variable positioned at the location indicated by the second asterisk.

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.

<readln procedure> syntax:

```
---- READLN ------------------------------------------------------------>

>-+-----------------------------------------------------------+-------
  !                                                           !
  !                             +<-------- , --------+        !
  !                             !                    !        !
  + ( +------------------------+-+- <read parameter> -+-+ ) -+
      !                        !                        !
      +- <textfile variable>  , -+                      !
      !                                                 !
      +- <textfile variable> ---------------------------+
```

If no <textfile variable> is specified, the <readln procedure> applies to the textfile named input.

An error occurs 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.

Reset Procedure

The <reset procedure> positions the file to the beginning. 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.

<reset procedure> syntax:

```
---- RESET ---- ( --+-- <file variable> ------+-- ) -----------------
                    !                         !
                    +-- <textfile variable> --+
```

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 10 characters (translated to upper case) of the <file variable> or <textfile variable> identifier. If the identifier is the predefined file identifier "input," a search is made for a remote file. This search can be modified by changing certain file attributes, such as TITLE, or through file equation.

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.

Following a <reset procedure>, the file is 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 the file.

Rewrite Procedure

The <rewrite procedure> creates a new, empty file. 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 10 characters (translated to upper case) of the <file variable> or <textfile variable> identifier is created. (If the identifier is the predefined file identifier "output," a remote file is created.)

<rewrite procedure> syntax:

```
---- REWRITE -- ( --+-- <file variable> ------+-- ) -----------------
                    !                         !
                    +-- <textfile variable> --+
```

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> returns true as long as the file is in generation mode.

Seek Procedure

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 (that is, zero relative). If the value of the specified <integer expression> is less than 0, an error occurs.

<seek procedure> syntax:

```
---- SEEK -- ( -- <file variable> -- , -- <integer expression> -- ) --
```

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

Write Procedure

The <write procedure> causes the specified <expression>s to be written sequentially to the file denoted by <file variable>.

<write procedure> syntax:

```
---- WRITE -- ( -- <file variable> -- , -- <expression> -- ) ---------
```

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 not open.

Write Textfile Procedure

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 named output.

<write textfile procedure> syntax:

```
                                          +<------- , ---------+
                                          !                    !
---- write   ( -+----------------------+-+ <write parameter> -+-- ) -
                !                      !
                + <textfile variable> , +
```

<write parameter> syntax:

```
----+-- <Boolean expression> --+-+---------------------------------+---
    !                          ! !                                 !
    +-- <char expression> -----+ + : <field width>-----------------+
    !                          !                                   !
    +-- <integer expression> --+                                   !
    !                                                              !
    +-- <real expression> --+--------------------------------------+
                            !                                      !
                            + : <field width> --+------------------+
                                                !                  !
                                                + : <frac digits> +
```

<field width> syntax:

```
---- <integer expression> -------------------------------------------
```

<frac digits> syntax:

```
---- <integer expression> -------------------------------------------
```

An error occurs if the textfile is not open. Also, an error occurs if the 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 in the following paragraphs.

<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:

Procedure	Result
write(f,b)	" TRUE" if b is true "FALSE" if b is false
write(f,true:2)	"TR"
write(f,true:10)	" TRUE"

Quotation marks show spacing.

<char expression>

For a value of the <char type>, the character is simply moved to the buffer variable and "put" into the file. The default <field width> for a <char expression> is 1 character. If a <field width> greater than 1 is specified, leading blanks are written.

Examples: (c is a <char variable> that contains the value $)

Procedure	Result
write(f,c)	"$"
write(f,c:3)	" $"

Quotation marks show spacing.

<integer expression>

Values of the <integer type> are formatted with a sign (minus 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: (i is an integer with value -12345)

Procedure	Result
write(f,i)	" -12345"
write(f,i:3)	"-12345"
write(f,i:12)	" -12345"

Quotation marks show spacing.

<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.

Floating-Point Format

In floating-point format, the number contains the following components:

1. A sign; minus if the number is negative, blank if it is positive.

2. The first significant digit, or zero, if the number is zero.

3. A decimal point (.)

4. The fractional part (at least one digit).

5. The exponent symbol (E)

6. The sign of the exponent (+ or -).

7. 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 only one fractional digit. If the specified <field width> is larger, the number is expanded by adding trailing zeros to the fractional part.

Fixed-Point Format

In fixed-point format, the number contains the following components:

1. A minus sign (-) if the number is negative.

2. The integral part of the number -- trunc(<real expression>).

3. A decimal point (.).

4. <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 the 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 representation of the number, the number is rounded at the last digit written.

Examples:

```
Procedure                        Result
------------------------         ------------------------------
write(f, 1.2345:6:4)             "1.2345"
write(f,1.2345:20)               " 1.2344999313354E+00"
write(f,-27.1828E-3:14)          "-2.7182801E-02"
write(f,0.31:3)                  " 3.1E-01"
write(f,-96E12:7)                "-9.6E+13"
write(f,0.317269:3)              " 3.2E-01"
write(f,-965E12:7)               "-9.6E+14"
write(f,0.31726E7:7:3)           "3172600.031"
write(f,-965E12:1:7)             "-964999961853027.3437500"
write(f,0.31726E7:13:3)          "  3172600.031"
write(f,-965E-2:12:7)            "  -9.6499996"
write(f,3.1776E-1:13:3)          "        0.318"
write(f,-962.5E-2:12:2)          "      -9.625"
```

Quotation marks show spacing.

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 named output. If no <write parameter>s are specified, a single blank line is written to the textfile named output. Following the execution of the <writeln procedure>, the value of the buffer variable becomes undefined.

An error occurs if the file is not open.

<writeln procedure> syntax:

```
---- WRITELN ----------------------------------------------------->

>-+-----------------------------------------------------------+------
  !                                                           !
  !                                 +<-------- , --------+     !
  !                                 !                    !     !
  + ( +-------------------------+-+- <write parameter> +-+ ) -+
      !                         !                        !
      +- <textfile variable>  , -+                        !
      !                                                   !
      +- <textfile variable> -----------------------------+
```

Type Transfer Functions

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.

<type transfer function> syntax:

```
----+-- <chr function> --+---------------------------------------
    !                    !
    +-- <ord function> --+
```

CHR Function

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.

<chr function> syntax:

```
---- CHR -- ( -- <integer expression> -- ) -------------------------
```

Examples:

```
var c1, c2 : char;
begin
c1 := chr(129);
c2 := chr(240);
end;
```

The character a is assigned to c1 and the character 0 is assigned to c2

ORD Function

The <ord function> returns, as an integer value, the ordinal number of the specified <ordinal expression>.

<ord function> syntax

```
---- ORD -- ( -- <ordinal expression> -- ) ---------------------------
```

Examples:

```
var i1, i2 : integer;
begin
i1 := ord('a');
i2 := ord(true);
end;
```

In the standard EBCDIC character set, i1 is assigned the integer value 129 and I2 is assigned the integer value 1.

Dynamic Allocation Procedures

The dynamic allocation procedures, used in conjunction with <pointer <pointer variables>, 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 allocation procedure> syntax:

```
----+-- <mark procedure--------+-------------------------------------!
    !                          !
    +-- <new procedure> -------+
    !                          !
    +-- <release procedure> --+
```

Dynamic variables are allocated in a storage area called the "heap." Creation of dynamic variables and manipulation of the heap is performed through the use of the three predefined procedures new, 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 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 the mark procedure 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 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.

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, that is, the current top-of-stack variable. Subsequent calls to the <new procedure> allocate dynamic variables "above" this mark; such variables are referred to as marked variables.

<mark procedure> syntax:

```
---- MARK -- ( -- <pointer variable> -- ) ----------------------------
```

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.

New Procedure

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.

<new procedure> syntax:

```
---- NEW -- ( -- <pointer variable> -- ) -----------------------------
```

Release Procedure

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.)

<release procedure> syntax:

```
---- RELEASE -- ( -- <pointer expression> -- ) ----------------------
```

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.

Arithmetic Functions

The <arithmetic function>s provide functions for use in <arithmetic expression>s.

<arithmetic functions> syntax:

```
----+-- <abs function> ----+-----------------------------------------
    !                      !
    +-- <arctan function> -+
    !                      !
    +-- <cos function> ----+
    !                      !
    +-- <exp function> ----+
    !                      !
    +-- <ln function> -----+
    !                      !
    +-- <round function> --+
    !                      !
    +-- <sin function> ----+
    !                      !
    +-- <sqr function> ----+
    !                      !
    +-- <sqrt function> ---+
    !                      !
    +-- <tan function> ----+
    !                      !
    +-- <trunc function> --+
```

ABS Function

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>.

<abs function> syntax:

```
---- ABS -- ( -- <arithmetic expression> -- ) -----------------------
```

ARCTAN Function

The <arctan function> returns, as a real value in radians, the principal value of the arctangent function at the specified <arithmetic expression>.
p1.<arctan function> syntax:

```
---- ARCTAN -- ( -- <arithmetic expression> -- ) --------------------
```

COS Function

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.

<cos function> syntax:

```
----  COS -- ( -- <arithmetic expression> -- ) ----------------------
```

EXP Function

The <exp function> returns, as a real value, e (the base of the natural logarithms) raised to the <arithmetic expression> power.

<exp function> syntax:

```
----  EXP -- ( -- <arithmetic expression> -- ) ----------------------
```

LN Function

The <ln function> returns, as a real value, the natural logarithm of the specified <arithmetic expression>.

<ln function> syntax:

```
----  LN -- ( -- <arithmetic expression> -- ) -----------------------
```

ROUND Function

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.

<round function> syntax:

```
----  ROUND -- ( -- <real expression> -- ) --------------------------
```

Examples:

```
    round(3.5)   yields the value   4

    round(-3.5)  yields the value  -4
```

SIN Function

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.

<sin function> syntax:

```
----  SIN -- ( -- <arithmetic expression> -- ) ----------------------
```

SQR Function

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.

<sqr function> syntax:

```
----  SQR -- ( -- <arithmetic expression> -- ) ----------------------
```

SQRT Function

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.

<sqrt function> syntax:

```
----  SQRT -- ( -- <arithmetic expression> -- ) ---------------------
```

TAN Function

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.

<tan function> syntax:

```
----  TAN -- ( -- <arithmetic expression> -- ) ----------------------
```

TRUNC Function

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.

<trunc function> syntax:

```
-6658902 to 665800
----  TRUNC -- ( -- <real expression> -- ) --------------------------
```

Examples:

```
    trunc(3.5)   yields the value   3

    trunc(-3.5)  yields the value  -3
```

General Procedures and Functions

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.

<general procedure> syntax:

```
----+-- <abort procedure> -----+-----------------------------------
    !                          !
    +-- <accept procedure> ----!
    !                          !
    +-- <date procedure> ------!
    !                          !
    +-- <display procedure> ---!
    !                          !
    +-- <time procedure>-------!
    !                          !
    +-- <wait procedure> ------!
```

<general function> syntax:

```
----+-- <odd function>-------+------------------------------------
    !                        !
    +-- <pred function> -----!
    !                        !
    +-- <runtime function> --!
    !                        !
    +-- <succ function> -----!
```

Abort Procedure

The <abort procedure> forces an immediate, abnormal termination of the program.

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

<abort procedure> syntax:

```
---- ABORT ------------------------------------------------------------
```

Accept Procedure

The <accept procedure> displays the contents of the <string constant> or <string variable> 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> with either blank fill or truncation if the message size is not the same size as the <string variable>. The maximum length of the <string variable> is 255 bytes.

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

<accept procedure> syntax:

```
---- ACCEPT -- ( -+ <string constant> +- , -- <string variable> -- ) -
                  !                   !
                  + <string variable> +
```

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 is placed in str.

Date Procedure

The <date procedure> returns the current date in the parameters <year>, <month>, and <day>. Values returned are all of the <integer type> and are in the following ranges:

parameter	range
<year>	0..9999
<month>	1..12
<day>	1..31

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

<date procedure> syntax:

```
---- DATE -- ( -- <year> -- , -- <month> -- , -- <day> -- ) ----------
```

<year> syntax:

```
---- <variable> ---------------------------------------------------
```

<month> syntax:

```
---- <variable> ---------------------------------------------------
```

<day> syntax:

```
---- <variable> ---------------------------------------------------
```

Example:

```
    var year  : integer;
        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

The <display procedure> displays the contents of the string on the ODT. The maximum length of the display string is 255 bytes.

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

<display procedure> syntax:

```
---- DISPLAY -- ( --+-- <string constant> --+---- ) ----------------
                    !                        !
                    +-- <string variable> --+
```

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.

<odd function> syntax:

```
---- ODD -- ( -- <integer expression> -- ) ---------------------------
```

Example:

```
var b : Boolean;
begin
b := odd(79 mod 27);
end;
```

PRED Function

The <pred function> returns the predecessor of the <ordinal expression>; that is, a value whose ordinal number is one less than that of the <ordinal expression>. If the <ordinal expression> has no predecessor value, an error occurs.

The function returns a result of the same type as the <ordinal expression>.

<pred function> syntax:

```
---- PRED -- ( -- <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

The <runtime function> returns, as a real value (units: seconds), the processor time that has been charged to the program.

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

<runtime function> syntax:

```
---- RUNTIME ---------------------------------------------------------
```

SUCC Function

The <succ function> returns the value of the successor of the <ordinal expression>; that is, the value whose ordinal number is one greater than that of the <ordinal expression>. 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>.

<succ function> syntax:

```
---- SUCC -- ( --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> syntax:

```
---- TIME -- ( -- <hours> -- , -- <minutes> -- , -- <seconds> -- ) ---
```

<hours> syntax:

```
---- <variable> ------------------------------------------------------
```

<minutes> syntax:

```
---- <variable> ------------------------------------------------------
```

<seconds> syntax:

```
---- <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 are of <integer type> and within the following ranges:

parameter	range
<hours>	0..23
<minutes>	0..59
<seconds>	0..59

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

Example:

```
var hours   : integer;
    minutes : integer;
    seconds : 0..59;
begin
time (hours, minutes, seconds);
end;
```

The hour is placed in the variable hours, the number of minutes past the hour is placed in the variable minutes, and the number of seconds into the minute is placed in the variable seconds.

SECTION 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 that determines the values that may be assigned. Another characteristic of a <variable> is its "access." This refers to the method by which it is identified when its value is to be referenced or changed.

This section has three parts: VARIABLES BY ACCESS, VARIABLES BY TYPE, and UNDEFINED VARIABLES. Variables of specific types, such as <array variable>s and <Boolean variable>s, are described in the Variables by Type portion of this section.

VARIABLES BY ACCESS

The access characteristic is basically independent of the type of the variable. In general, the access characteristic depends on whether or not the variable is a component of a structured variable and, if so, on the type of the structured variable of which it is a component. For the variables described in the following paragraphs (entire, indexed, dynamic, and buffer variables, and field designators), the possible access characteristics are defined.

<variable> syntax:

```
----+-- <entire variable> ---+-------------------------------------
    !                        !
    +-- <indexed variable> --+
    !                        !
    +-- <field designator> --+
    !                        !
    +-- <dynamic variable> --+
    !                        !
    +-- <buffer variable> ---+
```

Entire Variables

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.

<entire variable> syntax:

```
---- <variable identifier> ----------------------------------------
```

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

An <indexed variable> denotes a variable that is a component of an array. In order to access an <indexed array variable>, the <array variable> of which it is a component must be identified and the location of the variable 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.

<indexed variable> syntax:

```
---- <indexed array variable> ----------------------------------------
```

<indexed array variable> syntax:

```
                                 +<--------- , -----------+
                                 !                        !
---- <array variable> -- [ --+-- <index expression> --+-- ] ----------
```

<index expression> syntax:

```
---- <ordinal expression> --------------------------------------------
```

Examples:

```
    var x : array [char] of char;
        a : array [Boolean] of 1..10;

    a[false], x['a'], and x['4'] are
    <indexed variable>s.
```

Field Designators

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>.

<field designator> syntax:

```
----+-----------------------------+-- <field identifier> -------------
    !                             !
    +-- <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 any of the following ways:

(1) as the <record variable> of a <with statement>,

(2) as an actual variable parameter in an <actual parameter list>, or

(3) as the left-hand side of an <assignment statement>.

For additional information, refer to Actual Parameter Lists and Parameter Matching in section 3, and Assignment Statements and With Statements in section 4.

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

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>.

<dynamic variable> syntax:

```
---- <pointer variable> -- @ ------------------------------------------
```

An error occurs if the <pointer variable> is NIL, is undefined, contains a mark value, or references a dynamic variable that has been deallocated through the use of the <release procedure>. (See Mark Procedure and Release Procedure in section 6.) It is an error to "release" a dynamic variable while it is being referenced in any of the following ways:

(1) as the <record variable> of a <with statement>,

(2) as an actual variable parameter in an <actual parameter list>, or

(3) as the left-hand side of an <assignment statement>.

Refer to Actual Parameter Lists and Parameter Matching in section 3, Assignment Statements and With Statements in section 4, and Dynamic Allocation Procedures in section 6.

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.

Bufffer Variables

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.

<buffer variable> syntax:

```
----+-- file variable> -------+-- @ ---------------------------------
    !                         !
    +-- <textfile variable> --+
```

It is an error to alter the position of a file while the buffer variable is in use in one of the following ways:

(1) As the <record variable> of a <with statement>,

(2) as an actual variable parameter in an <actual parameter list>, or

(3) as the left-hand side of an <assignment statement>.

Refer to Actual Parameter Lists and Parameter Matching in section 3, and Assignment Statements and With Statements in section 4 for additional information.

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.

VARIABLES BY TYPE

Following are definitions of the variable types.

Array Variable

A <variable> declared of an <array type>.

Boolean Variable

A <variable> declared of the <Boolean type> or of a <subrange type> whose host type is the <Boolean type>.

Char Variable

A <variable> declared of the <char type> or of a <subrange type> whose host type is the <char type>.

Enumerated Variable

A <variable> declared of an <enumerated type> or of a <subrange type> whose host type is an <enumerated type>.

File Variable

An <entire variable> declared of a <file type>.

Integer Variable

A <variable> declared of the <integer type> or of a <subrange type> whose host type is the <integer type>.

Pointer Variable

A <variable> declared of a <pointer type>.

Real Variable

A <variable> declared of the <real type>.

Record Variable

A <variable> declared of a <record type>.

Set Variable

A <variable> declared of a <set type>.

String Variable

A <variable> declared of a <string type>.

Textfile Variable

An <entire variable> of the <textfile type>.

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 the read procedure).

Example:

```
var i : integer;
    j : integer;
begin
j := i;    { ERROR -- the value of i is undefined. }
end;
```

SECTION 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. These components are both simple and widely distributed throught the text. For this reason, they are not explained in place in the text but are explained oncein this section. The components include characters, identifiers, and numbers.

Section 9, Interpretation of Program Text, describes a different set of basic components -- 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 those 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

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. For example, ´´´A´´´ is a <character string> containing the three characters ´A´ (apostrophe, A, apostrophe). A <character string> that contains no values (´´) is a null string.

<character string> syntax:

```
         +<-------------------------------------+
         !                                      !
---- ´ --+--+--------------------------------+--+-- ´ --------------!
            !                                !
            +-- <non-apostrophe character> --+
            !                                !
            +-- ´´ --------------------------+
```

<character literal> syntax:

```
---- ´ --+-- <non-apostrophe character> --+-- ´ ----------------------
         !                                !
         +-- ´´ --------------------------+
```

<non-apostrophe character> definition:

Any <character> except the apostrophe (´).

<character> definition:

Any one of the characters in the standard character set. The standard character set is EBCDIC.

IDENTIFIERS

Identifiers may be of any length greater than 0, subject to the constraint that an identifier may not 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>. (Refer to section 9, Interpretation of Program Text.)

Allowing underscores in identifiers is a Burroughs extension to ANSI Pascal.

<identifier> syntax:

```
                  +<-------------------+
                  !                    !
---- <letter> --+--+--------------+--+-------------------------------
                   !              !
                   +-- <digit> ---+
                   !              !
                   +-- <letter> --+
                   !              !
                   +-- _ ---------+
```

<letter> definition:

Any one of the letters A through Z or a through z. The lower-case characters (a through z) are synonymous with the upper-case characters (A through Z).

<digit> definition:

Any one of the decimal numbers 0 through 9.

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

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 nteger> can be referred to by the predefined <integer constant identifier> maxint.

A number that includes a fractional part or an <exponent part> is an <unsigned real> number. Up to seven significant digits of precision are maintained.

In the <exponent part>, the letter E introduces a decimal exponent. (E has the meaning "times 10 to the power of".) The exponent can range from -47 to +68. The routines that print real numbers print a maximum of six significant digits. This is done so that the last digit can be guaranteed to be accurate.

<number> syntax:

```
----+ --------+-- <unsigned number> ---------------------------------
    !         !
    +--   +   --+
    !         !
    +--   -   --+
```

<unsigned number> syntax:

```
----+-- <unsigned integer> --+------------------------------------
    !                        !
    +-- <unsigned real> -----+
```

<unsigned integer> syntax:

```
    +<------------+
    !             !
----+-- <digit> --+-----------------------------------------------
```

<unsigned real> syntax:

```
    +<------------+  +<------+
    !             !  !       !
----+-- <digit> --+--+-- . --+-- <digit> --+---------------------+----
                  !                        !                     !
                  !                        +-- <exponent part> --+
                  !                                              !
                  +-- <exponent part>----------------------------+
```

<exponent part> syntax:

```
                                +<----------------------+
                                !                       !
----+-- E --+--+----------+--+----/3\---- <digit> --+----------------
    !       !  !          !
    +-- e --+  +--  +   --+
               !          !
               +--  -   --+
```

Examples:

```
    123     -1000     +2              0       { integers }
    0.0     -23.45    24567.4e-20     9E15    { reals }
```

FILE ATTRIBUTES AND MNEMONIC VALUES

File attributes and values are system-defined identifiers describing characteristics of files.

Certain file attributes either may 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 paramente list> may be used in the <setattribute procedure>. If an <attribute parameter list> is used, it must immediately follow the name of the attribute to be accessed.

Attributes:

<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>

<attribute parameter list> syntax:

```
---- [ <integer expression> --+------------------------------+-- ] --
                              !                              !
                              +-- , -- <integer expression> --+
```

Example:

```
    type t = packed array [1..80] of char;
    var f : file of t;
        i : integer;
    begin
    i := 1;
    setattribute(f, TITLE, 'TAPE1');
    end.
```

SECTION 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. 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 dollar sign ($) in column 1 are <compiler control record>s, which are not part of the Pascal program; they are described in appendix A. Records that do not contain a dollar sign ($) 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

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.

<program text> syntax:

```
     +<-- <token separator> -----+
     !                           !
     +<--------------------------+
     !                           !
 ----+-- <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>.

<token> syntax:

```
----+-- reserved word> ------------------+---------------------------
    !                                    !
    +-- <predefined identifier> ---------+
    !                                    !
    +-- <context-sensitive identifier> --+
    !                                    !
    +-- <identifier> -------------------+
    !                                    !
    +-- <number> -----------------------+
    !                                    !
    +-- <character string> --------------+
    !                                    !
    +-- <character literal> -------------+
    !                                    !
    +-- <special token> -----------------+
```

RESERVED WORD

<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.

<reserved word> list:

AND	DIV	FUNCTION	NIL	PROGRAM	UNTIL
ARRAY	DO	GOTO	NOT	RECORD	VAR
BEGIN	DOWNTO	IF	OF	REPEAT	WHILE
CAND	ELSE	IN	OR	SET	WITH
CASE	END	LABEL	OTHERWISE	THEN	
CONST	FILE	LIBRARY	PACKED	TO	
COR	FOR	MOD	PROCEDURE	TYPE	

PREDEFINED IDENTIFIER

<Predefined identifier>s are <identifier>s that have a predefined meaning in Pascal. As with user-defined <identifier>s, <predefined identifier>s may be redefined, but the former definition becomes unavailable within the scope of the redefinition.

<predefined identifier> list

abort	exp	output	setattribute
abs	false	page	sin
accept	get	pred	sqr
arctan	input	put	sqrt
Boolean	integer	read	succ
char	length	readln	tan
chr	ln	real	text
close	log	release	time
cos	mark	reset	true
date	maxint	rewrite	trunc
display	new	round	write
eof	odd	runtime	writeln
eoln	ord	seek	

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.

<token separator> syntax:

```
----+-- <blank> -----------+------------------------------------
    !                      !
    +-- <comment>----------+
    !                      !
    +-- <record boundry> --+
```

BLANK

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 properly.

<blank> definition:

One or more blank characters.

COMMENT

Comments are used to include documentation in a program. A <comment> may appear anywhere that a <blank> can appear; a <comment> may not appear in a <character string> or in another <comment>. Comments may contain any <character>s except the delimiting characters } and *).

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.

<comment> syntax:

```
                  +<----------------+
                  !                 !
----+-- { ----+--+-- <character> --+-- }  --+------------------------
    !         !                    !        !
    +-- (*  --+                    +-- *) --+
```

Examples

```
{ This is a comment. }
(* This comment uses the two-character synonyms for braces. *)
```

RECORD BOUNDARY

The <record boundary> acts as an implicit token separator. Thus, a token cannot be split at the column 72 boundary 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.

<record boundary> definition:

A theoretical boundary between column 72 of one record and column 1 of the next record.

APPENDIX A

COMPILING, EXECUTING, AND ANALYZING A PASCAL PROGRAM

The input file to the B 1000 Pascal compiler is a standard data file created by any of the various editors. Only the first 72 characters of each record are significant. Sequence numbers may appear in positions 73 through 80. These are not used by the compiler but are printed on the listing. Any patch information that may be present in columns 81-90 also appears on the listing.

The Pascal code may be entered in free format, but the general rules for formatting, as illustrated in any Pascal textbook, should be followed to create readable source programs.

COMPILER OPTIONS

Certain aspects of the compilation of a Pascal program may be controlled by directives to the compiler in the form of compiler control images (CCIs).

The CCI enables a user to control options that are provided in the Pascal compiler. Each option falls into one of the following six categories:

- Source language inputs
- Source language output
- Optional compilation mechanism
- Printed outputs
- Compiler diagnostic messages
- Compiler debugging

A CCI contains compiler control statements comprised of options or groups of options and any associated parameters. CCIs are totally distinct from the Pascal language, although they are typically interspersed with program source lines. CCI syntax differs from Pascal source syntax. Also, the following conventions differ between Pascal source text and CCI text.

1. CCIs may not contain comments.

2. Only upper-case letters may be used in CCIs, except within character strings.

3. Character strings (for example, in file titles) are delimited by double quotation marks ("), not apostrophes (´).

Because a CCI is not part of the Pascal language, a Pascal comment cannot occlude a CCI. Any source image with a dollar sign ($) in column 1 is processed as a CCI by the Pascal compiler, even if a Pascal comment begins before and ends after the CCI.

CCI Syntax Diagrams

The syntax diagrams for CCIs are shown next. Options that are allowed within a Pascal source are listed in the paragraphs that follow under the headings Boolean Options, Value Options, and Immediate Options. Except as noted, the syntax and semantics of these options are as specified by the CCI Standard.

NOTE

> The CCI Standard is a Burroughs document. The full title is Burroughs Corporation CSG Standard for Compiler Control Images.

CCI Syntax:

```
------ $ -------------------------------------------------------------
    !      !       !                              !
    !- $$ -!       !  !<-----------------------!  !
                   !  !                        !  !
                   !----- <Boolean-option> -------!
                      !                        !
                      !-- <value-option> ------!
                      !                        !
                      !-- <immediate-option> --!

>--------------------------------------------------------------------
      !                                                        !
      !    !<---------------------------------------------!    !
      !    !                                              !    !
      !    !          !<-----------------------------!    !    !
      !    !          !                              !    !    !
      !------- SET ------ <Boolean-option-setting> ------------!
           !          !                              !    !
           !          !-- <value-option> ------------!    !
           !          !                              !    !
           !          !-- <immediate-option> --------!    !
           !                                              !
           !          !<---------------------!            !
           !          !                      !            !
           !-- RESET--!-- <Boolean-option> --!------------!
           !          !
           !-- POP ---!
```

<Boolean-option> syntax:

```
------ <user-specified Boolean option identifier> --------------------
   !                                                                  !
   !-- <one of the predefined Boolean options described below> --!
```

<value-option> syntax:

```
------<one of the value options described below> --------------------
```

<immediate-option> syntax:

```
------ <one of the immediate options described below> ---------------
```

<Boolean-option-setting> syntax:

```
---- <Boolean-option> ----------------------------------------------
                      !                                            !
                      !--- = -- <Boolean-option-expression> ---!
```

<Boolean-option-expression> syntax:

```
------ <option-term> -----------------------------------------------
                     !                              !
                     !   !<---------------------!   !
                     !   !                      !   !
                     !---- OR -- <option-term> ----!
```

<option-term> syntax:

```
------ <option-factor> ---------------------------------------------
                       !                                  !
                       !   !<-----------------------!   !
                       !   !                        !   !
                       !---- AND -- <option-factor> ----!
```

<option-factor> syntax:

```
------- ( -- <Boolean-option-expression> -- ) -----------------------
    !                                        !
    !----- NOT -- <option-factor> -----------!
    !                                        !
    !----- <Boolean-option> ----------------!
    !                                        !
    !----- TRUE -----------------------------!
    !                                        !
    !----- FALSE ----------------------------!
```

NOTE

$ must be in column 1 or $$ in columns 1 and 2 of a CCI. The listing of a CCI with $$ is controlled by LIST and LISTINCL, not by LISTDOLLAR. User options are implicity declared by their first use, which may not be in a Boolean-option-expression. The usual precedence of Boolean operators (NOT, AND, OR) is used.

Boolean Options

The following Boolean options are defined in the CCI Standard

ANSI
Default = FALSE. The ANSI option causes any extensions to the ANSI Pascal Reference Standard to be treated as errors. Enabling this option currently has no effect.

CODE
Default = FALSE. The CODE option causes the compiler to produce a listing of the object code produced by the compilation process.

LINEINFO
Default = FALSE. The LINEINFO option causes the compiler to generate operators to determine the source line number in case of abnormal termination. If the option is not enabled, the line number of the beginning of the active procedure is determined instead.

LIST
Default = TRUE. The LIST option causes the compiler to include in the listing the source derived from the CARD file.

LISTDOLLAR
Default = FALSE. The LISTDOLLAR option causes the compiler to include in the listing all CCIs (single $) encountered during the compilation. LIST must also be TRUE.

LISTINCL
Default = FALSE. The LISTINCL option causes the compiler to include in the listing that part of the source which was accepted for compilation as a result of the enabling of the INCLUDE option. LIST must also be TRUE.

MAP
Default = FALSE. The function normally associated with this option is to produce an output listing with information cross referencing line numbers to object code addresses. However, this function is not needed because the Pascal

compiler error message and the analyzer program output reference source line numbers rather than code addresses. The MAP option in this compiler is actually equivalent to the CODE option.

NOBOUNDS
Default = FALSE. The NOBOUNDS option causes the compiler to forego generating operators to check for subrange variables going out of range assignments.

NOTAGFIELD
Default = FALSE. VARIANT causes the compiler to forego generating operators to check tag values on accesses to fields of tagged record variants.

OMIT
Default = FALSE. The OMIT option causes all source language images to be ignored for the purpose of compilation until it is disabled. Any source language images encountered while this option is enabled are processed in the normal manner. A lower-case letter o is printed on the listing just before the sequence number field for all records that are omitted.

XREF
Default = FALSE. The XREF option produces a listing of the line number where each identifier is referenced. The XREF option may be SET and RESET to cross reference various portions of a program.

NOTE

The cross reference option currently uses a memory sort. If a program with a large number of identifiers is being cross referenced, then the compile will require more memory than when cross referencing is not being done. The code file is closed before the cross reference is started so that the code file is saved even if the cross reference routines run out of memory.

Value Options

The following value options are defined in the CCI Standard.

ERRORLIMIT
Default value = 100. Causes compilation to terminate when the number of errors detected by the compiler equals or exceeds the integer value specified.

ERRORLIMIT Syntax:

```
---- ERRORLIMIT ------ = ----------- 100 ------------------
                                   !              !
                                   !--- <integer>---!
```

STRINGS
Default = EBCDIC. Input to the compiler is assumed to be in EBCDIC. If this option is set to ASCII, all character and string literals generated to the code file are translated from EBCDIC to ASCII. No translation occurs with the option set to EBCDIC.

STRINGS Syntax:

```
---- STRINGS --------- = -------- EBCDIC ------------------
                                    !          !
                                    !--- ASCII ---!
```

Immediate Options

The following immediate options are defined in the CCI Standard.

CLEAR
This option causes the compiler to disable (set false) the following Boolean options: ANSI, CODE, LIST, LISTDOLLAR, LISTINCL, OMIT, XREF.

PAGE
This option causes the compiler to eject a page on the output listing if the appropriate list options are set.

INCLUDE
This option causes the compiler to suspend reading input from the CARD file and to begin reading input from the file specified by the parameter. An INCLUDE CCI may not appear in the included file. The file-title is specified using the ON syntax; that is, Y/Z ON X means file X/Z on pack X. No other option may follow the INCLUDE on the same input image. If file-title has a quotation mark (") within it, it must be represented by two quotation marks (""). A lower-case letter i is printed on the listing just before the sequence number field for all records that are included.

INCLUDE Syntax:

```
---- INCLUDE --- " --- <file-title> --- " ------------------!
```

COMPILING AND EXECUTING A PASCAL PROGRAM

The Pascal compiler, PASCAL, is itself a Pascal program. It has three external files:

1. CARD, the program source text, modified to be DISK.

2. LINE, the program listing, modified to PRINTER BACKUP.

3. CODE, the B 1000 code file.

The compiler is run by using the MCP COMPILE command, usually with file statements to name its external files and possibly a static memory (MS) specification for a large compilation. Standard memory size is 500,000 bits. The LIBRARY and SYNTAX options of the COMPILE command both have the same effect of compiling to LIBRARY.

The compiler automatically segments the object code. A code segment is filled with at least 1500 bytes of code. At the end of the procedure in which the code segment was filled to 1500 bytes, a segment is started for the next procedure. Procedures are never broken across segments, but several procedures may be placed into one segment.

The file CODE is saved unless the program being compiled has syntax errors. The saved file is locked into the directory with the name that was assigned in the COMPILE command

Example:

```
COMPILE PROG WITH PASCAL TO LIBRARY;
FILE CARD NAME = SOURCE/PROG;
FILE LINE NAME = LIST/PROG   USER.BACKUP.NAME;
```

Compile-Time Errors

Each error detected at compile time is printed on the listing following the line in error, with a special character that points to the token that was being scanned when the error was detected. In some instances, the symbol being pointed to follows the actual error point, because the compiler parsed ahead before the error was evident to it.

Run-Time Errors

Errors detected at run time are reported by means of the MCP DS OR DP message. A standard run-time error message contains a segment number and displacement, usually of the program's next instruction pointer. In the case of Pascal, however, the segment number is always zero and the displacement value is the source line number at which the program failed.

Example:

```
TEST = 1631 -- VALUE OUT OF RANGE: S=0, D=13 (@000@,@00000D@); DS OR DP
```

In this example, TEST =1631 is the job name and number supplied by the MCP, and D=13 shows that the error occurred on line 13 of the source listing.

Some standard routines such as the routine to read and write real numbers are contained in a library file (PASCAL/LIBRARY). When a program uses any of the routines, the library is bound with the code of the program. If an error occurs in a library routine, the line number of the error is in the library rather than in the invoking program. The best way to determine the program line from which the library routine was called is to run the PASCAL/ANALYZER program on a dump of the program. The dump analysis shows the appropriate line. The PASCAL/ANALYZER program is described later in this appendix.

A run-time error may occur incorrectly when a program is close to running out of memory. If an error seems questionable, try running the program again with more memory.

Following is a list of all the run-time errors with notes on possible causes.

INDEX OUT OF RANGE
: The value of the expression used to index an array is outside the bounds of the array.

VALUE OUT OF RANGE
: The value of the expression is outside the range of the variable to which the expression is being assigned.

INTEGER OVERFLOW
: The value the expression is greater that maxint or less than -maxint.

REAL OVERFLOW
: The exponent part of the real-valued expression is greater than the maximum exponent for real numbers.

INV PTR REFERENCE
: A pointer which was pointing above the current top of the heap was dereferenced. The item that the pointer is pointing to has already been released.

DIVIDE BY ZERO
: A division or modulo by zero was attempted.

STACK LIMIT
The program has run out of memory while trying to allocate space for local variables. Run the program again with more memory using the MCP MS command.

HEAP LIMIT
The program has run out of memory while trying to allocate space for a dynamic variable. Run the program again with more memory using the MCP MS command.

SET OUT OF RANGE
A member of the set expression is outside the range of the set to which it is being assigned.

INVALID OPCODE
The interpreter attempted to execute an invalid operator.

INV STD ROUTINE
The compiler generated faulty code which resulted in an attempt to call an invalid standard routine.

VARIANT ERROR
A field of a variant record was accessed and the value of the tag field does not correspond to the variant part containing this field.

NIL POINTER ERROR
A pointer with the value of NIL was dereferenced.

INVALID CASE
A CASE statement was executed but the value of the case selector does not correspond to any case label and the case statement has no OTHERWISE clause.

FILE AT EOF
A file operation was attempted but the end of the file was encountered.

PROGRAM ABORT
The program was terminated by calling the ABORT procedure.

TEXT BUF OVERFLOW
Too many WRITE operations without a WRITELN procedure to this textfile have been done. Either insert a WRITELN procedure or increase the size of the buffer associated with this textfile using the file attribute specification in the program heading.

FILE NOT OPEN
A file operation was attempted on an unopen file.

UNDEFINED POINTER
A pointer which has not been assigned any value has been dereferenced.

FILE NOT AT EOF
A file operation was attempted but the file was not at end of file.

INVALID CHAR READ
An invalid character was encountered during an attempt to read an integer from a textfile.

FILE NOT CLOSED
A file operation was attempted which required the file to be closed, but it is open.

USING THE PASCAL/ANALYZER PROGRAM

When a run-time error occurs, the user has the option of getting a dump file of the current state of the program.

The standard analyzer program (SYSTEM/IDA) can be used to analyze dumps of Pascal programs, but it is not based on the internal structure of the Pascal virtual machine and, thus, produces a very general analysis. It is invoked with the MCP PM command, with switch 1 set to 1, and analyzes standard program components such as the run structure nucleus and file information blocks. Values of variables and the nesting of procedures are not shown.

The PASCAL/ANALYZER program is written specifically to analyze dumps of Pascal programs and is based on the Pascal run-time system. It contains two external files:

!bu DUMPFILE, the input dump file created by the MCP.

!bu LINE, the output listing file.

The PASCAL/ANALYZER program gives a detailed analysis of the state of the program at the point at which the error occurred.

The output is organized as follows:

The program name and date and the name of the run-time error appear at the top of the printout.

The values of all of the scratchpad registers are next.

Information for each file that was declared in the program is given next.

Analysis of the stack appears next. Each activation record, beginning with the most recent one, is analyzed. The analysis of each activation record includes the local variable, stack temporaries, and parameters. The name and current value of each variable is included.

At the end, the contents of the heap are printed in hexadecimal.

The PASCAL/ANALYZER program is executed as follows:

```
EX PASCAL/ANALYZER;
FILE DUMPFILE NAME DUMPFILE/124;
FILE LINE   NAME PROG/DUMP USER.BACKUP.NAME
```

USING THE SYSTEM/IDA PROGRAM

The SYSTEM/IDA program (the standard analyzer) is executed as follows:

```
PM 124; SW 1 = 1
```

DUMPFILE/124 is removed when the analysis is done. To retain the dump, file invoke the SYSTEM/IDA program with the following command:

```
PM 124 SAVE; SW 1 = 1
```

APPENDIX B

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 (!ra) 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 (!vr).

Items contained in broken brackets (<>) are syntactic variables that are defined in the manual or are information that the user is required to supply.

Upper-case items not enclosed in broken brackets must appear literally. Minimum abbreviations of upper-case items are underlined.

Example:

```
                                     !<-/3\------- , -------!
                                     !                      !
-- A RAILROAD DIAGRAM CONSISTS OF -----<bridges>-------------------->
                                       !                  !
                                       !-<loops>----------!
                                       !                  !
                                       !-<optional items>-!
                                       !                  !
                                       !-<required items>-!

>- AND IS TERMINATED BY A VERTICAL BAR. ---------------------------!
```

The following syntactically valid statements can be constructed from the preceding diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items>, <required items>, <optional items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

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 must 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 statements can 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:

```
  !<- <bridges> <return character>-!
  !                                !
----<object of the loop>-----------------------------------------
```

Example:

```
  !<-/1\-------- , -------!
  !                       !
------<optional item-1>-------------------------------------------------
    !                  !
    !-<optional item-2>-!
```

The following statements can be constructed from the railroad diagram in the preceding example.

```
<optional item-1>
<optional item-2>
<optional item-1>,<optional item-1>
<optional item-1>,<optional item-2>
<optional item-2>,<optional item-1>
<optional item-2>,<optional item-2>
```

A loop must be traversed in the direction of the arrowheads, and the limits specified by bridges cannot be exceeded.

BRIDGES

A bridge illustrates the minimum or maximum number of times a path can be traversed in a railroad diagram.

There are two forms of bridges:

/n\ n is an integer that specifies the maximum number of times th
path may be traversed.

/n*\ n is an integer that specifies the maximum number of times t
path may be traversed. The asterisk (*) indicates that the pa
must be traversed at least once.

Example:

```
  !<-/2\---------- , ----------!
  !                            !
------<optional item-1>-------------------------------------------------
    !                        !
    !-/2*\-<optional item-2>-!
```

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 statements can be constructed from the preceding diagram:

<optional item-1>,<optional item-2>

<optional item-2>,<optional item-2>,<optional item-1>

<optional item-2>

APPENDIX C

EBCDIC AND ASCII CHARACTER SETS

Tables C-1 and C-2 show the hexadecimal representation and ordinal number for each EBCDIC and ASCII character. Table C-1 is sorted by EBCDIC ordinal number and represents the EBCDIC-to-ASCII translation that is performed when necessary. Table C-2 is sorted by ASCII ordinal number and represents the ASCII-to-EBCDIC translation that is performed when necessary.

NOTES

The graphic representations for the EBCDIC hex codes 15, 5F, 6A, 79, and A1 are hardware dependent. Therefore, no EBCDIC graphic is shown in table C-1 for those codes.

Similarly, the graphic representations for the ASCII hex codes 21, 5E, 6C, and 7C are hardware dependent. Therefore, no ASCII graphic is shown in table C-2 for those codes.

Table C-1. B 1000 Codes in EBCDIC Sequence

EBCDIC		ASCII (EBCDIC Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
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
26	38	17	23	ETB	End of Transmission Bloc
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

Table C-1. (continued)

EBCDIC Hex	EBCDIC Decimal	ASCII Hex	ASCII Decimal	Graphic	Meaning
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		
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		

Table C-1. (continued)

E B C D I C		A S C I I (EBCDIC Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
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		
6B	107	2C	44	,	Comma
6C	108	25	37	%	Percent
6D	109	5F	95	_	Underscore
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		
7A	122	3A	58	:	Colon
7B	123	23	35	#	Number Sign
7C	124	60	64	@	Commercial At
7D	125	27	39	'	Apostrophe, Closing Quot
7E	126	3D	61	=	Equal Sign
7F	127	22	34	"	Quotation Marks
80	128	C3	195		
81	129	61	97	a	Lower Case a
82	130	62	98	b	Lower Case b
83	131	63	99	c	Lower Case c
84	132	64	100	d	Lower Case d
85	133	65	101	e	Lower Case e
86	134	66	102	f	Lower Case f
87	135	67	103	g	Lower Case g
88	136	68	104	h	Lower Case h
89	137	69	105	i	Lower Case 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		

Table C-1. (continued)

EBCDIC		ASCII (EBCDIC Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
90	144	CA	202		
91	145	6A	106	j	Lower Case j
92	146	6B	107	k	Lower Case k
93	147	6C	108	l	Lower Case l
94	148	6D	109	m	Lower Case m
95	149	6E	110	n	Lower Case n
96	150	6F	111	o	Lower Case o
97	151	70	112	p	Lower Case p
98	152	71	113	q	Lower Case q
99	153	72	114	r	Lower Case 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		
A2	162	73	115	s	Lower Case s
A3	163	74	116	t	Lower Case t
A4	164	75	117	u	Lower Case u
A5	165	76	118	v	Lower Case v
A6	166	77	119	w	Lower Case w
A7	167	78	120	x	Lower Case x
A8	168	79	121	y	Lower Case y
A9	169	7A	122	z	Lower Case 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		
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		

Table C-1. (continued)

EBCDIC Hex	EBCDIC Decimal	ASCII Hex	ASCII Decimal	Graphic	Meaning
C0	192	7B	123	{	Opening Brace
C1	193	41	65	A	Upper Case A
C2	194	42	66	B	Upper Case B
C3	195	43	67	C	Upper Case C
C4	196	44	68	D	Upper Case D
C5	197	45	69	E	Upper Case E
C6	198	46	70	F	Upper Case F
C7	199	47	71	G	Upper Case G
C8	200	48	72	H	Upper Case H
C9	201	49	73	I	Upper Case 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	Upper Case J
D2	210	4B	75	K	Upper Case K
D3	211	4C	76	L	Upper Case L
D4	212	4D	77	M	Upper Case M
D5	213	4E	78	N	Upper Case N
D6	214	4F	79	O	Upper Case O
D7	215	50	80	P	Upper Case P
D8	216	51	81	Q	Upper Case Q
D9	217	52	82	R	Upper Case 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	Upper Case S
E3	227	54	84	T	Upper Case T
E4	228	55	85	U	Upper Case U
E5	229	56	86	V	Upper Case V
E6	230	57	87	W	Upper Case W
E7	231	58	88	X	Upper Case X
E8	232	59	89	Y	Upper Case Y
E9	233	5A	90	Z	Upper Case 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		

Table C-1. (continued)

EBCDIC		ASCII (EBCDIC Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
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		

Table C-2. B 1000 Codes in ASCII Sequence

ASCII		EBCDIC (ASCII Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
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 Bloc
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		
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, Single Quote
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
2F	47	61	97	/	Slant (Slash)

Table C-2. (continued)

ASCII Hex	ASCII Decimal	EBCDIC (ASCII Graphic) Hex	Decimal	Graphic	Meaning
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	Upper Case A
42	66	C2	194	B	Upper Case B
43	67	C3	195	C	Upper Case C
44	68	C4	196	D	Upper Case D
45	69	C5	197	E	Upper Case E
46	70	C6	198	F	Upper Case F
47	71	C7	199	G	Upper Case G
48	72	C8	200	H	Upper Case H
49	73	C9	201	I	Upper Case I
4A	74	D1	209	J	Upper Case J
4B	75	D2	210	K	Upper Case K
4C	76	D3	211	L	Upper Case L
4D	77	D4	212	M	Upper Case M
4E	78	D5	213	N	Upper Case N
4F	79	D6	214	O	Upper Case O
50	80	D7	215	P	Upper Case P
51	81	D8	216	Q	Upper Case Q
52	82	D9	217	R	Upper Case R
53	83	E2	226	S	Upper Case S
54	84	E3	227	T	Upper Case T
55	85	E4	228	U	Upper Case U
56	86	E5	229	V	Upper Case V
57	87	E6	230	W	Upper Case W
58	88	E7	231	X	Upper Case X
59	89	E8	232	Y	Upper Case Y
5A	90	E9	233	Z	Upper Case Z
5B	91	4A	74	[	Opening Bracket
5C	92	E0	224	\	Reverse Slant
5D	93	5A	90	]	Closing Bracket
5E	94	5F	95		
5F	95	6D	109	_	Underscore

Burroughs

Table C-2. (continued)

ASCII Hex	ASCII Decimal	EBCDIC Hex	EBCDIC Decimal	Graphic	Meaning
60	96	79	121		
61	97	81	129	a	Lower Case a
62	98	82	130	b	Lower Case b
63	99	83	131	c	Lower Case c
64	100	84	132	d	Lower Case d
65	101	85	133	e	Lower Case e
66	102	86	134	f	Lower Case f
67	103	87	135	g	Lower Case g
68	104	88	136	h	Lower Case h
69	105	89	137	i	Lower Case i
6A	106	91	145	j	Lower Case j
6B	107	92	146	k	Lower Case k
6C	108	93	147	l	Lower Case l
6D	109	94	148	m	Lower Case m
6E	110	95	149	n	Lower Case n
6F	111	96	150	o	Lower Case o
70	112	97	151	p	Lower Case p
71	113	98	152	q	Lower Case q
72	114	99	153	r	Lower Case r
73	115	A2	162	s	Lower Case s
74	116	A3	163	t	Lower Case t
75	117	A4	164	u	Lower Case u
76	118	A5	165	v	Lower Case v
77	119	A6	166	w	Lower Case w
78	120	A7	167	x	Lower Case x
79	121	A8	168	y	Lower Case y
7A	122	A9	169	z	Lower Case z
7B	123	C0	192	{	Opening Brace
7C	124	6A	106		
7D	125	D0	208	}	Closing Brace
7E	126	A1	161		
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		
8C	140	2C	44		
8D	141	09	9		
8E	142	0A	10		
8F	143	1B	27		

Table C-2. (continued)

ASCII		EBCDIC (ASCII Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
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		
BC	188	72	114		
BD	189	73	115		
BE	190	74	116		
BF	191	75	117		

Table C-2. (continued)

ASCII		EBCDIC (ASCII Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
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		
EC	236	CE	206		
ED	237	CF	207		
EE	238	DA	218		
EF	239	DB	219		

Table C-2. (continued)

ASCII		EBCDIC (ASCII Graphic)			
Hex	Decimal	Hex	Decimal	Graphic	Meaning
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		

INDEX